РЕКОМЕНДОВАНО
Sun Microsystems

В. Монахов

Язык программирования Java и среда NetBeans

3-е издание

- Основные концепции и методы объектно-ориентированного программирования
- Основные концепции UML-проектирования
- Программирование мобильных устройств
- Написание серверного программного обеспечения
- Численные расчеты с использованием Java
- Программы для систем с многоядерными процессорами
- Desktop Application — новый вид программ с графическим интерфейсом

+ dvd Дистрибутивы JDK 6 и NetBeans 6.9 для Windows и Linux

bhv

В. Монахов

Язык программирования Java и среда NetBeans

3-е издание

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.068
ББК 32.973.26-018.1
М77

Монахов В. В.

М77 Язык программирования Java и среда NetBeans. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 704 с.: ил. + DVD

ISBN 978-5-9775-0671-7

Книга написана на базе курса лекций, читаемых автором на кафедре вычислительной физики Санкт-Петербургского государственного университета. Изложены основные синтаксические конструкции Java, принципы объектно-ориентированного программирования, особенности проведения численных расчетов. Приводятся сведения о среде NetBeans, предназначенной для профессиональной разработки всех видов программного обеспечения Java (ME, SE и EE). Рассказывается о создании сетевых приложений и приложений для мобильных устройств. Разбираются методики написания многопоточных приложений Java для систем с многоядерными процессорами. Материал сопровождается большим количеством примеров. В третьем издании внесены исправления и на DVD добавлены новые примеры и дистрибутив NetBeans 6.9.1. Также на DVD расположены дистрибутивы JDK 6u22 и NetBeans 6.5 для Windows и Linux.

Для начинающих Java-программистов и студентов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.12.10.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 56,76.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0671-7

© Монахов В. В., 2011
© Оформление, издательство "БХВ-Петербург", 2011

Оглавление

Предисловие к третьему изданию **11**

Введение **13**

Глава 1. Общие представления о языке Java **17**
1.1. Java и другие языки программирования. Системное и прикладное программирование 17
1.2. Виртуальная Java-машина, байт-код, JIT-компиляция. Категории программ, написанных на языке Java 22
1.3. Алфавит языка Java. Десятичные и шестнадцатеричные цифры и целые числа. Зарезервированные слова 28
Алфавит языка Java 29
Десятичные и шестнадцатеричные цифры и целые числа 29
Зарезервированные слова и литералы языка Java 31
1.4. Управляющие последовательности. Символы Unicode. Специальные символы 32
Управляющие последовательности 32
Управляющие последовательности — символы формирования текста 32
Управляющие последовательности — символы Unicode 32
Простые специальные символы 33
Составные специальные символы 34
1.5. Идентификаторы. Переменные и типы. Примитивные и ссылочные типы 35
1.6. Процедурное и объектно-ориентированное программирование. Инкапсуляция 38
1.7. Работа со ссылочными переменными. Сборка мусора 43
1.8. Проекты. Пакеты. Уровни видимости классов. Импорт классов 47
1.9. Базовые пакеты и классы Java 51
1.10. Технологии Java, .NET, ASP, PHP 55
1.11. Среды разработки NetBeans, Eclipse, JDeveloper, JBuilder, IntelliJ IDEA 59
NetBeans 6.5 61
Eclipse 3.4.1 (Ganimede) 63
JDeveloper 11 65
JBuilder 2008 67
IntelliJ IDEA 7.0.4 69
Общий итог сравнения сред разработки 70
Краткие итоги по главе 70

Типичные ошибки ... 73
Задания ... 74

Глава 2. Среда NetBeans 6.5 ... 75
2.1. Установка JDK 6 ... 75
2.2. Установка среды NetBeans 6.5 ... 79
2.3. Создание в NetBeans простейшего приложения Java ... 87
2.4. Компиляция файлов проекта и запуск приложения. Jar-файлы ... 92
2.5. Структура проекта NetBeans ... 96
2.6. Создание в NetBeans приложения Java с графическим интерфейсом ... 100
2.7. Редактор экранных форм ... 105
2.8. Свойства компонентов ... 110
2.9. Внешний вид приложения — технология Look and Feel ... 116
2.10. Ведение проектов ... 119
2.11. Редактирование меню экранной формы ... 121
2.12. Создание нового класса ... 129
2.13. Документирование исходного кода в Java ... 130
2.14. Основные компоненты пакетов *swing* и *awt* ... 139
2.15. Менеджеры размещения (*Layout*) и якоря (*Anchor*) ... 144
2.16. Создание приложения Desktop Application ... 145
Краткие итоги по главе ... 147
Типичные ошибки ... 148
Задания ... 148

Глава 3. Примитивные типы данных и операторы для работы с ними ... 150
3.1. Булевый (логический) тип ... 150
3.2. Целые типы, переменные, константы ... 151
3.3. Основные операторы для работы с целочисленными величинами ... 155
3.4. Вещественные типы и класс *Math* ... 156
3.5. Правила явного и автоматического преобразования типа при работе с числовыми величинами ... 160
3.6. Оболочечные классы. Упаковка (*boxing*) и распаковка (*unboxing*) ... 162
3.7. Приоритет операторов ... 164
Краткие итоги по главе ... 165
Типичные ошибки ... 167
Задания ... 167

Глава 4. Работа с числами в языке Java ... 169
4.1. Двоичное представление целых чисел ... 169
Позиционные и непозиционные системы счисления ... 169
Двоичное представление положительных целых чисел ... 171
Двоичное представление отрицательных целых чисел. Дополнительный код ... 172
Сложение положительного и отрицательного чисел ... 173
Сложение отрицательных чисел ... 174
Проблемы целочисленной машинной арифметики. Целочисленное переполнение ... 174
Сложение положительных чисел ... 174
Умножение положительных чисел ... 175
Шестнадцатеричное представление целых чисел и перевод из одной системы счисления в другую ... 177
Преобразование чисел из системы с меньшим основанием в систему с большим основанием ... 178

Преобразование чисел из системы с большим основанием в систему с меньшим основанием ... 178
Преобразование чисел в системах счисления с кратными основаниями ... 179
4.2. Побитовые маски и сдвиги ... 180
4.3. Двоичное представление вещественных чисел ... 183
Двоичные дроби ... 183
Мантисса и порядок числа ... 183
Научная нотация записи вещественных чисел ... 185
Стандарт IEEE 754 представления чисел в формате с плавающей точкой ... 186
Краткие итоги по главе ... 190
Типичные ошибки ... 191
Задания ... 191

Глава 5. Управляющие конструкции ... 193
5.1. Составной оператор ... 193
5.2. Условный оператор *if* ... 194
5.3. Оператор выбора *switch* ... 198
5.4. Условное выражение *...?... :* ... 199
5.5. Операторы инкремента ++ и декремента -- ... 200
5.6. Оператор цикла *for* ... 200
5.7. Ошибки при использовании вещественного счетчика цикла ... 202
5.8. Эффективная организация циклов при вычислениях в формате с плавающей точкой ... 205
5.9. Особенности целочисленных вычислений — организация циклов, приоритет операторов и арифметическое переполнение ... 210
5.10. Оператор цикла *while* — цикл с предусловием ... 215
5.11. Оператор цикла *do...while* — цикл с постусловием ... 217
5.12. Операторы прерывания *continue*, *break*, *return*, *System.exit* ... 217
Краткие итоги по главе ... 221
Типичные ошибки ... 222
Задания ... 222

Глава 6. Начальные сведения об объектном программировании ... 223
6.1. Наследование и полиморфизм ... 223
6.2. Функции. Модификаторы. Передача примитивных типов в функции ... 231
6.3. Локальные и глобальные переменные. Модификаторы доступа и правила видимости. Ссылка *this* ... 235
6.4. Передача ссылочных типов в функции. Проблема изменения ссылки внутри подпрограммы ... 238
6.5. Наследование. Суперклассы и подклассы. Переопределение методов ... 245
6.6. Наследование и правила видимости. Зарезервированное слово *super* ... 250
6.7. Статическое и динамическое связывание методов. Полиморфизм ... 252
6.8. Базовый класс *Object* ... 254
6.9. Конструкторы. Зарезервированные слова *super* и *this*. Блоки инициализации ... 256
6.10. Удаление неиспользуемых объектов и метод *finalize*. Проблема деструкторов для сложно устроенных объектов ... 260
6.11. Перегрузка методов ... 261
6.12. Правила совместимости ссылочных типов как основа использования полиморфного кода. Приведение и проверка типов ... 263

Краткие итоги по главе ... 267
Типичные ошибки ... 268
Задания ... 269

Глава 7. UML-диаграммы. Прямое и обратное проектирование. Рефакторинг ... 271
7.1. UML-диаграммы ... 271
7.2. Структура окон проектов с UML-диаграммами ... 279
7.3. Панель инструментов проектов с UML-диаграммами ... 282
7.4. Прямое проектирование (Forward engineering) — построение кода классов по UML-диаграммам ... 286
7.5. Обратное проектирование (Reverse engineering) — построение UML-диаграмм по разработанным классам ... 293
7.6. Рефакторинг ... 298
Первый случай — переименование элементов программы ... 301
Второй случай — перемещение класса ... 303
Третий случай — копирование класса для его дальнейшей модификации ... 303
Четвертый случай — безопасное удаление элемента программы ... 304
Пятый случай — изменение списка параметров метода ... 304
Шестой случай — инкапсуляция полей данных ... 305
Краткие итоги по главе ... 306
Типичные ошибки ... 306
Задания ... 307

Глава 8. Важнейшие объектные типы ... 308
8.1. Массивы ... 308
8.2. Коллекции, списки, итераторы ... 314
8.3. Перебор в цикле элементов коллекций. Оператор цикла *for-each* ... 318
8.4. Работа со строками в Java. Строки как объекты. Классы *String*, *StringBuffer* и *StringBuilder* ... 319
8.5. Типы-перечисления (enum) ... 333
8.6. Работа с датами и временем. Класс *GregorianCalendar* ... 335
8.7. Работа с графикой и графическим пользовательским интерфейсом ... 338
Код визуализации и код бизнес-логики приложения ... 338
Графические примитивы ... 339
Пример метода, работающего с графикой ... 341
8.8. Исключительные ситуации ... 345
Обработка исключительных ситуаций ... 345
Иерархия исключительных ситуаций ... 348
Объявление типа исключительной ситуации и оператор *throw* ... 351
Объявление метода, который может возбуждать исключительную ситуацию. Зарезервированное слово *throws* ... 352
8.9. Работа с файлами и папками ... 355
Работа с файлами и папками с помощью объектов типа *File* ... 356
Выбор файлов и папок с помощью файлового диалога ... 359
Работа с потоками ввода/вывода ... 364
Краткие итоги по главе ... 372
Типичные ошибки ... 373
Задания ... 374

Глава 9. Интерфейсы и композиция 375
9.1. Проблемы множественного наследования классов. Интерфейсы 375
9.2. Отличия интерфейсов от классов. Проблемы наследования интерфейсов 378
9.3. Пример работы с интерфейсами 380
9.4. Композиция — еще одна альтернатива множественному наследованию 383
Краткие итоги по главе 385
Типичные ошибки 385
Задания 385

Глава 10. Многопоточное программирование и многоядерные системы 387
10.1. Потоки выполнения (threads) и синхронизация 387
10.2. Преимущества и проблемы при работе с потоками выполнения 388
10.3. Синхронизация по ресурсам и событиям 389
Синхронизация по ресурсам 389
Синхронизация по событиям 391
10.4. Класс *Thread* и интерфейсы *Runnable* и *Callable*. Создание и запуск потока выполнения 392
10.5. Поля и методы, заданные в классе *Thread* 394
10.6. Работа многопоточных приложений в многопроцессорных и многоядерных системах 396
Причины перехода к многопроцессорным и многоядерным системам 396
Пример многопоточной программы 398
Работа многопоточного приложения на многоядерном компьютере 406
10.7. Синхронизация на основе интерфейсов *Lock* и *Condition* 408
Интерфейс *Lock* 408
Интерфейс *Condition* 412
Краткие итоги по главе 414
Типичные ошибки 415
Задания 416

Глава 11. Введение в сетевое программирование 417
11.1. Краткая справка по языку HTML 417
Система WWW и язык HTML 417
Теги и их атрибуты 419
Математические и специальные символы, греческие буквы 425
11.2. Апплеты 427
Структура апплета 427
Примеры апплетов. Аннотация *@Override* 429
Создание проекта с апплетами 434
Редактор HTML-документов, стилей CSS и кода JavaScript 437
11.3. Сервлеты 442
Создание сервлета и конфигурирование сервера приложений 444
Локализация сервлетов — работа с языковыми форматами 451
11.4. Технология JSP — Java Server Pages 456
11.5. Технология JSTL — JSP Standard Tag Library 460
Краткие итоги по главе 462
Типичные ошибки 463
Задания 464

Глава 12. Вложенные классы ... 465
12.1. Виды вложенных классов ... 465
12.2. Статические (static) вложенные классы и интерфейсы ... 466
12.3. Внутренние (inner) классы ... 468
12.4. Локальные (local) классы ... 471
12.5. Анонимные (anonimous) классы и обработчики событий ... 471
12.6. Анонимные (anonimous) классы и слушатели событий (listeners) ... 472
Краткие итоги по главе ... 475
Типичные ошибки ... 477
Задания ... 477

Глава 13. Приложение с графическим интерфейсом — *DesktopApplication* ... 478
13.1. Структура простой заготовки *DesktopApplication* ... 478
13.2. Исходный код класса *DesktopApplication* ... 480
13.3. Исходный код класса *DesktopApplicationView* ... 482
13.4. Конструктор главной формы приложения. Ресурсы приложения и концепция управляющих синглетонов ... 486
13.5. Аннотация *@Action* и показ справки ... 491
13.6. Концепция управляющего объекта-модели. Модель *TaskMonitor*, класс *Task* и обработчик *propertyChange* ... 493
13.7. Создание задачи типа *Task*. Назначение иконок и добавление изображений ... 500
13.8. Локализация приложения ... 505
13.9. Измерение времени и досрочное прекращение выполнения задания ... 510
13.10. Работа с межпрограммным буфером обмена — Clipboard ... 511
Копирование строк и изображений из буфера обмена в программу ... 511
Копирование строк из программы в буфер обмена. Класс *StringSelection* ... 514
Копирование изображений из программы в буфер обмена. Создание класса *ImageSelection* ... 515
13.11. Многопоточная система вычислений и индикации ... 519
Краткие итоги по главе ... 523
Типичные ошибки ... 524
Задания ... 525

Глава 14. Программирование мобильных телефонов ... 526
14.1. Спецификация Java Micro Edition. Конфигурации и профили. Мидлеты ... 526
14.2. Создание приложений для мобильных устройств ... 527
14.3. Дизайнер пользовательского интерфейса мидлета ... 533
14.4. Исходный код мидлета *Hello, World!* ... 538
14.5. Заготовка мидлета "Крестики-нолики" ... 547
14.6. Код бизнес-логики мидлета ... 556
14.7. Создание собственного компонента, являющегося наследником *TableItem*, и мидлета на его основе. Создание библиотеки ... 563
14.8. Запуск мидлета "Крестики-нолики" на реальном телефоне ... 571
14.9. Пример мидлета с переключением экрана и списком выбора ... 573
14.10. Примеры приложений для мобильных телефонов, поставляемые со средой NetBeans ... 574
Краткие итоги по главе ... 577
Типичные ошибки ... 578
Задания ... 578

Глава 15. Компонентное программирование 579
15.1. Компонентная архитектура JavaBeans 579
15.2. Создание компонента в NetBeans 6.5 580
15.3. Пример создания компонента в NetBeans 6.5 — панель с заголовком 584
15.4. Добавление в компонент новых свойств 588
15.5. Добавление в компонент новых событий 590
Краткие итоги по главе 593
Типичные ошибки 594
Задания 595

Глава 16. Платформо-зависимые методы и модуль C/C++ Pack 596
16.1. Использование динамических библиотек. Объявление платформо-зависимых (native) методов 596
16.2. Интерфейс JNI (Java Native Interface) взаимодействия Java с C/C++. Соответствие типов Java и C++ 597
16.3. Пакет C/C++ Pack. Подключение компиляторов C++, C и библиотек 603
16.4. Создание приложений C++ и C 612
16.5. Примеры приложений с native-методами 616
Первый этап. Создание исходного кода Java для native-методов 617
Второй этап. Генерация h-файла заголовка C++, содержащего декларацию методов 617
Третий этап. Написание на C++ реализации методов, заголовки которых заданы в сгенерированном файле jnipackage_JniMultiply.h 619
Четвертый этап. Компиляция кода C++ в динамическую библиотеку (DLL) 620
Пятый этап. Запуск приложения вместе с DLL в NetBeans 621
Шестой этап. Распространение приложения вместе с DLL 621
Краткие итоги по главе 625
Типичные ошибки 626
Задания 626

Глава 17. Отладка, тестирование и профилирование программ 627
17.1. Отладка приложений в среде NetBeans 627
17.2. Команды режима пошагового выполнения 631
17.3. Пример отладки неправильно работающего приложения 638
17.4. Принцип "презумпции виновности" и тестирование классов 642
17.5. Профилирование приложений 656
17.6. Профилирование отдельных методов 664
17.7. Настройка профилировщика "на лету". Точки профилирования 670
17.8. Нахождение проблем с использованием памяти — телеметрия 678
17.9. Профилирование использования памяти классами 681
Краткие итоги по главе 687
Типичные ошибки 688
Задания 688

Заключение 689

Приложение. Описание прилагаемого DVD 691

Литература 698
Основная 698
Дополнительная 698

Предметный указатель 699

Предисловие к третьему изданию

Год, прошедший со времени выхода второго издания, оказался богат на события в мире Java и NetBeans. Во-первых, состоялся выход большого количества обновлений дистрибутивов Java. Во-вторых, появились версии среды NetBeans 6.7, 6.8 и 6.9. В-третьих, вышел дистрибутив Open JDK (см. **http://www.openjdk.org/**), позволивший гарантировать дальнейшее существование платформы Java и основанных на ней технологий независимо от судьбы корпорации Sun Microsystems. И, кроме того, корпорация Oracle, один из мировых лидеров в области индустрии программного обеспечения, приобрела фирму Sun Microsystems.

Предыдущее издание заняло достойное место среди компьютерной литературы по программированию на языке Java для начинающих. Ведь другие учебники на русском языке, в которых изложение материала основано на использовании современных сред разработки, просто-напросто отсутствуют, а учиться современному программированию с помощью морально устаревших технологий — не самый лучший выбор. Третье издание является исправленной версией предыдущего, с коррекцией найденных опечаток и мелких ошибок, а также с новыми дистрибутивами Java и NetBeans.

К книге приложен DVD с материалами, соответствующими ее тексту (примерами, а также дистрибутивами Java JDK 6u11 и NetBeans 6.5), для того чтобы желающие могли проделать все ровно так, как описано в тексте глав. Тем не менее, на DVD имеются и более современные дистрибутивы: текущие на момент верстки третьего издания версии JDK 6u22 и NetBeans 6.9.1 в вариантах для Windows и Linux. Автор рекомендует в любом случае устанавливать более новую версию JDK 6u22. Примеры из книги работают как в NetBeans 6.5, так и в NetBeans 6.9.1, при этом NetBeans 6.5 функционирует быстрее и занимает меньше дискового пространства, зато NetBeans 6.9.1 имеет гораздо более развитую среду редактирования и отладки. При необходимости можно установить на компьютере обе версии среды, т. к. они не мешают друг другу (хотя, конечно, это займет дополнительное дисковое пространство).

Замечания по найденным ошибкам и опечаткам просьба присылать автору по электронной почте на адрес **v.v.monahov@mail.ru**. Автор выражает благодарность всем, кто внес вклад в исправление ошибок предыдущих изданий (см. **http://barsic.spbu.ru/www/comlan/java_vm.html**).

Доцент кафедры вычислительной физики СПбГУ,
кандидат физико-математических наук Вадим Валериевич Монахов

Введение

Предлагаемый учебный курс можно рассматривать как краткое введение в современное программирование на Java и обзор возможностей NetBeans — одной из самых популярных и быстро развивающихся сред разработки программного обеспечения для языков Java, C, C++, Fortran, PHP, JavaScript, Ruby, Groovy и многих других.

О среде NetBeans в русскоязычной литературе рассказывается только в данной книге автора, а в иностранной — имеется около десятка таких книг, причем лишь в двух-трех из них излагается курс программирования на Java с использованием NetBeans. Поэтому читателю, желающему научиться программированию на языке Java, данное учебное пособие предоставляет уникальную возможность осуществить это с помощью наиболее современных средств, заметно облегчающих работу программиста и делающих ее гораздо более эффективной по сравнению с методиками, описанными в других книгах.

Первое издание книги, вышедшее в марте 2008 года, было полностью распродано к декабрю того же года. Оно оказалось очень популярным как среди начинающих программистов, так и среди профессионалов, переходящих с C или C++ на Java. Во втором издании была изложена работа с NetBeans версии 6.5 (первое издание основывалось на версии 6.0), а также исправлены допущенные ранее ошибки и опечатки. Кроме того, был добавлен раздел, в котором NetBeans сравнивается с другими средами разработки Java-программ, заметно расширено описание работы со строками и дописаны разделы про современные средства Java для работы с датой и временем, а также про применение в NetBeans тестовых пакетов и библиотеки JUnit 4.

В настоящем издании сохранено изложение, основанное на использовании NetBeans 6.5, однако все сказанное практически без изменений относится и к NetBeans 6.9. А различие в версии JDK вообще несущественно, и вместо JDK 6u11 рекомендуется использовать JDK 6u22, т. к. этот комплект содержит обновления системы безопасности и исправления ошибок.

Несмотря на то, что языку Java уже посвящены тысячи книг и учебных курсов, все книги, написанные до выхода JDK 5 (прежнее обозначение JDK 1.5), не соответствуют современным представлениям об этом языке.

Данное учебное пособие содержит информацию об основных синтаксических конструкциях современной версии языка Java (JDK 6), типичных ошибках их использования, рекомендации по написанию на языке Java эффективно работающих программ.

Одной из привлекательных особенностей языка Java с самого начала была бесплатность распространения базовых средств разработки SDK (Software Development Kit) и исполняющей среды Java (виртуальной Java-машины). Однако компилятор, входящий

в состав SDK, работал в режиме командной строки, т. е. отставал идеологически, по крайней мере, лет на 20 от современных профессиональных компиляторов. В 2006 году корпорация Sun Microsystems пошла на беспрецедентный шаг — сделала бесплатными свои профессиональные средства разработки программного обеспечения. В настоящее время все они основаны на бесплатно распространяемой среде разработки NetBeans (платформе NetBeans), подключаемых к ней модулях (packs) и наборе библиотечных компонентов (Beans — зерен). Интересна игра слов: язык Java получил наименование по названию кофе, который любили программисты, так что Java Beans может быть расшифровано как "зерна кофе Java".

Среда NetBeans дала возможность разрабатывать программное обеспечение на языке Java с помощью самых современных методик. Она имеет развитые средства визуального проектирования пользовательского интерфейса, не уступающие Delphi, развитый редактор исходного кода программы с подсветкой ошибок непосредственно во время редактирования, а также множество других возможностей, о которых рассказывается в данной книге.

В NetBeans начиная с версии 6.5 не только существенно усовершенствован редактор исходного кода и отладчик, но и изменен сам подход к развитию среды. Если предыдущие версии ориентировались только на язык Java и в небольшой степени на C/C++, то в NetBeans 6.5 произошел переход к поддержке широкого спектра языков программирования. И если раньше корпорация Sun Microsystems, разработчик языка Java и значительной части среды NetBeans, основное внимание уделяла языку Java, то в последнее время акцент сменился — среда NetBeans и технология Java Scripting (исполнение виртуальной Java-машиной программ, написанных на отличных от Java языках программирования) оказались удобными инструментами для поддержки огромного количества языков программирования и сопутствующих им технологий. При этом связующим звеном с NetBeans и Java-машиной остается язык Java.

В состав дистрибутивов NetBeans 6.5 и 6.9 (имеются на прилагаемом к книге DVD) входят все основные модули, необходимые для программирования на Java. Кроме того, в них входит модуль "C/C++ Pack", позволяющий программировать в среде NetBeans не только на Java, но и на языках C и C++, благодаря подключению к NetBeans внешних компиляторов C/C++. Это дает возможность разрабатывать полный спектр Java-программ в одной и той же среде. Раньше для создания платформо-зависимого (native) кода, используемого native-методами классов Java, требовались сторонние среды разработки программ, написанных на языках C/C++.

Среду разработки NetBeans можно установить с прилагаемого к книге DVD или свободно загрузить с сайта **http://netbeans.org/**. Она распространяется на условиях лицензии, дающей право использовать разработанное с помощью среды NetBeans программное обеспечение не только в некоммерческих, но и в коммерческих целях. Таким образом, теперь как начинающим разработчикам, так и крупным фирмам нет необходимости тратить деньги на покупку профессиональных средств разработки — можно работать в среде NetBeans.

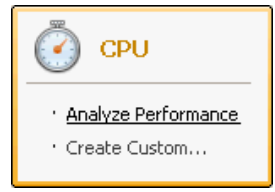
Почему же это не бесплатный сыр? Среда NetBeans разрабатывается сообществом разработки свободного программного обеспечения. Значительный вклад в разработку внесла фирма Sun Microsystems, которая получала деньги не у частных лиц или маленьких компаний, а у крупных корпоративных пользователей за техническую поддержку своих бесплатно распространяемых приложений. И лидерство в данной области косвенно обеспечивало гораздо бóльшие прибыли, чем продажа этих продуктов.

Java очень быстро развивается, особенно в последние годы. Из-за этого многие прежние представления и методы программирования на Java оказываются устаревшими. Среда NetBeans развивается еще быстрее — версия 6.5, появившаяся в 2008 году, заметно превосходит по возможностям версию 6.0, вышедшую в 2007 году. Данная книга — первый шаг к изучению современных технологий на основе Java и NetBeans.

Изложение ведется на основе языка Java для JDK версии 6 и среды NetBeans версии 6.5. Примеры, приведенные в данной книге, проверены в среде NetBeans версий 6.5 и 6.9 с JDK 6u11 и JDK 6u22.

Поскольку технология Java мультиплатформенна, то все изложенное справедливо как для MS Windows, так и для Solaris, Linux, Mac OS и других операционных систем.

Автор выражает благодарность рецензентам фирмы Sun Microsystems Алексею Акопяну и Алексею Уткину за полезные замечания по первоначальному варианту данного учебного пособия, Игорю Богдановичу Керницкому за содействие в подготовке примеров с native-кодом, а также жене Светлане за помощь в оформлении книги.



ГЛАВА 1

Общие представления о языке Java

1.1. Java и другие языки программирования. Системное и прикладное программирование

Язык программирования Java был создан в рамках проекта корпорации Sun Microsystems по разработке компьютерных программно-аппаратных комплексов нового поколения. Первая версия языка была официально опубликована в 1995 году. С тех пор язык Java стал стандартом "де-факто", вытеснив за десять лет языки C и C++ из многих областей программирования. В 1995 году они были абсолютными лидерами, но к 2006 году число программирующих на Java стало заметно превышать число программистов, использующих C и C++, и сейчас достигло более четырех с половиной миллионов человек. А число устройств, запрограммированных с помощью Java, превысило полтора миллиарда.

Как связаны между собой языки C, C++, JavaScript и Java? Что между ними общего и в чем они отличаются? В каких случаях следует, а в каких не следует их применять? Для того чтобы ответить на эти вопросы, нужно сначала остановиться на особенностях программного обеспечения предыдущих поколений и на современных тенденциях в развитии программного обеспечения.

Первоначально компьютеры программировались в машинных кодах. Затем появились языки ассемблера, которые заменили команды процессоров мнемоническими сокращениями, гораздо более удобными для человека, чем последовательности нулей и единиц. Их принято считать языками программирования *низкого уровня* (т. е. близкими к аппаратному уровню), т. к. они ориентированы на особенности конкретных процессоров. Программы, написанные на языках ассемблера, нельзя было переносить на компьютеры с другим типом процессора из-за несовместимости наборов команд (т. е. несовместимости на уровне исходного кода — source code).

Программы в *машинных кодах* (т. е. в виде последовательности нулей и единиц), соответствующих командам процессора и необходимым для них данным, нет надобности как-то преобразовывать. Их можно просто скопировать в нужное место памяти компьютера и передать управление первой команде программы (задать точку входа в программу).

Программы, написанные на каком-либо языке программирования, сначала нужно перевести из одной формы (текстовой) в другую (двоичную, т. е. в машинные коды). Про-

цесс такого перевода называется *трансляцией* (от англ. *translation* — "перевод", "перемещение"). Не обязательно переводить программу из текстовой формы в двоичные коды, возможен процесс трансляции с одного языка программирования на другой (или из кодов одного типа процессора в коды другого типа).

Имеются два основных вида трансляции — *компиляция* и *интерпретация*.

При *компиляции* первоначальный набор инструкций однократно переводится в исполняемую форму (машинные коды), а в последующем при работе программы используются только эти коды.

При *интерпретации* во время вызова необходимых инструкций каждый раз сначала происходит перевод инструкций из одной формы (текстовой или двоичной) в другую — в исполняемые коды процессора используемого компьютера. И только потом эти коды исполняются. Естественно, что интерпретируемые коды исполняются медленнее, чем скомпилированные, т. к. перевод инструкций из одной формы в другую обычно занимает в несколько раз больше времени, чем выполнение полученных инструкций. Но интерпретация обеспечивает бóльшую гибкость по сравнению с компиляцией, и в ряде случаев без нее не обойтись.

В 1956 году появился язык Fortran — первый язык программирования *высокого уровня* (т. е. не ориентированный на конкретную аппаратную реализацию компьютера). Он обеспечил переносимость программ на уровне исходных кодов, но довольно дорогой ценой. Во-первых, быстродействие программ, написанных на этом языке, было в несколько раз меньше, чем для ассемблерных. Во-вторых, такие программы занимали примерно в два раза больше места в памяти компьютера, чем ассемблерные. И, наконец, пришлось отказаться от поддержки особенностей периферийных устройств — ограничить общение с "внешним миром" простейшими возможностями, которые в программе одинаково реализовывались для ввода/вывода с помощью перфокарточного считывателя, клавиатуры, принтера, текстового дисплея и т. д. Тем не менее, языки программирования высокого уровня постепенно вытеснили ассемблер, поскольку обеспечивали не только переносимость программ, но и гораздо бóльшую их надежность, а также несоизмеримо более высокую скорость разработки сложного программного обеспечения. А Fortran до сих пор остается важнейшим языком программирования для высокопроизводительных численных научных расчетов.

Увеличение аппаратных возможностей компьютеров (рост объема оперативной и дисковой памяти, значительное повышение быстродействия), а также появление разнообразных периферийных устройств привело к необходимости пересмотра того, как должны работать программы. Массовый выпуск компьютеров потребовал унификации доступа из программ к различным устройствам, обеспечивая игнорирование особенностей их аппаратной реализации. Это можно осуществить, если обращение к устройству идет не напрямую, а через прилагающуюся программу — драйвер устройства (по-английски *driver* означает "водитель").

Появились операционные системы — наборы драйверов и специальных программ, распределяющих ресурсы компьютера между разными программами. Соответственно, программное обеспечение стало разделяться на *системное* и *прикладное*. Системное программное обеспечение — непосредственно обращающееся к аппаратуре, прикладное — решающее какие-либо прикладные задачи и использующее аппаратные возможности компьютера не напрямую, а через вызовы программ операционной системы.

Прикладные программы стали приложениями операционной системы (сокращенно *приложениями* — applications). Этот термин означает, что программа может работать только под управлением операционной системы. Если на том же компьютере установить другой тип операционной системы, то программа-приложение первой операционной системы, возможно, не будет работать.

Требования к прикладным и системным программам принципиально различаются. От системного программного обеспечения требуется максимальное быстродействие и минимальное количество занимаемых ресурсов, а также возможность доступа к любым необходимым аппаратным ресурсам. От прикладного — максимальная функциональность в конкретной предметной области. При этом быстродействие и занимаемые ресурсы не имеют значения до тех пор, пока не влияют на функциональность. Например, нет никакой разницы, реагирует ли программа на нажатие клавиши за одну десятую или за одну миллионную долю секунды. Правда, на первоначальном этапе создания прикладного программного обеспечения даже прикладные по назначению программы являлись системными по реализации, т. к. были вынуждены напрямую обращаться к аппаратуре.

Язык C был создан в 1972 году в одной из исследовательских групп Bell Laboratories при разработке операционной системы UNIX. Сначала была предпринята попытка написать операционную систему на ассемблере, но после появления в группе новых компьютеров возникла необходимость в новом платформо-независимом языке программирования высокого уровня, с помощью которого можно было бы писать операционные системы. Таким образом, язык C предназначался для создания системного программного обеспечения, и таким он остается до сих пор. Его идеология и синтаксические конструкции ориентированы на максимальную близость к аппаратному уровню реализации операций — в той степени, в какой он может быть обеспечен на аппаратно-независимом уровне. При этом главным требованием была максимальная скорость работы и минимальное количество занимаемых ресурсов, а также возможность доступа ко всем аппаратным средствам. Язык C — это язык *процедурного программирования*, т. к. его базовыми конструкциями являются *подпрограммы*. В общем случае подпрограммы принято называть подпрограммами-процедурами (откуда и идет название "процедурное программирование") и подпрограммами-функциями. Но в C имеются только подпрограммы-функции. Обычно их называют просто *функциями*.

Язык C произвел настоящую революцию в разработке программного обеспечения, получил широкое распространение и стал промышленным стандартом. Он до сих пор применяется для написания операционных систем и программирования микроконтроллеров. Но мало кто в полной мере осознает причины его популярности. В чем они заключались? В том, что он смог обеспечить необходимую функциональность программного обеспечения в условиях низкой производительности компьютеров, крайней ограниченности их ресурсов и неразвитости периферийных устройств! При этом повторилась та же история, что и с языком Fortran, но теперь уже для языка системного программирования. Переход на язык программирования высокого уровня, но с минимальными потерями по производительности и ресурсам, дал большие преимущества.

Большое влияние на развитие теории программирования оказал язык Pascal, разработанный в 1974 году швейцарским профессором Никлаусом Виртом. В этом проекте сочетались два оригинальных замысла. Первый — собственно язык Pascal, предназначен-

ный для обучения идеям *структурного программирования*. Второй — идея *виртуальной машины*. Никлаус Вирт предложил обеспечивать переносимость программ, написанных на Pascal, за счет компиляции их в набор команд некой абстрактной P-машины (P — сокращение от Pascal), а не в исполняемый код конкретной аппаратной платформы. А на каждой аппаратной платформе должна была работать программа, интерпретирующая эти коды. Говорят, что такая программа эмулирует (т. е. имитирует) систему команд несуществующего процессора. А саму программу называют виртуальной машиной.

В связи с ограниченностью ресурсов компьютеров и отсутствием в Pascal средств системного программирования этот язык не смог составить конкуренцию языку C, т. к. практически все промышленное программирование вплоть до середины последнего десятилетия XX века по реализации было системным. Идеи P-машины были в дальнейшем использованы и значительно усовершенствованы в Java.

Развитие теории и практики программирования привело к становлению в 1967—1972 годах нового направления — *объектного программирования*, основанного на концепциях работы с *классами* и *объектами*. Оно обеспечило принципиально новые возможности по сравнению с процедурным. Были предприняты попытки расширения различных языков путем введения в них конструкций объектного программирования.

В 1982 году Бьерн Страуструп путем такого расширения языка C создал язык, который он назвал "C с классами". В 1983 году после очередных усовершенствований им был создан первый компилятор языка C++. Два плюса означают: "язык C с очень большим количеством добавлений" и одновременно — "новое, увеличенное на один по сравнению с прежним, значение" (игра понятий: в C существует оператор "++").

Язык C++ является надмножеством над языком C — на нем можно писать программы как на "чистом C", без использования каких-либо конструкций объектного программирования. В связи с этим, а также и с дополнительными преимуществами объектного программирования, язык C++ быстро приобрел популярность и стал промышленным стандартом (сначала "де-факто", а потом и "де-юре"). Так что в настоящее время C++ признан базовым языком системного программирования. Длительное время он использовался и для написания прикладных программ. Но, как мы уже знаем, требования к прикладным программам совпадают с требованиями к системным только в том случае, когда быстродействие компьютера можно рассматривать как низкое, а ресурсы компьютера — малыми. Кроме этого, у языков C и C++ имеются еще два принципиальных недостатка: а) низкая надежность на уровне как исходного, так и исполняемого кода; б) отсутствие переносимости на уровне исполняемого кода. С появлением компьютерных сетей эти недостатки стали очень существенным ограничивающим фактором, поскольку вопросы безопасности при работе в локальных, а особенно в глобальных сетях приобретают первостепенную значимость.

В 1995 году появились сразу два языка программирования, имеющие в настоящее время огромное значение: Java, разработанный в корпорации Sun, и JavaScript, созданный в небольшой фирме Netscape Communication, получившей к тому времени известность благодаря браузеру Netscape Navigator.

Язык Java создавался как универсальный, предназначенный для прикладного программирования в неоднородных компьютерных сетях как со стороны клиентского компью-

тера, так и со стороны сервера. В том числе для использования на *тонких аппаратных клиентах* (устройствах малой вычислительной мощности с крайне ограниченными ресурсами). Скомпилированные программы Java работают только под управлением виртуальной Java-машины, поэтому их называют *приложениями Java*. Синтаксис операторов Java практически полностью совпадает с синтаксисом языка C, но в отличие от C++, Java это не расширение C, а совершенно независимый язык, со своими собственными синтаксическими правилами. Он гораздо сильнее типизирован по сравнению с C и C++, т. е. вносит больше ограничений на действия с переменными и величинами разных типов. Например, в C/C++ нет разницы между целочисленными числовыми, булевыми и символьными величинами, а также адресами в памяти. Так, можно умножить символ на булево значение `true` (истина), из которого вычтено целое число, и разделить результат на адрес! В Java введен вполне разумный запрет на почти все действия такого рода.

JavaScript создавался как узкоспециализированный прикладной язык программирования HTML-страниц, расширяющий возможности HTML, и он в полной мере отвечает этим потребностям до сих пор. Про язык HTML, являющийся основой для создания WWW-документов, более подробно будет рассказано в *главе 11*.

Следует подчеркнуть, что язык JavaScript *не имеет никакого отношения* к Java. Слово "script" означает сценарий. Включение слова "Java" в название JavaScript — рекламный трюк фирмы Netscape Communication. Язык JavaScript также C-образен, но в отличие от Java, интерпретируемый. Основное его назначение — программное управление элементами WWW-документов. Языки HTML и XML позволяют задавать статический, неизменный внешний вид документов, с их помощью невозможно запрограммировать реакцию на действия пользователя. JavaScript позволяет ввести элементы программирования в поведение документа. Программы, написанные на JavaScript, встраиваются в документы в виде исходных кодов (сценариев) и имеют небольшой размер. Для упрощения работы с динамически формируемыми документами JavaScript имеет свободную типизацию: переменные меняют тип по результату присваивания. Поэтому программы JavaScript гораздо менее надежны, чем C/C++, не говоря уже про Java.

Java, JavaScript и C++ относятся к объектно-ориентированным языкам программирования, все они имеют C-образный синтаксис операторов. Но как объектные модели, так и базовые конструкции этих языков (за исключением синтаксиса операторов) принципиально различны. Ни один из них не является версией или упрощением другого — это совсем разные языки, предназначенные для различных целей.

Итак:

- Java — универсальный язык прикладного программирования;
- JavaScript — узкоспециализированный язык программирования HTML-документов;
- C++ — универсальный язык системного программирования.

В 2000 году в корпорации Microsoft была разработана платформа .NET (читается "дотнет", DotNet — в переводе с английского "точка NET"). Она стала альтернативой платформе Java и во многом повторила ее идеи. Основное различие заключалось в том, что для этой платформы можно было использовать произвольное количество языков программирования, а не один. Причем классы .NET оказываются совместимыми как в

целях наследования, так и по исполняемому коду независимо от языка, на котором они написаны. Важнейшим языком .NET стал Java-образный язык C# (читается "Си шарп"). Фактически, C# унаследовал от Java большинство особенностей — динамическую объектную модель, сборку "мусора", основные синтаксические конструкции, хотя он вполне самостоятельный язык программирования, имеющий много привлекательных черт. В частности, компонентные модели Java и C# принципиально отличаются.

Java стал первым универсальным C-образным языком прикладного программирования, что обеспечило легкость перехода на этот язык большого числа программистов, знакомых с C и C++. А наличие средств строгой проверки типов, ориентация на работу с компьютерными сетями, переносимость на уровне исполняемого кода и поддержка платформо-независимого графического интерфейса, а также запрет прямого обращения к аппаратуре обеспечили выполнение большинства требований, предъявлявшихся к языку прикладного программирования. Чем больше становятся быстродействие и объем памяти компьютеров, тем больше потребность в разделении прикладного и системного программного обеспечения. Соответственно, для прикладных программ исчезает необходимость напрямую обращаться к памяти и другим аппаратным устройствам компьютера. Поэтому среди прикладных программ с каждым годом растет доля программного обеспечения, написанного на Java и языках .NET. Но как по числу программистов, так и по числу устройств, использующих соответствующие платформы, Java в настоящее время лидирует с большим отрывом.

1.2. Виртуальная Java-машина, байт-код, JIT-компиляция. Категории программ, написанных на языке Java

Первоначально слово "программа" означало последовательность инструкций процессора для решения какой-либо задачи. Эти инструкции представляли собой машинные коды, и разницы между исходным и исполняемым кодом программы не было. Разница появилась, когда программы стали писать на языках программирования. При этом программой стали называть как текст, содержащийся в файле с исходным кодом, так и исполняемый файл.

Для устранения неоднозначности термина "программа" исполняемый код программы принято называть приложением (application). Термин "приложение" — сокращение от фразы "приложение операционной системы". Он означает, что исполняемый код программы может работать только под управлением соответствующей операционной системы. Работа под управлением операционной системы позволяет избежать зависимости программы от устройства конкретного варианта аппаратуры на компьютере, где она должна выполняться. Например, как автору программы, так и пользователю совершенно безразлична аппаратная реализация устройства, с которого считывается информация: будет ли это жесткий диск с одной, двумя или шестнадцатью считывающими головками, или это будет CD-привод, DVD-привод, или еще какой-либо другой тип носителя. Но переносимость обычных приложений ограничивается одним типом операционных систем. Например, приложение MS Windows не будет работать под Linux, и наоборот. Существует возможность исключить зависимость приложения от

операционной системы, введя выполнение его команд не средствами операционной системы, а *исполняющей средой* — программой, контролирующей работу приложения и выполняющей его вызовы. Программы, написанные на языке Java, выполняются исполняющей средой — виртуальной Java-машиной, и поэтому обладают переносимостью на любую операционную систему, где имеется соответствующая Java-машина. Благодаря этому они являются не приложениями какой-либо операционной системы, а *приложениями Java*.

Программы, написанные на языке Java, представляют собой наборы классов (о классах будет рассказано несколько позже) и сохраняются в текстовых файлах с расширением java. При компиляции текст программы переводится (транслируется) в двоичные файлы с расширением class. Такие файлы содержат байт-код, представляющий собой совокупность инструкций для абстрактного Java-процессора в виде байтовых последовательностей команд этого процессора и данных к ним. Для того чтобы выполнить байт-код на каком-либо компьютере, его нужно перевести в инструкции для соответствующего процессора. Именно этим и занимается Java-машина. Первоначально байт-код всегда интерпретировался следующим образом: каждый раз, когда встречалась какая-либо инструкция Java-процессора, она переводилась в последовательность инструкций процессора компьютера, что, естественно, значительно замедляло работу приложений Java.

В настоящее время используется более сложная схема, называемая JIT-компиляцией (Just-In-Time), — это компиляция "по ходу дела", "на лету". Когда какая-либо инструкция (или набор инструкций) Java-процессора выполняется в первый раз, происходит компиляция соответствующего ей байт-кода с сохранением скомпилированного кода в специальном буфере. При последующем вызове той же инструкции вместо ее интерпретации скомпилированный код извлекается из буфера. Поэтому интерпретация происходит только при первом вызове инструкции.

Сложные оптимизирующие JIT-компиляторы действуют еще изощренней. Поскольку обычно компиляция инструкции идет гораздо дольше по сравнению с интерпретацией, время ее выполнения в первый раз при наличии JIT-компиляции может заметно отличаться в худшую сторону по сравнению с "чистой" интерпретацией. Поэтому бывает выгоднее сначала запустить процесс интерпретации, а параллельно ему в фоновом режиме компилировать инструкцию. Только после окончания процесса компиляции при последующих вызовах инструкции будет исполняться ее скомпилированный код (а до этого все ее вызовы будут интерпретироваться). Разработанная Sun виртуальная машина HotSpot осуществляет JIT-компиляцию только тех участков байт-кода, которые критичны ко времени выполнения программы. При этом по ходу работы программы происходит оптимизация скомпилированного кода.

Благодаря компиляции программ Java в платформо-независимый байт-код обеспечивается переносимость этих программ не только на уровне исходного кода, но и на уровне скомпилированных приложений. Конечно, при этом на компьютере, где выполняется приложение, должна быть установлена программа виртуальной Java-машины (Java Virtual Machine — JVM), скомпилированная в коды соответствующего процессора (native code — платформо-зависимый код). На одном и том же компьютере может быть несколько Java-машин разных версий или от разных производителей. Спецификация Java-машины и требования к компилятору языка Java открытые, поэтому различные фирмы (а не только Sun) разрабатывают компиляторы Java и Java-машины.

Обычно приложение операционной системы запускается с помощью средств самой операционной системы. Приложение Java, напротив, исполняется с помощью виртуальной Java-машины, которая сама является приложением операционной системы. Таким образом, сначала стартует Java-машина. Она получает в качестве параметра имя файла с компилированным кодом класса. В этом классе ищется и запускается на выполнение подпрограмма с именем `main`.

Приложения Java обладают не только хорошей переносимостью, но и высокой скоростью работы. Однако даже при наличии JIT-компиляции они все-таки выполняются медленнее, чем программы, написанные на C или C++. Это связано с тем, что JIT-компиляция создает не настолько оптимальный код, как многопроходный компилятор C/C++ (он имеет возможность тратить много времени и ресурсов на поиск конструкций программы, которые можно оптимизировать). А JIT-компиляция происходит "на лету", в условиях жесткой ограниченности времени и ресурсов. Для решения этой проблемы были разработаны компиляторы программ Java в код конкретных программно-аппаратных платформ (native code — платформо-зависимый код). Например, свободно распространяемый фондом GNU компилятор GCJ (**http://gcc.gnu.org/java/**). Правда, заметные успехи Sun в усовершенствовании Java-машины позволили практически достичь, а в ряде случаев даже обогнать по быстродействию программы, написанные на других языках, и отказаться от статической компиляции в платформо-зависимый код. В частности, приложения Java, активно занимающиеся выделением/высвобождением памяти, работают быстрее своих аналогов, написанных на C/C++, благодаря специальному механизму программных слотов памяти (slot — паз, отверстие для вставки чего-либо).

Виртуальная Java-машина не только исполняет байт-код (интерпретирует его, занимается JIT-компиляцией и исполняет JIT-компилированный код), но и выполняет ряд других функций (например, взаимодействует с операционной системой, обеспечивая доступ к файлам или поддержку графики), а также реализует автоматическое высвобождение памяти, занятой ненужными объектами, — так называемую сборку мусора (garbage collection).

Программы Java можно разделить на несколько основных категорий.

- Приложение (application) — аналог обычной прикладной программы.
- Апплет (applet) — специализированная программа с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера.
- Мидлет (midlet) — специализированная программа с ограниченными возможностями, работающая на мобильном устройстве. В настоящее время мидлеты — один из наиболее популярных сегментов рынка программного обеспечения, пишущегося на языке Java.
- Сервлет (servlet) — специализированная программа с ограниченными возможностями, работающая в WWW на стороне сервера. Используется преимущественно в рамках технологии JSP (Java Server Pages — серверных страниц Java) для программирования WWW-документов со стороны сервера и обработки отсылаемой на сервер информации.
- Серверное приложение (Enterprise application) — предназначено для многократного выполнения на стороне сервера.

- Библиотека (Java Class Library — библиотека классов, либо NetBeans Module — модуль платформы NetBeans) — предназначена для многократного использования программами Java.

Между приложениями и апплетами Java существует принципиальное различие: приложение запускается непосредственно с компьютера пользователя и имеет доступ ко всем ресурсам компьютера наравне с любыми другими программами. Апплет же загружается из WWW с постороннего сервера, причем из-за самой идеологии WWW-сайт, с которого загружен апплет, в общем случае не может быть признан надежным. А сам апплет может передавать данные на произвольный сервер в WWW. Поэтому, для того чтобы избежать риска утечки конфиденциальной информации с компьютера пользователя или совершения враждебных действий, у апплетов убраны многие возможности, имеющиеся у приложений.

Сервлеты — это приложения Java, запускаемые на стороне сервера. Они обладают возможностью доступа к файловой системе и другим ресурсам сервера через набор управляющих конструкций, предопределенных в рамках технологии JSP и пакета javax.servlet. Кроме того, они имеют доступ к содержимому HTTP-запросов со стороны клиентского компьютера, т. е. того компьютера, на котором работает пользователь. HTTP (HyperText Transport Protocol) — это протокол передачи WWW-документов через Интернет.

Технология JSP заключается в наличии дополнительных конструкций (тегов) в HTML- или XML-документах, которые позволяют серверу автоматически генерировать и компилировать сервлеты. В результате удается очень просто и удобно обрабатывать данные или элементы документа, а также внедрять в нужные места документа результаты обработки. Автоматическая генерация сервлетов происходит всего один раз — перед первой обработкой документа с тегами JSP на стороне сервера, поэтому код JSP выполняется достаточно быстро, но конечно требует, чтобы была установлена соответствующая Java-машина (например, входящая в состав сервера GlassFish — программного обеспечения, обеспечивающего поддержку большого количества необходимых серверных возможностей для работы в WWW). Отметим, что программный сервер GlassFish, основанный на кодах Sun Application Server, распространяется бесплатно и входит в состав NetBeans 6.5.

Более подробная информация о сетевом программировании и серверных приложениях Java будет изложена в *главе 11*.

Первоначально корпорация Sun Microsystems позиционировала Java как язык, обеспечивающий развитые графические возможности WWW-документов благодаря включению в них апплетов. Однако в настоящее время основные области применения Java — прикладное программирование на основе приложений, страниц JSP и сервлетов, а также других видов серверных программ. При этом апплеты играют незначительную роль.

Виртуальную Java-машину часто называют исполняющей средой (Java Runtime Environment — JRE).

Существуют два основных способа установки Java-машины на клиентский компьютер:

- JRE из поставки Java Development Kit (JDK) — комплект разработки программного обеспечения на языке Java. Как синоним может употребляться термин SDK — Software Development Kit (комплект разработки программного обеспечения);
- специализированный вариант JRE в составе интернет-браузера, называющийся Java plugin.

Комплект последних версий JDK можно свободно загружать с сайта Sun **http://java.sun.com/**.

При использовании апплетов требуется, чтобы в состав браузера входил специализированный комплект JRE. Как правило, он поставляется вместе с браузером и может при необходимости обновляться. Для MS Internet Explorer такой комплект и его обновления могут быть свободно загружены с сайта **http://www.java.com/**.

Имеется возможность установки Java-машины от различных производителей (не обязательно устанавливать комплект JDK от Sun). На одном и том же компьютере можно установить сразу несколько различных Java-машин (в том числе комплекты JDK разных версий). Правда, опыт показывает, что при этом некоторые программы, написанные на Java, теряют работоспособность (частично или полностью).

Комплекты JDK классифицируются исходя из версии Java (языка программирования и, соответственно, Java-машины) и типа создаваемых приложений. Так, ко времени написания данного текста выходили версии JDK 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6 и готовится к выходу 1.7. В каждой версии есть ряд подверсий, не сопровождающихся изменением языка программирования, а связанных в основном с исправлением ошибок или внесением небольших изменений в библиотеки (например, 1.4.1_01 или 1.5.0_04).

Версии Java 1.0 и 1.1 принято называть Java 1. Все версии Java, начиная с 1.2, называют Java 2. Однако более надежно классифицировать по номеру JDK, т. к. язык Java для версии JDK 1.5 заметно отличается по возможностям от языка Java для более ранних версий JDK — в него добавлено большое количество новых синтаксических конструкций, а также изменен ряд правил. Поэтому код, правильный в Java для версии JDK 1.5, может оказаться неправильным в Java для версии JDK 1.4 (не говоря уже про Java для версии JDK 1.3 или 1.2). Кроме того, начиная с JDK 1.5 компания Sun перестала использовать в названиях комплектов программного обеспечения термин "Java 2" и происходящие от него сокращения вида j2, а в нумерации комплектов JDK отказалась от префикса "1". Поэтому сейчас принято говорить о версиях JDK 5 или JDK 6, а не о JDK 1.5 или JDK 1.6. Но соответствующие папки при установке среды на компьютере все-таки называются jdk1.5 и jdk1.6.

Комплекты разработки JDK одной версии отличаются по типу создаваемых с их помощью приложений. По применяемым средствам разработки и условиям выполнения приложений Java различают три платформы:

- Микроиздание — Java Platform, Micro Edition (Java ME) **http://java.sun.com/javame/** для программирования "тонких аппаратных клиентов" (т. е. устройств, обладающих малыми ресурсами — наладонных компьютеров, сотовых телефонов, микроконтроллеров, смарт-карт). Старое название J2ME. Для разработки приложений применяется не JDK, а специальный набор инструментов (Sun Java Wireless Toolkit for CLDC, Sun Java Toolkit for CDC, NetBeans 6.5, инструментарий от других производителей). Приложения Java работают под управлением одной из двух имеющихся для этой платформы вариантов Java-машины: максимально облегченной килобайтной виртуальной машины KVM или несколько более мощной виртуальной машины CVM.
- Стандартное издание — Java Platform, Standard Edition (Java SE) **http://java.sun.com/javase/** для программирования "толстых клиентов" (т. е. устройств, обладающих достаточно большими ресурсами — обычных компьютеров и серверов).

Старое название J2SE. Основой средств разработки служит JDK. Приложения Java работают под управлением Java-машины JVM (Java Virtual Machine).

- Корпоративное издание — Java Platform, Enterprise Edition (Java EE) **http://java.sun.com/javaee/** для написания серверного программного обеспечения. Старое название J2EE. Является надстройкой над платформой Java SE, поэтому все приложения Java EE работают с использованием классов Java SE и под управлением JVM. В SDK данной платформы входит мощный программный сервер Sun Java System Application Server (сервер приложений Java), а также большое число других исполняющих сред, основанных на технологиях Java.

При распространении какого-либо продукта, написанного на Java для платформы Java SE, возможна установка только программного обеспечения Java-машины (JRE — Java Runtime Environment). Например, в случае Java 6 update 7 (обновление 7) комплект JDK назывался jdk1.6.0_07, и ему будет соответствовать комплект jre1.6.0_07. При этом создавалась папка с именем jre1.6.0_07 с вложенными папками bin и lib. Начиная с Java 6 update 10, среда jre устанавливается в папку jre6 без указания номера обновления — эта и более поздние версии допускают обновление JRE без скачивания и переустановки всего дистрибутива.

Фактически, Java 6 update 10 — новый этап в развитии Java, и его отличие от предыдущей версии Java 6 update 7 (обновления 8 и 9 отсутствовали) гораздо больше, чем различия между Java 5 и Java 6. Очередной большой этап — предстоящий переход к Java 7.

Состав jre6 достаточно прост. В папке bin содержатся файлы и папки, необходимые для работы Java-машины и дополнительных инструментов для работы с ней в специальных режимах. В папке lib находятся вспомогательные файлы и библиотеки, в основном связанные с параметрами настроек системы.

Также возможна установка целиком JDK. Например, при установке Java 1.6 update 11 создается папка jdk1.6.0_11 с вложенными папками:

- bin — содержатся файлы инструментов разработки;
- demo — файлы примеров с исходными кодами;
- include — заголовки файлов C для доступа к ряду библиотек Java и отладчику виртуальной Java-машины на платформо-зависимом уровне — на основе интерфейсов JNI (Java Native Interface) и JVMDI (Java Virtual Machine Debugging Interface) соответственно;
- jre — файлы, необходимые для работы с виртуальной Java-машиной;
- lib — ряд библиотек и сопроводительных файлов для работы инструментов из папки bin;
- sample — примеры с исходными кодами.

В папке jdk1.6.0_11 содержится также архив src.zip с исходными кодами стандартных классов Java.

Следует отметить, что комплекты JRE, входящие в состав JDK и поставляемые отдельно от JDK, заметно различаются. Это связано с тем, что в отдельно поставляемом комплекте не нужны средства, предназначенные для отладки приложений и специальных

отладочных режимов работы с Java-машиной (управления загрузкой, профилирования, проверки стека и т. д.).

К сожалению, в JDK отсутствует даже самая простейшая документация с описанием назначения имеющихся в нем инструментов, там даны лишь ссылки на сайт компании Sun, где можно найти эту информацию. В табл. 1.1 перечислено назначение основных инструментов.

Таблица 1.1. *Средства разработки приложений*

Утилита	Назначение
javac	Компилятор в режиме командной строки для программ, написанных на языке Java
java	Утилита для запуска в режиме командной строки откомпилированных программ-приложений
appletviewer	Утилита для запуска на исполнение и отладку апплетов без браузера. При этом не гарантируется работоспособность отлаженного апплета в браузере
jdb	Отладчик программ, написанных на языке Java
javadoc	Генератор документации по классам на основе комментариев, начинающихся с /**
jar	Создание и управление Java-архивами jar
javah	Генератор заголовочных файлов C/C++ для подключения к программам Java внешних библиотек C/C++ на основе интерфейса JNI
javap	Дизассемблер классов
extcheck	Утилита для обнаружения конфликтов между файлами архивов jar
native2ascii	Утилита для конвертации в режиме командной строки параметра, передаваемого в виде текста на национальном алфавите, в последовательность символов Unicode

Кроме того, имеются средства поддержки работы в WWW и корпоративных сетях (интранет) с интерфейсом RMI — интерфейсом удаленного вызова методов (Remote Methods Invocation). Это программы *rmic*, *rmiregistry*, *rmid*. Также присутствуют средства поддержки информационной безопасности *keytool*, *jarsigner*, *policytool* и ряд других категорий утилит.

Подчеркнем, что набор утилит JDK рассчитан на морально устаревший режим командной строки, и что в большинстве случаев гораздо удобнее и правильнее пользоваться современной профессиональной средой разработки NetBeans. Для ее работы из JDK необходим только комплект JRE.

1.3. Алфавит языка Java. Десятичные и шестнадцатеричные цифры и целые числа. Зарезервированные слова

Подробную информацию об алфавите языка Java и основных синтаксических конструкциях можно найти на сайте **http://java.sun.com/docs/books/jls/index.html** (на английском языке).

Алфавит языка Java

Алфавит языка Java состоит из букв, десятичных цифр, разделителей и специальных символов. Буквами считаются латинские буквы (кодируются в стандарте ASCII), буквы национальных алфавитов (кодируются в стандарте Unicode, кодировка UTF-16), а также соответствующие им символы, кодируемые управляющими последовательностями (о них будет рассказано чуть позже).

Буквы и цифры можно использовать в качестве идентификаторов (т. е. имен) переменных, методов и других элементов языка программирования. Правда, при наличии в идентификаторах букв национальных алфавитов в ряде случаев могут возникнуть проблемы — эти символы будут показываться в виде вопросительных знаков.

Непосредственно как буквы рассматривается только часть символов национальных алфавитов, остальные — специальные символы — используются в качестве операторов и разделителей языка Java и не могут входить в состав идентификаторов.

Латинские буквы ASCII:

- `ABCD...XYZ` — заглавные (прописные);
- `abcd...xyz` — строчные.

Дополнительные буквенные символы ASCII:

- `_` — знак подчеркивания;
- `$` — знак доллара.

Национальные буквы на примере русского алфавита:

- `АБВГ...ЭЮЯ` — заглавные (прописные);
- `абвг...эюя` — строчные.

Десятичные цифры:

```
0 1 2 3 4 5 6 7 8 9
```

Разделители (их девять, задаются как символы ASCII):

```
(       )       {       }       [       ]       ;       ,       .
```

Десятичные и шестнадцатеричные цифры и целые числа

Целые числовые константы в исходном коде Java (так называемые литерные константы) могут быть десятичными или шестнадцатеричными. Они записываются либо символами ASCII, либо символами Unicode следующим образом.

Десятичные константы записывают как обычно (например, `-137`).

Константы-значения, записанные в виде последовательности символов (литер), называют *литерными* (иногда — *литералами*). В Java литерные константы бывают не только числовыми, но и булевыми (значения `true` и `false`), символьными (например, `'a'`, `'b'`, `'1'`, `'2'`), строковыми (например, `"Это - строка"`). Также имеется особая литерная константа `null`.

Шестнадцатеричная литерная константа целого типа начинается с символов `0x` или `0X` (цифра `0`, после которой следует латинская буква `x`), затем идет само число в шестнадцатеричной нотации. Например:

`0x10` соответствует $10_{16} = 16$;

`0x2F` соответствует $2F_{16} = 47$ и т. д.

О шестнадцатеричной нотации будет рассказано чуть позже.

Ранее иногда применялись восьмеричные числа, в языках C/C++, а также и в Java их записывали в виде числа, начинающегося с цифры 0 (т. е. `010` означало $10_8 = 8$).

В настоящее время в программировании восьмеричные числа практически не применяют, а неадекватное использование ведущего нуля может приводить к логическим ошибкам в программе.

Целая литерная константа в обычной записи имеет тип `int`. Если после константы добавить букву `L` (или `l`, что хуже видно в тексте, хотя в среде разработки выделяется цветом), то она будет иметь тип `long`, обладающий более широким диапазоном значений, чем тип `int`.

Поясним теперь, что такое шестнадцатеричная нотация записи чисел и зачем она нужна.

Информация представляется в компьютере в двоичном виде — как последовательность битов. Бит — это минимальная порция информации, он может быть представлен в виде ячейки, в которой хранится ноль или единица. Но бит — слишком мелкая единица, поэтому в компьютерах информация хранится, кодируется и передается байтами — порциями по 8 битов.

В данной книге под "ячейкой памяти" будет пониматься непрерывная область памяти (с последовательно идущими адресами), выделенная программой для хранения данных. На рисунках мы будем изображать ячейку прямоугольником, внутри которого находятся хранящиеся в ней данные. Если у ячейки имеется имя, оно будет выведено рядом с этим прямоугольником.

Мы привыкли работать с числами, записанными в так называемой десятичной системе счисления. В этой системе имеются 10 цифр (от 0 до 9), а в числе имеются *десятичные разряды*. Каждый разряд слева имеет вес 10 по сравнению с предыдущим, т. е. для получения значения числа, соответствующего цифре в каком-то разряде, стоящую в нем цифру нужно умножать на 10 в соответствующей степени (т. е. $52 = 5 \cdot 10 + 2$; $137 = 1 \cdot 10^2 + 3 \cdot 10^1 + 7$, и т. п.).

В программировании десятичной системой счисления пользоваться не всегда удобно, т. к. в компьютерах информация организована в виде битов, байтов и более крупных порций. Человеку неудобно оперировать данными в виде длинных последовательностей нулей и единиц. В настоящее время в программировании стандартной является шестнадцатеричная система записи чисел. Например, с ее помощью естественным образом кодируется цвет, устанавливаются значения отдельных битов числа, осуществляется шифрование и дешифрование информации, и т. д. В этой системе счисления все очень похоже на десятичную, но только не 10, а 16 цифр, и вес разряда не 10, а 16. Первые десять цифр записывают обычными десятичными цифрами, а недостающие цифры (бóльшие 9) — заглавными латинскими буквами: `A`, `B`, `C`, `D`, `E`, `F`:

`0 1 2 3 4 5 6 7 8 9 A B C D E F`

То есть: `A = 10`, `B = 11`, `C = 12`, `D = 13`, `E = 14`, `F = 15`.

Заметим, что в шестнадцатеричной системе счисления числа от 0 до 9 записываются одинаково, а превышающие 9 отличаются. Для чисел от 10 до 15 в шестнадцатеричной системе счисления используют буквы от `A` до `F`, после чего задействуют следующий шестнадцатеричный разряд. Десятичное число 16 в шестнадцатеричной системе счисления записывается как 10. Для того чтобы не путать числа, записанные в разных системах счисления, около них справа пишут индекс с указанием *основания* системы счисления. Для десятичной системы счисления это 10, для шестнадцатеричной — 16. Для десятичной системы основание обычно не указывают (если это не приводит к путанице).

Точно так же в технической литературе часто не указывают основание для чисел, записанных в шестнадцатеричной системе счисления, если в записи числа встречаются не только обычные цифры от 0 до 9, но и буквенные цифры от `A` до `F`. Обычно употребляют заглавные буквы, но допустимы и строчные.

Примеры

$0x10 = 10_{16} = 16;$
$0x100 = 100_{16} = 16 \cdot 16 = 256;$
$0x1000 = 1000_{16} = (16)^3 = 4096;$
$0x20 = 20_{16} = 2 \cdot 16 = 32;$
$0x21 = 21_{16} = 2 \cdot 16 + 1 = 33;$
$0xF = F_{16} = 15;$
$0x1F = 1F_{16} = 1 \cdot 16 + 15 = 31;$
$0x2F = 2F_{16} = 2 \cdot 16 + 15 = 47;$
$0xFF = FF_{16} = 15 \cdot 16 + 15 = 255;$

Более подробно представление чисел в компьютере будет рассмотрено в *главе 4*.

Зарезервированные слова и литералы языка Java

В табл. 1.2 приведены ключевые слова (keywords), зарезервированные для синтаксических конструкций языка. Их назначение нельзя переопределять внутри программы, они не могут быть идентификаторами (именами переменных, подпрограмм и т. п.).

Таблица 1.2. *Зарезервированные слова языка Java*

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Кроме ключевых слов имеются предопределенные литералы `false`, `null` и `true`. Фактическое отличие предопределенных литералов от ключевых слов заключается только в том, что они представляют собой значения, а не элементы синтаксических конструкций. Их также нельзя переопределять внутри программы — использовать в качестве идентификаторов или изменять значение.

1.4. Управляющие последовательности. Символы Unicode. Специальные символы

Управляющие последовательности

Иногда в тексте программы в строковых константах требуются символы, которые обычным образом в текст ввести нельзя (например, кавычки, а также различные специальные символы). В этом случае прибегают к управляющей последовательности — символу обратной косой черты, после которой следует один управляющий символ.

Управляющие последовательности — символы формирования текста

В табл. 1.3 приведены управляющие последовательности, применяющиеся в языке Java для форматирования текста. Кроме очевидных символов форматирования текста, к ним относятся также кавычки (поскольку одни кавычки могут оказаться внутри других, что требует специальных соглашений в языке программирования).

Таблица 1.3. *Управляющие последовательности*

Символ	Что означает
`\b`	Возврат на один символ назад
`\f`	Перевод на новую страницу
`\n`	Перевод на новую строку
`\r`	Возврат к началу строки
`\t`	Горизонтальная табуляция
`\'`	Кавычка
`\"`	Двойные кавычки
`\\`	Обратная косая черта
`\u`	Начало кодировки символа Unicode

Управляющие последовательности Java, C и C++ в основном совпадают. В Java отсутствуют только последовательности, использование которых неактуально (`\a`, `\v`) или неудобно (`\?`).

Управляющие последовательности — символы Unicode

Управляющая последовательность может содержать несколько символов. Например, символы национальных алфавитов могут кодироваться последовательностью `\u`, после

которой идет код символа в шестнадцатеричной кодировке для кодовых таблиц UTF-16 или UTF-8, например:

`\u0030 – \u0039` — цифры ISO-LATIN от 0 до 9;

`\u0024` — знак доллара $;

`\u0041 – \u005a` — буквы от *A* до *Z*;

`\u0061 – \u007a` — буквы от *a* до *z*.

Простые специальные символы

В табл. 1.4 приведены простые специальные символы (т. е. нецифровые и небуквенные символы, имеющие в языке программирования специальное назначение).

Таблица 1.4. *Простые специальные символы*

Символ	Что означает
+	Оператор сложения
–	Оператор вычитания
*	Оператор умножения
/	Оператор деления
%	Оператор остатка от целочисленного деления
=	Оператор присваивания
~	Оператор побитового дополнения (побитовое НЕ)
?	Вопросительный знак — часть тернарного (состоящего из трех частей) условного оператора "? :"
:	Двоеточие — часть условного оператора "? :". Также используется для задания метки — ставится после имени метки
^	Оператор XOR (исключающее ИЛИ)
&	Оператор AND (побитовое И)
|	Оператор OR (побитовое ИЛИ)
!	Оператор NOT (НЕ)
>	Больше
<	Меньше
{	Левая фигурная скобка — разделитель, открывающий блок кода
}	Правая фигурная скобка — разделитель, закрывающий блок кода
,	Запятая — разделитель в списке параметров оператора; разделитель в составном операторе
.	Точка — десятичный разделитель в числовом литерном выражении; разделитель в составном имени для доступа к элементу пакета, класса, объекта, интерфейса
(	Левая круглая скобка — разделитель, открывающий список параметров в операторах и группируемую часть в выражениях

Таблица 1.4 *(окончание)*

Символ	Что означает
)	Правая круглая скобка — разделитель, закрывающий список параметров в операторах и группируемую часть в выражениях
[	Левая квадратная скобка — разделитель, открывающий индекс массива
]	Правая квадратная скобка — разделитель, закрывающий индекс массива
;	Точка с запятой — разделитель, обозначающий окончание оператора
'	Апостроф (одиночная кавычка) — открытие и закрытие символа
"	Двойные кавычки — открытие и закрытие строки символов
\	Обратная косая черта (backslash) — используется для задания управляющих последовательностей символов
	Знак пробела (невидимый)
	Знак табуляции (невидимый)
@	Коммерческое а ("эт") — знак начала аннотации
#	Не имеет специального назначения
¤	То же
«	То же
»	То же
№	То же
§	То же

Составные специальные символы

В табл. 1.5 приведены составные специальные символы — последовательности из двух, трех или четырех нецифровых и небуквенных символов, имеющие в языке программирования специальное назначение.

Таблица 1.5. *Составные специальные символы*

Символ	Что означает
++	Оператор инкремента (увеличения на 1); x++ эквивалентно `x = x + 1`
--	Оператор декремента (уменьшения на 1); x-- эквивалентно `x = x − 1`
&&	Оператор AND (логическое *И*)
||	Оператор OR (логическое *ИЛИ*)
<<	Оператор левого побитового сдвига
>>>	Оператор беззнакового правого побитового сдвига
>>	Оператор правого побитового сдвига с сохранением знака отрицательного числа

Таблица 1.5 *(окончание)*

Символ	Что означает
==	Равно
!=	Не равно
+=	`y += x` эквивалентно `y = y + x`
-=	`y -= x` эквивалентно `y = y - x`
*=	`y *= x` эквивалентно `y = y * x`
/=	`y /= x` эквивалентно `y = y / x`
%=	`y %= x` эквивалентно `y = y % x`
|=	`y |= x` эквивалентно `y = y | x`
^=	`y ^= x` эквивалентно `y = y ^ x`
>>=	`y >>= x` эквивалентно `y = y >> x`
>>>=	`y >>>= x` эквивалентно `y = y >>> x`
<<=	`y <<= x` эквивалентно `y = y << x`
/*	Начало многострочного комментария
/**	Начало многострочного комментария, предназначенного для автоматического создания документации по классу
*/	Конец многострочного комментария (открываемого как /* или /**)
//	Однострочный комментарий

1.5. Идентификаторы. Переменные и типы. Примитивные и ссылочные типы

Идентификаторы — это имена переменных, подпрограмм-функций и других элементов языка программирования. Идентификаторы могут состоять только из букв и цифр, причем первой всегда должна быть буква (в том числе символы подчеркивания и доллара), а далее может идти их произвольная комбинация. Некоторые символы национальных алфавитов рассматриваются как буквы, поэтому их можно применять в идентификаторах. Но некоторые служат в качестве символов-разделителей, и в идентификаторах недопустимы.

Язык Java *регистро-чувствительный*. Это значит, что идентификаторы чувствительны к тому, в каком регистре (верхнем или нижнем) набираются символы. Например, имена `i1` и `I1` соответствуют разным идентификаторам. Это правило привычно для тех, кто знаком с языками C/C++, но может на первых порах вызвать сложности у тех, кто изучал язык Pascal, который допускает произвольный регистр символов.

Длина идентификатора в Java любая (но в пределах разумного). Так, даже при длине идентификатора во всю ширину экрана компилятор NetBeans правильно работает.

Переменная — это именованная ячейка памяти, которой сопоставлен *тип* и содержимое которой может изменяться. Перед тем как задействовать какую-либо переменную, она должна быть задана в области программы, предшествующей тому месту, где эта переменная используется. При объявлении переменной сначала указывается ее тип, а затем идентификатор. Указание типа позволяет компилятору задавать размер ячейки (объем памяти, выделяемой под переменную, или значение данного типа), а также допустимые правила действий с переменными и значениями этого типа.

В Java существует несколько предопределенных типов:

- `int` — целое число;
- `float` — вещественное число;
- `boolean` — логическое значение;
- `Object` — самый простой объектный тип (класс) Java;
- и т. д.

Также имеется возможность задавать собственные объектные типы (классы), о чем будет рассказано позже.

Переменные `a1` и `b1`, имеющие некий тип `MyType1`, объявляют так:

```
MyType1 a1,b1;
```

При этом `MyType1` — имя типа этих переменных.

Другой пример — объявление переменной `j` типа `int`:

```
int j;
```

Типы бывают предопределенные и пользовательские.

Например, `int` — предопределенный тип, а `MyType1` — пользовательский.

Для объявления переменной не требуется никакого зарезервированного слова, а имя типа пишется перед именами задаваемых переменных.

Объявление переменных может сопровождаться их *инициализацией* — присваиванием начальных значений. Приведем пример такого объявления целочисленных переменных `i1` и `i2`:

```
int i1=5;
int i2=-78;
```

либо

```
int i1=5,i2=-78;
```

Для начинающих программистов отметим, что символ "=" используется в Java и многих других языках в качестве символа *присваивания*, а не символа равенства, как это принято в математике. Он означает, что значение, стоящее с правой стороны от этого символа, *копируется* в переменную, стоящую в левой части. Так, присваивание `b = a` означает, что в переменную (ячейку) с именем `b` следует скопировать значение из переменной (ячейки) с именем `a`. Поэтому неправильное с точки зрения математики выражение:

```
x = x + 1
```

в программировании вполне корректно. Оно означает, что нужно взять значение, хранящееся в ячейке с именем `x`, прибавить к нему `1` (это будет происходить где-то вне ячейки `x`), после чего получившийся результат записать в ячейку `x`, заменив им прежнее значение.

После объявления переменных их можно использовать в выражениях присваивания:

```
переменная = значение;
переменная = выражение;
переменная1 = переменная2;
```

и т. д.

Например:

```
i1 = i2 + 5 * i1;
```

Примитивными типами (primitive types) (или, что то же самое, *простыми типами*) называются такие, для которых данные содержатся в одной ячейке памяти, не имеющей вложенных ячеек.

Ссылочными типами называются такие, для которых в ячейке памяти *(ссылочной переменной)* содержатся не сами данные, а только адреса этих данных, т. е. *ссылки* на данные. При присваивании в ссылочную переменную заносится новый адрес, а не сами данные. Но непосредственного доступа к адресу, хранящемуся в ссылочных переменных, нет. Это сделано для обеспечения безопасности работы с данными — как с точки зрения устранения непреднамеренных ошибок, характерных для работы с данными по их адресам в языках C/C++/Pascal, так и для предотвращения намеренного взлома информации.

Если ссылочной переменной не присвоено ссылки, то в ней хранится нулевой адрес, которому дано символическое имя `null`. Ссылки можно присваивать друг другу, если они совместимы по типам, а также присваивать им значение `null`. При этом из одной ссылочной переменной в другую копируется адрес. Ссылочные переменные можно проверять на равенство, в том числе на равенство `null`. При этом сравниваются не данные, а их адреса, хранящиеся в ссылочных переменных.

В Java все типы делятся на примитивные и ссылочные. К примитивным типам относятся следующие предопределенные типы:

- целочисленные типы `byte`, `short`, `int`, `long`, `char`;
- типы данных в формате с плавающей точкой `float`, `double`;
- булевый (булевский, логический) тип `boolean`;

Все остальные типы Java ссылочные.

В Java действуют следующие соглашения о регистре букв в идентификаторах:

- имена примитивных типов следует писать в нижнем регистре (строчными буквами), например: `int`, `float`, `boolean` и т. д.;
- имена ссылочных типов следует начинать с заглавной (большой) буквы, а далее для имен, состоящих из одного слова, писать все остальные буквы в нижнем регистре (например, `Object`, `Float`, `Boolean`, `Collection`, `Runnable`). Но если имя составное, то новую часть имени начинают с заглавной буквы (например, `JButton`, `JTextField`, `JFormattedTextField`, `MyType` и т. д.). Обратите внимание, что типы `float` и `Float`, `boolean` и `Boolean` различны — язык Java чувствителен к регистру букв;

- для переменных и методов имена, состоящие из одного слова, следует писать в нижнем регистре (например, `i`, `j`, `object1`). Если имя составное, новую часть имени начинают с заглавной буквы: `myVariable`, `jButton2`, `jTextField2.getText()` и т. д.;
- имена пакетов *(см. разд. 1.8)* следует писать полностью в нижнем регистре даже в том случае, когда имя составное: `mypackage`, `desktopapplication1`, `org.jdesktop.swingworker` и т. д.;
- имена констант следует писать в верхнем регистре (большими буквами), разделяя входящие в имя составные части символом подчеркивания "_", например: `Double.MIN_VALUE`, `Double.MAX_VALUE`, `JOptionPane.INFORMATION_MESSAGE`, `MY_CHARS_COUNT` и т. п.;
- символ подчеркивания "_" рекомендуется для разделения составных частей имени только в именах констант и пакетов.

Перечисленные соглашения о написании идентификаторов заданы в качестве стандарта языка Java, но, в отличие от соглашений о синтаксисе языка, компиляторами не отслеживаются. Хотя следование им заметно повышает читаемость программного кода, а нарушение — свидетельствует о низкой квалификации программиста.

Переменная примитивного типа может быть отождествлена с ячейкой, в которой хранятся данные. У нее всегда есть имя. Присваивание переменной примитивного типа какого-либо выражения меняет значение данных. Для ссылочных переменных при присваиваниях действия производятся с адресами ячеек, в которых хранятся данные, а не с самими данными.

Для чего нужны такие усложнения? Ведь человеку гораздо естественнее работать с ячейками памяти, в которых хранятся сами данные, а не адреса этих данных. Ответ заключается в том, что в программах часто требуются динамически создаваемые и уничтожаемые данные. Для них нельзя заранее создать необходимое число переменных, т. к. это число неизвестно на этапе написания программы и зависит от внешних параметров и выбора пользователя. Такие данные приходится помещать в динамически создаваемые и уничтожаемые ячейки. А с этими ячейками удается работать только с помощью ссылочных переменных.

Ссылочные типы Java используются в объектном программировании (в частности, для работы со строками, файлами, элементами пользовательского интерфейса). Все *пользовательские типы* (задаваемые программистом) являются ссылочными (в том числе типы-перечисления).

1.6. Процедурное и объектно-ориентированное программирование. Инкапсуляция

Объектно-ориентированное программирование (ООП) — это методология программирования, опирающаяся на три базовых принципа:

- инкапсуляцию;
- наследование;
- полиморфизм.

Язык Java объектно-ориентированный и в полном объеме реализует эти принципы. В данном разделе рассматривается принцип инкапсуляции. Наследованию и полиморфизму посвящены отдельные разделы.

Построение программ, основанных на ООП, принципиально отличается от более ранней методики *процедурного программирования*, в которой основой служили *подпрограммы*.

Программа — это набор инструкций процессору и данных, объединенных в единую функционально законченную последовательность, позволяющую выполнять какую-нибудь конкретную деятельность.

Подпрограмма — это набор инструкций и данных, объединенных в относительно самостоятельную последовательность, позволяющую выполнять какую-нибудь конкретную деятельность внутри программы. При этом подпрограмма не может работать самостоятельно — она запускается из программы, может получать из нее данные или передавать их в программу.

Подпрограммы принято делить на подпрограммы-процедуры и подпрограммы-функции. Подпрограмму-процедуру вызывают для выполнения каких-либо действий (например, для распечатки текста на принтере). Подпрограмма-функция выполняет какие-либо действия и возвращает некоторое значение. Например, вычисляет синус и возвращает вычисленное значение или создает сложно устроенный объект и возвращает ссылку на него (адрес ячейки, в которой он находится).

Программы, написанные в соответствии с принципами процедурного программирования, состоят из набора подпрограмм, причем для решения конкретной задачи программист явно указывает на каждом шаге, *что* делать и *как* делать. Эти программы практически полностью (процентов на девяносто) состоят из решения конкретных задач.

Программы, написанные в соответствии с принципами ООП, пишут совершенно иначе. В них основное время занимает продумывание и описание того, как устроены классы. Код с описанием классов предназначен для многократного использования без внесения каких-либо изменений. При этом обеспечивается разграничение областей влияния частей программного кода друг на друга и упорядочение способов взаимодействия этих частей благодаря ограничению области видимости элементов программ. В результате только небольшая часть времени посвящается решению конкретной задачи — написанию на основе классов такого кода, который в других задачах подобного типа не применить.

Именно благодаря такому подходу объектное программирование приобрело огромную популярность — при необходимости решения сходных задач можно взять уже готовый код, модифицируя только ту часть программы, которая относится к решению конкретной задачи.

Подробно мы разберем принципы ООП позже. Пока же в общих чертах разъясним их суть.

Самый простой из указанных в начале раздела принципов — *инкапсуляция*. Это слово в общем случае означает "заключение внутрь капсулы" (т. е. ограничение доступа к внутреннему содержимому снаружи и отсутствие такого ограничения внутри капсулы). В объектном программировании инкапсуляция означает использование *классов* — таких *типов*, в которых кроме *данных* описаны подпрограммы, позволяющие работать

с этими данными, а также выполнять другие действия. Такие подпрограммы, инкапсулированные в класс, называют *методами*. Поля данных и методы, заданные в классе, часто называют *членами класса* (class members).

Класс — это описание того, как будет устроен *объект*, являющийся *экземпляром данного класса*, а также какие методы объект может вызывать. Заметим, что методы, в отличие от других подпрограмм, могут напрямую обращаться к данным своего объекта. Поскольку экземплярами классов ("воплощением" в реальность того, что описано в классе) служат объекты, то классы называют *объектными типами*.

Все объекты, являющиеся экземплярами некоторого класса, имеют одинаковые наборы полей данных (*атрибуты* объекта), но значения этих полей свои для каждого объекта. Поля данных — это переменные, заданные на уровне описания класса, а не при описании метода. В процессе жизни объекта эти значения могут изменяться. Значения полей данных объекта задают его *состояние*. А методы задают *поведение* объекта. Причем в общем случае на это поведение влияет состояние объекта — методы пользуются значениями его полей данных.

Классы в Java задают следующим образом. Сначала указывают зарезервированное слово `class`, затем имя класса, после чего в фигурных скобках записывают реализацию класса — задают его поля *(глобальные переменные)* и методы.

Объектные переменные — такие переменные, которые имеют объектный тип. В Java объектные переменные — это не сами объекты, а только *ссылки* на них (т. е. все объектные типы являются ссылочными).

Объектную переменную объявляют так же, как и другие переменные: сначала пишут тип, а затем через пробел имя объявляемой переменной.

Например, задать переменную `obj1` типа `Circle` ("окружность") можно так:

```
Circle obj1;
```

Объектную переменную связывают с объектом путем присваивания. Поскольку объектная переменная ссылочная, в правой части присваивания необходимо указать ссылочное значение: либо функцию, возвращающую ссылку на объект (адрес объекта), либо имя другой ссылочной переменной, либо ссылочную константу. Если ссылочной переменной не присвоено никакого значения, то в ней хранится значение по умолчанию — константа `null`. Объектные переменные можно проверять на равенство, в том числе на равенство `null`. При этом сравниваются не сами объекты, а их адреса, хранящиеся в объектных переменных.

Создается объект с помощью вызова специальной подпрограммы, задаваемой в классе и называемой *конструктором*. Конструктор возвращает ссылку на созданный объект. Имя конструктора в Java *всегда совпадает с именем класса*, экземпляр которого создается. Перед именем конструктора во время вызова ставится оператор `new` — "новый", означающий, что создается новый объект. Например, вызов

```
obj1 = new Circle();
```

означает, что создается новый объект типа `Circle` (окружность), и ссылка на него (адрес объекта) записывается в переменную `obj1`. Переменная `obj1` до этого уже должна быть объявлена. Оператор `new` отвечает за динамическое выделение памяти под создаваемый объект.

Часто совмещают задание объектной переменной и назначение ей объекта. В нашем случае оно будет выглядеть так:

```
Circle obj1 = new Circle();
```

У конструктора, как и у любой подпрограммы, может быть список параметров. Они нужны для того, чтобы задать начальное состояние объекта при его создании. Например, мы хотим, чтобы у создаваемой окружности можно было при вызове конструктора задать координаты `x`, у ее центра и радиус `r`. Тогда при написании класса `Circle` можно предусмотреть конструктор, в котором первым параметром задается координата `x`, вторым — `y`, третьим — радиус окружности `r`. Тогда задание переменной `obj1` может выглядеть так:

```
Circle obj1 = new Circle(130,120,50);
```

Эта запись означает, что создается объект-окружность с центром в точке с координатами `x = 130`, `y = 120` и радиусом `r = 50`.

Если разработчики класса не создали ни одного конструктора, то в реализации класса автоматически создается конструктор по умолчанию, имеющий пустой список параметров. И его можно вызывать в программе так, как мы это первоначально делали для класса `Circle`.

Отметим еще одно правило, касающееся используемых имен. Как мы помним, имена объектных типов принято писать с заглавной буквы, а имена объектных переменных — со строчной. Если объектная переменная имеет тип `Circle`, то она служит ссылкой на объекты-окружности. Поэтому имя `obj1` не очень удачно — мы выбрали его только для того, чтобы подчеркнуть, что именно с помощью этой переменной осуществляется связь с объектом, и чтобы читатель не путал тип переменной, ее имя и имя конструктора. В Java принято называть объектные переменные так же, как их типы, но начинать имя со строчной буквы. Поэтому предыдущий оператор мог бы выглядеть так:

```
Circle circle = new Circle(130,120,50);
```

Если требуется работа с несколькими объектными переменными одного типа, то их принято называть в соответствии с указанным выше правилом, но добавлять порядковый номер. Следующие строки программного кода создают два независимых объекта с одинаковыми начальными параметрами:

```
Circle circle1 = new Circle(130,120,50);
Circle circle2 = new Circle(130,120,50);
```

С помощью объектных переменных осуществляется доступ к полям данных или методам объекта: сначала указывают имя переменной, затем точку, после чего пишут имя поля данных или метода. Например, если имя объектной переменной `obj1`, а имя целочисленного поля данных `x`, то присваивание ему нового значения будет выглядеть так:

```
obj1.x = 5;
```

А если имя подпрограммы `show`, у нее нет параметров, и она не возвращает никакого значения, то ее вызов будет выглядеть так:

```
obj1.show();
```

Методы делятся на *методы объектов* и *методы классов*. Чаще всего встречаются методы объектов. Они так называются потому, что пользуются полями данных объектов,

и поэтому их можно вызывать только из самих объектов. Методы классов, напротив, не пользуются полями данных объектов, они могут работать при отсутствии объекта. Поэтому их можно вызывать как из классов, так и из объектов. Формат вызова:

```
имяКласса.имяМетода(список параметров)
```

или

```
имяОбъекта.имяМетода(список параметров).
```

Модификатором называется зарезервированное слово, задающее правила доступа к элементу или особенность его реализации. Его ставят перед типом и именем задаваемого поля, метода или класса. Разрешается писать подряд несколько модификаторов, поясняющих разные особенности элемента.

Про то, как задавать методы в классе, какие бывают модификаторы и какие существуют правила видимости у методов, полей данных и классов, будет рассказано в *главе 6*, где излагаются начальные сведения об объектном программировании.

При задании метода класса перед его именем необходимо поставить модификатор `static` — "статический". Мы никогда не будем употреблять для таких методов название "статические методы", так как оно крайне неудачно, а будем называть их методами класса, как это принято в теории программирования.

Точно так же переменные (поля данных) делятся на *переменные объектов* и *переменные классов*. При задании переменной класса перед ее именем необходимо поставить модификатор `static`. Переменные класса, как и методы класса, можно вызывать как из классов, так и из объектов.

Формат вызова:

```
имяКласса.имяПеременной
```

или

```
имяОбъекта.имяПеременной.
```

Не следует путать классы, объекты и объектные переменные. Класс — это тип, т. е. описание того, как устроена ячейка памяти, в которой будут располагаться поля данных объекта. Объект — это содержимое данной ячейки памяти. А в переменной объектного типа содержится адрес объекта, т. е. адрес ячейки памяти. Сказанное относится только к языкам с динамической объектной моделью, в частности к Java. Для C++ это не так.

Как уже было сказано, кроме полей данных в классе описываются методы. Несмотря на схожесть задания в классе полей и методов их реальное размещение во время работы программы отличается. Методы не хранятся в объектах, но объекты могут их вызывать. Каждый объект имеет свой комплект полей данных — "носит свои данные с собой". Если есть сотня объектов одного типа, т. е. экземпляров одного и того же класса, то в памяти компьютера будет сотня ячеек памяти, устроенных так, как это описано в классе. Причем у каждого объекта значения этих данных могут быть свои. Например, если объект является окружностью, отрисовываемой на экране, то у каждой окружности будет свой набор координат, радиусов и цветов. Если мы будем рисовать окружности с помощью метода `show()`, нет необходимости в каждом объекте хранить код этого метода — он для всех ста объектов будет одним и тем же. Поэтому методы хранятся не в объектах, а в классах.

Класс — более общая сущность, чем объект, и до того, как во время работы программы в памяти компьютера будет создан объект, сначала должен быть загружен в память со-

ответствующий ему класс. В Java можно создавать переменные типа "класс", и с их помощью обращаться к классам так, будто это объекты особого рода. Но, в отличие от обычных объектов, такие "объекты" не могут существовать в нескольких экземплярах, и правила работы с ними принципиально отличаются от работы с объектами. Такие сущности называются *метаобъектами*.

Переменные можно объявлять либо в классе, либо в методе. В первом случае мы будем говорить, что переменная *глобальная*, т. е. является полем данных объекта. Во втором — что она *локальная*. Переменные, объявленные в классе, можно использовать внутри методов независимо от того, в каком месте исходного кода задана эта переменная — до или после объявления соответствующего метода. Считается, что область задания элементов класса предшествует области кода внутри заданных в этом классе методов.

1.7. Работа со ссылочными переменными. Сборка мусора

Объекты, как мы уже знаем, — это экземпляры ссылочных типов. Работа с переменными ссылочного и примитивного типов принципиально различается. В каждой переменной примитивного типа содержится свое значение, и его изменение не влияет на те значения, которые можно получить с помощью других переменных. Причем имя переменной примитивного типа можно рассматривать как имя ячейки с данными, и у такой ячейки может быть только одно имя, которое не может меняться по ходу работы программы. Для ссылочных переменных это не так.

Переменные ссылочного типа содержат не сами данные, а их адреса. Поэтому присваивания для таких переменных меняют адреса, но не данные. Кроме того, из-за этого под них выделяется одинаковый объем памяти независимо от типа объектов, на которые они ссылаются. А имена ссылочных переменных можно рассматривать как псевдонимы имен ячеек с данными — у одной и той же ячейки с данными может быть сколько угодно псевдонимов, т. к. адрес одной и той же ячейки можно копировать в произвольное число переменных соответствующего типа. И все они будут ссылаться на одну и ту же ячейку с данными.

В Java ссылочные переменные служат для работы с объектами: в этом языке программирования используется *динамическая объектная модель*, и все объекты создаются динамически, с явным указанием в программе момента их создания. Это отличает Java от C++, языка со *статической объектной моделью*. В C++ могут существовать как статически заданные объекты, так и динамически создаваемые. Но это не преимущество, как могло бы показаться, а проблема, т. к. в ряде случаев могут возникнуть принципиально неразрешимые ситуации.

Следует отметить, что сами объекты безымянны, и доступ к ним осуществляется только через ссылочные переменные. Часто говорят про ссылочную переменную как про сам объект, поскольку долго и неудобно произносить "объект, на который ссылается данная переменная". Но этого по мере возможности следует избегать.

Ссылочной переменной любого типа может быть присвоено значение `null`, означающее, что она никуда не ссылается. Попытка доступа к объекту через такую переменную вызовет ошибку. В Java все ссылочные переменные первоначально инициируются значением `null` (если им не назначена ссылка на объект прямо в месте объявления). Очень

часто встречающаяся ошибка — попытка доступа к полю или методу с помощью ссылочной переменной, которой не сопоставлен объект. Такая ошибка не обнаруживается на этапе компиляции и проявляется при выполнении (Run-time error). При этом приложение Java генерирует исключительную ситуацию (ошибку) попытки доступа к объекту через ссылку `null` с сообщением примерно такого вида:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException.
```

Последовательность действий со ссылочными переменными и объектами наглядно иллюстрируют рисунки, на которых каждой такой переменной и каждому объекту сопоставлен прямоугольник — символическое изображение ячейки (рис. 1.1). Около ячейки пишут ее имя (если оно есть), а внутри — значение, содержащееся в ячейке.

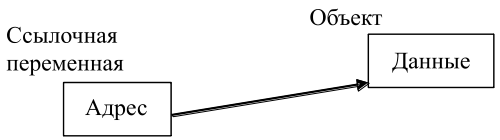


Рис. 1.1. Ссылочная переменная и связанный с ней объект

Если переменная является ссылочной, из нее выходит стрелочка-ссылка. Она кончается на том объекте, на который указывает ссылка. Если в ссылочной переменной содержится значение `null` (ссылка "в никуда", то есть `адрес==0`), рисуется "висящая" короткая стрелка, у которой находится надпись `null`. Отметим, что в Java символом равенства служит `==`, а не символ `=`, который является оператором присваивания.

При использовании предлагаемых обозначений в публикациях или при обсуждениях удобно руководствоваться следующими правилами. Если ссылка "перещелкивается", то либо создается новый рисунок (в книжке), либо перечеркивается крестиком прежняя стрелочка и рисуется новая (на листе бумаги или на доске). При новом перещелкивании эта "старая" стрелка перечеркивается двумя крестиками, а "более свежая" (которая была перещелкнута) — одним крестиком, и т. д. Такая система обозначений позволяет наглядно представить, что происходит при работе со ссылками, и не запутаться в том, куда они указывают и с какими ссылками в какой момент связаны динамически создаваемые объекты.

При выполнении оператора `new`, после которого указан вызов конструктора, динамически выделяется новая безымянная ячейка памяти, имеющая тип, соответствующий типу конструктора, а сам конструктор после окончания работы возвращает адрес этой ячейки. Если у нас в левой части присваивания стоит ссылочная переменная, то в результате ссылка "перещелкивается" на динамически созданную ячейку.

Рассмотрим этот процесс подробнее. Будем считать, что сначала в ячейке `circle1` типа `Circle` хранится нулевой адрес (значение ссылки равно `null`). Изобразим это как стрелку "в никуда" с надписью `null` (рис. 1.2).

После оператора

```
circle1 = new Circle(x1, y1, r1);
```

в динамически выделенной безымянной ячейке памяти будет создан объект-окружность с координатами центра `x1`, `y1` и радиусом `r1` (это какие-то значения, конкретная величина которых в данном случае не важна) (рис. 1.3).

Рис. 1.2. Начальное состояние ссылочной переменной `circle1`

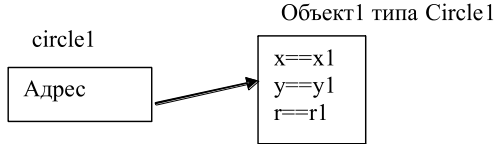


Рис. 1.3. Ссылочная переменная `circle1` и связанный с ней объект типа `Circle`

Поля объекта доступны через ссылку как по чтению, так и по записи. До тех пор пока ссылочная переменная `circle1` содержит адрес `Объект1`, имя `circle1` является псевдонимом, заменяющим имя этого объекта. Его можно использовать так же, как имя обычной переменной в любых выражениях и операторах, не связанных с изменением адреса в переменной `circle1`.

Поскольку между ссылочными переменными одного типа разрешены присваивания, переменная по ходу программы может сменить объект, на который она ссылается. Если `circle2` также имеет тип `Circle`, то допустимы присваивания вида:

```
circle2 = circle1;
```

Такие присваивания изменяют адреса в ячейках ссылочных переменных, но не меняют содержания объектов.

Рассмотрим следующий участок кода (рис. 1.4):

```
Circle circle1 = new Circle(x1, y1, r1);
Circle circle2 = new Circle(x2, y2, r2);
Circle circle3;
```

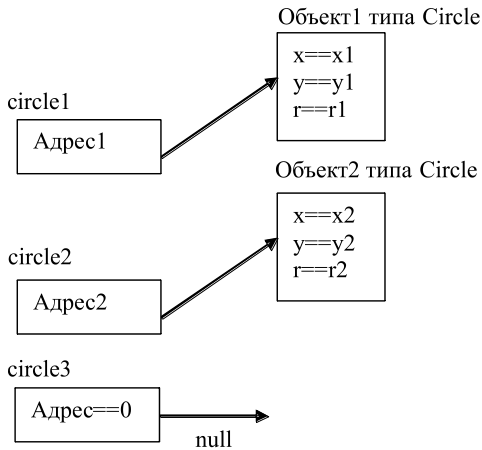


Рис. 1.4. Ссылочные переменные и связанные с ними объекты (состояние до присваивания)

Проведем присваивание:

```
circle3 = circle2;
```

В результате получится состояние переменных и объектов, изображенное на рис. 1.5.

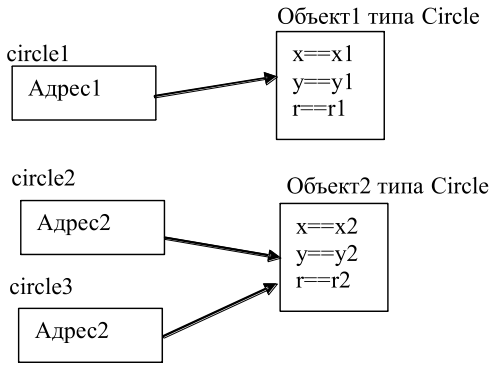


Рис. 1.5. Ссылочные переменные и связанные с ними объекты (состояние после присваивания `circle3=circle2`)

Обе переменные (как `circle2`, так и `circle3`) теперь ссылаются на один и тот же объект — в них находится один и тот же `Адрес2` (т. е. оба имени — синоним имени `Объект2`). Напомним, что сам объект, как все динамически создаваемые величины, безымянный. Таким образом, `circle2.x` даст значение `x2`, точно так же, как и `circle3.x`. Более того, если мы поменяем значение `circle2.x`, это приведет к изменению `circle3.x` — ведь это одно и то же поле `x` нашего `Объект2`.

Рассмотрим теперь, что произойдет при присваивании:

```
circle1 = circle2;
```

Этот случай (рис. 1.6) отличается от предыдущего только тем, что переменная `circle1` до присваивания уже была связана с объектом.

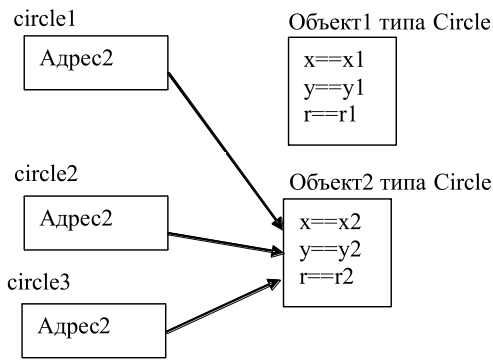


Рис. 1.6. Ссылочные переменные и связанные с ними объекты (состояние после присваивания `circle1=circle2`)

В результате у `Объект2` окажется сразу три ссылочные переменные, которые с ним связаны и имена которых являются его псевдонимами в данном месте программы: `circle1`, `circle2` и `circle3`. При этом программная связь с `Объект1` окажется утерянной — он занимает место в памяти компьютера, но программный доступ к нему невозможен, поскольку адрес этого объекта программой утерян. Таким образом, он бесполезен и напрасно занимает ресурсы компьютера.

Подобные ячейки памяти называют *мусором*. В Java предусмотрен механизм высвобождения памяти, занятой такими бесполезными объектами, — "сборка мусора" (garbage collection), работающий автоматически. Этим в фоновом режиме занимается специальная часть виртуальной Java-машины — *сборщик мусора*. При программировании на Java, в отличие от таких языков, как C/C++ или Object Pascal, программисту нет необходимости самому заботиться о высвобождении памяти, занятой под динамически создаваемые объекты.

Следует подчеркнуть, что намеренная потеря связи ссылочной переменной с ненужным уже объектом — это одно, а непреднамеренная — совсем другое. Если вы не планировали потерю связи с объектом, а она произошла, возникнет логическая ошибка. И хотя она не приведет к зависанию программы или ее неожиданному закрытию, такая программа будет работать не так, как вы предполагали, т. е. неправильно или не совсем правильно (что иногда еще опасней, т. к. ошибку можно не заметить или, заметив, очень трудно понять ее причину).

1.8. Проекты. Пакеты. Уровни видимости классов. Импорт классов

Современное программное обеспечение построено по модульному (блочному) принципу. Программы давно перестали состоять из одного файла. Поэтому слово "программа" лучше заменять на "проект". Тем более что термин "программа", как уже говорилось, неоднозначен.

Идеология Java подразумевает работу в компьютерных сетях и возможность подгрузки в необходимый момент через сеть требуемых классов и ресурсов, в которых нуждается программа и которые не были до того загружены. Для обеспечения такого рода работы приложения Java разрабатываются и распространяются в виде большого числа независимых классов. Однако такой способ разработки приводит к чрезвычайно высокой фрагментации программы. Даже небольшие учебные проекты часто состоят из десятков классов, а реальные проекты — из сотен. При этом каждому общедоступному (public) классу соответствует свой файл, имеющий то же имя. Для того чтобы справиться с таким обилием файлов, в Java предусмотрено специальное средство группировки классов, называемое пакетом (package). Пакеты обеспечивают независимые *пространства имен* (namespaces), а также ограничение доступа к классам.

Классы всегда задают в каком-либо пакете. Пакеты могут быть вложенными с произвольным уровнем вложения (ограничивается только операционной системой и, как правило, не менее 256). Каждому пакету сопоставлена папка с исходными кодами соответствующих классов, при этом пакету, вложенному в другой, соответствует папка, вложенная в другую.

Для того чтобы поместить класс в пакет, требуется продекларировать имя пакета в начале файла, в котором объявлен класс, в следующем виде:

```
package имя_пакета;
```

Кроме того, необходимо поместить исходный код класса в папку, соответствующую пакету.

Если декларация имени пакета отсутствует, считается, что класс принадлежит пакету с именем `default`.

Вложенным пакетам соответствуют составные имена. Например, если мы имеем пакет с именем `pkg1`, в который вложен пакет с именем `pkg2`, в который вложен пакет с именем `pkg3`, то объявление, что класс с именем `MyClass1` находится в пакете `pkg3`, будет выглядеть так:

```
package pkg1.pkg2.pkg3;
class MyClass1 {
....
}
```

Внутри фигурных скобок должно содержаться описание класса. Оно заменено многоточием.

Разделителем имен пакетов в программе служит точка независимо от типа операционной системы. Хотя в разных операционных системах вложенность папок будет обозначаться по-разному:

- в MS Windows — `pkg1\pkg2\pkg3\`
- в UNIX и Linux — `pkg1/pkg2/pkg3/`
- в Mac OS — `pkg1:pkg2:pkg3:`

При создании проекта в среде NetBeans класс автоматически помещается в пакет.

Не следует путать пакеты (packages) Java и модули (packs) среды NetBeans. Модули обеспечивают добавление новых возможностей среде NetBeans, а пакеты — средство группировки классов в языке Java.

При декларации класса можно указывать, что он общедоступен, с помощью *модификатора доступа* `public`:

```
public class MyClass2 {
....
}
```

При этом возможен доступ к данному классу из других пакетов.

Если же модификатор `public` отсутствует, как в случае `MyClass1`, то доступ к классу разрешен только из классов, находящихся с ним в одном пакете. Про такие файлы говорят, что у них *пакетный* вариант доступа (видимости). В C++ аналогичный вид доступа называется "дружественным" — friend, а в Java официальное, но не очень часто используемое название для пакетного способа видимости — default, то есть "видимость по умолчанию".

В файле с расширением .java можно располагать только один общедоступный класс и произвольное число классов с пакетным уровнем видимости.

Класс может использовать общедоступные (public) классы из других пакетов напрямую, с указанием полного имени класса в пространстве имен, включающего имя пакета. Например, доступ к классу `MyClass2` в таком варианте осуществляется так:

```
pkg1.pkg2.pkg3.MyClass2
```

Для того чтобы задать переменную объектного типа, нужно прежде указать ее класс. В нашем случае это будет выглядеть так:

```
pkg1.pkg2.pkg3.MyClass2 myObject;
```

Чтобы отличать имена классов от имен пакетов, в Java принято имена пакетов писать только строчными буквами, имена классов начинать с заглавной буквы, а имена полей данных (в том числе имена объектных переменных) и методов начинать со строчной буквы. Если имя класса, поля данных или метода (но не пакета!) состоит из нескольких слов, то каждое новое слово принято писать с заглавной буквы. Новое слово также можно отделять от предыдущего символом подчеркивания. Таким образом, из названия

```
javax.swing.JMenuItem
```

понятно, что:

`javax` и `swing` — пакеты,

`JMenuItem` — имя класса.

Существует способ доступа к именам из другого пакета напрямую, без указания каждый раз полного пути в пространстве имен. Это делается с помощью оператора `import`. Если мы хотим импортировать имя класса `MyClass2` из пакета `pkg3`, то после объявления имени нашего пакета (например, `mypack1`), но до объявления нашего класса (например, `MyClass3`) пишут

```
import pkg1.pkg2.pkg3.MyClass2;
```

При этом в классе `MyClass3` имя `MyClass2` можно указывать напрямую, без имени пакета перед ним: `pkg1.pkg2.pkg3`. Задание переменной будет выглядеть так:

```
MyClass2 myObject;
```

Но если мы импортируем пакеты, содержащие классы с одинаковыми именами, то требуется указание полного имени класса, квалифицированного именем пакета.

Если мы хотим импортировать имена всех классов из пакета, в операторе `import` после имени пакета вместо имени класса следует написать `*`.

Пример:

```
import pkg1.pkg2.pkg3.*;
```

Заметим, что импортируются только имена файлов, находящихся точно на уровне указанного пакета. Импорта имен из вложенных в него пакетов не происходит. Например, если записать

```
import pkg1.*;
```

или

```
import pkg1.pkg2.*;
```

то имя класса `MyClass2` не будет импортировано, поскольку он находится внутри `pkg1` и `pkg2` не непосредственно, а через пакет `pkg3`.

Имеется одно исключение из правила для импорта: классы ядра языка Java, содержащиеся в пакете `java.lang`, импортируются автоматически без указания имени пакета.

Например, объявить графический объект `g`, имеющий тип `java.awt.Graphics`, можно тремя способами:

- напрямую с указанием имени пакета и класса:

```
java.awt.Graphics g;
```

- с предварительным импортом класса `Graphics` из пакета `java.awt` и последующим указанием имени этого класса без его спецификации именем пакета:

```
import java.awt.Graphics;
...
Graphics g;
```

- с предварительным импортом всех классов (в том числе `Graphics`) из пакета `java.awt` и последующим указанием имени этого класса без его спецификации именем пакета:

```
import java.awt.*;
...
Graphics g;
```

Следует отметить, что обращение к переменным класса и методам класса должно идти только через имя класса. Например, константа `PI` и тригонометрические функции `sin`, `cos` и др. заданы в классе `java.lang.Math`.

Поэтому присваивание, выглядящее в математической форме записи как $y=\frac{\sin \pi x}{\pi x}$, в программе Java в традиционном варианте выглядит очень не похоже на оригинал, приходится писать совершенно нечитаемый текст:

```
y=Math.sin(Math.PI*x)/(Math.PI*x);
```

Пакет `java.lang` и все входящие в него классы импортируются по умолчанию, поэтому нам не придется писать `java.lang.Math.PI` или `java.lang.Math.sin`. Но писать просто `PI` или `sin` раньше было невозможно, либо приходилось в каждом классе задавать константу:

```
final double PI=Math.PI;
```

и функцию:

```
double sin(double x){
       return Math.sin(x);
}
```

Если же была нужна не только функция `sin`, то приходилось перезадавать все другие математические функции. Характерная особенность Java — небольшой размер классов и методов, так что такой подход оказывался бесполезным.

Начиная с JDK 5, в Java появилась возможность *статического импорта* из пакета классов переменных класса и методов класса.

Можно импортировать как отдельные переменные или методы класса:

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
```

так и все члены класса:

```
import static java.lang.Math.*;
```

После такого импорта выражение приобретает более читаемый вид:

```
y = sin(PI * x) / (PI * x);
```

Имена предопределенных пакетов Java начинаются с имени java или javax. Пользовательским пакетам задавать такие имена нельзя.

При работе с пакетами возникает проблема конфликта имен. Допустим, два разработчика, участвующие в одном проекте, создали пакеты с одинаковыми именами. Или же вы пытаетесь использовать библиотеки классов от двух разных производителей и обнаруживаете, что в них содержатся пакеты с одинаковыми именами, хотя эти пакеты не имеют друг к другу никакого отношения. Возникает конфликт имен, т. к. некоторые классы пакетов имеют одинаковые имена. Очевидно, что попытки замены одних осмысленных названий пакетов и содержащихся в них классов на другие не дадут никакой гарантии. Поэтому должна быть выстроена некоторая система построения уникальных имен.

В Java предлагается методика, гарантирующая отсутствие совпадений имен пакетов (а значит, и имен классов, т. к. полное имя класса состоит из имени пакета и "короткого" имени класса). Предлагается давать пакетам имена, совпадающие с доменными WWW-адресами, но записанными в обратном порядке.

Например, если у физического факультета СПбГУ доменное имя phys.spbu.ru (веб-адрес **http://www.phys.spbu.ru/**), то имя головного пакета для всех разработчиков с физического факультета должно быть ru.spbu.phys. При этом работающие на кафедре вычислительной физики (computational physics) сотрудники будут создавать свои пакеты и классы в пакете ru.spbu.phys.cph, т. к. для данной кафедры сокращенное имя в сети cph. А работающие на кафедре высшей математики и математической физики (mathematical physics) сотрудники будут создавать свои пакеты и классы в пакете ru.spbu.phys.mph, т. к. для данной кафедры сокращенное имя в сети mph.

Имя самого внешнего пакета для всех организаций России будет ru, для Франции — fr и т. д.

Поэтому, если вы будете изучать состав папок и файлов какого-нибудь пакета (например, в JDK), не удивляйтесь, если сначала будет папка, в которую вложена только одна папка, в ней — в свою очередь только одна папка, и т. д. И только в самой внутренней папке найдутся файлы классов. Такая система не только гарантирует отсутствие конфликта имен, но и дает возможность легко найти сайт разработчика и подразделение, в котором разработан соответствующий пакет.

1.9. Базовые пакеты и классы Java

Стандартные библиотеки являются гораздо более важной частью языка программирования, чем это кажется. Начинающие программисты часто сравнивают, "какой язык программирования лучше", но автору ни разу не довелось услышать при этом обсуждения достоинств и недостатков стандартных библиотек этих языков. В то же время сам по себе язык программирования — это лишь инструмент для создания программ ("строительный кран"), а "строительный материал", из которого, как из кирпичиков и

блоков, создается строение (программа), — библиотеки. Конечно, и в самом языке обычно имеются конструкции, позволяющие программисту самостоятельно создавать "кирпичики" и "строительные блоки" программы. Но в нашей аналогии это скорее песок и цемент. При строительстве реального (а не игрушечного) дома они годятся только для того, чтобы создать бетон, скрепляющий готовые блоки или кирпичи. Ведь не будете же вы строить дом, разрабатывая для него каждый кирпич из отдельных песчинок!

Совершенно аналогично постройке домов идет строительство современных программных комплексов. Операторы языка программирования используются в основном для того, чтобы скреплять между собой уже готовые программные конструкции, оформленные в виде библиотек. В Java такие библиотеки оформляют в виде стандартных пакетов.

В состав JDK 1.6 входят следующие пакеты: `java`, `javax`, `org` и `com.sun`. В них, в свою очередь, находится большое число вложенных пакетов.

В табл. 1.6 приведено содержимое пакета `java`.

Таблица 1.6. *Содержимое пакета* `java`

Пакет, класс	Краткое описание
`java.applet`	Поддержка работы с апплетами
`java.awt`	Базовый пакет работы с графическим пользовательским интерфейсом (Abstract Window Toolkit — Абстрактный инструментарий графического окна)
`java.beans`	Поддержка компонентной модели JavaBeans
`java.io`	Поддержка базовых средств ввода/вывода
`java.lang`	Содержит базовые классы языка Java. Автоматически импортируется в любую программу без указания имени пакета
`java.lang.reflect`	Поддерживает механизм доступа к классам как метаобъектам, обеспечивает динамическое выяснение программой, какие возможности поддерживает класс. Данный механизм называется reflection — "отражение"
`java.lang.Math`	Класс, обеспечивающий поддержку основных математических функций, а также простейшее средство генерации псевдослучайных чисел
`java.math`	Поддержка вычислений с целыми числами произвольной длины, а также числами в формате с плавающей точкой произвольной точности
`java.net`	Поддержка работы в Интернете, а также соединений через сокеты (sockets)
`java.nio`	Содержит классы и пакеты для поддержки сетевых соединений, расширяющие возможности пакета `java.io`. В частности, содержит классы контейнеров (буферов) для создания списков с данными различных примитивных типов, а также пакеты `channels` (каналы соединения, коннекции) и `charset` (национальный набор символов). Пакет `charset` обеспечивает поддержку перекодирования из символов Unicode в последовательность байтов для передачи через канал связи, а также обратное преобразование
`java.rmi`	Поддержка вызовов удаленных методов

Таблица 1.6 *(окончание)*

Пакет, класс	Краткое описание
java.security	Поддержка специальных средств, обеспечивающих безопасность приложения, в том числе при работе в компьютерных сетях (списки доступа, сертификаты безопасности, шифрование и т. д.)
java.sql	Поддержка SQL-запросов к базам данных
java.text	Поддержка специальных средств, обеспечивающих локализацию программ, — классы, поддерживающие работу с текстом, датами, текстовым представлением чисел. Кроме того, содержит средства зависимого от локализации сравнения строк
java.util	Содержит важнейшие классы для работы со структурами данных (в том числе необходимых для работы с событиями и датами). В частности — поддержку работы с массивами (сортировка, поиск), а также расширенные средства генерации псевдослучайных чисел
java.util.jar	Поддержка работы с jar-архивами (базовым видом архивов в Java)
java.util.zip	Поддержка работы с zip-архивами

Содержимое пакета `javax`, обеспечивающего поддержку возможностей, появившихся в Java 2, приведено в табл. 1.7.

Таблица 1.7. *Содержимое пакета* `javax`

Пакет	Краткое описание
javax.accessibility	Обеспечивает настройку специальных возможностей представления информации для людей с плохим зрением, слухом и т. п., а также других случаев, когда требуется специализированный доступ для управления информационными объектами
javax.activity	Вспомогательный пакет для работы с компонентами
javax.crypto	Поддержка шифрования-расшифровки данных
javax.imageio	Поддержка работы с изображениями (ввод/вывод)
javax.management	Поддержка работы с управляющими компонентами (MBean — Management Bean)
javax.naming	Поддержка работы с пространством имен компонентов
javax.net	Поддержка работы в Интернете, а также соединений через сокеты (sockets). Расширение возможностей `java.net`
javax.print	Поддержка работы с печатью документов
javax.rmi	Поддержка вызовов удаленных методов. Расширение возможностей `java.rmi`
javax.security	Поддержка специальных средств, обеспечивающих безопасность приложения. Расширение возможностей `java.security`
javax.sound	Поддержка работы со звуковыми потоками и файлами

Таблица 1.7 *(окончание)*

Пакет	Краткое описание
javax.sql	Поддержка SQL-запросов к базам данных. Расширение возможностей java.sql
javax.swing	Библиотека основных графических компонентов в Java 2
javax.transaction	Поддержка работы с транзакциями
javax.xml	Поддержка работы с XML-документами и парсерами

В пакете org (табл. 1.8) находятся пакеты, предоставляемые свободным сообществом разработчиков.

Таблица 1.8. *Содержимое пакета* org

Пакет	Краткое описание
org.ietf	Поддержка защищенных соединений по протоколу GSS (Kerberos v5)
org.omg	Средства для использования из программ на Java технологии CORBA, применяемой для создания распределенных объектных приложений
org.w3c	Интерфейсы для работы с XML-документами в соответствии со спецификацией DOM
org.xml	Поддержка работы с XML-документами

Пакет com.sun (табл. 1.9) от фирмы Sun Microsystems в основном обеспечивает расширение возможностей пакета javax.

Таблица 1.9. *Содержимое пакета* com.sun

Пакет	Краткое описание
com.sun.accessibility	Дополнение к пакету javax.accessibility
com.sun.beans	Дополнение к пакету java.beans
com.sun.corba	Поддержка работы в компьютерных сетях с базами данных по технологии CORBA (Common Object Request Broker Architecture)
com.sun.crypto	Дополнение к пакету javax.crypto
com.sun.image	Поддержка работы с изображениями
com.sun.imageio	Дополнение к пакету javax.imageio
com.sun.java	Поддержка стилей показа приложений *(см. разд. 2.9)*, а также служебные утилиты для работы с браузерами и WWW-документами
com.sun.java_cup	Поддержка технологии JavaCup
com.sun.jlex	Поддержка работы лексического анализатора
com.sun.jmx	Дополнение к пакету javax.management
com.sun.jndi	Пакет в процессе разработки
com.sun.management	Дополнение к пакету javax.management

Таблица 1.9 *(окончание)*

Пакет	Краткое описание
com.sun.media	Поддержка работы со звуком
com.sun.naming	Дополнение к пакету javax.naming
com.sun.net	Дополнение к пакету javax.net
com.sun.org	Поддержка взаимодействия с сервером Apache, средства работы с базами данных по технологии CORBA
com.sun.rmi	Дополнение к пакету javax.rmi

Также имеется пакет org.jdesktop, не входящий в состав JDK, но поставляемый с NetBeans. В нем заданы классы, обеспечивающие работу приложений с графическим пользовательским интерфейсом. Они являются надстройкой над некоторыми классами, заданными в пакетах java.beans, javax.swing, org.jdesktop.swingworker, и используются в компонентной модели NetBeans. В частности, в пакете org.jdesktop находится пакет application, обеспечивающий поддержку работы приложений Desktop Application *(см. главу 13)*, а также пакет beansbinding, обеспечивающий поддержку технологии связывания (binding) компонентов Java Beans *(см. главу 15)*.

1.10. Технологии Java, .NET, ASP, PHP

Язык Java был создан в 1995 году как платформо-независимый язык прикладного программирования. Он очень быстро приобрел широкую популярность и заметно потеснил языки C и C++ в области разработки прикладного программного обеспечения. В результате стали говорить о технологии Java и о платформе Java, подчеркивая, что это больше, чем просто язык программирования. В 1998 году появилась компонентная модель Java Beans, и ряд сред разработки приложений Java стал успешно конкурировать со средами, обеспечивающими визуальное проектирование пользовательского интерфейса — Microsoft Visual Basic и Borland Delphi. Казалось, что язык Java завоевал лидирующие позиции в области прикладного программирования.

Однако в 2000 году фирмой Microsoft была предложена новая технология, названная .NET, в большой степени вобравшая в себя основные черты технологии Java: динамическую объектную модель, повышенную безопасность приложений (в том числе обеспечиваемую применением ссылок и сборщика мусора), использование виртуальной машины и платформо-независимого байтового кода, JIT-компиляцию этого кода. Но технология .NET имела ряд новых черт.

Во-первых, вместо одного языка программирования в .NET стало возможно использование произвольного числа языков программирования. От них требовалось только, чтобы они удовлетворяли спецификации, позволяющей скомпилированным классам работать под управлением виртуальной машины, называемой в .NET Common Language Environment или Common Language Runtime — общей исполняющей средой поддержки языков программирования. Базовым языком программирования стал созданный одновременно с .NET язык C#, фактически явившийся усовершенствованным вариантом языка Java, но несовместимый с ним как по ряду синтаксических конструкций, так и по

скомпилированному коду. Андерс Хейлсберг, создатель Turbo Pascal и один из ведущих разработчиков среды Delphi (язык Object Pascal), перейдя из компании Borland Ltd в Microsoft, создавая C# и технологию .NET, заменил в Java ряд синтаксических конструкций и добавил новые на основе идей, заимствованных из Delphi.

Во-вторых, если технология Java рассматривалась всеми в качестве языка программирования, то технология .NET фактически создавалась как платформо-независимая часть операционной системы MS Windows. Поэтому важной частью .NET стал набор базовых классов .NET Framework, обеспечивающий поддержку прикладного программирования в большинстве практически важных областей.

В .NET основой программирования (как и в Java) служат классы. Исходный код класса, написанный на любом из языков .NET (т. е. удовлетворяющий спецификации Common Language Environment), компилируется в платформо-независимый код. Этот код уже не имеет специфики языка программирования, на котором был написан, работает под управлением исполняющей среды .NET и может использоваться любыми другими классами .NET. Причем скомпилированный код класса может использоваться не только для вызовов, но и для наследования и обеспечения полиморфизма. Такие классы называются *компонентами*. Важно то, что для использования каким-либо приложением необходимого класса .NET Framework как при разработке приложения, так и при его запуске на компьютере пользователя нет необходимости загружать класс через Интернет или устанавливать его каким-либо другим образом. Достаточно того, чтобы был установлен свободно распространяемый пакет компонентов .NET Framework, что делается в версиях MS Windows (начиная с Windows XP) непосредственно во время установки операционной системы. Причем набор компонентов в пакете стандартизован, что обеспечивает гарантированное нахождение нужного компонента и его работоспособность. Именно эти особенности обеспечивают преимущество оболочки операционной системы по сравнению с отдельными программами и пакетами.

То, что технология .NET была сделана открытой и стандартизирована ISO (International Standard Organization — Международная организация по стандартам) в 2004 году, безусловно, сделало ее привлекательной для многих разработчиков.

Появление проекта Mono с реализацией .NET под Linux, а также ряда других проектов по реализации .NET под разными операционными системами позволило этой технологии выйти за пределы семейства операционных систем одной фирмы и на первый взгляд сделало позиции Java неконкурентоспособными в данной области. Действительно, язык программирования — это одно, а оболочка операционной системы с набором большого числа компонентов — совсем другое. Тем более что язык Java в варианте J.NET стал одним из возможных языков .NET, т. е. вошел в эту технологию как составная часть.

Правда, сходство J.NET с Java касается только синтаксиса языка, но не библиотек, а мы уже знаем, что это дает не очень много. Ведь именно стандартные библиотеки обеспечивают основную часть возможностей языка программирования. Поэтому реальное программирование на языке J.NET очень сильно отличается от программирования на Java.

Отметим, что слово Java — торговая марка Sun, поэтому называть .NET-версию языка как Java.NET другие фирмы не имели права. По той же причине Microsoft называла свой компилятор для Java как Visual J++, а Borland — как J-Builder.

Через относительно короткий срок неутешительные для Java тенденции изменились. Стало ясно, что надежды на реализацию .NET кем-либо, кроме Microsoft, оказались несостоятельны, а в рамках технологии Java удалось найти достойный ответ на вызов со стороны .NET. Им стала платформа NetBeans, технология Java Scripting и идеология Open Source (открытый исходный код).

NetBeans — это технология компонентного программирования, созданная первоначально на основе модели Java Beans, о которой речь пойдет позже. Свободно распространяемая среда разработки NetBeans позволяет создавать различные типы программ на языке Java, в том числе с использованием компонентов NetBeans, а также создавать такие компоненты. Более того, в настоящее время среда разработки NetBeans позволяет разрабатывать программы и на других языках программирования. В частности, при установке модуля C/C++ Pack возможна разработка и компиляция программ на языках C и C++. Возможно подключение модулей (Packs), поддерживающих языки программирования JavaScript, Ruby, Groovy, PHP, JavaFX и др. Причем набор языков ничем не ограничен — необходимо только удовлетворять требованиям подключения модулей к среде NetBeans.

Особенно интересным направлением развития в Java, начиная с JDK 6, стала поддержка технологии *Java Scripting* (см. официальную документацию на сайте **http://java.sun.com/javase/6/docs/technotes/guides/scripting/programmer_guide/index.html**). В этой технологии с помощью программного интерфейса (*API*, application programming interface), заданного в пакете javax.script, сценарии различных интерпретируемых языков могут быть универсальным образом вызваны в момент выполнения Java-кода. Такой код может не только вызывать внешние сценарии, но и предоставлять этим сценариям доступ к избранным Java-объектам. В настоящее время на основе этой технологии поддерживается около трех десятков языков (см. **https://scripting.dev.java.net/**), причем число поддерживаемых языков быстро растет. Особенно удобно то, что среда NetBeans поддерживает редактирование исходного кода для таких языков (с подсветкой синтаксиса и диагностикой ошибок).

Платформой NetBeans (NetBeans Platform) называется базовая часть среды разработки NetBeans без подключенных внешних модулей. Это та часть, которая не ориентирована на поддержку языка Java. Она гораздо меньше по размеру и может быть использована для создания программистами собственных средств разработки путем подключения к ней внешних модулей. Как платформа NetBeans, так и все модули среды NetBeans доступны в виде исходных кодов на сайте **http://www.netbeans.org/downloads/zip.html**.

Подключение новых языков программирования к платформе NetBeans возможно двумя способами.

Первый — реализация всех интерфейсов среды. Это наиболее мощный и гибкий способ, но самый сложный и затратный.

Второй способ, годящийся для интерпретируемых языков, т. е. преимущественно для языков сценариев, — это использование технологии, разработанной в рамках проекта Schliemann (**http://www.netbeans.org/community/magazine/html/03/schliemann/**).

Движение свободного программного обеспечения (Open Source), набирающее популярность в последние годы, стремится к всеобщей открытости программного кода. При этом следует различать два варианта открытости. Первый из них (freeware — свободно распространяемое программное обеспечение) подразумевает свободное распростране-

ние и использование программ и их исходных кодов, как правило, с единственным ограничением — сохранением открытости и свободы распространения программ и кодов программных продуктов на основе данного исходного кода. Второй требует открытости исходного кода для изучения и, при необходимости, исправления ошибок, но не всегда означает передачу авторских прав на какое-либо другое использование этого исходного кода.

Среда и компоненты NetBeans 6.5 распространяются на основе лицензий GNU General Public License Version 2 with Classpath Exception ("GPLv2 with Classpath Exception") или Common Development and Distribution License ("CDDL"), пользователь имеет право выбрать ту из них, которая ему больше подходит.

Лицензия GNU GPL предоставляет пользователю права свободно запускать, копировать, модифицировать и распространять программу, а также изучать, как она устроена и работает. При этом обязательно, чтобы вместе с программой свободно распространялись и ее исходные коды, включая все сделанные изменения, и по той же самой лицензии.

Лицензия CDDL ("Общая лицензия на разработку и распространение") разработана корпорацией Sun Microsystems и основана на Mozilla Public License (MPL) версии 1.1. Она требует предоставления разработчиками исходных кодов программного обеспечения, но только для ознакомления — без права их модификации кем-либо, кроме самого разработчика.

Мультиплатформенность и наличие множества свободно распространяемых компонентов NetBeans в сочетании с качественной бесплатной средой разработки и очень широким использованием языка Java, а также активно развивающейся поддержкой других языков программирования дает возможность надеяться, что NetBeans сможет стать унифицированной оболочкой разработки программных продуктов для различных операционных систем, и что компоненты NetBeans (в первую очередь компоненты Java) выступят в роли базовых элементов для различных операционных систем.

Но для серьезного соперничества с .NET потребуется наличие стандартизированных пакетов компонентов (одной библиотеки Swing мало). Более того, необходимо, чтобы все эти пакеты входили в поставку JDK. Наличие разрозненных нестандартизированных пакетов не даст преимуществ перед конкурирующей технологией .NET.

Одним из решений этой проблемы стало расширение базового набора пакетов и классов в составе JDK. Даже самый старый из имеющихся (пакет `java`) в новых версиях JDK усовершенствуется, не говоря уже о более новом пакете `javax`. Кроме того, в дополнительных модулях NetBeans (Web & Java EE, UML Pack и др.) есть большое число дополнительных библиотечных пакетов (packages).

Еще одно важное направление конкуренции — сектор так называемого корпоративного программного обеспечения (Enterprise Software), т. е. программного обеспечения, предназначенного для корпоративных сетей. Это, в первую очередь, серверное программное обеспечение. Обычные пользователи, работающие на персональном компьютере, с такими программами практически не сталкиваются. А вот в крупных фирмах (не только в западных, но и во многих российских) такое программное обеспечение — основа документооборота и любой деятельности. По оценкам экспертов рынок корпоративного программного обеспечения по деньгам во много раз превышает рынок программ для "персоналок". И на этом рынке в настоящее время безоговорочно лидирует Java, в пер-

вую очередь благодаря надежности, гибкости и развиваемости приложений Java. В *главе 11* будет вкратце рассказано о некоторых серверных технологиях Java.

Естественно, в данной области наблюдается жесткая конкуренция технологий.

До сих пор имеется некоторое количество серверных программ, написанных на языках C/C++. Они иногда дают более высокое быстродействие, чем Java, но не обеспечивают надежность, развиваемость, скорость разработки и документируемость больших программных продуктов. Программы C/C++ могут подключаться к проектам Java в виде так называемых native (родных) методов в классах Java *(см. главу 16)* и "хранимых процедур" серверных приложений. Поэтому самые критичные к быстродействию участки кода могут быть выполнены на C/C++, что объединяет преимущества Java и C/C++.

Другая конкурирующая серверная технология — ASP (Active Server Pages) от Microsoft. Во многом она схожа с Java-технологией JSP, о которой рассказывается в *главе 11*, но работает только на платформе Windows, что сильно ограничивает ее применение.

Быстро развивающаяся и ставшая весьма популярной технология PHP во многом похожа на ASP и JSP, но в подавляющем большинстве случаев является чисто процедурной, в то время как ASP и JSP основаны на объектном подходе. При этом PHP развивается в рамках идеологии Open Source и бесплатна, как и JSP. Исполняющая среда PHP5 обеспечивает высокое быстродействие серверных приложений, а разработка этих приложений для не очень больших проектов оказывается дешевле. В то же время у PHP-приложений имеются проблемы с надежностью и развиваемостью, не говоря уж о документировании. Поэтому PHP-проекты обычно остаются уделом относительно небольших компаний и университетов. Тем не менее, число потребителей этой технологии достаточно велико. Поэтому радует, что NetBeans 6.5 обеспечивает развитую поддержку PHP-программирования.

Таким образом, несмотря на то, что платформа Java лидирует во многих областях современной программной индустрии, соревнование продолжается, что идет на пользу качеству решений и приводит к необходимости постоянно совершенствовать как язык Java, так и все связанные с ним технологии.

1.11. Среды разработки NetBeans, Eclipse, JDeveloper, JBuilder, IntelliJ IDEA

В настоящее время имеется несколько широко известных интегрированных сред разработки (Integrated Development Environment — IDE) для программирования на языке Java (табл. 1.10).

Таблица 1.10. *Интегрированные среды разработки для Java*

Среда	Разработчики	Краткое описание
NetBeans 6.5	Sun, netbeans.org	Бесплатная среда с открытым исходным кодом. Может быть свободно загружена с сайта **http://www.netbeans.org/**. Размер дистрибутива 248 Мбайт. По мнению автора на момент выхода данной книги является наиболее развитой из бесплатных сред разработки для Java

Таблица 1.10 *(окончание)*

Среда	Разработчики	Краткое описание
JDeveloper 11	Oracle	Бесплатная среда, ориентированная на работу с базами данных Oracle на основе технологий Java EE, но может использоваться как универсальная среда разработки для Java. Может быть свободно загружена с сайта **http://oss.oracle.com/jdeveloper.html**. Размер дистрибутива 656 Мбайт
Eclipse 3.4.1	IBM, Eclipse Foundation	Бесплатная среда с открытым исходным кодом. Может быть свободно загружена с сайта **http://www.eclipse.org/**. Размер дистрибутивов 86 Мбайт (комплект "Eclipse IDE for Java Developers") и 163 Мбайт (комплект "Eclipse IDE for Java EE Developers"). Является основой (платформой) для многих коммерческих сред разработки
JBuilder 2008	Borland, CodeGear	Имеется три редакции — бесплатная (JBuilder 2008 Turbo) и две коммерческие (JBuilder 2008 Professional и JBuilder 2008 Enterprise). Бесплатная редакция и пробные версии коммерческих могут быть свободно загружены с сайта **http://www.codegear.com/**. Размер дистрибутива 1,8 Гбайт, он идентичен для всех редакций
IntelliJ IDEA 7.0.4	JetBrain	Коммерческая среда. Очень популярна в западных странах. Пробная версия может быть свободно загружена с сайта **http://www.jetbrains.com/idea/**. Размер дистрибутива 88 Мбайт

Перечислим ряд черт, характерных для всех приведенных сред. Начинающие читатели встретят в этом описании много непонятных терминов и названий технологий, но не следует пугаться — о большинстве из них будет рассказано в последующих главах этой книги. Итак, укажем наиболее важные черты современных интегрированных сред разработки программного обеспечения на языке Java.

- Развитый редактор исходного кода, включающий не только средства редактирования, но и *синтаксический анализатор* исходного кода, обеспечивающий подсветку проблемных мест *(см. главу 2)*.
- В каждой IDE имеется *отладчик*, позволяющий проводить пошаговую проверку выполнения алгоритмов *(см. главу 17)*.
- Во всех средах разработки обеспечивается поддержка *рефакторинга* — изменения структуры проекта без изменения его функциональности (переименования элементов, перемещение их из одной области программного кода в другую и т. п. — *см. главу 7*).
- Поддержка интеграции с различными *системами контроля версий* (Vesrion Control Systems — VCS, или, что то же самое, Revision Control Systems, Source Control Systems или Source Code Management — SCM). Системы контроля версий обеспечивают документирование любых изменений в проекте и, при необходимости, откат назад на нужное число шагов во всем проекте или его части. Кроме того, в настоящее время они стали базовыми средствами для профессиональной групповой работы над проектами.

- В наиболее развитых средах разработки (NetBeans, JDeveloper, IntelliJ IDEA) имеется встроенный *профилировщик*, позволяющий проводить оптимизацию приложения по потребляемым ресурсам и скорости выполнения *(см. главу 17)*.

Рассмотрим теперь каждый из перечисленных продуктов более подробно, причем только базовую поставку — без дополнительных программных средств от сторонних производителей. Это, конечно, заметно понижает в данном сравнении рейтинг ряда сред, и, в особенности, Eclipse, т. к. эта среда славится именно возможностями подключения средств от сторонних производителей. В то же время, как считает автор, если бы появился готовый комплект таких средств, обеспечивающий нужную степень интеграции друг с другом и устойчивую работу на платформе Eclipse, его можно было бы рассматривать как самостоятельную среду разработки. Как, например, JBuilder в данном обзоре. А конструктор "сделай сам" может претендовать только на роль платформы для разработки конечного изделия, но не на роль самого изделия. Поэтому для всех сред учитываются только те средства, которые с ними поставляются или могут быть загружены через Интернет из стандартного централизованного *репозитория* (специализированного хранилища) среды, заданного в ней по умолчанию.

NetBeans 6.5

NetBeans ("сетевые компоненты") — бесплатная среда с открытым исходным кодом. Очень быстро развивается. Разрешено использование в коммерческих целях.

Обладает развитыми средствами для добавления в среду новых возможностей с помощью специальных дополнений к среде — плагинов. Имеется централизованный сайт репозитория, в котором содержатся надежные плагины, рекомендуемые и проверенные разработчиками NetBeans, а также можно назначать свои адреса репозиториев. Поэтому установка и удаление стандартных плагинов в NetBeans — чисто техническая работа, не требующая поисков, консультаций и исследований.

Среда отличается самым развитым из имеющихся редактором форм графического пользовательского интерфейса (GUI) и наиболее развитыми средствами разработки программного обеспечения для мобильных устройств (Java ME, *см. главу 14*).

Содержит наиболее полный из бесплатных сред комплект дополнительных средств разработки приложений (генерация различных типов проектов на основе набора шаблонов, генерация проектов на основе большого числа имеющихся примеров, генерация заготовок документационных комментариев — *см. главу 2*, переход в исходные коды JDK при навигации по классам проекта).

Имеет развитые средства развертывания (deploy) проектов — установки их с рабочего компьютера на заданный, в том числе на сервер.

Допускает работу с языками Java, C,C++, Ruby, Groovy, JavaScript, JavaFX, JRuby, Rails, Grails, HTML, CSS, XML, XHTML, технологиями JSP, EJB, AJAX, Hibernate, JPA, Spring, Web Services, Struts.

Поддерживает тестирование с помощью модулей JUnit *(см. главу 17)*. Имеет развитые средства UML-проектирования. Поддерживает импорт проектов Eclipse и (при установке из репозитория соответствующего плагина) JBuilder.

Недостатки. Как и все IDE, перечисленные далее, потребляет относительно много ресурсов (285 Мбайт оперативной памяти — сама среда, 315 Мбайт при работе простей-

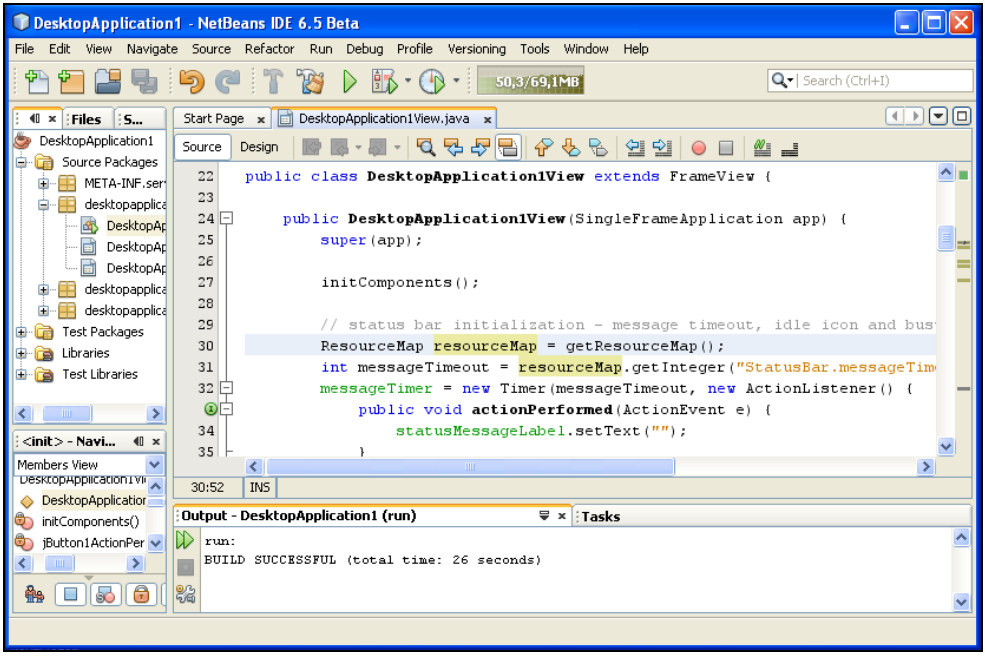


Рис. 1.7. Редактирование исходного кода в среде NetBeans

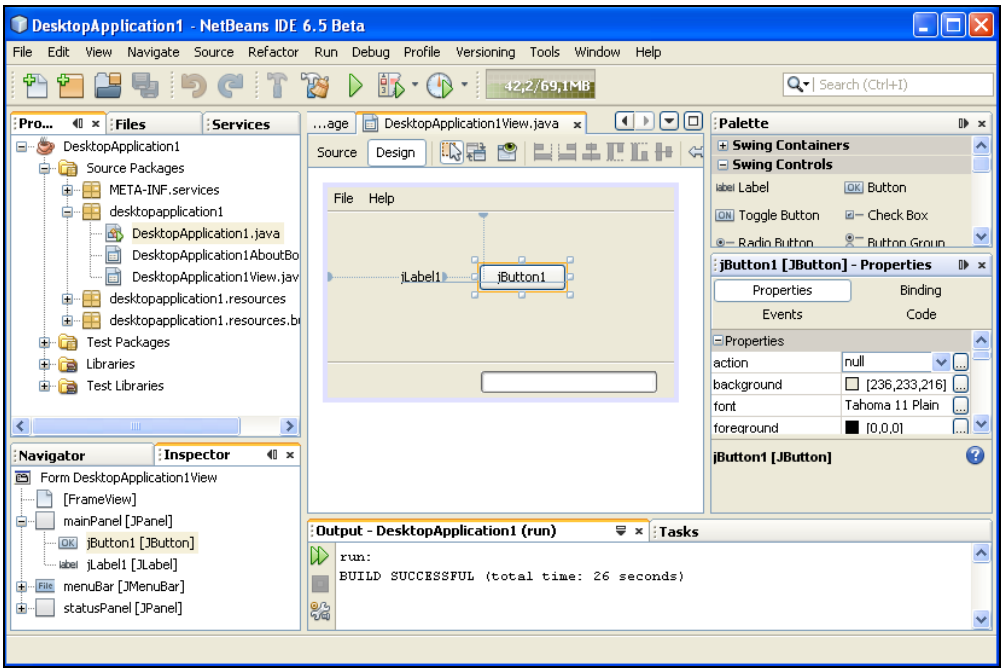


Рис. 1.8. Редактирование формы GUI в среде NetBeans — дизайнер форм

шего приложения с графическим интерфейсом). Поэтому предъявляет повышенные требования к объему оперативной памяти компьютера (не менее 512 Мбайт) и быстродействию процессора (не ниже Celeron 1000). Не поддерживает импорт проектов IntelliJ Idea и JBuilder. Имеет не очень удобную справочную систему.

На рис. 1.7 приведен внешний вид среды NetBeans при редактировании исходного кода приложения DesktopApplication (*см. главы 2* и *13*). На рис. 1.8 приведен вид экрана дизайнера при редактировании в среде NetBeans экранной формы этого приложения.

Eclipse 3.4.1 (Ganimede)

Eclipse ("затмение") — бесплатная среда с открытым исходным кодом. В последнее время многими рассматривалась как главный конкурент NetBeans на рынке свободных продуктов, причем опережающий NetBeans. Но это было до выхода NetBeans 6.5 и JDeveloper 11, так что теперь ситуация, вероятно, изменится. Среда очень быстро развивается. Разрешено ее применение в коммерческих целях. Широко используется для программирования на Java.

Первой среди сред разработки Java обеспечила развитые средства для добавления в среду новых возможностей с помощью плагинов. Благодаря этому стала основой (платформой) для многих коммерческих сред разработки — MyEclipse, BEA Workshop, IBM Rational Application Developer, Borland JBuilder (в настоящее время — CodeGear JBuilder) и многих других. Существует несколько сайтов репозиториев по умолчанию, в которых содержится большое количество плагинов, а также можно назначать свои адреса репозиториев. Имеется разделение плагинов по категориям в зависимости от области применения. Однако отсутствует сертификация плагинов и подробное описание их назначения и функциональности. Поэтому установка и удаление плагинов в Eclipse является увлекательной исследовательской работой с непредсказуемым результатом. Работа с плагинами настраивается через пункт меню **Software Updates…** (Обновления программного обеспечения), находящийся … в пункте меню **Help**. Если вы не знаете об этой особенности Eclipse, то вряд ли найдете! Следует отметить, что эту странную черту Eclipse унаследовали некоторые другие среды, построенные на ее основе, либо пытающиеся повторить ее функциональность — JDeveloper, JBuilder.

Среда Eclipse является промышленным стандартом де-факто для разработки программ на языке C++ (хотя в NetBeans 6.5 появилась мощная поддержка средств разработки на языке C++, так что, возможно, в ближайшее время в этой области ситуация изменится — *см. главу 16*).

Eclipse Ganimede в отличие от старых версий не требует инсталляции на компьютере — запускается непосредственно из набора файлов дистрибутива. Имеет удобный отладчик и встроенный органайзер. При редактировании исходного кода при навигации по классам проекта не обеспечивает переход в исходный код классов JDK, но позволяет легко настроить путь к архиву src.zip из JDK, после чего навигация работает. Подобным же образом можно настроить переход в исходные коды любых используемых классов. Имеет удобное переключение "перспектив" в правой верхней части окна — наборов показываемых на экране окон (один набор служит для редактирования исходного кода, другой — для отладки, и т. д.).

Отличается интересной, но несколько раздражающей особенностью редактора исходного кода — при наведении мыши на имя элемента сразу показывается всплывающая подсказка по этому элементу.

Поддерживает работу с языками Java, C, C++, Ruby, Groovy, JavaScript, JRuby, Rails, Grails, Flex, XSL, HTML, CSS, XML, XHTML, технологиями JSP, EJB, AJAX, GWT, Hibernate, JPA, Spring, Web Services, Struts.

Поддерживает тестирование с помощью модулей JUnit и импорт проектов JBuilder.

Недостатки. Потребляет относительно много ресурсов (260 Мбайт оперативной памяти — сама среда, 335 Мбайт при работе простейшего приложения с графическим интерфейсом) — примерно как NetBeans. Практически отсутствуют шаблоны и примеры для создания заготовок проектов. Нет встроенного редактора форм GUI, его нужно устанавливать как плагин (к тому же имеющиеся редакторы GUI заметно уступают редактору NetBeans). Отсутствуют средства UML-проектирования — нужно устанавливать плагин. Часто встречается несовместимость плагинов от разных производителей, вызывающая неустойчивость среды (хотя с каждым по отдельности Eclipse работает). Не реализована встроенная поддержка работы с мобильными устройствами. Отсутствует встроенный профилировщик, что не позволяет рассматривать среду в базовом варианте поставки в качестве полноценного средства разработки. Не умеет самостоятельно опознавать установленные на компьютере JDK. Не поддерживает импорт проектов NetBeans, IntelliJ Idea.

На рис. 1.9 показан экран Eclipse в режиме редактирования исходного кода.

Вывод: Eclipse очень мощная платформа для создания средств разработки программного обеспечения, но в имеющихся вариантах поставки не может рассматриваться как конечный продукт, конкурентоспособный с NetBeans и JDeveloper. Для создания конфигурации, пригодной для реальной Java-разработки, требуется поиск и установка многочисленных плагинов.

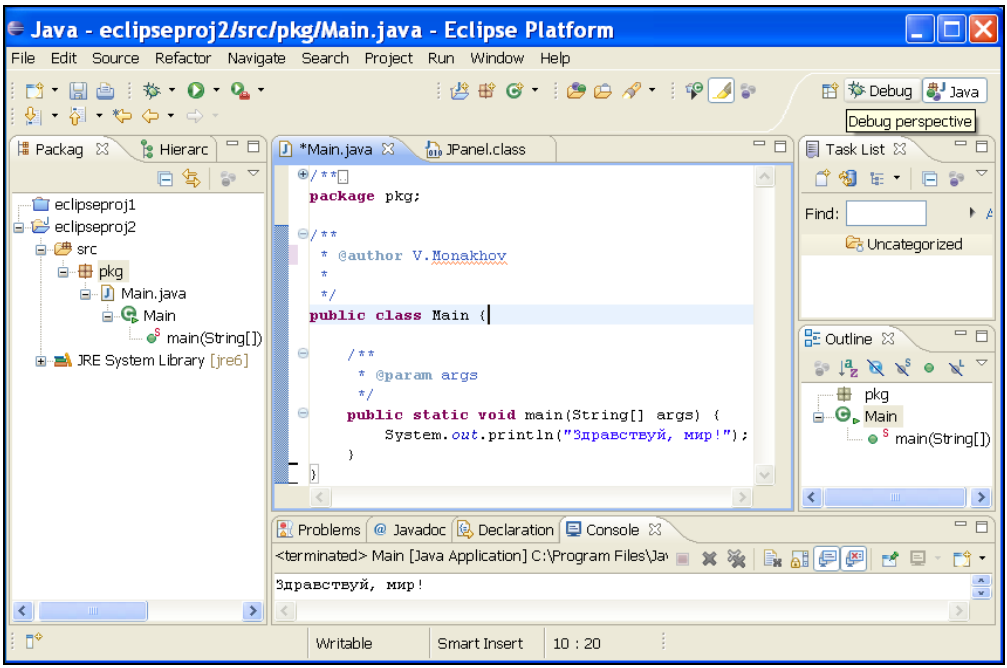


Рис. 1.9. Редактирование экранной формы в среде Eclipse

JDeveloper 11

JDeveloper ("Java-разработчик") — бесплатная среда, ориентированная на работу с базами данных Oracle на основе технологий Java EE. Стала популярной в основном благодаря высокой степени интеграции с программными серверными системами Oracle, но вполне пригодна в качестве универсальной среды разработки программного обеспечения Java.

Имеет полноценные средства визуального проектирования *графического пользовательского интерфейса* (GUI), качественный отладчик и встроенный профилировщик. Обладает специализированными средствами, облегчающими работу с базами данных.

Включает развитые средства развертывания (deploy) проектов, позволяющие устанавливать их с рабочего компьютера на заданный, в том числе на сервер. Имеет очень удобную и понятную справочную систему (на английском языке). Обеспечивает переход в исходные коды JDK при навигации по классам проекта, но только после указания в первый раз пути к архиву с исходными файлами JDK (файлу src.zip в папке JDK). Подобным же образом можно настроить переход в исходные коды любых используемых классов.

Поддерживает тестирование с помощью модулей JUnit. Имеет развитые средства UML-проектирования.

Недостатки. Потребляет много ресурсов, несколько больше, чем NetBeans (360 Мбайт оперативной памяти — сама среда, 410 Мбайт при работе простейшего приложения с графическим интерфейсом). Небольшое по сравнению с NetBeans количество заготовок для создаваемых приложений. Отсутствуют примеры создаваемых приложений. Среда не умеет самостоятельно опознавать установленные на компьютере JDK. Нельзя устанавливать программу в папку или запускать инсталлятор из папки, в имени которой содержатся пробелы (например, "Program Files") или символы кириллицы. Файловый диалог инсталлятора очень долго анализирует список файлов папки, в которую заходит пользователь, из-за чего продолжительное время в ней показывается список файлов из старой папки, вводя пользователя в заблуждение. Установка JDeveloper невозможна без установки сервера приложений WebLogic, входящего в состав дистрибутива. Нельзя изменить конфигурацию установки (например, добавить новый путь к JDK) без деинсталляции хотя бы одного элемента комплекта. На ноутбуке автора с операционной системой Windows XP SP2 после установки пакета не удалось запустить JDeveloper (выдавалась ошибка создания экземпляра java-машины), хотя остальные программы запускались. И только после установки пакета XP SP3 он заработал (также требуется достаточно новая версия JDK: с JDK 6 update3 инсталлятор сообщал о необходимости установить более новую версию). Нет поддержки Java ME. При создании новых элементов проекта (экранной формы, класса, UML-диаграммы и ее элементов, и т. п.) не работает откат назад (Undo). Не устанавливаются некоторые менеджеры размещения (например, XYLayout — *см. разд. 2.15*). Связь созданных экранных форм с приложением приходится делать вручную.

Несмотря на большое число перечисленных недостатков, они носят второстепенный характер и не снижают очень хорошего впечатления от продукта.

На рис. 1.10 приведен вид окна, появляющегося при запуске JDeveloper и позволяющего выбирать режим работы (поддержка всех технологий, работа только с базами дан-

ных, работа с технологиями Java EE, работа только с базовыми средствами разработки на языке Java).

На рис. 1.11 показан экран JDeveloper в режиме редактирования исходного кода, а на рис. 1.12 — в режиме редактирования экранной формы GUI.

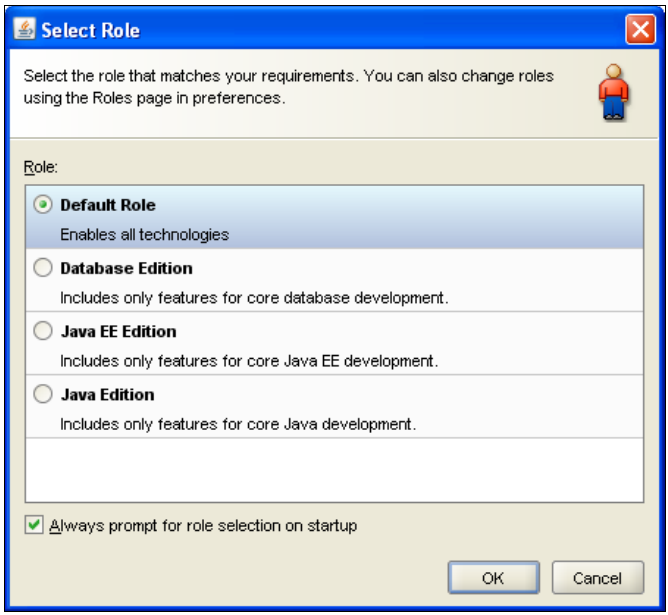


Рис. 1.10. Запуск JDeveloper

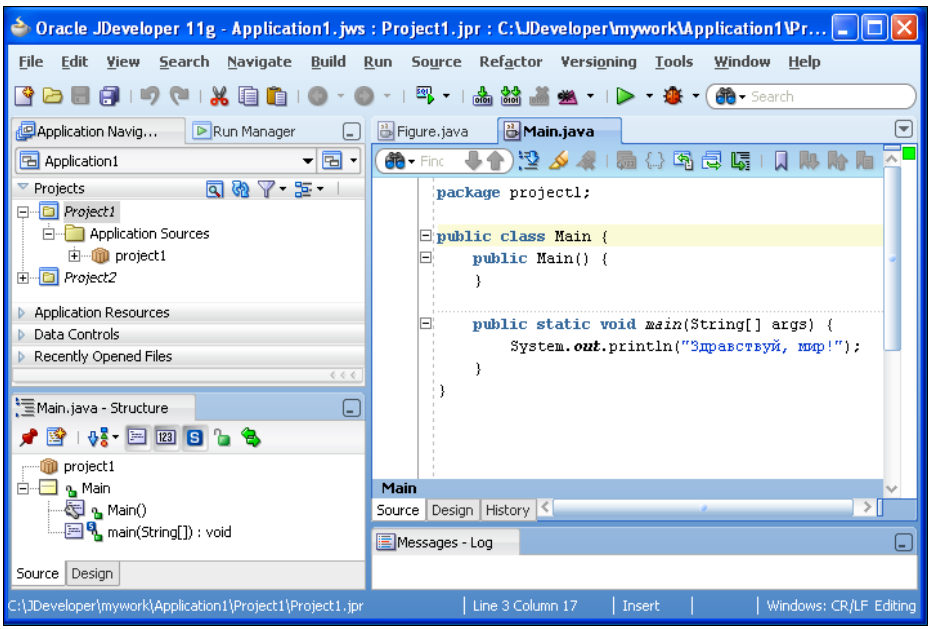


Рис. 1.11. Работа в среде JDeveloper

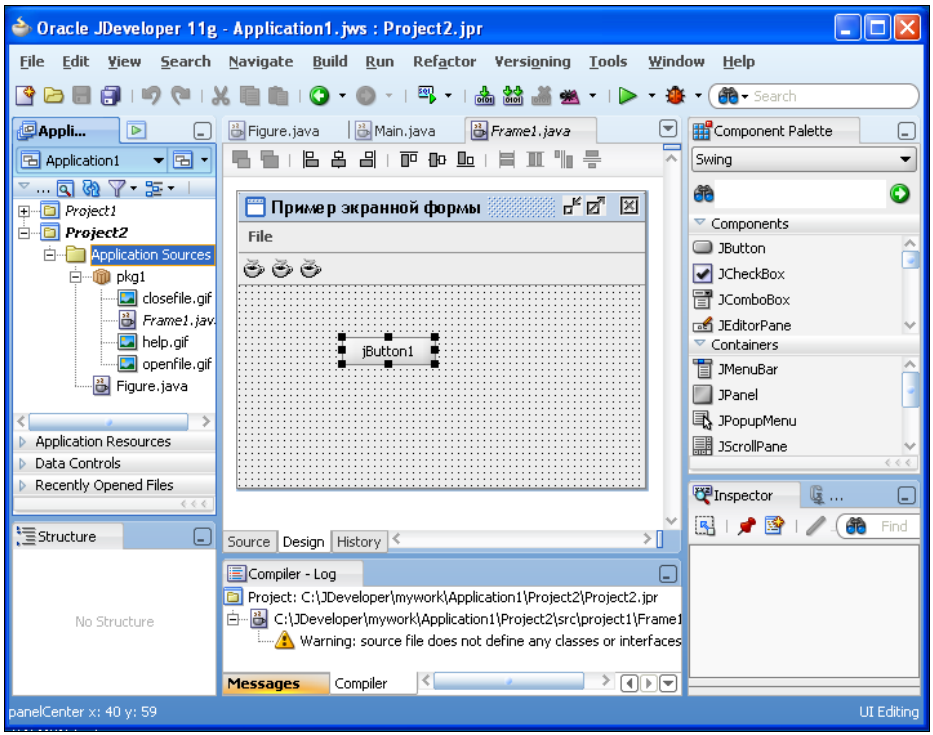


Рис. 1.12. Редактирование экранной формы в среде JDeveloper

Несмотря на некоторые недоработки, дизайнер экранных форм, редактор исходного кода, средства UML-проектирования, а также отладчик и профилировщик JDeveloper весьма развиты, а проблемы с инсталляцией и запуском, вероятно, в скором времени будут решены. В общем и целом, несмотря на ряд проблем, корпорации Oracle удалось вывести свой продукт на очень высокий уровень, по многим параметрам не уступающий среде NetBeans, а кое в чем и превосходящий.

Вывод: JDeveloper в ближайшее время сможет серьезно конкурировать с NetBeans и другими средами в области разработки программного обеспечения Java на платформах Java SE и Java EE. В области Java ME эта среда пока заметно проигрывает среде NetBeans.

JBuilder 2008

Существуют три редакции — бесплатная (JBuilder 2008 Turbo) и две коммерческие (JBuilder 2008 Professional и JBuilder 2008 Enterprise). Для всех трех имеется один и тот же дистрибутив. Отличие заключается в регистрационном ключе, получаемом после регистрации (бесплатно для бесплатной версии или за соответствующую плату для коммерческих). Тип лицензии может быть в любой момент изменен. Среда основана на платформе Eclipse и обладает характерным для Eclipse интерфейсом. Она обеспечивает все базовые возможности Eclipse, а также ряд дополнительных. В частности, поддержку дополнительных вариантов рефакторинга, аспектно-ориентированного программирования (одного из перспективных направлений современного программирования)

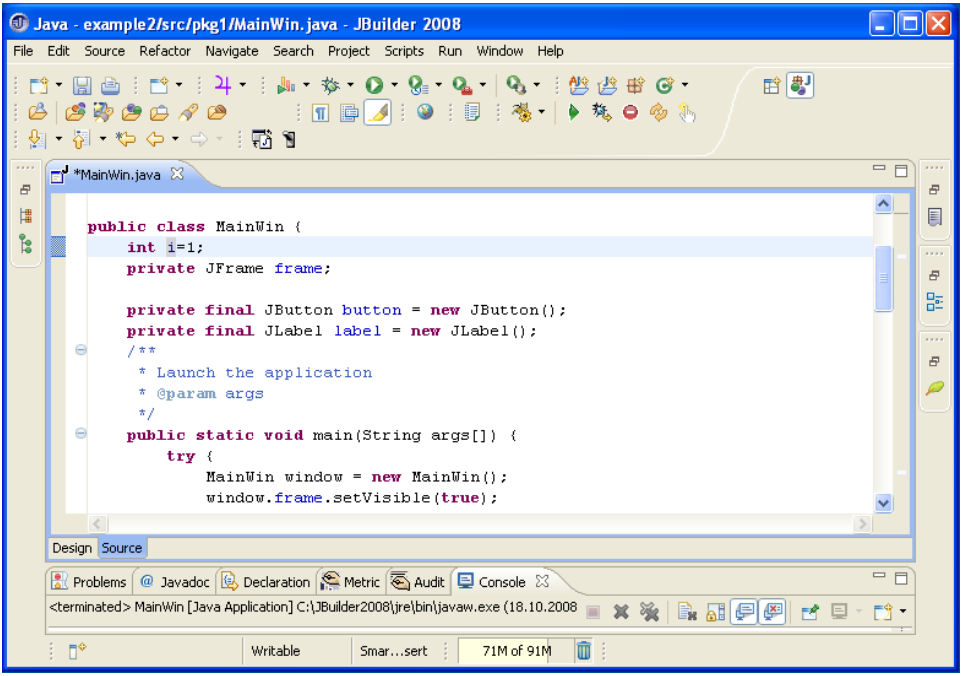


Рис. 1.13. Работа в среде JBuilder

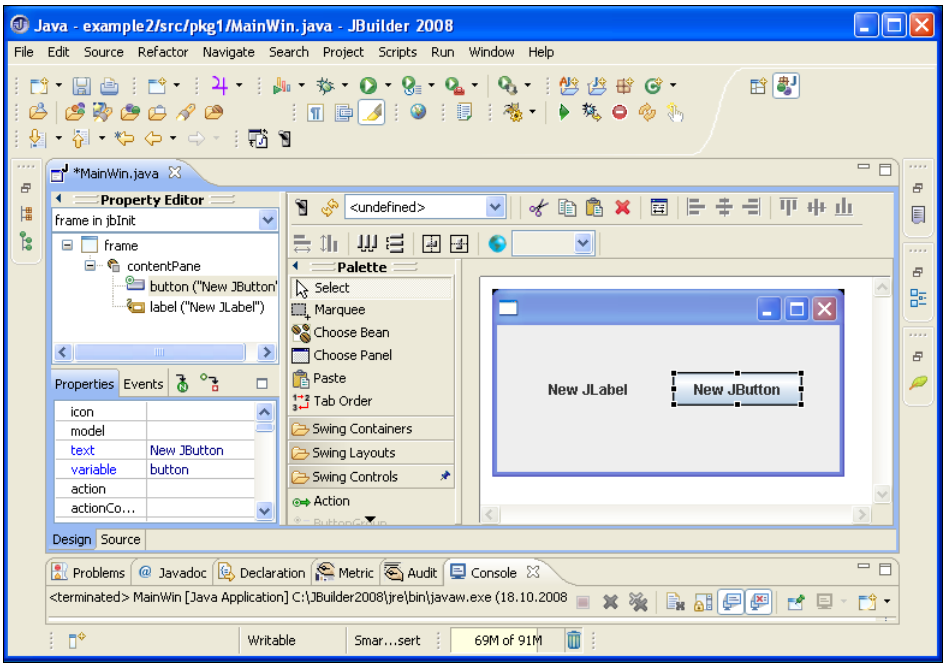


Рис. 1.14. Редактирование экранной формы в среде JBuilder

с помощью языка AspectJ — расширения языка Java, визуальное проектирование экранных форм пользовательского интерфейса GUI (в коммерческих версиях), UML-проектирование (в коммерческих версиях).

Недостатки. Потребляет много ресурсов, больше, чем NetBeans (350 Мбайт оперативной памяти — сама среда, при работе простейшего приложения с графическим интерфейсом вместе с приложением 450 Мбайт). Очень большой объем дистрибутива (1,8 Гбайт). Нельзя устанавливать в папку, в имени которой содержатся пробелы (в частности, в папку "Program Files"). Бесплатная версия не позволяет редактировать формы GUI. Недостаточно развитая система ведения проектов: не создается jar-архив с выходными файлами, не предусмотрена отдельная папка для создания документации по проекту. Возникают проблемы с кодировкой при генерации документации для проектов с документационными комментариями, написанными на русском языке. Не поддерживает импорт проектов NetBeans, IntelliJ Idea.

На рис. 1.13 показан экран JBuilder в режиме редактирования исходного кода, а на рис. 1.14 — в режиме редактирования экранной формы GUI.

Вывод: платная среда JBuilder не обеспечивает преимуществ по сравнению со своим бесплатным конкурентом NetBeans, а по ряду параметров уступает ему.

IntelliJ IDEA 7.0.4

Коммерческая среда. Имеет развитый редактор исходного кода, отладчик, профилировщик, обеспечивает поддержку разработки программного обеспечения для мобильных устройств (Java ME). Являлась первой из сред, которая стала обеспечивать поддержку рефакторинга в проектах Java, благодаря чему приобрела популярность среди профессиональных Java-разработчиков. Помимо синтаксического анализатора, встроенного в редактор исходного кода, содержит мощный анализатор качества кода с более чем 600 видами проверки, позволяющий обнаружить проблемные места в программе (проблемы абстракции, присваивания, финализации, многопоточности, файловых потоков, портирования, областей видимости, структуры классов, стиля кодирования, обработки исключительных ситуаций, повторения фрагментов кода и неиспользуемые коды). Обеспечивает интеграцию документации по проекту, созданной на основе документационных комментариев, с внешней документацией. Обеспечивает поддержку языков Java, Ruby, Groovy, JavaScript, JRuby, Rails, Grails, Flex, XSL, HTML, CSS, XML, XHTML, технологий JSP, EJB, AJAX, GWT, Hibernate, JPA, Spring, Web Services, Struts. Поддерживает интеграцию с различными системами контроля версий. Поддерживает импорт проектов JBuilder и Eclipse. Имеет развитые средства для добавления в среду новых возможностей с помощью плагинов.

Недостатки. Потребляет много ресурсов, больше, чем NetBeans (300 Мбайт оперативной памяти — сама среда, при работе простейшего приложения с графическим интерфейсом вместе с приложением 450 Мбайт). Обладает достаточно бедными возможностями редактора — отсутствие показа документационных комментариев в "умной подсказке" выводит данную среду из группы лидеров. Не позволяет создавать обработчики событий "одним щелчком мыши", как другие среды. Эту операцию приходится от начала до конца делать вручную, т. к. в Idea автоматическая связь форм с исходным кодом отсутствует. Поэтому средства визуального проектирования пользовательского интерфейса Idea следует считать недостаточно развитыми. Отсутствует централизован-

ный сайт репозитория, в котором содержатся надежные плагины, рекомендуемые и проверенные разработчиками IntelliJ IDEA, отсутствует группировка плагинов по сайтам и по категориям. Нет встроенного профилировщика, что не позволяет рассматривать среду в базовом варианте поставки в качестве полноценного средства разработки. Отсутствуют средства UML-проектирования. Не умеет самостоятельно опознавать установленные на компьютере JDK. Недостаточно развита система ведения проектов: не создается jar-архив с выходными файлами, не предусмотрена отдельная папка для создания документации по проекту. Не поддерживает импорт проектов NetBeans.

Вывод: в настоящее время платная среда IntelliJ IDEA не может рассматриваться как лидер в области сред разработки программного обеспечения Java, и по большинству параметров заметно уступает своему бесплатному конкуренту — NetBeans. Единственное серьезное преимущество IntelliJ IDEA — мощный анализатор кода.

Общий итог сравнения сред разработки

Бесплатная среда NetBeans 6.5 может рассматриваться как лидер в области сред разработки программного обеспечения Java. Она начала опережать по большинству параметров не только бесплатную среду Eclipse, являвшуюся до последнего времени лидером среди свободно распространяемых IDE, но и коммерческие среды IntelliJ IDEA и JBuilder. Существенно расширились возможности бесплатной среды JDeveloper, ее версия 11 уже может рассматриваться как один из претендентов на лидерство, особенно в области разработки приложений Java EE. Основным преимуществом IntelliJ IDEA перед NetBeans 6.5 и другими средами до сих пор остается мощный анализатор кода.

Краткие итоги по главе

- Алфавит языка Java состоит из букв, десятичных цифр и специальных символов. Буквами считаются латинские буквы (кодируются в стандарте ASCII), буквы национальных алфавитов (кодируются в стандарте Unicode), а также соответствующие им символы, кодируемые управляющими последовательностями.
- В программах разрешается пользоваться десятичными и шестнадцатеричными целыми числовыми константами. Шестнадцатеричная константа начинается с символов `0x` или `0X`, после чего идет само число в шестнадцатеричной нотации.
- Java — универсальный язык прикладного программирования, JavaScript — узкоспециализированный язык программирования HTML-документов, C++ — универсальный язык системного программирования. Java — компилируемый, платформо-независимый, объектно-ориентированный язык с С-образным синтаксисом.
- Программы Java переносимы на уровне как исходных, так и скомпилированных исполняемых кодов (байт-кода). Байт-код платформо-независимый, т. к. не содержит инструкций процессора конкретного компьютера и интерпретируется виртуальной Java-машиной (JVM).
- JIT-компиляция (Just-In-Time) — компиляция байт-кода в код конкретной платформы в момент выполнения программы, т. е. "по ходу дела", "на лету". Она позволяет ускорить работу программ за счет замены интерпретации байт-кода на выполнение скомпилированного кода.

- Основные категории программ Java:
  - приложение (application) — аналог обычной прикладной программы;
  - апплет (applet) — специализированная программа с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера;
  - мидлет (midlet) — специализированная программа с ограниченными возможностями, работающая на мобильном устройстве;
  - сервлет (servlet) — специализированная программа с ограниченными возможностями, работающая в WWW на стороне сервера;
  - серверное приложение (Enterprise application) — предназначено для многократного выполнения на стороне сервера;
  - библиотека (Java Class Library — библиотека классов, либо NetBeans Module — модуль платформы NetBeans) — предназначена для многократного использования программами классов Java.
- Версии Java 1.0 и 1.1 принято называть Java 1. Все версии Java, начиная с 1.2, принято называть Java 2.
- Поставить на компьютер исполняющую среду Java (виртуальную Java-машину) можно путем установки SDK (Software Development Kit) — комплекта разработки программного обеспечения. Имеются три типа SDK:
  - Java ME — комплект Java Micro Edition, предназначенный для программирования "тонких аппаратных клиентов";
  - Java SE — комплект Java Standard Edition, предназначенный для программирования обычных компьютеров;
  - Java EE — комплект Java Enterprise Edition, предназначенный для написания серверного программного обеспечения.
- Язык Java регистро-чувствительный. Исходные коды программ Java набираются в виде последовательности символов Unicode.
- Управляющая последовательность применяется в случае, когда требуется использовать символ, который обычным образом в текст программы ввести нельзя. Простая управляющая последовательность начинается с символа "\", после которого идет управляющий символ. Управляющая последовательность для кодирования символа Unicode начинается с последовательности из двух символов "\u", после которой следуют четыре цифры номера символа в шестнадцатеричной нотации, например `\u1234`.
- Специальные символы служат в качестве операторов и разделителей языка Java и не могут входить в состав идентификаторов. Специальные символы бывают простые и составные. Они используются в операторах для форматирования текста и как разделители.
- Идентификаторы — это имена переменных, процедур, функций и т. д. В идентификаторах можно применять только буквы и цифры, причем первой всегда должна быть буква, а далее может идти произвольная комбинация букв и цифр. Длина идентификатора в Java не ограничена.

- Переменная — это именованная ячейка памяти, содержимое которой может изменяться. При объявлении переменной сначала указывается тип переменной, а затем идентификатор задаваемой переменной.
- Типы в Java делятся на примитивные и ссылочные. Существует несколько предопределенных примитивных типов, все остальные — ссылочные. Все пользовательские типы (кроме типов-перечислений) являются ссылочными. Значение `null` соответствует ссылочной переменной, которой не назначен адрес ячейки с данными.
- Существуют три базовых принципа объектно-ориентированного программирования: *инкапсуляция*, *наследование*, *полиморфизм*.
- *Класс* — это описание того, как устроен объект. И поля данных, и методы задаются в классах. Но при выполнении программы поля данных хранятся в объектах, а методы — в классах. Методы задают *поведение*, а поля данных — *состояние* объекта.
- Переменные, описываемые в классах, называются *глобальными*. Они задают поля данных объектов. Переменные, описываемые в методах, называются *локальными*. Они являются вспомогательными и существуют только во время вызова метода.
- Переменные ссылочного типа содержат адреса данных, а не сами данные. Поэтому присваивания для таких переменных меняют адреса, но не данные. Все объектные типы являются ссылочными. Потеря ссылки на объект приводит к сборке мусора.
- Объект создается с помощью вызова *конструктора* — специальной подпрограммы-функции, задаваемой в классе.
- Методы делятся на *методы объектов* и *методы классов*. Метод объекта можно вызывать только из объекта соответствующего типа. А метод класса может работать и при отсутствии объекта и вызываться из класса.
- При декларации класса можно указывать, что он общедоступен, с помощью *модификатора доступа* `public`. В этом случае возможен доступ к данному классу из других пакетов. В файле java можно располагать только один общедоступный класс и произвольное число классов с другим уровнем видимости. Если модификатор `public` отсутствует, то доступ к классу разрешен только из классов, находящихся с ним в одном пакете. Про такие файлы говорят, что у них *пакетный* вариант доступа.
- Важнейшие пакеты — `java` и `javax`, а также вложенные в них пакеты. Информацию о содержащихся в них элементах можно получить в среде разработки, набрав `java.` или `javax.` и прочитав появившуюся подсказку.
- Для того чтобы при обращении к классу не указывать полный путь с именем пакета, можно *импортировать* класс `import pkg1.pkg2.Имя;` или весь пакет `import pkg1.pkg2.*;`.
- Для *статического импорта* из пакета классов, переменных класса и методов класса можно импортировать как отдельные переменные или методы класса:

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
```

  так и все члены класса:

```
import static java.lang.Math.*;
```

- Все классы и объекты приложения вызываются и управляются из метода `main`, который имеет сигнатуру `public static void main(String[] args)`. Так как это метод класса, для его работы нет необходимости в создании объекта, являющегося экземпляром класса.
- Бесплатная среда NetBeans 6.5 может рассматриваться как лидер в области сред разработки программного обеспечения Java , начинающий опережать по большинству параметров не только Eclipse (лидировавшую до последнего времени среди бесплатных сред), но и коммерческие среды IntelliJ IDEA и JBuilder.
- Существенно расширились возможности бесплатной среды JDeveloper, ее версия 11 уже может рассматриваться как один из претендентов на лидерство, особенно в области разработки приложений Java EE.
- Основным преимуществом IntelliJ IDEA перед NetBeans 6.5 и другими средами до сих пор остается мощный анализатор кода.

Типичные ошибки

- Путают языки Java и JavaScript, либо считают, что JavaScript — это интерпретируемый вариант Java. Хотя эти языки не имеют друг к другу никакого отношения. Возможна трансляция команд Java в конструкции языка JavaScript — технология AJAX (Asynchronous JavaScript and XML — асинхронный JavaScript и XML). Но это выходит за предмет обсуждения данной книги.
- Ошибочно считают, что приложение Java может быть запущено на любом компьютере без установки исполняющей среды (JRE).
- Не различают приложения (applications) и апплеты (applets).
- При записи шестнадцатеричного числа вида `0x...` вместо ведущего нуля пишут букву `O`.
- Ошибочно считают, что в идентификаторах Java нельзя использовать символы национальных алфавитов.
- Ошибочно считают, что не имеет значения, в каком регистре набраны символы идентификатора (характерно для тех, кто раньше программировал на Pascal или Fortran).
- Очень часто встречающаяся ошибка: путают классы, объекты и объектные переменные. Но это совершенно разные сущности. Класс — это тип, т. е. описание того, как устроена ячейка памяти, в которой будут располагаться поля данных объекта, и какие методы можно вызывать. Объект — это содержимое ячейки памяти объектного типа. А в переменной объектного типа содержится адрес объекта.
- Ошибочно считают, что методы хранятся в объектах.
- Пытаются обратиться к объекту с помощью ссылочной переменной, которой назначено значение `null`.
- Непреднамеренная потеря связи с объектом. Обычно возникает при изменении ссылки на объект внутри подпрограммы.

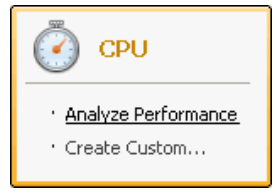
Задания

- Написать в 16-ричном виде числа 0, 1, 8, 15, 16, 255, 256.
- Дать ответ, допустимы ли следующие идентификаторы:

  `i1, i_1, 1i, i&1, i1234567891011, IJKLMN.`

- Указать, какие из перечисленных идентификаторов допустимы и какие из допустимых различны:

  `myObject, MyObject, myobject, Myobject, my object, my_object, $myobject`



ГЛАВА 2

Среда NetBeans 6.5

NetBeans — это интегрированная среда разработки (Integrated Development Environment, IDE), т. е. такая программа (или набор программ в одной оболочке), которая позволяет пользователю решать практически все задачи в его проблемной области без выхода из этой программы. Все современные средства разработки программного обеспечения представляют собой интегрированные среды.

До появления интегрированных сред разработки приходилось набирать программы в текстовом редакторе, запускать компилятор командной строки и читать его сообщение о месте предполагаемой ошибки компиляции, после чего в текстовом редакторе искать нужное место исходного кода, исправлять ошибку, читать следующее сообщение об ошибке и т. д.

Интегрированные среды позволяют сразу перейти в место с предполагаемой ошибкой, а некоторые наиболее продвинутые (NetBeans относится к ним) подчеркивают ошибки красной волнистой линией непосредственно в процессе редактирования исходного кода программы.

2.1. Установка JDK 6

Комплект программного обеспечения, находящийся на прилагаемом к книге компакт-диске, в первую очередь рассчитан на пользователей Windows, однако работа с NetBeans возможна и из других сред. В частности, для пользователей Linux предназначены файлы, в названии которых присутствует слово linux *(см. описание компакт-диска)*.

Требования к аппаратуре у новых версий программного обеспечения, как обычно, повышенные. Среда NetBeans 6.5 с открытым простейшим приложением занимает в памяти примерно 353 Мбайт. Из них на ноутбуке автора (с Windows XP SP3 и 1,5 Гбайт оперативной памяти) примерно 176 Мбайт располагаются в оперативной памяти, а 177 Мбайт операционная система хранит в виртуальной памяти (в файле подкачки на жестком диске). При этом файлы самой среды занимают в памяти всего около 7 Мбайт, а остальную память потребляет виртуальная Java-машина (процесс java.exe).

Работа с виртуальной памятью примерно в 1000 раз медленнее, чем с оперативной, и ее использование заметно замедляет процесс в тех случаях, когда операционная система вынуждена к ней обращаться. Исходя из личного опыта, автор делает вывод, что

512 Мбайт оперативной памяти можно рассматривать как нижний предел, с которым еще можно работать в среде NetBeans 6.5. На компьютерах, где памяти заметно меньше, реально работать нельзя.

Требования к тактовой частоте процессора в данном случае не столь существенны — процессоры Pentium 4 (1,6 ГГц) и Celeron (2,4 ГГц) вполне справлялись с работой.

На прилагаемом к книге компакт-диске содержится дистрибутив среды NetBeans 6.5. Для его установки в папке NetBeans 6.5 в операционной системе Windows следует запустить файл netbeans-6.5-ml-windows.exe.

Но предварительно требуется установить JDK. При отсутствии на компьютере необходимой версии JDK (5 или 6) выдается сообщение, показанное на рис. 2.1.

Рис. 2.1. Сообщение, выдаваемое в случае, когда необходимая версия JDK на компьютере не установлена

Следует нажать кнопку **Exit Installer** (Выйти из программы установки) и запустить с диска файл jdk-6u11-windows-i586-p.exe.

По функциональности JDK 5 и JDK 6 примерно одинаковы, при установке полных объемов дистрибутивов заметных изменений в версии 6 по сравнению с 5-й нет. Но предпочтительнее ставить JDK 6 даже в том случае, если на компьютере уже имеется JDK 5, поскольку версия 6 обеспечивает более высокую скорость работы приложений, лучшую многопоточность (в частности, на многоядерных процессорах), меньшее время старта Java-машины, дополнительные возможности управления режимами Java-машины и обладает большей устойчивостью. Кроме того, начиная с JDK 6 update 10, имеется возможность при появлении очередных обновленных версий JDK не переставлять весь дистрибутив, а устанавливать только обновления. При этом нет необходимости удалять предыдущие версии JDK и JRE — они могут работать на компьютере одновременно. Хотя при запуске приложений Java возможны проблемы из-за наличия пути к неправильной версии исполняющей среды.

JDK 6 рассчитана на достаточно новые версии операционных систем. Например, для MS Windows поддерживаются Windows 2000 SP4 (пакет обновлений версии 4) и выше, XP SP2, Vista. При попытке установить JDK для более старой версии (например, XP SP1) выдается предупреждение о том, что требуется более новая версия операционной системы, но можно на свой страх и риск продолжить установку и практически все будет работать правильно (но без гарантий Sun). Если же вы захотите работать не только с NetBeans, но и с JDeveloper 11, необходима XP SP3 и достаточно новая версия

JDK 6 (update 7 или более поздний). Для установки JDK зайдите в папку jdk1.6 и запустите файл jdk-6u11-windows-i586-p.exe.

На короткое время появится окно с предложением подождать, пока идет подготовка к процессу установки, после чего появится окно с лицензией, предлагаемой Sun (рис. 2.2).

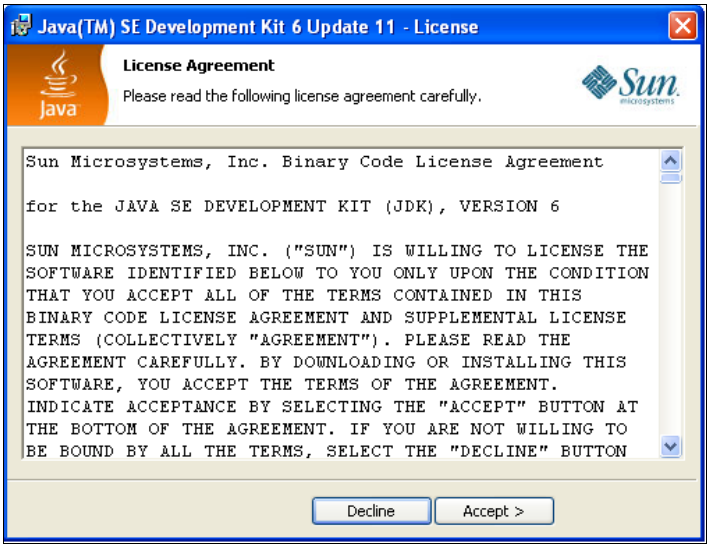


Рис. 2.2. Окно с лицензией

Замечание

Указанные на рисунках цифры соответствуют JDK 1.6u11 (JDK 6 обновление 11). На компакт-диске может оказаться более новая версия JDK, поэтому внешний вид изображений на экране может несколько отличаться от приводимых в книге. Указанная далее информация по объему дискового пространства, требуемого для модулей NetBeans 6.5, также приблизительна.

Нажатие на кнопку **Accept >** (Принять) означает ваше согласие с данной лицензией и начало установки JDK. Нажатие на кнопку **Decline** (Отклонить) — отказ от продолжения установки. В лицензии говорится о том, что Sun дает право на бесплатное использование программ JDK в целях запуска Java-программ при условии, что не будет проводиться модификация кодов программ и текстов документации JDK, а также о том, что программное обеспечение поставляется в соответствии с принципом "AS IS" (как есть), без какой-либо ответственности Sun за применение этого программного обеспечения пользователем.

После нажатия кнопки **Accept >** появляется диалог выбора устанавливаемых элементов JDK (рис. 2.3). Рекомендуется установить их все. Это займет примерно 300 + 46 + 49 + 173 + 25 = 593 Мбайт дискового пространства. В крайнем случае, если дискового пространства не хватит, можно установить только Public JRE (исполняющую среду Java для приложений — 173 Мбайт). Но лучше все-таки установить JDK на другой диск (кнопка **Change...** — изменить место установки) или просто удалить ненужные программы с рабочего диска.

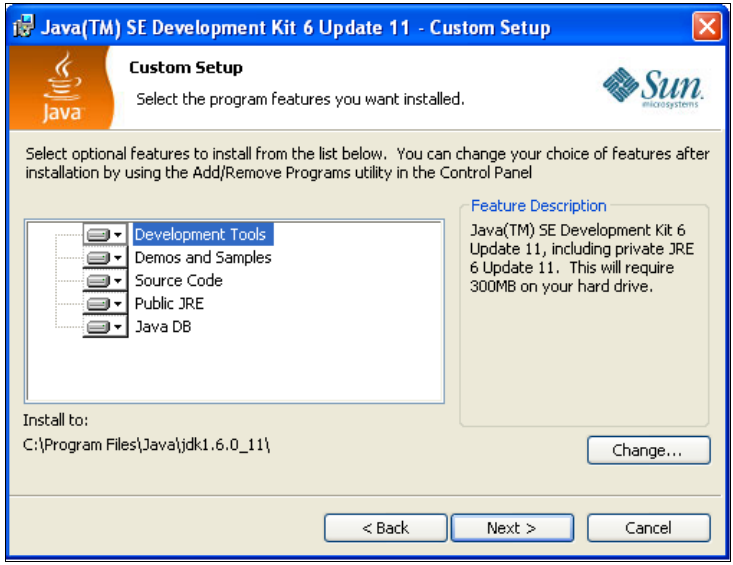


Рис. 2.3. Диалог выбора устанавливаемых элементов JDK

Нажатие на кнопку **Next >** (Далее, следующий) продолжает установку. По очереди появляются сообщения о выполняемых при установке действиях (это занимает 2—3 минуты), после чего открывается окно с предложением выбрать папку (рис. 2.4), где будет установлена исполняющая среда JRE 6. Напомним, что это комплект JRE, предназначенный для обычного выполнения, а не тот вариант JRE, который входит в состав JDK и служит для отладки при разработке приложений.

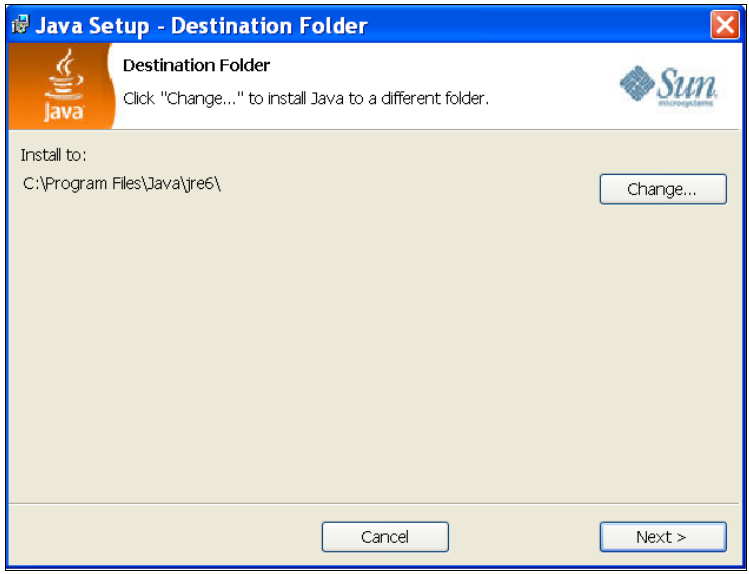


Рис. 2.4. Окно с предложением о смене папки, в которую устанавливается JRE

Рис. 2.5. Сообщение об успешном завершении установки JDK

Нажимаем кнопку **Next >** и продолжаем установку. Наконец, через 3—4 минуты появляется сообщение, показанное на рис. 2.5, означающее успешное завершение установки.

Для завершения установки следует нажать кнопку **Finish** (Завершить).

2.2. Установка среды NetBeans 6.5

Как уже говорилось, для установки NetBeans IDE следует запустить файл netbeans-6.5-ml-windows.exe. Напомним, что IDE означает "интегрированная среда разработки".

Сначала появится окно, показывающее прогресс в конфигурации инсталлятора (рис. 2.6).

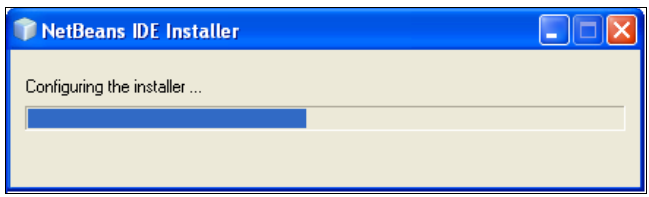


Рис. 2.6. Окно, показывающее прогресс в конфигурации инсталлятора

Это не очень быстрый процесс, придется подождать 2—3 минуты до тех пор, пока не появится следующее окно (рис. 2.7), в котором необходимо нажать кнопку **Customize...**, после чего откроется окно для выбора устанавливаемых модулей (рис. 2.8).

Замечание

Даже если у вас большой жесткий диск, не следует устанавливать ненужные модули. Особенно это относится к исполняющим средам (runtime) — после установки они по умолчанию будут автоматически запускаться каждый раз при включении компьютера, замедляя загрузку операционной системы и отнимая у компьютера много оперативной памяти.

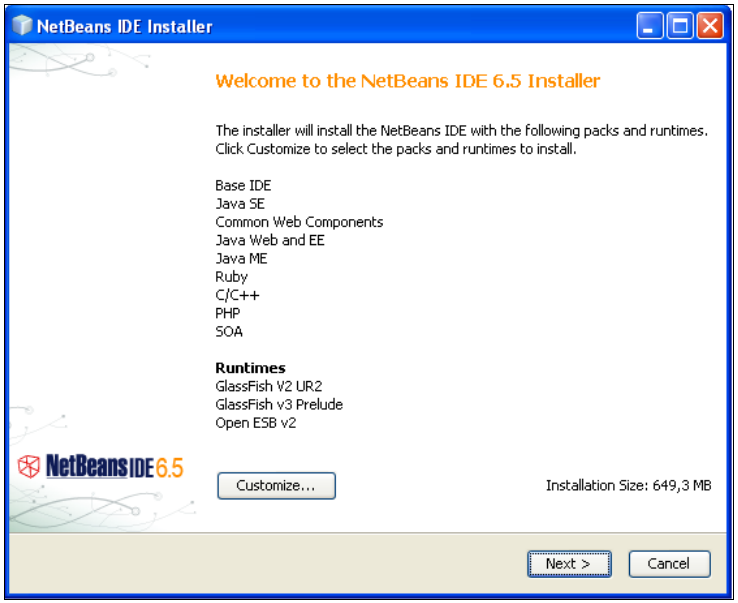


Рис. 2.7. Окно, в котором показываются устанавливаемые модули

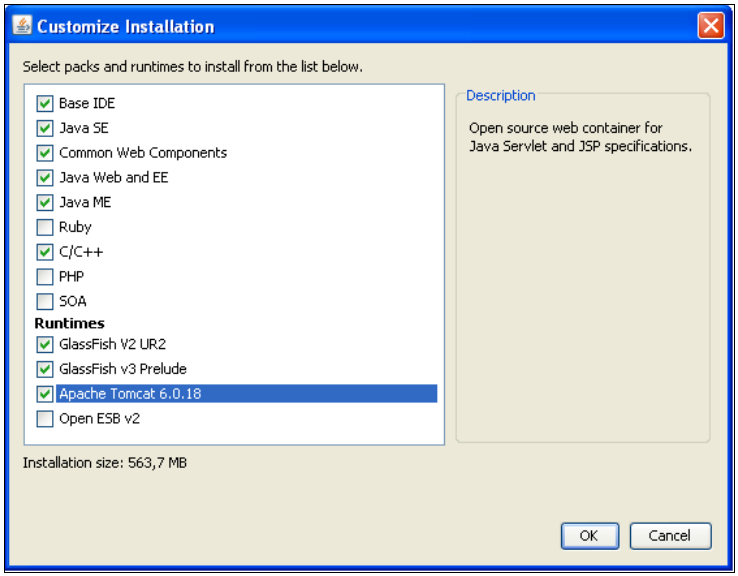


Рис. 2.8. Окно выбора устанавливаемых модулей

Перечислим пункты выбора в данном окне (в скобках приведен перевод английского текста, поясняющего соответствующую опцию установки, а затем даны комментарии по использованию модуля в рамках данного учебного курса).

- **Base IDE.** 70,8 Мбайт. (Основные компоненты интегрированной среды разработки.) Платформа NetBeans без модулей расширения.

  Данный модуль следует оставить отмеченным как необходимый.

- **Java SE.** 63,9 Мбайт. (Основные инструменты для программирования на языке Java.) Интегрированная среда разработки Java. Включает редактор, отладчик, профилировщик, поддержку рефакторинга и инструменты проектирования для создания графического пользовательского интерфейса (ранее известные как Matisse).

  Модуль расширения платформы NetBeans для использования языка Java. Следует оставить отмеченным как необходимый.

- **Common Web Components.** 5,2 Мбайт. (Компоненты, используемые всеми веб-ориентированными проектами для Java Web, PHP, Ruby on Rails, Grails. Поддержка работы с библиотеками JavaScript).

  Данный модуль следует оставить отмеченным как необходимый.

- **Java Web and EE.** 110,1 Мбайт. (Инструменты для создания WWW-приложений Java, а также корпоративных (серверных) приложений, совместимых со спецификациями J2EE 1.4 и Java EE 5.) Включает поддержку сервлетов, приложений JSP (Java Server Pages), библиотек тегов, Spring (библиотека с набором "легких" переносимых оболочечных компонентов для всех основных технологий Java EE — JDBC, ORM, JTA, Servlets/Jsp, JMX и др.), Struts ("распорки" — технология для создания WWW-приложений), Java Persistence API (поддержка технологии Java Persistence — устойчивость Java), Enterprise Java Beans (серверные компоненты Java Beans), JAX-WS (веб-службы JAX. Аббревиатура JAX расшифровывается как Java API for XML-Based Remote Procedure Calls — программируемый пользовательский интерфейс для основанных на XML удаленных вызовах процедур), RESTful (веб-службы, использующие принцип определения ресурсов по универсальным идентификаторам ресурсов URI), Java Server Faces (лица сервера Java), языка Groovy, комплекса Grails. Обеспечивает окружение для визуального проектирования пользовательского интерфейса соответствующих приложений.

  Модуль следует оставить отмеченным как необходимый. Он нужен для работы с серверными технологиями Java *(см. главу 11)*.

- **Java ME.** 100,0 Мбайт. (Всеобъемлющий набор инструментов для создания приложений Java Micro Edition для сотовых телефонов и других портативных устройств.) Включает визуальный дизайнер, мастер веб-соединений и инструменты для решения проблем фрагментации памяти устройств.

  Модуль следует оставить отмеченным как необходимый. Он нужен для работы с технологиями Java для мобильных телефонов и других тонких аппаратных клиентов *(см. главу 14)*.

- **Ruby.** 38,3 Мбайт. (Инструментальные средства для программирования серверных приложений на языке Ruby (Рубин) и Ruby on Rails.) Включает интерпретатор

JRuby, исполняющую среду (виртуальную машину) Rails, мощный редактор, отладчик, "менеджер драгоценного камня", интерактивную оболочку Ruby.

В рамках данного учебного курса не требуется.

- **C/C++.** 13,4 Мбайт. (Инструменты для создания приложений C и C++.) Включает заготовки проектов, поддержку существующих проектов, развитый редактор, отладчик, мастер управления конфигурациями. Модуль необходим для работы с платформо-зависимыми (native) методами — *глава 16*.
- **PHP.** 4,7 Мбайт. (Инструменты для создания в среде NetBeans приложений PHP.)

  В данном учебном курсе не требуется.
- **SOA.** 23,7 Мбайт. (Средства для создания составных приложений в рамках SOA (Service Oriented Architecture) — архитектуры, основанной на использовании сервисов.) Включает графический пакет BPEL, предназначенный для создания и визуального проектирования приложений, ориентированных на использование сервисов, в том числе графический редактор, отладчик, поддержка рефакторинга.

  Требует, чтобы были установлены исполняющие среды GlassFish и Open ESB 2.0.

  В рамках данного учебного курса не нужен, следует снять флажок.

Runtimes (исполняющие среды):

- **GlassFish v2 UR2.** 84,8 Мбайт. (Бесплатный высокопроизводительный сервер приложений Java.) Базовая реализация спецификации Java EE 5. Проект с открытым исходным кодом.

  Предназначен для разработки и использования совместимых с платформой Java EE 5 приложений и веб-серверов в крупномасштабных промышленных средах. Основан на исходных кодах серверов приложений Sun Java System Application Server PE 9 фирмы Sun и TopLink persistence фирмы Oracle.

  Модуль необходим для проверки работы с серверными приложениями (*см. разд. 11.3* и *11.4*). В рамках данного курса не требуется подключения к Интернету, сервер позволяет отлаживать приложения на локальном компьютере.
- **GlassFish v3 Prelude.** 36,8 Мбайт. Предварительный вариант разрабатываемой новой версии сервера приложений GlassFish. Отличается от версии 2 модульностью, небольшим количеством потребляемых ресурсов, расширяемостью архитектуры. Компонент не обязателен. Но желательно подключить его, чтобы при работе с серверными приложениями можно было осуществлять выбор между разными программными серверами.
- **Apache Tomcat 6.0.18.** 10,4 Мбайт. (Бесплатный веб-сервер с открытым исходным кодом.) Базовый пример реализации сервера для технологий сервлетов и JSP (JavaServer Pages).

  Компонент не обязателен. Но желательно подключить его, чтобы при работе с серверными приложениями можно было сравнивать этот легкий (потребляющий мало ресурсов) сервер с GlassFish, а также осуществлять выбор между разными программными серверами.
- **Open ESB v2.** 12,2 Мбайт. (Открытая реализация исполняющей среды Enterprise Service Bus (ESB — корпоративная сервисная шина), использующая как основу Java

Business Integration (JBI — Java-интеграция бизнеса).) Open ESB реализует открытые стандарты, имеет открытый исходный код и является платформой для построения решений, основанных на SOA.

В рамках данного учебного курса не требуется.

После выбранных опций требуемое на диске место уменьшится до 564 Мбайт. Нажатие на кнопку **OK** возвращает нас в начальный диалог установки NetBeans. И теперь уже можно нажать кнопку **Next >**.

Появится окно с лицензией. В нижней части окна следует отметить флажком пункт выбора **I accept the terms in the license agreement** (Я принимаю условия лицензионного соглашения). Лицензия дает право на бесплатное использование среды NetBeans и поставляемых с ней модулей.

Для продолжения установки следует нажать кнопку **Next >**.

Появится следующая форма, в которой можно выбрать папку, куда устанавливается NetBeans и модули к ней, а также папку с версией JDK, под управлением которой будет работать среда NetBeans (рис. 2.9).

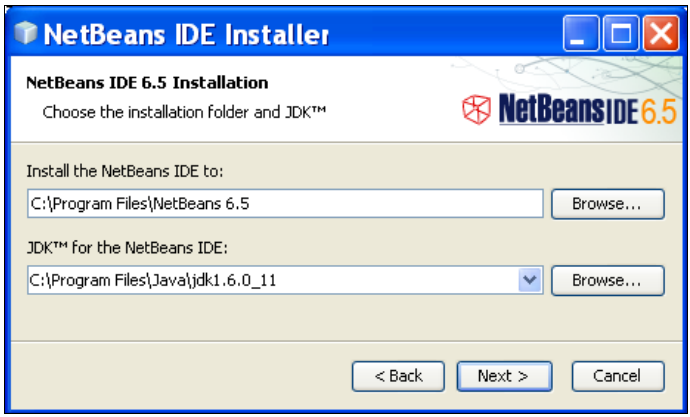


Рис. 2.9. Форма выбора папки, в которую устанавливается среда NetBeans

После нажатия на кнопку **Next >** появляется диалоговое окно с параметрами администрирования сервера **GlassFish** (рис. 2.10).

Пароль администратора, конечно, желательно изменить, чтобы никто не смог зайти на сервер приложений в качестве администратора по стандартному паролю. Остальные параметры лучше не менять.

После нажатия кнопки **Next >** появится форма с возможностью выбора папки для установки сервера **Glassfish-v3-prelude**. Ничего не меняем и нажимаем кнопку **Next >**. Будет выведена форма выбора папки для установки сервера **Apache Tomcat 6.0**. Ничего не меняем и опять нажимаем кнопку **Next >**.

Появится форма с информацией о выбранных параметрах. Нажатием на кнопку **Install** (Установить) запустим установку. Откроется диалоговое окно, в котором несколько минут будет выводиться текущая информация о ходе установки, после чего отобразится сообщение **Setup Complete** (установка завершена) — рис. 2.11.

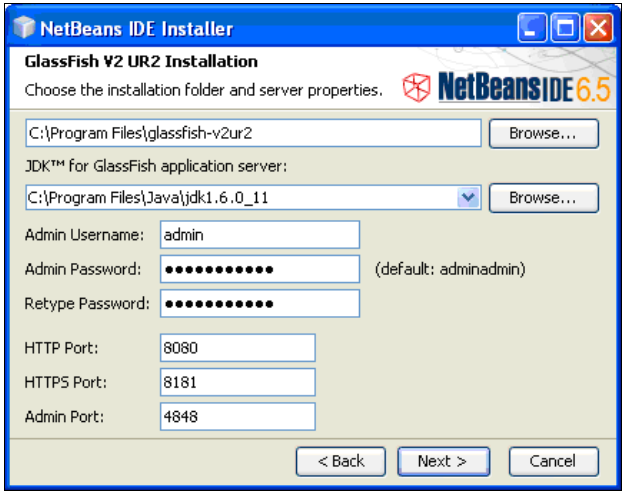


Рис. 2.10. Диалоговое окно с параметрами администрирования сервера **GlassFish**

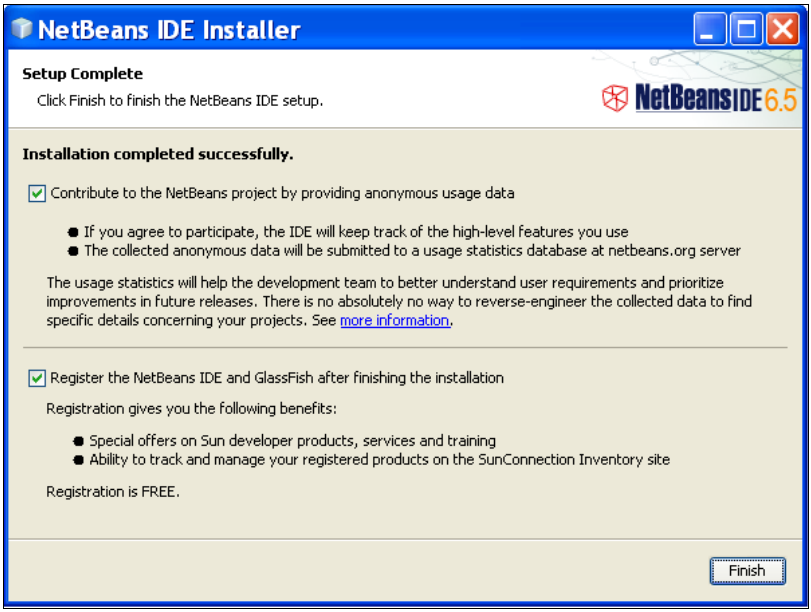


Рис. 2.11. Окно окончания установки NetBeans 6.5

Если ваш компьютер подключен к Интернету, и вы согласны ответить на ряд вопросов по среде NetBeans, чтобы помочь ее разработчикам усовершенствовать продукт, или если вы хотите зарегистрироваться как пользователь NetBeans и GlassFish, сразу нажмите кнопку **Finish** (Завершить) и ответьте на вопросы на английском языке в появляющихся после этого формах. Остальным пользователям следует снять галочки в обоих пунктах выбора (квадратиках) и только после этого нажать кнопку **Finish**, что и завершит установку. На рабочем столе компьютера появится ярлык NetBeans, а в меню **Пуск** — рабочая группа **Программы/ NetBeans 6.5/NetBeans IDE**.

Для того чтобы в дальнейшем добавить еще не установленные модули и исполняющие среды, нужно еще раз запустить инсталлятор среды NetBeans, нажать кнопку **Customize…** и отметить необходимые пункты. Не следует опасаться того, что при повторном запуске инсталлятора затрется предыдущая установка или потеряются какие-то параметры или файлы.

Изменить состав установленных модулей и исполняющих сред также можно с помощью пункта меню среды NetBeans **Tools/Plugins** (Инструменты/Присоединяемые модули).

Удалить среду NetBeans и/или исполняющие среды (серверы приложений или Open ESB, если она была установлена) можно либо из панели управления в пункте **Установка и удаление программ**, либо с помощью запуска программы Uninstall в папке **Program Files\ NetBeans 6.5**.

Отметим, что при наличии на компьютере антивирусных программ (например, Kaspersky Antivirus с включенной проактивной защитой) после окончания работы инсталлятора они могут выдавать предупреждение о потенциально опасных действиях элемента инсталлятора (программы nbi-cleaner, запускаемой из временной папки). Это вовсе не означает, что дистрибутив заражен вирусом или является "троянской" программой. Данной программе следует разрешить выполнять необходимые действия — она удаляет ненужные файлы и записывает в реестр компьютера информацию о конфигурации прошедшей инсталляции (по-видимому, именно это вызывает тревогу).

После запуска NetBeans 6.5 появляется страница приветствия (welcome page, start page) этой среды (рис. 2.12). Если не хочется, чтобы данное окно появлялось каждый раз после старта среды, снимите галочку в нижней части окна в пункте **Show On Startup** (Показывать при запуске).

На этом наша работа по установке программ еще не закончена. Читателям, знакомым с английским языком (а для профессионального программирования знать технический английский абсолютно необходимо), полезно установить документацию к JDK, находящуюся в файле с архивом jdk-6-doc.zip на прилагаемом к книге компакт-диске. Архив следует целиком скопировать в папку на жестком диске (например, в папку C:\Program Files\Java), после чего запустить интегрированную среду разработки (IDE) NetBeans и выбрать в главном меню пункт **Tools/Java Platforms**.

Появится диалоговое окно (рис. 2.13), в котором следует переключиться на вкладку **Javadoc**, нажать кнопку **Add ZIP/Folder...** и выбрать путь к архиву (при попадании файлового диалога в папку с архивом придется подождать — среда долго читает заголовки архивных файлов, это не "зависание" программы).

После подтверждения выбора архива (установки на него курсора и нажатия кнопки **Add ZIP/Folder…**) документация к JDK установлена. Можно осуществлять быстрый доступ к документации из среды NetBeans: в меню **Help** (справка) открывать появляющийся после этого пункт **Javadoc References/Java Platform SE 6**, а также выполнять поиск **Help/Javadoc Index Search** (<Shift>+<F1>) (рис. 2.14).

Если в исходном коде курсор стоит на каком-либо имени, то появляющаяся при нажатии комбинации клавиш <Shift>+<F1> справка оказывается контекстной. Однократный щелчок по имени в списке найденных вхождений приводит к показу в нижнем окне информации, соответствующей этому вхождению, двойной щелчок — к отображению той же информации (HTML-документа) в независимом окне браузера, которое можно развернуть на весь экран.

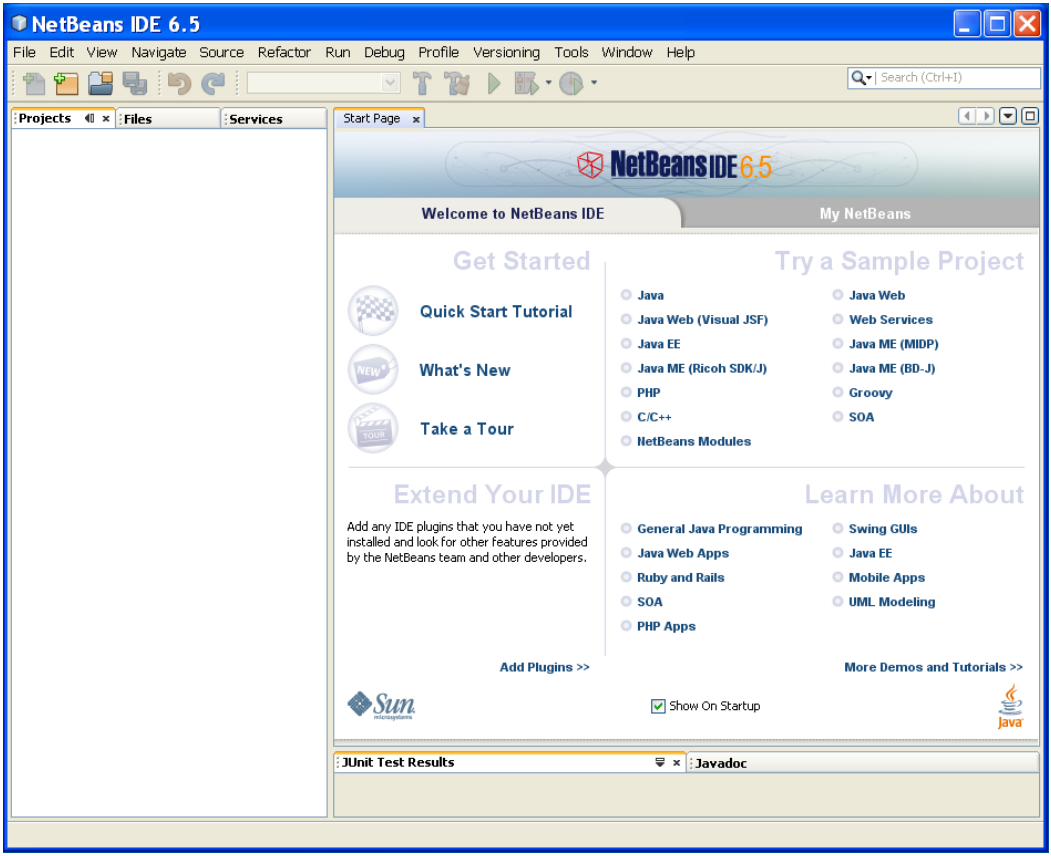


Рис. 2.12. Окно приветствия среды NetBeans 6.5

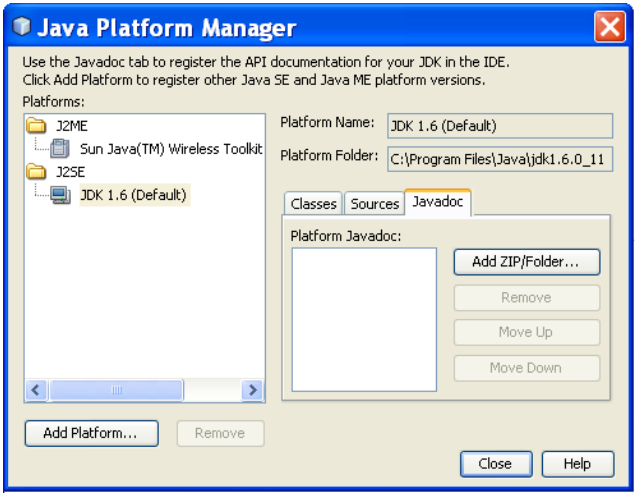


Рис. 2.13. Установка документации к JDK

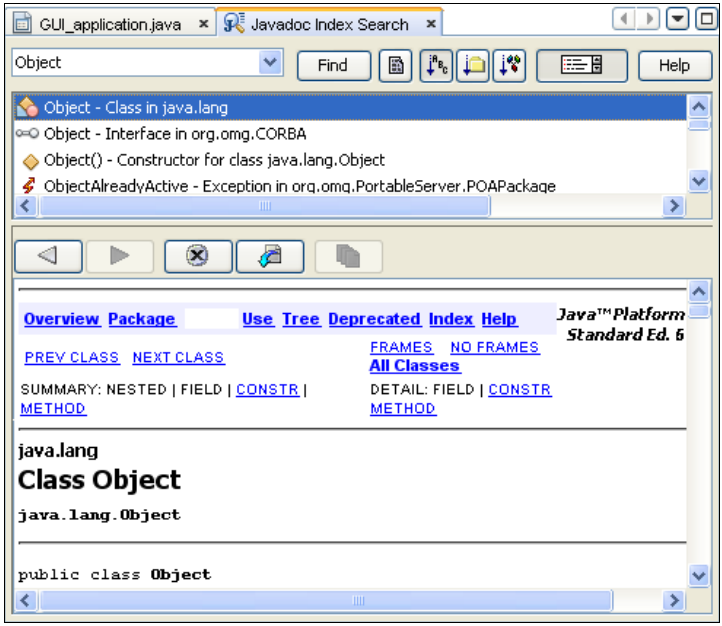


Рис. 2.14. Справка **Javadoc Index Search**

2.3. Создание в NetBeans простейшего приложения Java

Создадим с помощью среды NetBeans приложение Java. Для этого запустим интегрированную среду разработки (IDE) NetBeans, в главном меню **File/New Project...**, в открывшемся диалоговом окне (рис. 2.15) в столбце **Categories** (Категории) выберем пункт **Java**, после чего в столбце **Projects** (Проекты) укажем вид проекта **Java Application** и нажмем кнопку **Next >**.

Появится форма (рис. 2.16), в которой можно изменить некоторые параметры создаваемого проекта.

Нажмем кнопку **Finish**. Значения по умолчанию для начала менять не стоит — это можно будет сделать потом, когда вы освоитесь со средой разработки.

На рис. 2.17 показано, как выглядит редактирование исходного кода приложения в среде NetBeans.

Отметим, что после запуска идет довольно долгая процедура индексации архива rtl в JDK, и картина в окне навигации появляется не скоро. Но это происходит только при первом запуске.

В левом верхнем окне **Projects** отображается дерево проектов. В нашем случае это дерево для проекта **JavaApplication1**. В данном окне можно видеть одновременно произвольное число проектов. Узлы деревьев сворачиваются щелчком по значку с минусом и разворачиваются щелчком по значку с плюсом (или двойным щелчком по соответствующему имени).

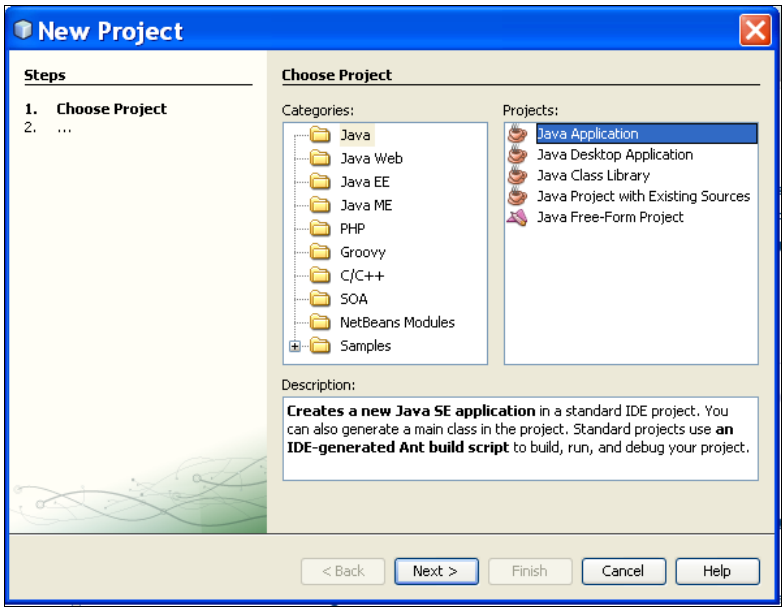


Рис. 2.15. Создание нового проекта. Шаг 1

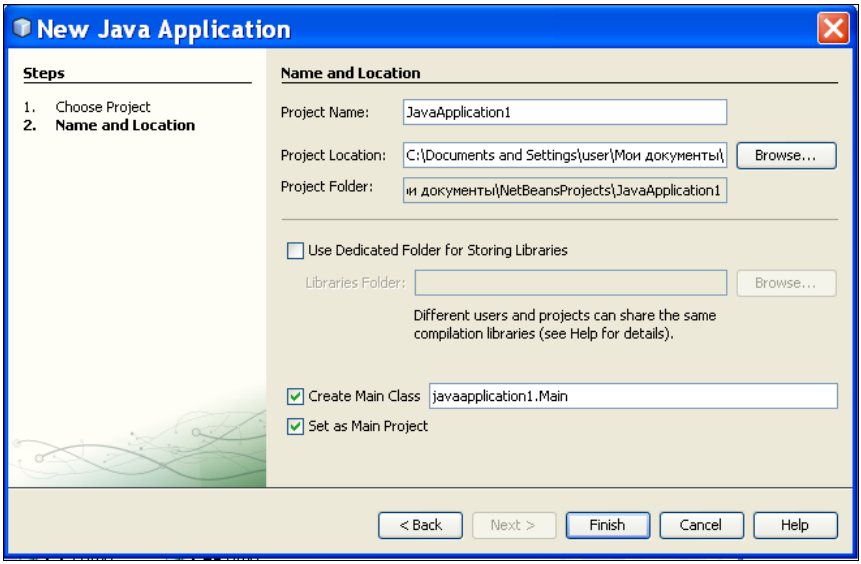


Рис. 2.16. Создание нового проекта. Шаг 2

В правом (главном) окне редактора проекта показывается исходный код проекта. Имя окна (**Source** — сокращение от Source code — исходный код) отображается только при наличии в главном окне вкладок для других типов редакторов (например, для редактора Design (дизайн) в приложениях с графическим пользовательским интерфейсом).

В левом нижнем окне **Main.java — Navigator** выводится список имен членов класса приложения — имена переменных и подпрограмм. В этом окне двойной щелчок по

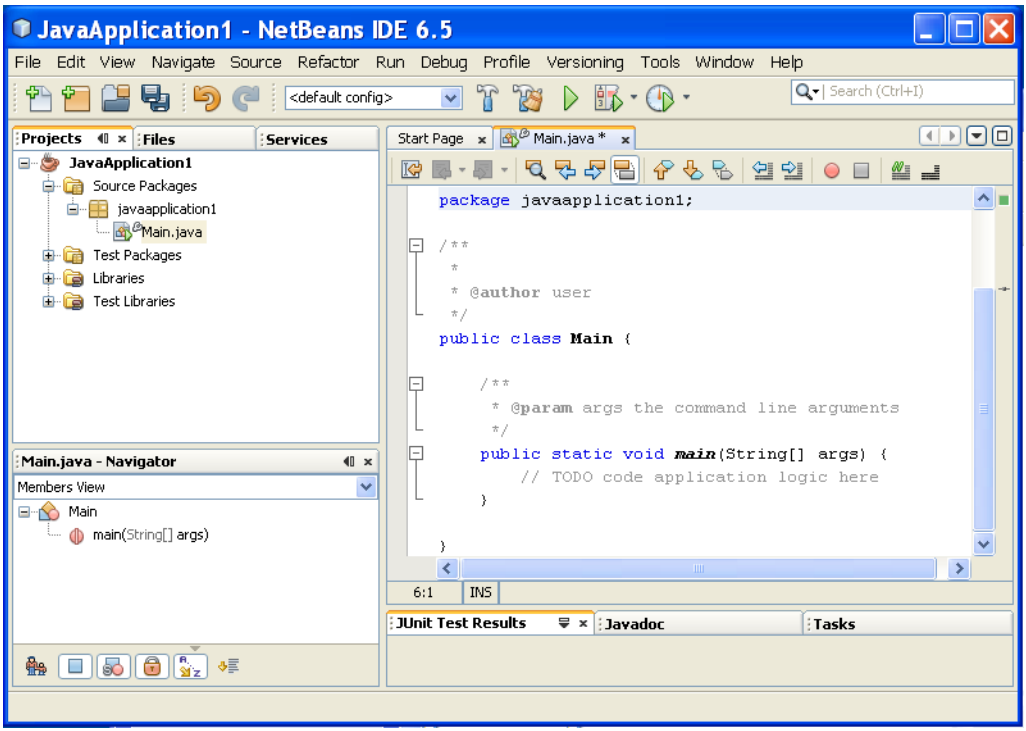


Рис. 2.17. Редактирование исходного кода приложения

имени приводит к тому, что в окне редактора исходного кода происходит переход на то место, где задана соответствующая переменная или подпрограмма.

Рассмотрим, как выглядит сгенерированный исходный код нашего приложения Java (листинг 2.1).

Листинг 2.1. Исходный код приложения

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package javaapplication1;

/**
 *
 * @author User
 */
public class Main {

    /**
     * @param args the command line arguments
     */
```

```
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

Сначала идет многострочный комментарий `/* ... */`, который содержит информацию о том, как изменить шаблон (template) создания проекта.

Затем объявляется, что наш класс будет находиться в пакете `javaapplication1`.

После этого идет многострочный комментарий `/** ... */`, предназначенный для автоматического создания документации по классу. В нем присутствует инструкция задания метаданных с помощью выражения `@author` — информация об авторе проекта для утилиты создания документации `javadoc`.

Метаданные — это некая информация, которая не относится к работе программы и не включается в нее при компиляции, но сопровождает программу и может быть использована другими программами для проверки прав на доступ к ней или ее распространения, проверки совместимости с другими программами, указания параметров при запуске класса и т. п. В данном месте исходного кода имя `User` берется средой разработки из операционной системы по имени папки пользователя. Его следует заменить на имя реального автора (в нашем случае на "Вадим Монахов").

В Java начиная с версии JDK 5 метаданные задаются с помощью *аннотации* — выражения, начинающегося с символа @, после которого следует идентификатор, а затем могут идти круглые скобки, внутри которых указываются параметры аннотации. Об аннотациях более подробно будет рассказано далее в *разд. 2.13*, *9.1*, *11.2*, *11.4*, *13.5* и *17.4*.

Все классы и объекты приложения вызываются и управляются из метода `main`, который объявлен далее и выглядит следующим образом:

```
public static void main(String[] args) {
 }
```

Он является методом класса *(см. разд. 1.6)* и поэтому для его работы нет необходимости в создании объекта — экземпляра класса `Main`. Если данный объект создается, то это происходит во время работы метода `main`.

Метод `main` — главный метод приложения, он управляет работой запускаемой программы и автоматически вызывается при запуске приложения. Параметр `args` этого метода — массив строк, имеющий тип `String[]`. Это параметры командной строки, которые передаются в приложение при его запуске. Слово `String` означает "строка", а квадратные скобки обозначают, что это массив строк.

После окончания выполнения метода `main` приложение завершает свою работу.

При объявлении любого метода в Java сначала указывают модификатор видимости, обозначающий права доступа к методу, затем другие модификаторы, после чего следует тип возвращаемого методом значения. Если модификатор видимости отсутствует, считается, что это пакетный (default) вариант видимости, разрешающий доступ к методу только в пределах пакета, в котором находится класс.

Конструкторы — это вид подпрограмм, предназначенных для создания объектов данного класса. Они представляют особый случай — в них имя типа и имя метода совпадают. В Java они не считаются методами, хотя в других языках программирования такого тонкого различия не делается.

Далее следует имя метода, после чего в круглых скобках идет список параметров (аргументов), передаваемых в данный метод при его вызове. После этого в фигурных скобках идет *тело метода*, т. е. его *реализация* (алгоритм, который будет выполняться при вызове метода).

В языке Java, как и в C/C++, подпрограммы всегда являются подпрограммами-функциями, возвращающими какое-либо значение. Если нужно написать подпрограмму-процедуру, не возвращающую никакого значения, то в C/C++/Java пользуются подпрограммами-функциями с типом возвращаемого значения `void` (пустота, пустое пространство), как и происходит в случае метода `main`.

Среда NetBeans создает заготовку методов — в них имеется пустое тело. Чтобы метод выполнил какие-либо действия, следует дописать свой собственный код. Напишем традиционный пример — вывод сообщения "Привет!". В заготовке метода `main` имеется комментарий:

```
// TODO code application logic here
(здесь описать логику работы приложения).
```

Этот комментарий напоминает программисту, что в данном месте следует написать программный код, реализующий логику работы приложения после запуска приложения.

Заменим комментарий строкой вывода текста:

```
System.out.println("Привет!");
```

Класс `System` (система) имеет поле `out` (наружу). Это объект, предназначенный для поддержки вывода. У него есть метод `println` для вывода текста в режиме консоли.

Консольный ввод/вывод ранее широко применялся в операционных системах, ориентированных на работу в режиме командной строки. При этом основным средством взаимодействия пользователей с программами служила текстовая консоль (пульт управления). Устройство ввода — клавиатура, а устройство вывода — окно операционной системы, обеспечивающее вывод текста в режиме пишущей машинки (системным шрифтом с буквами, имеющими одинаковую ширину). Очень много примеров программ в учебных курсах по Java ориентированы на работу в таком режиме.

В настоящее время в связи с тем, что подавляющее большинство пользователей имеют дело с программами в графическом режиме, работу с консолью нельзя рассматривать как основную форму ввода/вывода. Тем более что NetBeans позволяет без особых усилий создавать графический пользовательский интерфейс (Graphics User's Interface, GUI) приложения. А консольный режим следует применять только как промежуточный, удобный как средство вывода вспомогательной информации при отладке программы.

В редакторе исходного кода NetBeans синтаксические конструкции наглядно подсвечиваются разным цветом и стилем (рис. 2.18).

Например, имена методов отображаются наклонным полужирным шрифтом, а имена объекта `out` подсвечиваются зеленым и выводятся наклонным шрифтом (курсивом).

Конструктор `Main()` также выделяется полужирным, но, в отличие от остальных методов, не наклонным, а прямым шрифтом.

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Привет!");
    }
```

Рис. 2.18. Подсветка элементов в редакторе исходного кода

Замечание

Не путайте конструктор `Main()` с классом `Main`. Они имеют одинаковые имена, но после имени конструктора всегда идут круглые скобки.

Еще одна полезная возможность — автоматическое преобразование строковой константы в сумму строк при нажатии клавиши <Enter>. При этом нет необходимости вручную вставлять кавычки и знак суммы. Напомним, что символ перевода на новую строку в Java кодируется как `\n`, и в редакторе использовать для этого нажатие на клавишу <Enter> нельзя.

Следует отметить, что в NetBeans проекты по умолчанию хранятся в папке C:\Documents and Settings\User\Мои документы\NetBeansProjects, что позволяет отделить их от папок других программных сред. Но если вы откроете проект из другой папки, эта папка станет текущей. Поэтому полезно помнить путь к основному месту расположения проектов NetBeans. Также следует знать, что недавно открывавшиеся проекты можно вызвать гораздо проще — со страницы приветствия (их имена там отображаются) или с помощью пункта меню **File/Open Recent Project** (Файл/Открыть недавний проект).

2.4. Компиляция файлов проекта и запуск приложения. Jar-файлы

В современных средах разработки предусмотрены два режима компиляции — compile (скомпилировать) и build (построить). В режиме compile происходит компиляция только тех файлов проекта, которые были изменены в процессе редактирования после последней компиляции. А в режиме build перекомпилируются заново все файлы.

Для компиляции проекта следует выбрать в меню среды разработки команду **Run/Build Main Project** (или нажать клавишу <F11>, или пиктограмму с молотком на панели инструментов). При этом будут заново скомпилированы из исходных кодов все классы проекта.

Команда **Run/Clean and Build Main Project** (комбинация клавиш <Shift>+<F11> или иконка с молотком и веником на панели инструментов) удаляет все выходные файлы

проекта (очищает папки build и dist), после чего заново компилируются все классы проекта.

Команда **Run/Generate Javadoc (JavaApplication1)** запускает создание документации по проекту. При этом из исходных кодов классов проекта выбирается информация, заключенная в документационные комментарии `/** ... */`, и на ее основе создается гипертекстовый HTML-документ.

Команда **Run/Complile "Main.java"** (или клавиша <F9>) компилирует редактируемый файл проекта (в нашем случае файл Main.java, в котором хранятся исходные коды класса `Main`).

Для того чтобы запустить скомпилированное приложение из среды разработки, следует выбрать в меню среды разработки команду **Run/Run Main Project** (или нажать клавишу <F6>, или иконку с зеленым треугольником на панели инструментов). При запуске приложение всегда автоматически компилируется (но не строится), так что после внесения изменений для запуска обычно достаточно нажать клавишу <F6>.

После запуска нашего проекта на выходной консоли (которая находится в нижней части окна проекта) появится служебная информация о ходе компиляции и запуске (рис. 2.19).

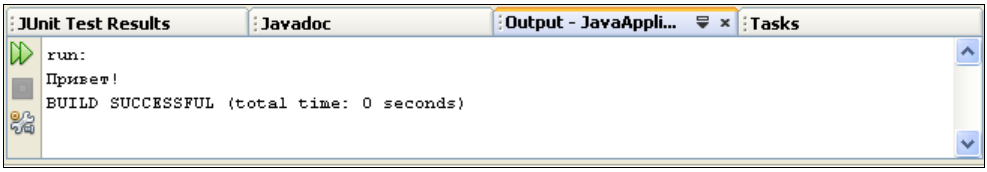


Рис. 2.19. Информация о ходе компиляции и запуске на выходной консоли

На эту же консоль осуществляется вывод методов `System.out.print` и `System.out.println`.

Метод `System.out.print` отличается от `System.out.println` только тем, что в `println` после вывода происходит автоматический переход на новую строку, а в `print` продолжается вывод в ту же строку консоли.

Поэтому вывод:

```
System.out.println("Привет!");
System.out.println("Привет!");
```

даст текст:

```
Привет!
Привет!
```

а

```
System.out.print("Привет!");
System.out.print("Привет!");
```

даст:

```
Привет!Привет!
```

Если требуется запустить скомпилированное приложение без использования среды NetBeans и исходных кодов, можно ввести в командную строку следующую команду:

```
java -jar "C:\Documents and Settings\User
\Мои документы\NetBeansProjects\JavaApplication1\dist\JavaApplication1.jar"
```

Вам, конечно, следует указать путь к вашей папке, в которой создается скомпилированный файл.

Скомпилированные коды классов хранятся в файлах с расширением class, но распространять их в таком виде очень неудобно. В нашем простейшем примере имеется всего один класс, но в реально действующих программах обычно используется от нескольких единиц до нескольких сотен классов. Для того чтобы процесс распространения (deployment) файлов приложения был упорядочен, в Java принято упаковывать все файлы приложения в один архивный файл с расширением jar (от Java Archive — архив Java).

Архивирование обычно уплотняет данные, содержащиеся в файлах архива, экономя не столько место на диске, сколько время пересылки таких файлов в компьютерных сетях. Для файлов небольшого размера вспомогательная информация, записываемая в заголовок архивного файла, занимает больше места, чем сэкономлено на сжатии данных (но для маленьких приложений это некритично).

Запустить приложение из командной строки можно с помощью команды меню **Пуск/Выполнить...** (либо из оболочек Far, Windows Commander, Midnight Commander и т. п.). При этом на мгновение появляется окно текстовой консоли и сразу закрывается. Действительно, программа сделала все, что нужно, вывела сообщение и закончила работу.

В бесплатно распространяемой программе Far (**http://www.farmanager.com/**) работать несколько удобнее: результат вывода появляется в окне, но только в режиме показа консоли (нажать комбинацию клавиш <Ctrl>+<O>). Однако русский текст представлен неправильными символами, т. к. Unicode текстовая консоль Far пока не поддерживает.

Задержать результаты вывода текстовой консоли с экрана можно, записав метод `main` так, как показано в листинге 2.2 (пример 02_04_console на прилагаемом к книге компакт-диске).

Листинг 2.2. Задержка результатов вывода

```
public static void main(String[] args) {
  System.out.println("Привет!");
  try {
    System.in.read();
  } catch (Exception e) {
  }
}
```

Мы добавили ожидание ввода с клавиатуры с помощью вызова `System.in.read()`. Про блок `try-catch` будет рассказано в *разд. 8.6*.

Никаких специальных средств для извлечения файлов приложения из архива jar не потребуется — этим самостоятельно занимается Java-машина. Точно так же не нужно заботиться о создании jar-файлов — этим занимается среда NetBeans по команде меню **Run/Build Main Project** (скомпилировать главный проект) (дублируется клавишей <F11>).

По умолчанию создаются jar-файлы без сжатия. Это бывает необходимо, когда хочется уменьшить время компиляции (архивация — процесс относительно длительный и ресурсоемкий), или когда на клиентском компьютере приложение должно запускаться максимально быстро или потреблять минимум ресурсов, а время загрузки и размер архива не имеет принципиального значения. В этом случае архив просто объединяет несколько файлов в один, а его распаковка идет максимально быстро.

Для того чтобы включить сжатие jar-файлов, следует выполнить следующие действия:

1. Сделать интересующий вас проект главным (команда **Set as Main Project** в контекстном меню, открываемом при щелчке правой кнопкой мыши по имени проекта в дереве проектов).
2. Запустить команду **Run/Set Project Configuration/Customize...** (Запуск/Установить конфигурацию проекта/Задать параметры…) главного меню среды NetBeans, либо в окне **Projects** щелкнуть правой кнопкой мыши по имени проекта и в открывшемся контекстном меню выбрать самый нижний пункт — **Properties** (Свойства).
3. В открывшейся форме выбрать пункт **Build/Packaging** (упаковка) и установить флажок **Compress JAR File** (рис. 2.20).

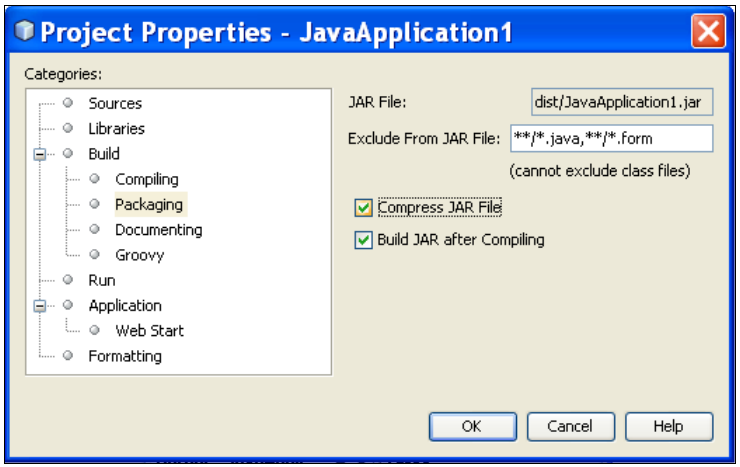


Рис. 2.20. Установка опции сжатия jar-архива

Данная форма полезна и при установке других опций компиляции (например, для задания ключей компиляции).

В ряде случаев (например, после переименования проекта) компиляцию лучше выполнить командой **Run/Clean and Build Main Project** (очистить и скомпилировать главный проект) (дублируется комбинацией клавиш <Shift>+<F11> или кнопкой панели инструментов с изображением веника и молотка). При этом в проекте сначала удаляется все содержимое папок, где хранятся выходные и вспомогательные файлы, созданные средой разработки, и только после этого заново формируются служебные файлы и компилируется проект.

2.5. Структура проекта NetBeans

Рассмотрим, из каких частей состоит проект NetBeans. В окне **Projects** (рис. 2.21) видны основные элементы проекта: **Source Packages** (Пакеты исходного кода), **Test Packages** (Пакеты тестирования), **Libraries** (Библиотеки) и **Test Libraries** (Библиотеки поддержки тестирования). Как уже говорилось, ветви дерева представления элементов проекта можно разворачивать/сворачивать путем нажатия на узлы, отмеченные знаком плюса/минуса. Мы пока будем пользоваться только пакетами исходного кода.

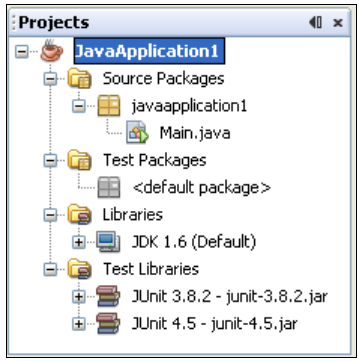


Рис. 2.21. Окно **Projects**

В компонентной модели NetBeans пакеты приложения объединяются в единую конструкцию — модуль. Модули NetBeans являются базовой конструкцией не только для создания приложений, но и для написания библиотек. Они представляют собой оболочку над пакетами (а также могут включать в себя другие модули).

Скомпилированный модуль — это не набор большого количества файлов, а всего один файл — архив jar (Java Archive — архив Java). В нашем случае он имеет то же имя, что и приложение, и расширение jar (это файл JavaApplication1.jar). Модули NetBeans гораздо лучше подходят для распространения, поскольку не только обеспечивают целостность комплекта взаимосвязанных файлов, но и хранят их в заархивированном виде в одном файле, что намного ускоряет копирование и уменьшает объем занимаемого места на носителях.

Отметим не очень удобную особенность NetBeans — после записи проекта и закрытия среды разработки не сохраняется конфигурация развернутых деревьев проекта, хотя конфигурация открытых окон остается той же. Поэтому после каждого нового запуска NetBeans деревья проектов показываются в свернутом виде.

Чтобы попасть в режим редактирования исходного кода нашего приложения, в окне **Projects** (проекты) (левом верхнем окне среды разработки) следует развернуть последовательность узлов **JavaApplication1/SourcePackages/javaapplication1/**. Это делается нажатием на плюсики в соответствующих узлах или двойным щелчком по имени узла. Затем нужно сделать двойной щелчок левой кнопкой мыши по имени узла **Main.java**, либо с помощью щелчка правой кнопки мыши по этому имени открыть контекстное меню и выбрать в нем первый пункт — **Open**.

Есть и более простой способ. По умолчанию сначала открывается окно **Welcome** (привет, приветствие). Поскольку среда разработки сохраняет список открытых окон, то в

верхней части окна редактирования кода щелчком мыши можно выбрать нужное имя окна (но при этом не видна структура проекта). Так что первый способ во многих случаях может быть предпочтительнее.

Если вы открываете новый проект, старый не закрывается. И в дереве проектов видны все доступные проекты. То же относится и к списку открытых окон. Это позволяет работать сразу с несколькими проектами (например, копировать в текущий проект участки кода из других проектов). Один из открытых проектов — главный (**Main Project**) — будет запускаться на исполнение по команде **Run/Run Main Project**.

Чтобы установить какой-либо из открытых проектов в качестве главного, следует в дереве проектов правой кнопкой мыши щелкнуть по имени проекта и выбрать пункт меню **Set Main Project**. Закрыть какой-либо из открытых проектов можно, щелкнув правой кнопкой мыши по имени проекта и выбрав пункт меню **Close Project**.

Работа в редакторе исходного кода в режиме по умолчанию, показанном на рис. 2.17, очень неудобна, поскольку рабочая область редактора кода занимает слишком малую часть окна. Максимизируем область редактора, нажав на иконку , расположенную в ряду иконок в правой верхней части окна редактора — на том же уровне, что закладки **Start Page** и **Main.java**. При этом среда разработки приобретет вид, показанный на рис. 2.22.

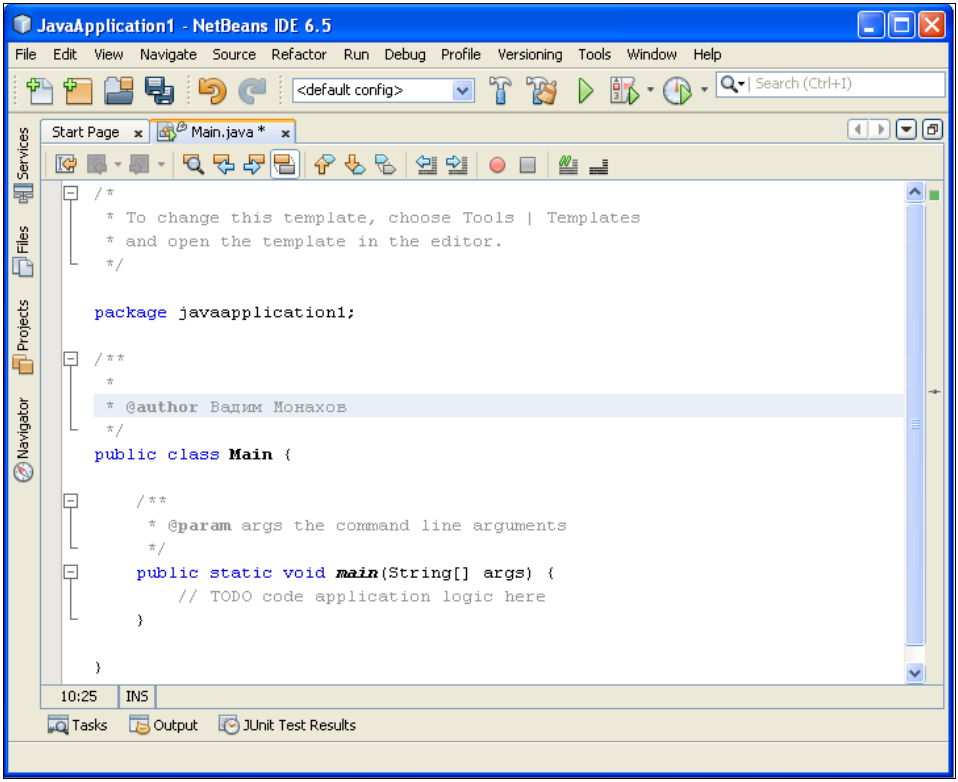


Рис. 2.22. Вид среды разработки при максимизации окна редактора

Необходимые окна легко раскрыть щелчком по соответствующему названию. Например, окно **Projects** иллюстрирует рис. 2.23.

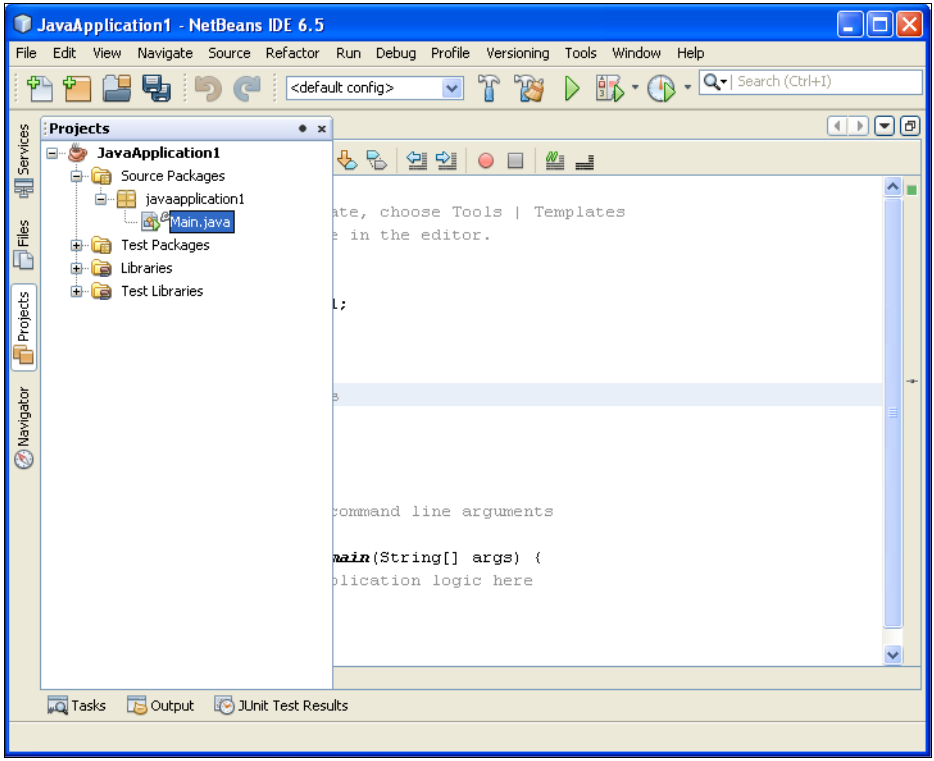


Рис. 2.23. Вид окна **Projects** при максимизации окна редактора

Щелчок по имени Projects в левой части окна сворачивает окно **Projects**, возвращая среду разработки к виду, показанному на рис. 2.22. А нажатие на иконку ▢ в правой верхней части окна редактора возвращает среду разработки к первоначальному виду, принятому по умолчанию.

Рассмотрим теперь структуру папок проекта NetBeans. По умолчанию головная папка проекта располагается в папке пользователя. В операционной системе Windows XP проект по умолчанию находится в папке C:\Documents and Settings\ИмяПользователя\.

Приведем дальнейшее расположение папок и файлов (листинг 2.3). Для наглядности имена главных папок выделены полужирным шрифтом, а имена вложенных папок и файлов записаны под ними и сдвинуты относительно них вправо. Папки и файлы, находящиеся на одном уровне вложенности, располагаются друг под другом с одинаковым сдвигом от левого края страницы.

Листинг 2.3. Структура папок и файлов

```
build
    classes
        javaapplication1
```

```
            Main.class
               ... .class
        META-INF
dist
    javadoc
    lib
    JavaApplication1.jar
    README.TXT
nbproject
src
    javaapplication1
        Main.java
            ... .java
            ... .form
    META-INF
test
build.xml
manifest.mf
```

Рассмотрим, что хранится в главных папках:

- **build** — скомпилированные файлы классов, имеющие расширение class;
- **dist** — файлы и папки, предназначенные для распространения как результат компиляции (модуль JAR-приложения или библиотеки, а также документация к нему — набор HTML-файлов в папке javadoc);
- **nbproject** — служебная информация по проекту;
- **src** — исходные коды классов. Кроме того, там же хранится информация об экранных формах (диалогах), если они есть. Эти формы будут видны на экране в виде окон с кнопками, текстом и т. п. Информация об экранных формах содержится в XML-файлах, имеющих расширение form;
- **test** — сопроводительные тесты, предназначенные для проверки правильности работы классов проекта. Тесты пишутся для проверки правильности работы проекта и составляющих его классов при изменениях и дополнениях, возникающих во время цикла жизни проекта.

Листинг 2.4 содержит перевод файла README.TXT, находящегося в папке dist (там же, где архив jar, предназначенный для распространения как файл приложения).

Листинг 2.4. Файл README.TXT из папки dist

```
==============================
ОПИСАНИЕ ВЫВОДА КОМПИЛЯЦИИ
==============================
Когда вы компилируете проект приложения Java, которое имеет главный класс, среда
разработки (IDE) автоматически копирует все файлы jar-архивов, указанные в classpath ваших
проектов, в папку dist/lib. Среда разработки также автоматически прибавляет путь к каждому
из этих архивов в файл манифеста приложения (MANIFEST.MF).


Чтобы запустить проект в режиме командной строки, зайдите в папку dist и наберите в режиме
командной строки следующий текст:
     java -jar "JavaApplication1.jar"
```

Чтобы распространять этот проект, заархивируйте папку dist (включая папку lib) и распространяйте zip-архив.

Замечания

* Если два jar-архива, указанные в classpath ваших проектов, имеют одинаковое имя, в папку lib будет скопирован только первый из них.

* Если в classpath указана папка с классами или ресурсами, ни один из элементов classpath не будет скопирован в папку dist.

* Если в библиотеке, указанной в classpath, также есть элемент classpath, то имеющиеся в нем элементы должны быть учтены в пути classpath времени выполнения проектов.

* Чтобы установить главный класс в стандартном проекте Java, щелкните правой кнопкой мыши в окне **Projects** и выберите **Properties**. Затем выберите **Run** и введите данные о названии класса в поле **Main Class**. Кроме того, вы можете вручную ввести название класса в элементе **Main-Class** манифеста.

2.6. Создание в NetBeans приложения Java с графическим интерфейсом

Как уже говорилось, экранной (диалоговой) формой называется область, которая видна на экране в виде окна с различными компонентами — кнопками, текстом, выпадающими списками и т. п.

Среды, позволяющие в процессе разработки приложения в интерактивном режиме помещать на формы компоненты и задавать их параметры, называются RAD-средами. RAD расшифровывается как Rapid Application Development (быстрая разработка приложений).

В NetBeans и других современных средах разработки такой процесс основан на объектной модели компонентов, поэтому он называется Объектно-ориентированным дизайном (Object-Oriented Design, OOD).

NetBeans является RAD-средой и позволяет быстро и удобно создавать приложения с развитым графическим пользовательским интерфейсом (Graphics User's Interface, GUI). Хотя языковые конструкции Java, позволяющие это делать, не очень просты, на начальном этапе работы с экранными формами и их элементами нет необходимости вникать в эти тонкости. Достаточно знать основные принципы работы с такими проектами.

С точки зрения автора изучение того, как создавать приложения с графическим интерфейсом, весьма важно для начинающих программистов, и это следует делать с самых первых шагов по изучению Java.

Во-первых, с самого начала осваивается создание полноценных приложений, которые пригодны для практического использования. Трудно месяцами изучать абстрактные концепции, и только став профессионалом, иметь возможность сделать нечто такое, что можно показать окружающим. Гораздо интереснее и полезнее сразу применять полученные знания на практике.

Во-вторых, такой интерфейс при решении какой-либо задачи позволяет лучше сформулировать, какие параметры нужно вводить, какие действия и в какой последовательности выполнять, и что, в конце концов, получится. И отобразить все это на экране: вводимым параметрам будут соответствовать пункты ввода текста, действиям — кнопки и пункты меню, результатам — пункты вывода текста.

В NetBeans 6.5 имеется заготовка GUI-приложения Desktop Application, но она слишком сложна для начального изучения. Поэтому для дальнейшей работы следует скопировать пример 02_06_ GUI_example с прилагаемого к книге компакт-диска. Сначала в папку с вашими проектами целиком перепишите с компакт-диска папку NetBeansProjects. В нашем случае ее необходимо скопировать в папку:

C:\Documents and Settings\User\Мои документы\NetBeansProjects.

Затем запустите среду NetBeans (если она не была запущена) и закройте имеющиеся открытые проекты (чтобы они не мешали). Это делается щелчком правой кнопки мыши по имени проекта с последующим выбором команды меню **Close** (закрыть) (рис. 2.24).

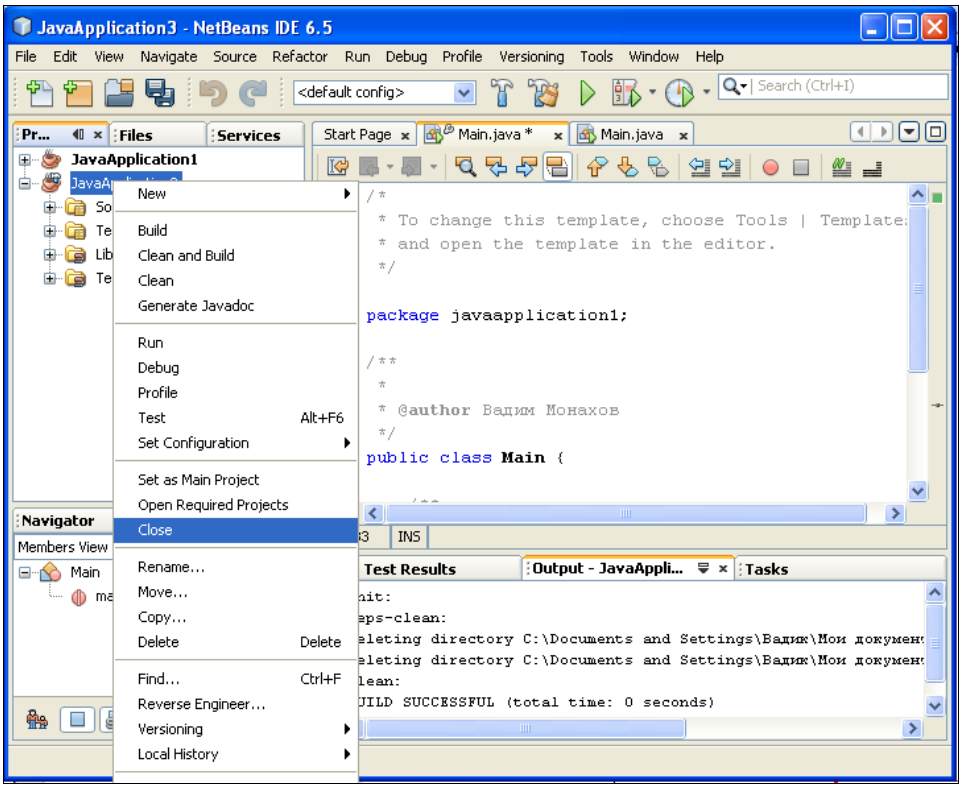


Рис. 2.24. Закрытие уже открытых проектов

Если необходимо закрыть несколько проектов (например, три), то следует выделить их имена и вызвать контекстное меню щелчком правой кнопки мыши в выделенной области, где и выбрать команду **Close 3 Projects**.

После этого можно открывать пример из файла. В главном меню выберите команду **File/Open Project** (либо на панели инструментов иконку с открывающейся папкой, либо нажмите комбинацию клавиш <Shift>+<Ctrl>+<O>). В открывшемся диалоговом окне выделите папку **02_06_GUI_example** (лучше в нее не заходить, а просто установить выделение на эту папку), после чего нажать кнопку **Open Project** (Открыть папку с проектом), выбрать тип файла **Project Folder** (рис. 2.25).

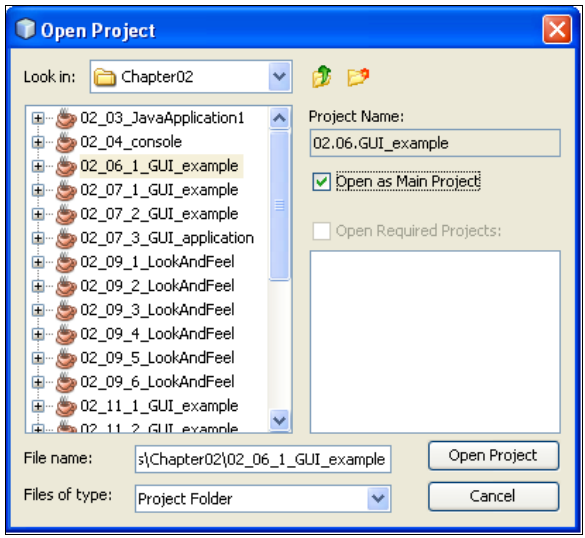


Рис. 2.25. Пример открытия проекта с существующим исходным кодом

Если не снимать флажок **Open as Main Project** (Открыть как главный проект), то проект автоматически становится главным. Можно выделить сразу несколько проектов, тогда при нажатии кнопки **Open Project** откроются все выбранные проекты — при этом флажок **Open as Main Project** автоматически снимается.

В окне редактора исходного кода появится текст, приведенный в листинге 2.5.

Листинг 2.5. Исходный код примера

```
/*
 * GUI_application.java
 *
*/

package examples;

/**
 *
 * @author  Вадим Монахов
 */

import javax.swing.*;

public class GUI_application extends javax.swing.JFrame {
    /**
     * Creates new form GUI_application
     */
    public GUI_application() {
        initComponents();
     }
```

```
    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    +Generated Code

    private void exitMenuItemActionPerformed(java.awt.event.ActionEvent evt)
    {
        System.exit(0);
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new GUI_application().setVisible(true);
            }
        });
    }

    // Variables declaration – do not modify
    private javax.swing.JMenuItem aboutMenuItem;
    private javax.swing.JMenuItem contentsMenuItem;
    private javax.swing.JMenuItem copyMenuItem;
    private javax.swing.JMenuItem cutMenuItem;
    private javax.swing.JMenuItem deleteMenuItem;
    private javax.swing.JMenu editMenu;
    private javax.swing.JMenuItem exitMenuItem;
    private javax.swing.JMenu fileMenu;
    private javax.swing.JMenu helpMenu;
    private javax.swing.JMenuBar menuBar;
    private javax.swing.JMenuItem openMenuItem;
    private javax.swing.JMenuItem pasteMenuItem;
    private javax.swing.JMenuItem saveAsMenuItem;
    private javax.swing.JMenuItem saveMenuItem;
    // End of variables declaration
}
```

Поясним некоторые его части. Указание пакета `examples`, в котором будет располагаться код класса приложения, нам уже знакомо. Декларация самого класса `GUI_application` в данном случае несколько сложнее, чем раньше:

```
public class GUI_application extends javax.swing.JFrame
```

Эта декларация означает, что задается общедоступный класс `GUI_application`, который является наследником класса `JFrame`, заданного в пакете `swing`, вложенном в пакет `javax`. Слово `extends` переводится как "расширяет" (класс-наследник всегда расширяет возможности класса-прародителя).

Общедоступный конструктор `GUI_application()` создает объект приложения и инициализирует все его компоненты методом `initComponents()`, автоматически генерируемым средой разработки и скрываемым в исходном коде узлом **+Generated Code**.

Развернув узел, можно увидеть реализацию этого метода, но изменить код нельзя. Мы не будем останавливаться на том, что в нем делается.

Далее следует закрытый (`private`) метод, который будет обсуждаться чуть позже:

```
private void exitMenuItemActionPerformed
```

Метод

```
public static void main(String[] args)
```

нам уже знаком — это главный метод приложения. Он является методом класса нашего приложения и автоматически выполняется Java-машиной при запуске приложения. В данном примере метод создает экранную форму приложения и делает ее видимой. Для понимания кода данного метода требуется предварительно изучить работу с интерфейсами *(глава 9)*, многопоточностью *(глава 10)* и безымянными классами *(глава 12)*. Поэтому мы не будем заниматься его разбором.

Далее следует область объявления компонентов-пунктов меню нашей формы. Она автоматически создается редактором экранных форм и недоступна для изменения в редакторе исходного кода.

После запуска приложения командой **Run/Run Main Project** (либо нажатием клавиши <F6>, либо нажатием иконки с зеленым треугольником) форма выглядит так, как показано на рис. 2.26 или 2.27. В ней уже имеется заготовка меню, которое способно разворачиваться и сворачиваться, и даже работает команда **Выход**, при запуске которой происходит выход из приложения.

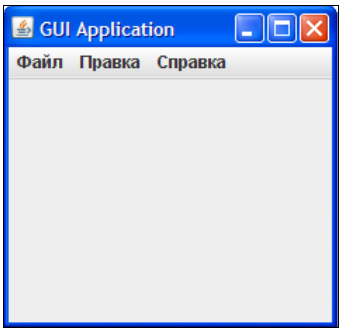


Рис. 2.26. Запущенное приложение

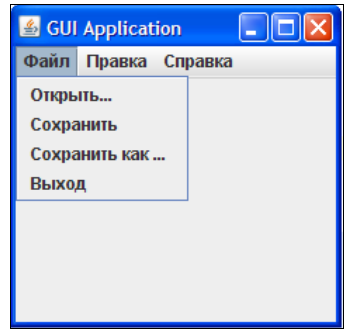


Рис. 2.27. Приложение с раскрытым меню

Именно за нажатие на этот пункт меню несет ответственность оператор `exitMenuItemActionPerformed`. При проектировании экранной формы он назначен в качестве *обработчика события* — подпрограммы, которая выполняется в результате наступления события (в нашем случае событие — выбор пункта меню **Выход**). Внутри обработчика `exitMenuItemActionPerformed` имеется всего одна строчка:

```
System.exit(0);
```

Эта команда вызывает прекращение выполнения метода `main` и выход из приложения с нулевым кодом завершения. Как правило, ненулевой код завершения возвращают при аварийном завершении приложения (чтобы по его значению можно было выяснить причины "вылета" программы).

2.7. Редактор экранных форм

Нажмем вкладку **Design** (Дизайн) в левой верхней части редактора исходного кода. При этом мы переключимся из режима редактирования исходного кода (активна вкладка **Source** (исходный код)) в режим редактирования экранной формы (активна вкладка **Design**), как показано на рис. 2.28.

Вместо исходного кода показывается внешний вид экранной формы и находящиеся на ней компоненты. Справа от окна, в котором отображается экранная форма в режиме редактирования, расположены окна **Palette** (Палитра) палитры компонентов и окно **Properties** (свойства) показа и редактирования свойств текущего компонента.

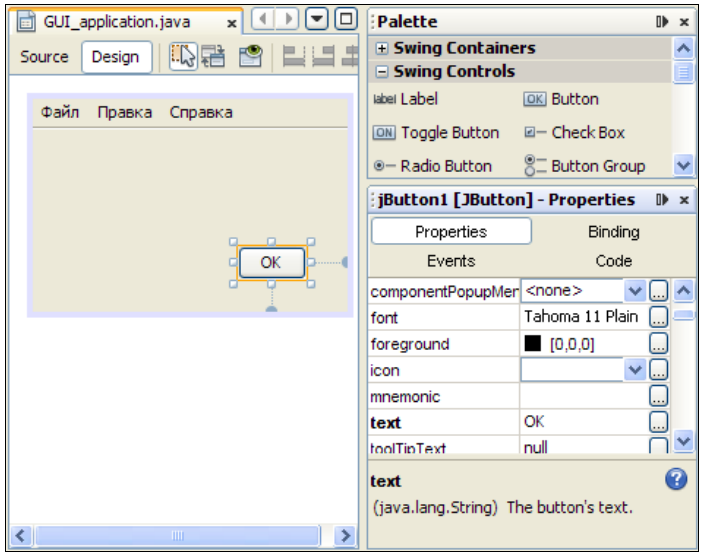


Рис. 2.28. Редактирование экранной формы

Свойство — это поле данных, которое после изменения значения может произвести какое-либо действие. Например, при изменении значения ширины компонента отрисовать на экране компонент с новой шириной. Обычное поле данных на такое не способно. Таким образом, свойство — это "умное поле данных".

Палитрой компонентов пользуются при выборе типа компонента, который нужен программисту для размещения на экранной форме. В NetBeans 6.5 компоненты базового набора графических компонентов `Swing` удобно разложены на разных страницах палитры:

- **Swing Containers** (контейнеры) — это компоненты, предназначенные для расположения на них других компонентов;

- **Swing Controls** (управляющие элементы) — компоненты для выполнения конкретных функций: вывода текста, редактирования текста, показа кнопки и т. п.;
- **Swing Menus** (меню) — компоненты для создания главных и контекстных (popup) меню формы;
- **Swing Windows** (окна) — компоненты для создания оконных диалогов (экранных форм, а также диалогов выбора);
- **AWT** — набор устаревших компонентов AWT, первоначально использовавшихся в Java;
- **Beans** — применяется преимущественно для компонентов, установленных пользователем;
- **Java Persistence** — технология взаимодействия с объектами, которые могут сохраняться после окончания работы приложения и загружаться "из внешнего мира" (например, из базы данных).

Рассмотрим, как добавить на нашу форму компонент типа `JButton` (сокращение от Java Button). Для этого развернем в палитре компонентов меню **Swing Controls** (управляющие элементы Swing), если оно было свернуто, щелкнем мышью по пункту **Button** на палитре и передвинем ее указатель в нужное место экранной формы. При попадании указателя мыши в область экранной формы на ней появляется кнопка стандартного размера, которая передвигается вместе с мышью. Щелчок в нужном месте формы приводит к тому, что кнопка остается в этом месте. Вокруг нее показываются рамка и маленькие квадратики, обозначающие, что наш компонент выделен. Для него осуществляется показ и редактирование свойств (**Properties**) и событий (**Events**) в окне **Properties**. По умолчанию в окне свойств сначала отображаются свойства (вкладка **Properties**), показ событий осуществляется при переходе на вкладку **Events.**

Кроме того, от выделенного компонента исходят линии, к которым идет привязка для задания положения компонента на форме.

По умолчанию надписи на компонентах задаются как имя типа, после которого идет номер компонента. Но вместо заглавной буквы, в отличие от имени типа, используется строчная. Поэтому первая кнопка будет иметь надпись **jButton1**, вторая — **jButton2** и т. д. Такие же имена будут приобретать автоматически создаваемые в исходном коде переменные, соответствующие кнопкам.

Изменить надпись на кнопке можно несколькими способами.

Во-первых, перейдя в окно **Properties**, изменив значение свойства `text` и нажав клавишу <Enter> для завершения ввода.

Во-вторых, изменив аналогичным образом свойство `label`.

В-третьих, сделав по нашей кнопке щелчок правой кнопкой мыши (т. е. вызвав контекстное меню), выбрав пункт **Edit Text** (Редактировать текст) и отредактировав текст.

Наконец, можно в окне **Properties** отредактировать текст не в однострочном поле ввода значений для свойств `text` или `label`, а открыв многострочный редактор путем нажатия на кнопку с тремя точками (находящуюся справа от пункта редактирования значения свойства) (рис. 2.29).

В старых версиях NetBeans можно было войти в режим редактирования текста непосредственно в области любого компонента, сделав по нему двойной щелчок, но в

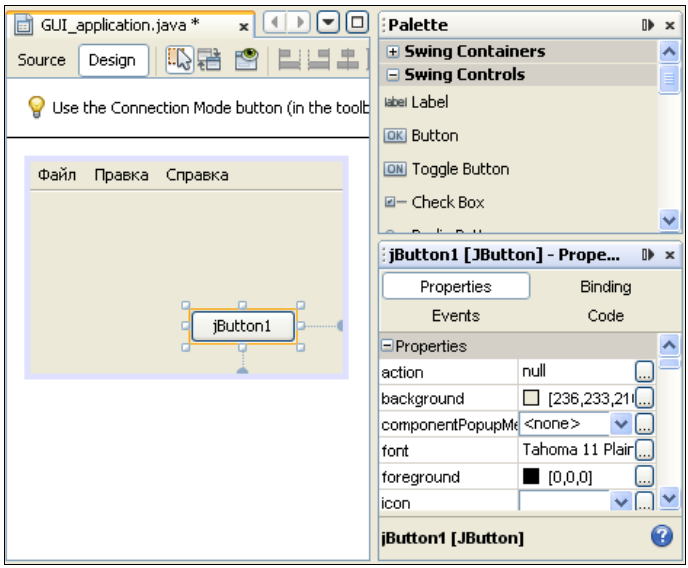


Рис. 2.29. Редактирование свойств компонента

NetBeans 6.5 это работает только для некоторых компонентов (например, для меток `JLabel` и многострочных пунктов редактирования текста `JTextArea`). А для кнопок `JButton`, пунктов `JTextField` и ряда других компонентов по двойному щелчку либо происходит переход в область обработчика события в исходном коде, либо вызывается диалог установки свойств акции, назначаемой компоненту. Про акции и этот диалог будет рассказываться в *главе 13*, посвященной созданию приложений с графическим интерфейсом (Desktop Applications). Тем не менее, для кнопок в режим редактирования текста можно войти повторным (не двойным!) щелчком в области текста кнопки. Первый щелчок приводит к выбору кнопки, а второй — к выделению текста на кнопке.

Введем на кнопке надпись **OK** и назначим эту кнопку для выхода из программы.

Размер компонента можно задать мышью (захватить за рамку и расширить либо сузить по соответствующим направлениям). Установка на новое место осуществляется перетаскиванием компонента мышью.

Получившийся результат будет примерно таким же, как в проекте 02_07_GUI_example1 на прилагаемом к книге компакт-диске.

Упомянутый ранее многострочный редактор текста не помогает сделать надпись на кнопке многострочной. Но этого можно добиться с помощью HTML-текста, о котором будет рассказано в *разд. 11.1*, посвященном языку HTML. Те читатели, которые уже знакомы с HTML, могут выполнить простейшее HTML-форматирование текста в любых компонентах `Swing`. Для этого достаточно в начале текста написать тег `<html>`, после которого следует HTML-форматированный текст. Отметим, что набор HTML-тегов, которые можно применять, очень ограничен, и их необходимо вводить в нижнем регистре.

Пояснение

В HTML тегом называется управляющая последовательность, заключенная в треугольные скобки. Она задает правила показа текста — разметку текста.

Введем текст (рис. 2.30):

```
"<html><p align=center>OK</p>Выход из программы"
```

После этого текст на кнопке будет выглядеть так, как показано на рис. 2.31.

В качестве еще одного примера добавим всплывающую подсказку (в Java она называется `toolTip`, а в традиции Microsoft — `hint`).

Для нашей кнопки введем текст: "Эта кнопка предназначена для выхода из программы" в поле, соответствующее свойству `toolTipText`.

Без применения HTML-форматирования подсказка также может быть только однострочной — символы перевода на новую строку при выводе подсказки игнорируются, даже если они заданы в строке программным путем. С использованием языка HTML перевод на новую строку осуществляется так: с помощью тега `<html>` в начале подсказки указываем, что далее идет HTML-текст, и вставляем тег `<br>` в тех местах, где нужен перевод текста на новую строку.

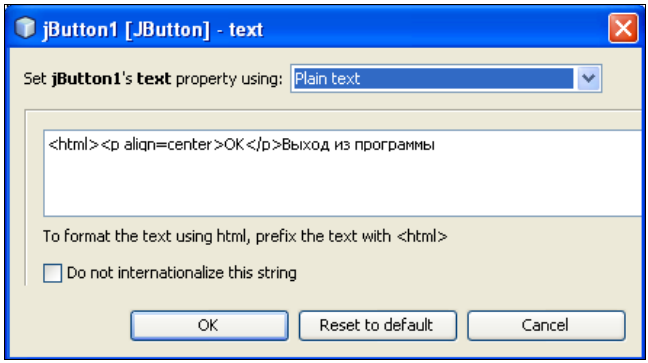


Рис. 2.30. Ввод текста на кнопку

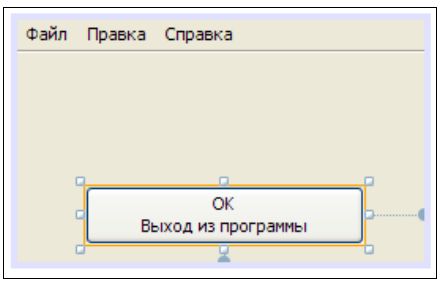


Рис. 2.31. Кнопка с HTML-форматированным текстом

Наконец, зададим действие, которое будет выполняться при нажатии на кнопку, — (`event handler`) *обработчик события* нажатия на кнопку. Для этого сначала выделим кнопку, после чего щелкнем по ней правой кнопкой мыши и в появившемся контекстном меню выберем пункт **Events/Action/actionPerformed** (рис. 2.32).

Events означает "события", **Action** — "действие", **actionPerformed** — "выполненное действие".

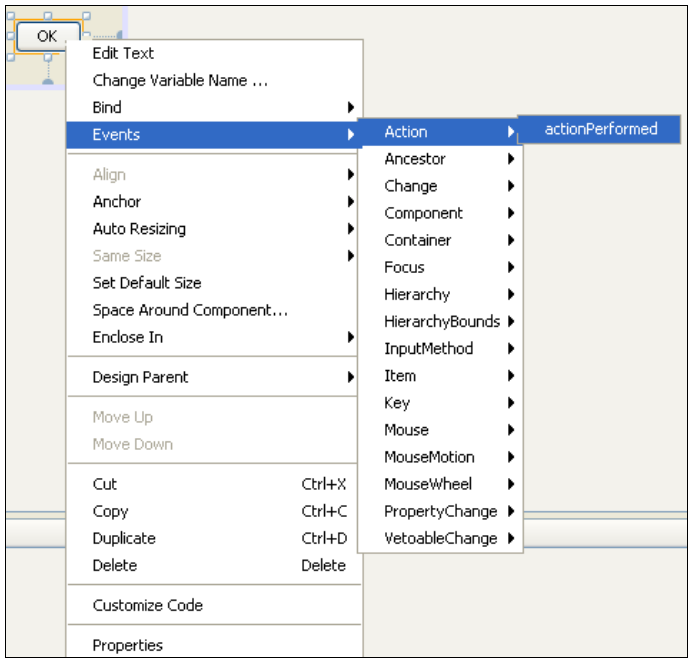


Рис. 2.32. Назначение обработчика события

После этого произойдет автоматический переход в редактор исходного кода, и там появится заготовка обработчика события:

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
// TODO add your handling code here:
    }
```

Аналогичный результат можно получить быстрее — двойным щелчком по кнопке.

Имеется и еще один вариант. После того как мы выделим кнопку в окне редактирования формы (**Design**), в окне **Inspector** показывается и выделяется имя этой кнопки (рис. 2.33). В нашем случае это **jButton1 [JButton]** (в квадратных скобках показывается тип компонента). Двойной щелчок по этому имени в окне инспектора приводит к переходу в исходный код назначенного обработчика события. Если же обработчик не назначен, то вызывается диалог назначения акции.

Рядом с обработчиком `jButton1ActionPerformed` будет расположен уже имеющийся обработчик события, срабатывающий при нажатии на пункт меню **Выход**:

```
private void exitMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}
```

Заменим в нашем обработчике события строку с комментарием на код, вызывающий выход из программы:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}
```

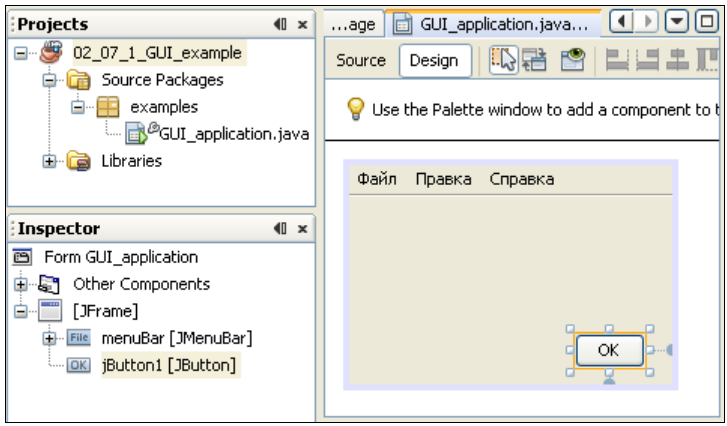


Рис. 2.33. Назначение обработчика события с помощью инспектора компонентов

Теперь после запуска нашего приложения подведение курсора мыши к кнопке приведет к появлению всплывающей подсказки, а нажатие на кнопку — к выходу из программы.

Получившийся результат показан в проекте 02_07_ GUI_example2 на прилагаемом к книге компакт-диске.

Часто встречается ситуация, когда необходимо вывести сообщение при наступлении какого-либо события (например, при нажатии на кнопку). В этом случае вызывают панель с сообщением:

```
javax.swing.JOptionPane.showMessageDialog(null,"Меня нажали");
```

Если классы пакета `javax.swing` импортированы, то префикс `javax.swing` при вызове не нужен.

2.8. Свойства компонентов

Некоторые свойства выделенного компонента (его размер, положение, текст) можно изменять непосредственно в области экранной формы. Однако большинство свойств просматривают и меняют в окне редактирования свойств. Оно состоит из двух столбцов: в левом показываются имена свойств, в правом — их значения. Значения, стоящие в правом столбце, во многих случаях могут быть отредактированы непосредственно в ячейках таблицы. Ввод заканчивается нажатием клавиши <Enter> или выходом из редактируемой ячейки, а отменить результаты неоконченного ввода можно нажатием клавиши <Esc>.

В правой части каждой ячейки имеется кнопка с изображением троеточия "..." — в современных операционных системах принято добавлять три точки в названия тех пунктов меню и кнопок, после нажатия на которые открывается диалоговое окно. В данном случае раскрывается окно специализированного редактора соответствующего свойства, если такой редактор существует.

Если требуется просматривать и редактировать большое количество свойств компонента, бывает удобнее щелкнуть правой кнопкой мыши по нужному компоненту и в появившемся контекстном меню выбрать пункт **Properties**. При этом откроется отдельное

окно редактирования свойств компонента. Можно держать открытыми одновременно произвольное количество таких окон.

Булевы свойства в колонке значений свойств показываются в виде кнопок выбора `checkbox` — квадратиков с возможностью установки флажка внутри. Если флажка нет, то значение свойства `false`, если есть — `true`.

В табл. 2.1 перечислены на примере кнопки `JButton` важнейшие свойства, которые можно устанавливать для компонентов (многие из них относятся и к другим компонентам).

***Таблица 2.1.** Важнейшие свойства, которые можно устанавливать для компонентов*

Название свойства	**Что оно задает**
Категория Properties (свойства)	
`action`	Объект типа `javax.swing.Action`, выполняющий действие при наступлении события `actionEvent`. В NetBeans 6.5 осуществлен переход от обработчиков событий `ActionPerformed` к обработчикам типа `Action` (про акции *см. главу 13*)
`background`	Цвет фона
`font`	Шрифт надписи на компоненте
`foreground`	Цвет надписи на компоненте
`icon`	Пиктограмма (небольшая картинка, иконка), которая рисуется на компоненте рядом с текстом. Устанавливается ссылка на объект типа Icon (*см. разд. 2.11, 13.3—13.4* и *13.7*)
`mnemonic`	Код клавиши-акселератора на клавиатуре (например, при вводе в это поле символа `q` наша кнопка активизируется и нажимается по комбинации <Alt>+<q>)
`text`	Текст (надпись) на компоненте
`toolTipText`	Всплывающая подсказка, появляющаяся через некоторое время при наведении курсора мыши на компонент
Категория Other Properties (другие свойства)	
`UIClassID`	Строка, задающая имя класса, определяющего внешний вид компонента в технологии Look and Fill *(см. разд. 2.9)*
`actionCommand`	Строка, посылаемая в акцию-обработчик события при наступлении соответствующего события, например, нажатия кнопки *(см. главу 13)*. Обработчик может выполнять те или иные действия в зависимости от переданной строки
`alignmentX`	Предпочтительное горизонтальное смещение компонента
`alignmentY`	Предпочтительное вертикальное смещение компонента
`autoscrolls`	Наличие скроллеров — полос прокрутки, возникающих, когда видимый размер компонента меньше, чем нужный для вывода текста или других элементов этого компонента
`baselineResizeBehavior`	Перечисление *(см. разд. 3.8)*, указывающее, как должна меняться базовая линия компонента при изменении его размера (обычно базовая линия проходит по центру компонента и относительно нее выравнивается текст и изображения)
`border`	Тип рамки вокруг компонента

Таблица 2.1 *(продолжение)*

Название свойства	Что оно задает
borderPainted	Рисуется ли рамка вокруг компонента
buttonGroup	Позволяет объединять кнопки в группы с помощью назначения этому полю имени компонента типа ButtonGroup. Важно для объединения в группы кнопок типа ToggleButton и RadioButton
componentPopupMenu	Позволяет назначать контекстное меню, появляющееся по нажатию правой кнопкой мыши в области компонента
contentAreaFilled	Указывает, имеется ли заполнение цветом внутренней области компонента (для кнопок оно создает эффект трехмерности, без заполнения кнопка выглядит плоской)
defaultCapable	Указывает, способна ли кнопка быть кнопкой по умолчанию: при нажатии клавиши <Enter> автоматически происходит нажатие кнопки по умолчанию (такая кнопка на экранной форме должна быть единственной)
disabledIcon	Картинка, которая рисуется на компоненте рядом с текстом в случае, когда компонент недоступен (disabled) (т. е. когда его свойство enabled установлено в false)
disabledSelectedIcon	Картинка, которая рисуется на компоненте рядом с текстом в случае, когда выделенный компонент недоступен (disabled) (т. е. когда его свойство enabled установлено в false)
displayedMnemonicIndex	Код символа-акселератора, назначенного компоненту (т. е. соответствующее этому символу число типа char)
doubleBuffered	Обеспечивает режим двойной буферизации для отрисовки на экране при установке флажка для данного свойства (оно при этом приобретает значение true). Этот режим следует задавать только для компонентов, на которых будет проводиться вывод динамических графических данных. Двойная буферизация устраняет мерцание при динамическом выводе графики, но потребляет в два раза больше ресурсов
enabled	Указывает, доступен ли компонент. По умолчанию все создаваемые на форме компоненты доступны. Недоступные компоненты рисуются более блеклыми красками и на них нельзя установить фокус (т. е. перейти на них мышью или с помощью горячих клавиш)
focusCycledRoot	Указывает, является ли компонент-контейнер корневым для цикла навигации при нажатии на клавиатуре клавиш навигации. Если компонент корневой, навигация осуществляется только внутри него между дочерними компонентами. См. также далее информацию о свойстве nextFocusableComponent
focusTraversalPolicy	Задает способ перемещения фокуса от компонента к компоненту (навигации) при нажатии на клавиатуре клавиш навигации. Способ зависит от того, ссылка на объект какого типа назначена данному свойству: ContainerOrderFocusTraversalPolicy или DefaultFocusTraversalPolicy — по порядку компонентов в контейнере; SortingFocusTraversalPolicy — сортирует компоненты с помощью метода-компаратора; LayoutFocusTraversalPolicy — сортирует компоненты в зависимости от их положения и размера. См. также далее информацию о свойстве nextFocusableComponent

Таблица 2.1 *(продолжение)*

Название свойства	Что оно задает
focusTraversalPolicyProvider	Булево свойство, указывающее, что свойство focusTraversalPolicy именно этого компонента-контейнера будет использовано в качестве провайдера политики навигации
focusPainted	Указывает на необходимость перерисовки компонента при получении им фокуса. Если пользовательские средства отрисовки компонента не используются, то ни на что не влияет
focusable	Указывает, способен ли компонент захватывать фокус. Свойство важно для компонентов редактирования текста и компонентов-контейнеров
hideActionText	Булево свойство, предназначенное для управления компонентом, которому назначена акция *(см. главу 13)*. Если значение равно false (по умолчанию), при выполнении акции компонент начинает отображать текст, переданный в акцию (свойство actionCommand). В противном случае этот текст не отображается
horizontalAlignment	Горизонтальное выравнивание текста и/или картинки, нарисованной на компоненте. Имеет значения: CENTER (по центру — значение по умолчанию); LEADING (ведущий, лидирующий); LEFT (по левому краю); RIGHT (по правому краю); TRAILING ("прицепленный сзади"). Для горизонтально пишущегося текста разницы между LEADING и LEFT, а также между RIGHT и TRAILING нет
horizontalTextPosition	Горизонтальное положение текста по отношению к картинке, нарисованной на компоненте. Имеет значения: CENTER (картинка по центру текста); LEADING, LEFT (текст слева от картинки); RIGHT, TRAILING (текст справа от картинки)
verticalAlignment	Вертикальное выравнивание текста и/или картинки, нарисованной на компоненте. Имеет значения: CENTER (по центру); TOP (расположение по верху компонента); BOTTOM (расположение по низу компонента)
verticalTextPosition	Вертикальное выравнивание текста по отношению к картинке, нарисованной на компоненте. Имеет значения: CENTER (картинка по центру текста); TOP (картинка по верху текста); BOTTOM (картинка по низу текста)
iconTextGap	Расстояние (зазор) в пикселах (точках экрана) между картинкой, нарисованной на компоненте (если она есть), и текстом, написанном на компоненте

Таблица 2.1 *(продолжение)*

Название свойства	Что оно задает
`inheritsPopupMenu`	Наследует контекстное меню. Если установить это поле в `true` (установить флажок в редакторе свойств), то компонент унаследует контекстное меню от того группирующего компонента (контейнера), на котором он расположен
`inputVerifier`	Ссылка на объект типа `javax.swing.InputVerifier`, предназначенный для проверки правильности ввода текста в компонент
`insets`	Задает для компонента-контейнера отступы сверху, слева, снизу и справа, которые не должны быть заняты дочерними компонентами
`label`	Практически то же, что и Text, но имеется только для компонентов с неизменяемым текстом (реально это ссылка на объект типа java.awt.Label).
`maximumSize`	Максимально допустимый размер компонента — в случае, когда его размер может изменяться во время работы программы. Важен для форм, у которых свойство `resizable` установлено в `true`
`minimumSize`	Минимально допустимый размер компонента (в случае, когда его размер может изменяться во время работы программы). Важен для форм, у которых свойство `resizable` установлено в `true`
`model`	Ссылка на объект модели представления данных для компонента. Обычно создание специальных объектов-моделей необходимо только для таблиц, и то в NetBeans имеются средства создания таких объектов автоматически — в режиме визуального проектирования таблицы
`multiclickThreshold`	Задает для компонента порог (в миллисекундах) для двойного щелчка мыши
`name`	Строка с именем компонента. Это просто строковое значение, не имеющее ничего общего с именем компонента в Java-коде
`nextFocusableComponent`	Ссылка на компонент, который должен быть следующим при нажатии на клавишу навигации <Tab> на клавиатуре. По умолчанию компонентам могут быть не назначены такие ссылки, и навигация будет отсутствовать. Это зависит от стиля LookAndFeel — *см. разд. 2.9*. При стиле System под Windows навигация присутствует, при стиле Metal — отсутствует
`opaque`	Непрозрачный. При снятии флажка, установленного по умолчанию, компонент делается прозрачным (эта возможность появилась только начиная с Windows 2000). Не следует пользоваться прозрачностью без особой необходимости — она занимает во время работы программы много ресурсов процессора
`paintingForPrint`	Булево свойство. Если задать его значение равным `true` (установить флажок), то при печати родительского компонента-контейнера методом `print` будет осуществлена печать данного дочернего компонента (вызван его метод `print`)
`preferredSize`	Предпочтительный размер компонента (задается размер в пикселах по горизонтали и по вертикали)

Таблица 2.1 *(окончание)*

Название свойства	Что оно задает
pressedIcon	Картинка, которая рисуется на компоненте рядом с текстом в случае, когда кнопка нажата (свойство относится только к кнопкам)
requestFocusEnabled	Булево свойство, задающее способность компонента захватывать фокус после щелчка по нему мышью. Не влияет на захват фокуса при навигации с помощью клавиатуры
rolloverEnabled	Булево свойство, задающее способность компонента менять цвет при наведении на него курсора мыши
rolloverIcon	Картинка, которая рисуется на компоненте рядом с текстом при наведении на компонент курсора мыши
rolloverSelectedIcon	Картинка, которая рисуется на выделенном компоненте рядом с текстом при наведении на этот компонент курсора мыши. Данное свойство наиболее важно для пунктов меню и элементов списков
selected	Булево свойство. Если задать его значение равным true (установить флажок), компонент будет отмечен как выделенный. Данное свойство наиболее важно для радиокнопок, элементов независимого выбора (JCheckBox), кнопок с фиксацией (JToggleButton), пунктов меню, элементов списков
selectedIcon	Картинка, которая рисуется на выделенном компоненте рядом с текстом
verifyInputWhenFocusTarget	Булево свойство, задающее необходимость проверки компонентом введенного в него текста при получении компонентом фокуса. Если компоненту не назначен объект inputVerifier, данное свойство ни на что не влияет

О свойствах, относящихся к категории **Layout** (Менеджер размещения), будет рассказано в *разд. 2.15*. Отметим, что набор свойств компонента зависит от менеджера размещения для компонента-контейнера, на котором располагается этот компонент. Указанные свойства относятся к менеджеру, используемому по умолчанию (**Free Design**).

Также полезно учесть, что имена свойств, значения которых были изменены пользователем, показываются в окне **Properties** полужирным шрифтом.

У экранных форм, как особых компонентов, имеются несколько дополнительных свойств, которые полезно знать (табл. 2.2).

Таблица 2.2. *Дополнительные свойства экранных форм*

Название свойства	Что оно задает
defaultCloseOperation	Задает операцию, которая выполняется по умолчанию при закрытии формы. По умолчанию это HIDE (сделать форму невидимой, но не удалять). Также возможны значения EXIT_ON_CLOSE (выход из приложения после закрытия формы), DO_NOTHING (ничего не делать), DISPOSE (удаление формы после закрытия)
title	Задает заголовок формы

Таблица 2.2 *(окончание)*

Название свойства	Что оно задает
Категория Other Properties (другие свойства)	
`alwaysOnTop`	Булево свойство, задающее способность формы всегда находиться поверх других форм в раскрытом окне операционной системы
`alwaysOnTopSupported`	Булево свойство, доступное только для чтения, показывающее, поддерживает ли операционная система управление формами с помощью свойства `alwaysOnTop`
`bounds`	Размер формы задается пиксельными координатами левого верхнего угла и правого нижнего угла формы. Пиксел (светящаяся точка экрана), соответствующий верхнему левому краю экрана, имеет координаты [0,0]. Следующий пиксел по горизонтали имеет координату [1,0], а по вертикали [0,1]. Форма, расположенная в левом верхнем углу и имеющая ширину 200 пикселов и высоту 100 пикселов, будет иметь границы [0,0,199,99]. Значение `bounds` игнорируется в случае, если задаваемые им размеры выходят за пределы, определяемые свойствами `minimumSize` или `maximumSize`
`cursor`	Задает вид курсора мыши при его нахождении над любыми компонентами формы. Например, при установке значения `Crosshair Cursor` он будет иметь вид вертикального крестика. Обычно значение этого свойства меняют динамически во время работы программы. Чаще всего — на `Hand Cursor` (в виде руки), обозначая доступность какого-либо действия с компонентом, над которым находится курсор, или на `Wait Cursor` — ожидание в виде песочных часов
`locationByPlatform`	Когда отсутствует программная установка положения формы, а свойство имеет значение `true` (по умолчанию `false`), то при показе форма устанавливается в позицию, задаваемую операционной системой. Особенно хорошо видно влияние этого свойства в случае, когда несколько раз запустили одно и то же приложение. Например, Windows располагает окна приложений с некоторым сдвигом друг относительно друга, чтобы было видно несколько запущенных программ
`sizable`	Позволяет разрешать (по умолчанию) или запрещать пользователю во время работы программы изменять мышью размер формы
`state`	Задает состояние окна: минимизированное (значение 1) или нормальное (значение 0 или любое другое, отличное от 1)

2.9. Внешний вид приложения — технология Look and Feel

На этапе редактирования приложения внешний вид его компонентов соответствует платформе — стандартному виду приложений используемой операционной системы. Однако после запуска он становится совсем другим, поскольку по умолчанию все приложения Java показываются в платформо-независимом виде (стиль Metal). Пример 02_09_LookAndFeel1 на компакт-диске (рис. 2.34).

Кроме того, наше приложение появляется в левом верхнем углу экрана, а хотелось бы, чтобы оно появлялось в центре.

Для того чтобы показать приложение в платформо-ориентированном виде (т. е. в том виде, который использует компоненты и настройки операционной системы) (рис. 2.35), требуется изменить код конструктора приложения, вставив перед вызовом метода `initComponents` задание типа пользовательского интерфейса (User's Interface, сокращенно UI), листинг 2.6, пример 02_9_LookAndFeel2 на компакт-диске.

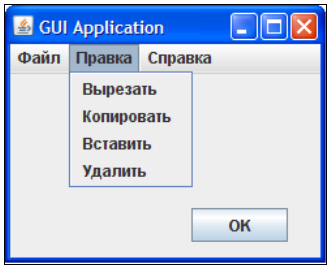


Рис. 2.34. Внешний вид запущенного приложения с платформо-независимым пользовательским интерфейсом, задаваемым по умолчанию

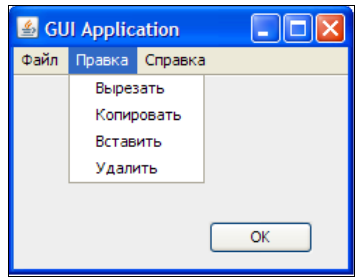


Рис. 2.35. Внешний вид запущенного приложения с платформо-ориентированным пользовательским интерфейсом в операционной системе Windows XP

Листинг 2.6. Код конструктора приложения

```
import javax.swing.*;
import java.awt.*;

...

public GUI_application() {

  try{
     UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
     }catch(Exception e){}

  initComponents();

  Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
  Dimension frameSize = getSize();
  setLocation(new Point( (screenSize.width-frameSize.width)/2,
                  (screenSize.height-frameSize.width)/2 )
       );
}
```

Код, следующий после вызова `initComponents()`, предназначен для установки окна приложения в центр экрана. Хотя, возможно, более правильным будет просто установить в `true` свойство формы `locationByPlatform`.

Имеется возможность задания еще одного платформо-независимого вида приложения — в стиле Motif (рис. 2.36), используемого в операционной системе Solaris (пример 02_09_LookAndFeel3 на компакт-диске).

Для установки такого вида вместо вызова:

```
UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName()
```

следует написать:

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

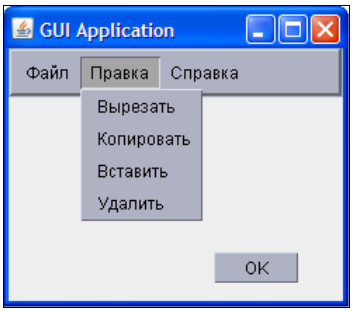


Рис. 2.36. Внешний вид запущенного приложения с платформо-независимым пользовательским интерфейсом в стиле Motif

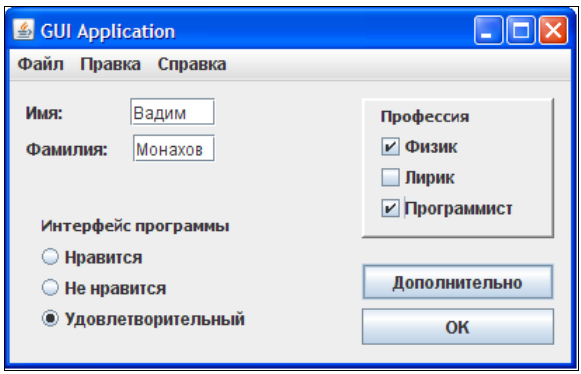


Рис. 2.37. Платформо-независимый стиль Metal

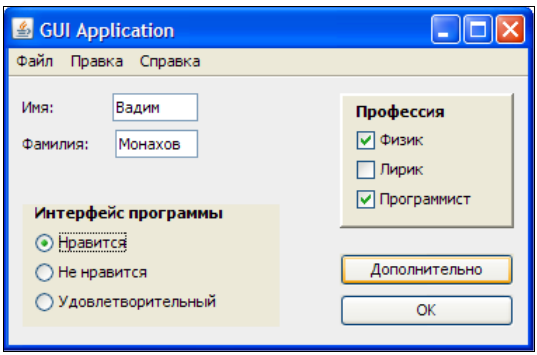


Рис. 2.38. Стиль Windows XP

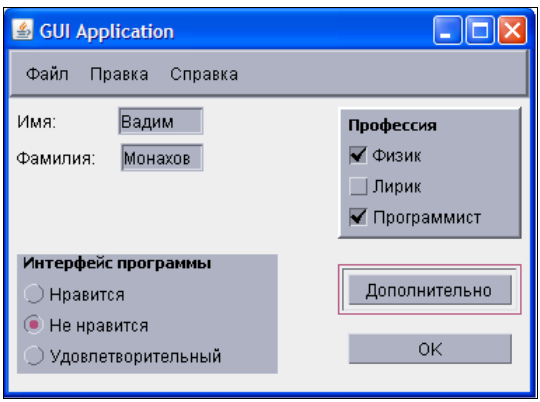


Рис. 2.39. Стиль Motif

Использованные конструкции станут понятны читателю после изучения дальнейших разделов учебного пособия.

Приведенные примеры выглядят по-разному, но все-таки не дают хорошего представления о стилях оформления GUI для разных платформ. Для того чтобы стало более понятно, как выглядят реальные приложения, приведем еще варианты внешнего вида экранов (примеры 02_09_LookAndFeel4 — 02_09_LookAndFeel6 на компакт-диске) (рис. 2.37—2.39).

2.10. Ведение проектов

Чтобы не запутаться в разных проектах и их версиях (особенно с учетом того, что учебные проекты часто бывает необходимо переносить с одного компьютера на другой), следует серьезно отнестись к ведению проектов. Автором в результате многолетней работы с разными языками и средами программирования выработана следующая система (откорректированная применительно к среде NetBeans).

- Под каждый проект создается папка с названием проекта. Будем называть ее папкой архива для данного проекта. Названия папок могут быть русскоязычными, как и имена приложений и файлов.
- При создании нового проекта среда разработки предлагает ввести имя папки, где его хранить (следует указать имя папки архива). Кроме того, предлагается ввести имя проекта. Это имя будет использовано средой NetBeans как для создания папки проекта, так и для названия вашего приложения. Для того чтобы облегчить работу с вашим приложением в разных странах, рекомендуется делать это название англоязычным. В папке проекта среда разработки автоматически создаст систему вложенных папок проекта и сформирует все его файлы. Структура папок проектов NetBeans была описана ранее.
- Если берется проект с существующим исходным кодом, то его папка копируется в папку нашего архива либо вручную, либо выбором соответствующей последовательности действий в мастере создания проектов NetBeans.
- При получении сколько-нибудь работоспособной версии проекта следует делать его архивную копию. Для этого в открытом проекте в окне **Projects** достаточно щелкнуть правой кнопкой мыши по имени проекта, и в появившемся контекстном меню выбрать пункт **Copy**. Откроется диалоговая форма (рис. 2.40), в которой предлагается автоматически образованное имя копии (к первоначальному имени проекта добавляется подчеркивание и номер копии — для первой копии это _1, для второй _2 и т. д.). Причем головная папка архива по умолчанию остается той же, что и у первоначального проекта (что очень удобно, поскольку дает возможность создавать копию всего тремя щелчками мыши без набора чего-либо с клавиатуры.

Скопированный проект автоматически появляется в окне **Projects**, но не становится главным (т. е. вы продолжаете работать с прежним проектом, и все его открытые окна сохраняются). Можно сразу закрыть новый проект. Напомним, что для этого правой кнопкой мыши следует щелкнуть по его имени, и в появившемся контекстном меню выбрать пункт **Close**.

Для чего нужна такая система ведения проектов? Дело в том, что у начинающих программистов имеется обыкновение разрушать результаты собственного труда. Они раз-

вивают проект, не сохраняя архивов. Доводят его до почти работающего состояния, после чего еще немного усовершенствуют, затем еще, а потом все перестает работать. А поскольку они вконец запутываются, то восстановить работающую версию уже нет возможности. И им нечего предъявить преподавателю или заказчику!

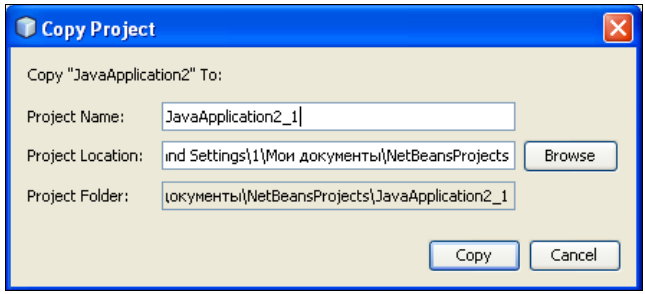


Рис. 2.40. Создание рабочей копии проекта

Поэтому следует приучиться копировать в архив все промежуточные версии проекта, более работоспособные, чем уже сохраненные. В реальных проектах трудно запомнить все изменения, сделанные в конкретной версии, и (что еще важнее) все взаимосвязи, вызвавшие эти изменения. Поэтому даже опытным программистам время от времени приходится констатировать: "Ничего не получается!", после чего восстанавливать версию, где еще не было тех нововведений, которые привели к путанице. Кроме того, часто бывает, что новая версия в каких-то ситуациях работает неправильно. И приходится возвращаться на десятки версий назад в поисках той, где не было таких ошибок. А затем внимательно сравнивать работу двух версий, выясняя причину неправильной работы более поздней версии. Или убеждаться, что все предыдущие версии также работали неправильно, просто ошибку не замечали.

Изложенная методика работы с проектами хороша только для ведения учебных проектов. В промышленном программировании основу совместной разработки проектов составляет система поддержки версий. В NetBeans существует встроенный механизм взаимодействия с системами поддержки версий программ, разработанными внешними производителями. Данный механизм, в частности, может быть использован для подключения к NetBeans наиболее известных бесплатных систем поддержки версий (пункт меню **Versioning**) — CVS, SVN (то же, что Subversion) и Mercurial (на ее основе разрабатываются, например, Mozilla, OpenJDK и NetBeans).

Еще один полезный совет — ненужные окна в проектах можно закрывать. Также можно менять размер окон и их взаимное расположение. На компьютерах с небольшим экраном это может оказаться очень полезно. Правда, иногда в результате таких экспериментов случайно закрываются не только лишние, но и очень нужные окна, например, окно инспектора объектов. Как его вернуть обратно?

Для этого служит меню **Window** (окно) (рис. 2.41).

Аналогично, если было закрыто окно свойств компонентов, его можно вернуть на экран либо через пункт меню **Window/Properties**, либо с помощью комбинации клавиш <Ctrl>+<Shift>+<7>. Если была закрыта палитра компонентов, ее можно вернуть на экран либо через пункт меню **Window/Palette**, либо комбинацией клавиш <Ctrl>+ +<Shift>+<8>.

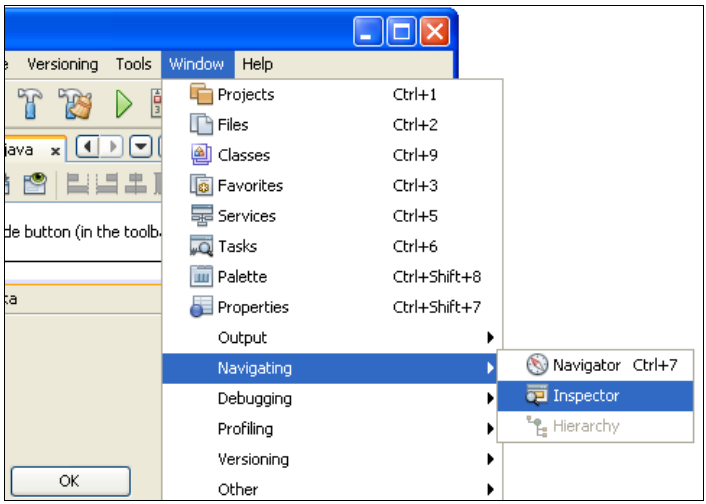


Рис. 2.41. Меню **Window** (возврат на экран инспектора компонентов)

2.11. Редактирование меню экранной формы

Имеются случаи, когда процесс редактирования компонентов несколько отличается от описанного ранее. Например, для главного меню экранной формы и для контекстного меню. В NetBeans 6.5 процесс редактирования этих компонентов очень удобен — с точки зрения автора даже лучше, чем в известных RAD-средах Delphi и Visual Basic.

Рассмотрим, как изменить текст пунктов меню формы с русского языка на английский. Для этого откроем наш проект (например, 2_11_GUI_example1), сделаем его копию и закроем первоначально открытый проект. Если щелкнуть мышью по какому-либо заголовочному пункту (item) меню, раскроются вложенные пункты меню. Причем так можно открывать элементы меню любого уровня вложенности. Открытые элементы меню закрываются только при щелчке мышью по другим заголовкам меню или вне меню. Текст любого пункта меню легко изменить. Для этого достаточно сделать двойной щелчок по соответствующему пункту, отредактировать текст и нажать клавишу <Enter>.

Изменить место какого-нибудь пункта меню можно путем простого перетаскивания мышью.

При щелчке мышью по какому-либо пункту меню в окне редактора свойств появятся значения свойств этого пункта. С помощью редактора свойств можно менять не только текст пунктов меню, но и другие свойства этого пункта.

Таким образом изменим названия: "Файл" на "File", "Правка" на "Edit", "Справка" на "Help" (рис. 2.42). Для того чтобы без компиляции и запуска программы посмотреть, как будет выглядеть наша экранная форма, можно нажать иконку **Preview Design** (третья по счету после вкладки **Design** в окне редактирования экранной формы — с изображением глаза, расположенного над формой).

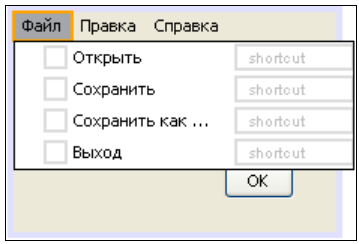


Рис. 2.42. Редактирование пунктов меню

В открытых подменю рядом с пунктами меню видны надписи **shortcut** (сокращенный путь). Это информация о специальных комбинациях клавиш — горячих клавишах. Следует отличать их от клавиш-акселераторов.

Горячие клавиши (Hot Keys, в NetBeans они названы акселераторами, Accelerators) — это отдельные клавиши или комбинации клавиш, которые срабатывают непосредственно при нажатии на клавиатуре заданной комбинации и предназначены для дублирования пунктов меню без навигации по меню.

Обычно в качестве горячих используются функциональные клавиши <F1>, <F2> и т. д., а также сочетание обычных клавиш со специальными клавишами <Ctrl>, <Alt> или <Shift>.

Замечание

В англоязычной традиции понятия горячих клавиш (Hot Keys) и акселераторов (Accelerator Keys) обычно не различают. В русскоязычной традиции акселератором принято называть клавишу, обеспечивающую ускоренный доступ к пункту меню с помощью нажатия этой клавиши при нажатой клавише <Alt>. Соответствующая буква в строке меню изображается подчеркнутой.

В нашем случае зададим для пункта **Файл/Сохранить** в качестве горячих клавиш комбинацию <Ctrl>+<S>, а в качестве обработчика выбора этого пункта меню — вызов диалога с сообщением о сохранении файла, чтобы было понятно, что пункт меню сработал.

Задать комбинацию можно путем двойного щелчка мышью в области, помеченной как **shortcut**. При этом появляется диалоговая форма (рис. 2.43), в которой можно установить нужные сочетания клавиш.

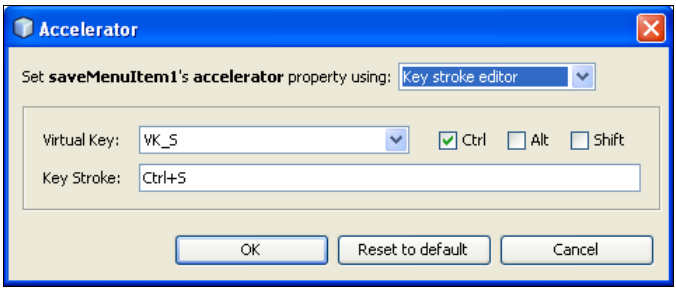


Рис. 2.43. Задание горячих клавиш

В поле **Key Stroke** (нажатая клавиша) следует ввести латинскую букву `S` (в любом регистре и на любом языке — учитывается только аппаратный SCAN-код нажатой клавиши). В результате в поле **Virtual Key** (виртуальная клавиша) код клавиши автоматически преобразуется в **VK_S**. После этого следует установить флажок **Ctrl** и нажать кнопку **OK**.

Акселераторы — это клавиши, ускоряющие навигацию по пунктам меню. Они никак не связаны с горячими клавишами (напомним про путаницу с англоязычной терминологией) и могут назначаться для пункта меню заданием символа-акселератора в свойстве `mnemonic` с помощью инспектора объекта. Обычно в качестве акселератора задают одну из букв пункта меню, при этом во время ускоренной навигации клавиши-акселераторы оказываются подчеркнутыми. Так, если заголовочному пункту главного меню назначен акселератор, то переход в режим ускоренной навигации происходит при нажатии на клавишу <Alt> и отпускании ее. В отличие от большинства программ для MS Windows нажатие на клавишу-акселератор при нажатой клавише <Alt> не приводит к ускоренной навигации. После отпускания клавиши <Alt> буква, соответствующая акселератору, остается подчеркнутой, и нажатие на клавиатуре этой буквы приводит к ускоренной навигации. В нашем случае нажатие на клавишу <Ф> или <ф> приведет к открытию вложенного меню с пунктами **Открыть**, **Сохранить**, **Сохранить как...**, **Выход** (рис. 2.44). После чего нажатие на русское <С> или <с> приведет к вызову пункта меню **Сохранить**, а нажатие на <В> или <в> — к выходу из программы.

Напомним, что в пунктах меню, выбор которых сопровождается открытием диалогов, принято в конце изображать три точки.

Существует еще один способ редактирования вложенных пунктов меню и других элементов экранной формы. В режиме дизайна при щелчке в области экранной формы в левом нижнем окне (**Inspector** — инспектор компонентов) среды разработки появляется список компонентов экранной формы. Инспектор позволяет просматривать деревья вложенности различных элементов проекта. Обратите внимание, что на рис. 2.45 и 2.46 подсвечена вкладка **Inspector** (вкладка **Navigator** не имеет отношения к тому, что показывается в окне).

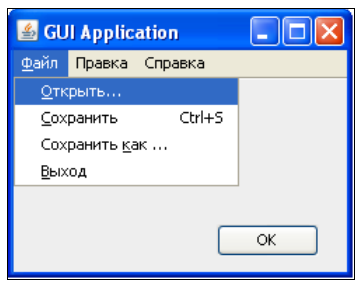


Рис. 2.44. Вид пунктов меню с акселераторами и горячей клавишей при работе приложения

Сама экранная форма является экземпляром класса `JFrame` (от Java Frame — окно, кадр, предоставляемое языком Java). В окне инспектора после схематического изображения компонента и имени соответствующей ему переменной в квадратных скобках указывается тип компонента. Развернем узел для нашей формы типа **JFrame**, а также вложенные узлы **menuBar1** типа **JMenuBar** и **fileMemu1** типа **JMenu** (рис. 2.45).

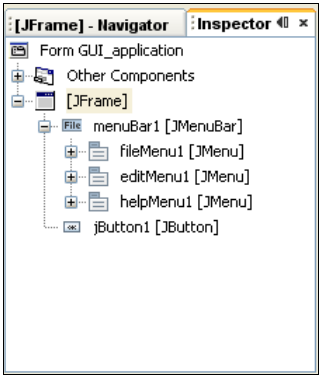


Рис. 2.45. Окно инспектора компонентов

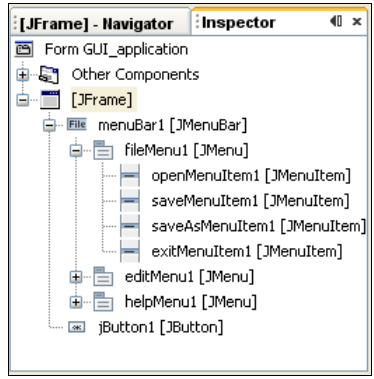


Рис. 2.46. Развернутое дерево вложенности

Мы увидим имена переменных, соответствующих всем пунктам меню, вложенным в файловое меню: **openMenuItem1**, **saveMenuItem1**, **saveAsMenuItem1**, **exitMenuItem1** (рис. 2.46). Щелчок по имени **openMenuItem1** в окне инспектора компонентов приведет к тому, что в окне редактирования свойств появятся значения свойств данного пункта меню. В поле **Text** заменим слово **Open** на **Открыть**. Затем перейдем на пункт **saveMenuItem1**, и т. д. В результате получим экранную форму с пунктами меню на русском языке.

Удаление пунктов меню в NetBeans 6.5 производится так же, как и любых других элементов экранной формы. Если нам нужно удалить какой-либо пункт меню (в том числе заголовочный, содержащий подменю), то достаточно в режиме дизайна (визуального проектирования) выделить этот пункт меню мышью и нажать клавишу <Delete>. Либо можно правой кнопкой мыши вызвать контекстное меню и выбрать в нем пункт **Delete**. Также можно удалять пункты меню в окне инспектора объектов нажатием клавиши <Delete> или выбором пункта **Delete** в контекстном меню.

Удалим таким образом все пункты главного меню приложения за исключением пунктов в группе **Файл**.

Отметим, что если мы случайно удалили в проекте что-либо лишнее (строки исходного кода или компоненты на экранной форме), их можно восстановить с помощью пункта главного меню **Undo** (отменить сделанное) (рис. 2.47). Действие этого пункта дублируется иконкой на панели инструментов, а также комбинацией клавиш <Ctrl>+<Z>.

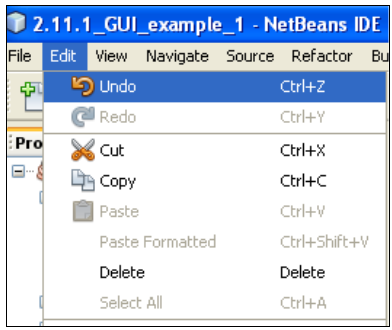


Рис. 2.47. Отмена совершенного действия (в частности — удаления кода или компонента)

Вернуться к состоянию до действия **Undo** можно с помощью пункта **Redo** (заново сделать) или комбинации клавиш <Ctrl>+<Y>.

Пусть нам теперь требуется добавить новый пункт главного меню. Для этого имеются три пути. Во-первых, можно щелкнуть правой кнопкой мыши в пустой области заголовка меню (`menuBar`) и в контекстном меню выбрать пункт **Add Menu** (Добавить меню) (рис. 2.48).

Появится новый заголовок меню. Если по имеющемуся заголовочному пункту меню щелкнуть правой кнопкой мыши и в контекстном меню выбрать пункт **Add From Palette** (добавить из палитры), то в выбранное место меню можно вставить либо новое подменю (**Menu**), либо новый пункт меню (**Menu Item**), либо разделительную линию (**Separator**) (рис. 2.49).

Пункты меню делятся на три категории:

- **Menu Item** — обычный пункт;
- **Menu Item/CheckBox** — пункт меню с элементом независимого выбора (установкой флажка перед текстом);
- **Menu Item/RadioButton** — пункт меню с элементом зависимого выбора (радиокнопкой).

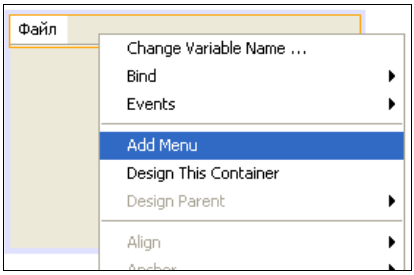


Рис. 2.48. Добавление нового главного пункта меню

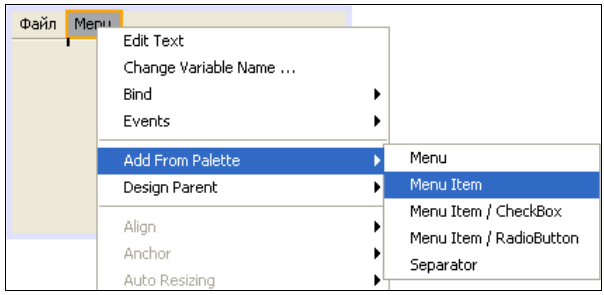


Рис. 2.49. Добавление нового вложенного меню или пункта меню

Новый элемент появится в конце соответствующего этому заголовку подменю, и путем перетаскивания мышью его можно установить на нужное место. Заметим, что щелчок по обычному пункту меню не приводит к появлению в контекстном меню пункта **Insert**.

На рис. 2.50 и 2.51 показан внешний вид меню в режиме дизайна и в режиме выполнения приложения.

Между областью, предназначенной для показа флажков элементов **CheckBox** или переключателей элементов **RadioButton**, в режиме дизайна видны незаполненные пустые квадратики (они предназначены для расположения иконок, соответствующих пунктам меню). Вызвать диалог назначения иконки можно либо двойным щелчком по такому квадратику, либо нажатием кнопки с тремя точками (...) в поле **icon** окна **Properties**.

В появившемся диалоговом окне (рис. 2.52) можно либо выбрать одно из уже имеющихся в проекте изображений, либо загрузить из файла без добавления в проект, либо сначала импортировать это изображение в проект. У нас в проекте пока нет изображе-

ний, а использовать изображение без добавления в проект не имеет смысла — ведь тогда при переносе проекта на другой компьютер оно туда не будет перенесено. Поэтому мы импортируем изображение в проект.

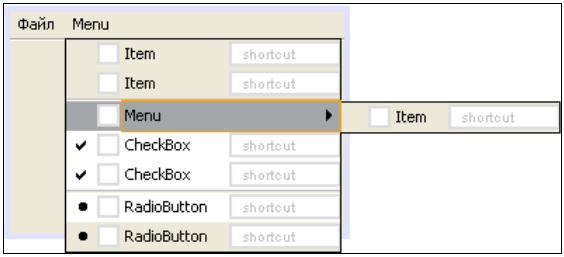


Рис. 2.50. Меню в режиме проектирования

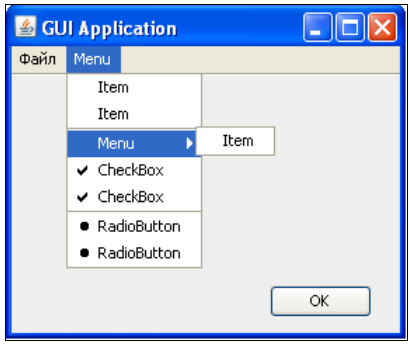


Рис. 2.51. Меню в режиме работы приложения

Для нормальной работы пунктов меню с радиокнопками необходимо предпринять те же действия, что и с обычными радиокнопками — назначить пунктам радиогруппу `buttonGroup`. Так что нам придется перетащить на форму компонент `Button Group` из палитры компонентов **Swing Controls**.

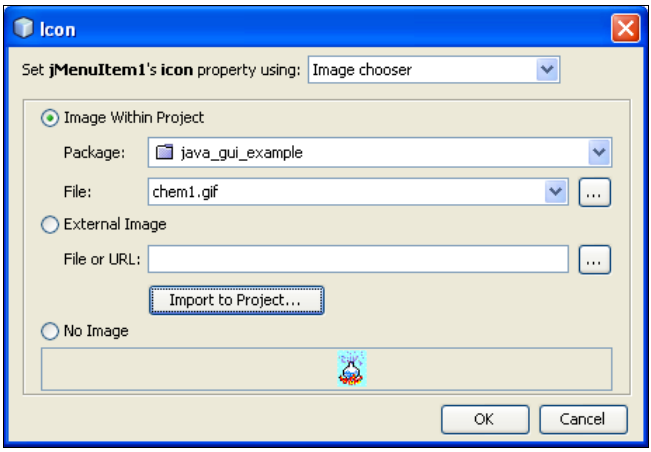


Рис. 2.52. Назначение изображения пункту меню

После этого для каждого из пунктов меню **Menu Item/RadioButton** в окне **Properties** в пункте **buttonGroup** в выпадающем списке нужно выбрать группу **buttonGroup1**, а также снять флажок **selected** для всех элементов радиогруппы (кроме тех, которые не должны быть отмечены при запуске приложения). В противном случае будет отмечен последний из элементов радиогруппы.

На рис. 2.53 и 2.54 показан внешний вид меню с иконками, радиокнопками и пунктами независимого выбора в режиме дизайна и в режиме выполнения приложения.

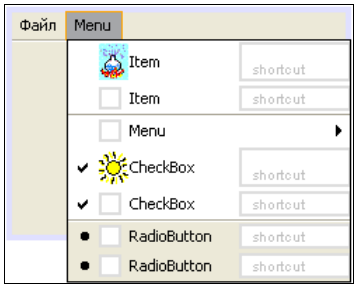


Рис. 2.53. Меню в режиме проектирования

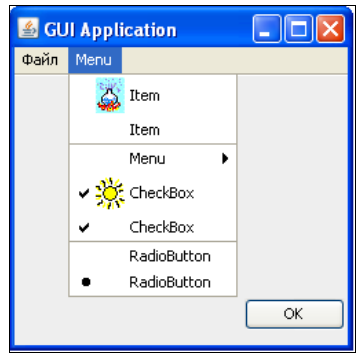


Рис. 2.54. Меню в режиме работы приложения

Следует обратить внимание на проблему, которая часто возникает при проектировании меню — добавление или перемещение сепаратора. На приведенных рисунках имеются два сепаратора, разделяющих горизонтальной линией различные группы пунктов меню (по нашему выбору). Если мы решили добавить новый сепаратор, то он возникает после последнего пункта меню, а переместить его мышью на нужное место в списке пунктов меню оказывается практически невозможно.

Но порядок пунктов меню можно легко изменить в окне **Inspector** путем перетаскивания мышью на нужное место. Например, на рис. 2.55 показано перетаскивание пункта **jSeparator2** на место между пунктами **jCheckBoxMenuItem2** и **jRadioButtonMenuItem1**.

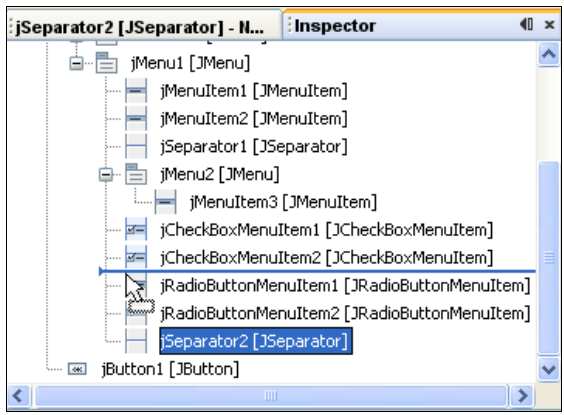


Рис. 2.55. Изменение положения пункта **jSeparator2** в меню с помощью окна **Inspector**

Мы уже говорили, что существует несколько способов добавления элементов меню. Рассмотрим второй способ. Следует раскрыть в палитре компонентов (**Palette**) пункт **Swing Menus** (меню компонентов `Swing`) и выбрать там необходимый компонент меню, после чего перетащить его в область главного меню и отпустить в нужном месте этого меню.

Есть и третий вариант, менее критичный к "меткости" пользователя при попадании мышью в требуемое место меню. Можно вызвать в окне инспектора компонентов (**Inspector**) контекстное меню, щелкнув правой кнопкой мыши по имени одного из за-

головков меню, и запустить пункт **Add From Palette/Menu Item**. Если же нам необходимо добавить заголовок в главное меню, то в окне инспектора следует вызвать контекстное меню для компонента **menuBar1**, и там запустить команду **Add From Palette/Menu** (Добавить из палитры/Меню).

Заметим, что пункты меню можно перетаскивать как в окне визуального проектирования, так и в окне инспектора компонентов (**Inspector**). Их можно не только менять местами, но и перетаскивать во вложенные пункты меню (или, напротив, извлекать наружу из вложенных пунктов).

В проекте 02_11_GUI_example3 вы найдете образец того, что у нас получилось. Отметим, что формировать меню следует разумно, без излишеств.

Наконец, рассмотрим создание контекстного меню, появляющегося при щелчке по какому-либо компоненту нашей формы. В качестве примера назначим контекстное меню кнопке выхода. Для других компонентов процесс будет аналогичным.

Рассмотрим работу с контекстными меню в NetBeans 6.5. В режиме дизайна (вкладка **Design**) выберем мышью в палитре компонентов (окно **Palette** в правом верхнем окне) компонент **JPopupMenu** и перетащим его в область экранной формы. Либо можно щелкнуть правой кнопкой мыши по форме и в контекстном меню выбрать **Add From Palette/Swing Menus/Popup Menu**.

При этом контекстное меню на форме не появится, но в окне инспектора компонентов в дереве **Other Components** возникнет новый узел **jPopupMenu1[JPopupMenu]**. Если щелкнуть по узлу, то окажется, что кроме самого компонента **jPopupMenu1** в нем ничего нет.

Щелкнем правой кнопкой мыши по этому узлу и в появившемся контекстном меню выберем **Add From Palette /Menu Item** (рис. 2.56).

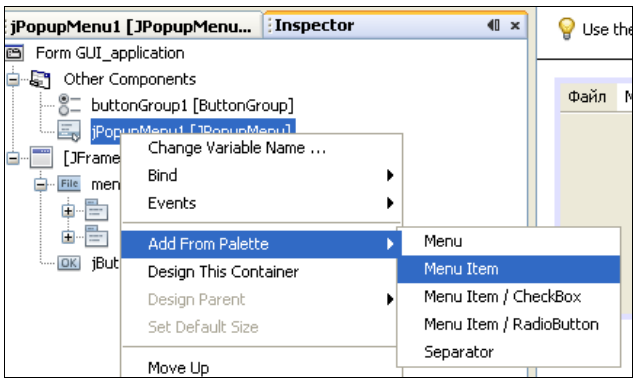


Рис. 2.56. Создание нового пункта контекстного меню

После этого в дереве **jPopupMenu1** появится узел **jMenuItem4[JMenuItem]**, а в редакторе свойств компонентов можно задать значение свойству **text** данного компонента. Введем текст "Выйти из программы".

Далее уже известным нам способом зададим обработчик нажатия на этот пункт меню (либо сделаем по нему двойной щелчок в окне **Inspector**, либо выберем в контекстном меню, возникающем при щелчке правой кнопкой мыши по имени **jMenuItem4** в окне

Inspector, пункт **Events/Action/ActionPerformed**). А в обработчике напишем оператор выхода из программы:

```
System.exit(0);
```

Мы пока только создали контекстное меню, которое доступно в нашей форме, но еще не назначили его никакому компоненту. Для того чтобы назначить меню **jPopupMenu1** кнопке **JButton1**, выделим эту кнопку и в редакторе свойств компонентов в пункте **componentPopupMenu** нажмем мышью стрелку вниз, разворачивающую выпадающий список. Кроме значения <**none**>, заданного по умолчанию этому свойству для каждого компонента, там имеется имя **jPopupMenu1**. Его мы и выберем.

Теперь контекстное меню, состоящее из одного пункта **Выйти из программы**, появится при щелчке правой кнопкой мыши по кнопке. Добавление других пунктов меню и назначение им обработчиков событий проводится совершенно аналогично.

2.12. Создание нового класса

Пусть мы хотим создать в нашем проекте новый класс. Для этого щелкнем правой кнопкой мыши по имени пакета и выберем в появившемся контекстном меню пункт **New/Java Class...** (рис. 2.57).

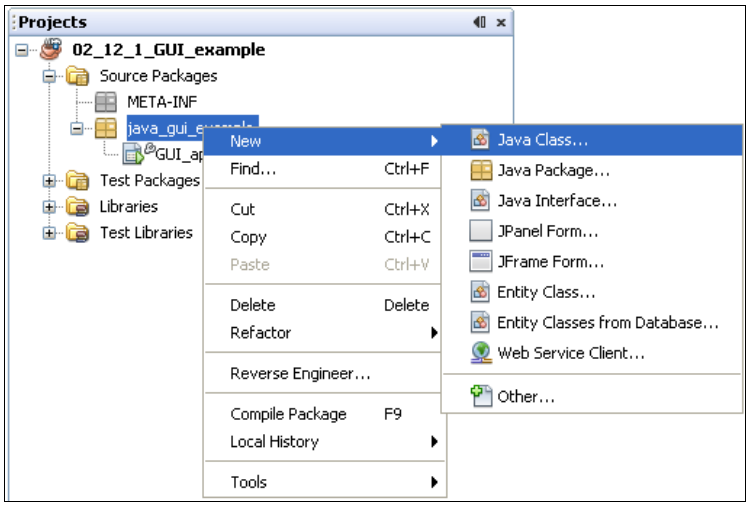


Рис. 2.57. Создание нового класса. Шаг 1

Появится диалоговое окно создания нового класса Java (рис. 2.58). В пункте ввода **Class Name** (имя класса) следует указать имя создаваемого класса, заменив имя по умолчанию. Зададим имя **Figure**.

В качестве пакета, в котором расположен класс, будет автоматически задан пакет нашего приложения (если первоначальный щелчок правой кнопкой мыши был по имени этого пакета).

Кроме описанной процедуры для создания нового класса можно воспользоваться мастером создания нового класса в главном меню среды NetBeans (**File/New File.../** , категория **Java**, тип файла **Java Class/Next>**). В результате появится то же диалоговое окно.

После нажатия на кнопку **Finish** (закончить) в редакторе исходного кода появляется заготовка класса (рис. 2.59). Отметим, что в прежних версиях NetBeans в заготовке класса создавался конструктор с пустым телом, а в версии 6.5 это не делается.

Далее можно редактировать текст для класса — вводить новые поля и методы. Скажем, можно ввести поля:

```
private int x=0;
private int y=0;
```

Пример 02_12_GUI_example2 показывает, что получилось в результате наших действий. Разверните дерево просмотра для пакета `java_gui_example`, посмотрите, как выглядит класс `Figure`, и сравните с тем, что получилось у вас.

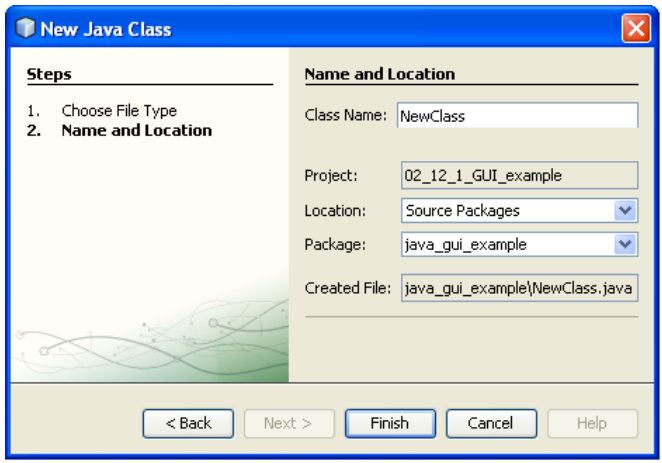


Рис. 2.58. Создание нового класса. Шаг 2

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package java_gui_example;

/**
 *
 * @author В.Монахов
 */
public class Figure {

}
```

Рис. 2.59. Заготовка нового класса

2.13. Документирование исходного кода в Java

Одна из важнейших частей написания программного обеспечения — документирование создаваемого кода. В Java для этих целей применяется средство, обеспечивающее поддержку на уровне синтаксиса языка программирования — специализированные

комментарии. Они начинаются с комбинации символов /** и заканчиваются комбинацией символов */ , в то время как обычный многострочный комментарий начинается с комбинации символов /* и заканчивается комбинацией */.

Часть комментариев автоматически создает среда разработки при генерации класса.

Пример:

```
/**
 * Creates new form GUI_application
 */
```

Средством обработки внедренных в исходный код комментариев и создания для класса справочных HTML-файлов является инструмент javadoc, входящий в состав JDK. Но в среде NetBeans для генерации документации удобнее пользоваться вызовом через главное меню: **Run/Generate Javadoc (...)** (вместо имени конкретного проекта мы поставили многоточие). При этом на основе документационных комментариев создается документация к проекту в виде набора HTML-файлов *(см. разд. 11.1)* с описаниями классов, методов, полей и т. д.

Документационные комментарии бывают для:

- пакетов (пока не функционируют);
- классов;
- интерфейсов *(см. главу 9)*;
- пользовательских типов-перечислений (на уровне пакетов пока не функционируют, но можно использовать для типов, заданных в классах);
- методов;
- переменных.

Документационные комментарии пишут непосредственно перед заданием соответствующей конструкции — пакета, класса, интерфейса, типа-перечисления, метода или переменной. Следует учитывать, что по умолчанию документация создается только для элементов, имеющих уровень видимости `public` или `protected`.

Пример фрагмента кода с документационными комментариями приведен в листинге 2.7.

Листинг 2.7. Пример фрагмента кода с документационными комментариями

```
/**
 * Пример приложения Java с документационными комментариями <br>
 * В приложении заданы типы-перечисления Months и Spring и показано,
 * как с ними работать.
 * Кроме того, дан пример использования класса из другого пакета.
 * @see enumApplication.Months Информация о типе-перечислении Months
 * @see enumApplication.Spring
 * @see enumApplication#m1
 * @version Версия 0.2 Вторая версия. Первоначальная версия проверена
 * при компиляции в среде NetBeans 5.5, вторая — в средах NetBeans 6.0, 6.1, 6.5
 * @author Вадим Монахов
 */
```

```
public class enumApplication extends javax.swing.JFrame {
    int i=1;
/**
  * Spring — задает перечисление из 3 весенних месяцев года:
  * march,apr,may.
  *  <ul>
  *   <li>march
  *   <li>apr
  *   <li>may
  *  </ul>
  *   Идентификатор для марта записан отличающимся от соответствующего
  * идентификатора в перечислении Months, а для апреля и мая записаны так
  * же — чтобы подчеркнуть, что их пространства имен независимы.
  */
    public enum Spring {march,apr,may}

    /**
     * Months — задает перечисление из 12 месяцев года: <br>
     * jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec <br>
     * (январь, февраль и т. д.)
     */
    public enum Months {jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec}
    Spring spr1= Spring.apr, spr2;

    /**
     *Переменная, иллюстрирующая работу с перечислениями
     */
    public Months m1,m2=Months.mar,m3;
```

Имеются два типа кода внутри блока документационного комментария — HTML-текст и метаданные (команды документации, начинающиеся с символа @ — *аннотации*). Если пишется обычный текст, то он рассматривается как HTML-текст, поэтому все пробелы и переносы на новую строку при показе приводятся к одному пробелу. Для того чтобы очередное предложение при показе начиналось с новой строки, следует вставить последовательность символов `<br>`, называющуюся тегом HTML. Возможно использование произвольных тегов HTML, а не только тега переноса на новую строку: теги неупорядоченного списка `<ul>` и `<li>`, теги гиперссылок, изображений и т. д. В то же время рекомендуется отказаться от заголовков и фреймов, поскольку их наличие может привести к проблемам — javadoc создает на основе документационного кода собственную систему заголовков и фреймов. Кроме того, при преобразовании в HTML-документ из документационного кода удаляются символы `*`, если они стоят на первом значимом месте в строке (символы пробелов не значимые).

Некоторая информация по HTML приведена в *главе 11*. Для более подробного изучения тегов и атрибутов HTML следует читать справочную или учебную литературу по этому языку разметки документов. Соответствующие ссылки и документы можно найти, например, на сайте автора:

http://barsic.spbu.ru/www/comlan/html_r.html.

Рассмотрим команды документации (символы метаданных).

- `@see` (смотри) — применяется для создания в документе гиперссылок на другие комментарии. Допустима для любых конструкций (классов, методов и т. д.).

  Формат:

  `@see ИмяКласса` — для класса;

  `@see ИмяКласса.ИмяПеречисления` — для типа-перечисления, заданного в классе;

  `@see ИмяКласса#ИмяЧлена` — для метода или переменной; для интерфейса — аналогично классу. При этом имя класса или интерфейса может быть либо коротким, либо квалифицировано именем пакета.

- `@version` (версия) — информация о версии. Применяется для классов и интерфейсов.

  Формат:

  ```
  @version Информация о версии в произвольной форме.
  ```

- `@author` (автор) — информация об авторе. Применяется для классов и интерфейсов.

  Формат:

  ```
  @author Информация об авторе (в произвольной форме).
  ```

  Может включать не только имя, но и данные об авторских правах, а также об электронной почте автора, его сайте и т. д.

- `@since` (начиная с) — информация о версии JDK, начиная с которой введен (либо работоспособен) класс или интерфейс.

  Формат:

  ```
  @since Информация в произвольной форме.
  ```

- `@param` (сокращение от parameter (параметр)) — информация о параметре метода. Комментарий `/** @param ... */` ставится в месте декларации метода в списке параметров перед соответствующим параметром.

  Формат использования:

  ```
  @param ИмяПараметра Описание.
  ```

- `@return` (возвращает) — информация о возвращаемом методом значении и его типе.

  Формат:

  ```
  @return Информация в произвольной форме.
  ```

- `@throws` (возбуждает исключение) — информация об исключительных ситуациях, которые могут возбуждаться методом.

  Формат:

  ```
  @throws ИмяКлассаИсключения Описание.
  ```

- `@deprecated` (устаревшее) — информация о том, что данный метод устарел и в последующих версиях будет ликвидирован. При попытке использования таких методов компилятор выдает программисту предупреждение (warning) о том, что метод устарел, хотя и компилирует проект.

Формат:

```
@deprecated Информация в произвольной форме.
```

Признак окончания команды документации — начало новой команды или окончание комментария.

В NetBeans 6.5 появились новые возможности работы с документационными комментариями — проверка наличия необходимых комментариев, их правильности, а также возможность автоматической генерации заготовок комментариев. Для примера поместим следующий фрагмент кода, приведенный в листинге 2.8, в некий класс `Functions1` и вызовем проверку документационных комментариев проекта с помощью пункта меню **Tools/Analyze Javadoc** (рис. 2.60).

Листинг 2.8. Фрагмент кода без документационных комментариев

```
public int a;
public int f1(int x) {
    return 2 * x;
}

public float f2(float x,float y) {
    return x+y;
}
```

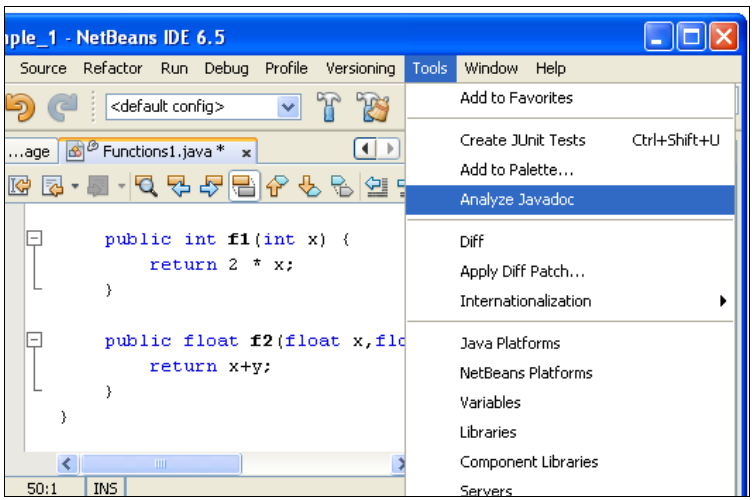


Рис. 2.60. Вызов анализатора документационных комментариев

В результате в нижней части экрана в той же области вкладок, что и окно **Output**, появится окно вкладки **Analyzer** (анализатор) — рис. 2.61.

Поставим флажок в пункте выбора напротив имени **Functions1.java**, после чего автоматически отметятся все три проблемные пункты, помеченные изображением лампочки (по-видимому, этот символ обозначает предупреждение "Внимание!"). После этого кнопка **Fix Selected** (исправить отмеченное), находящаяся в левом нижнем углу на

рис. 2.61, станет доступной, и нажатие на нее приведет к генерации заготовок документационных комментариев как для переменной `a`, так и для функций `f1` и `f2` (листинг 2.9).

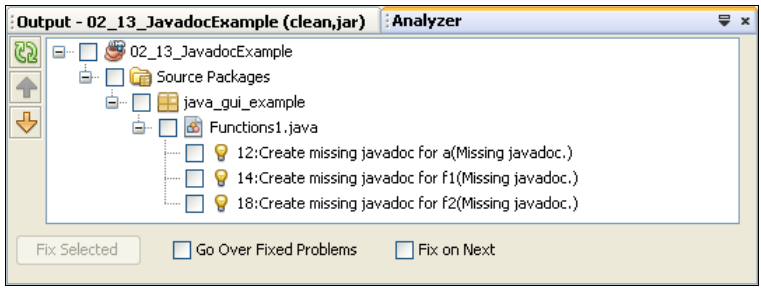


Рис. 2.61. Вкладка **Analyzer**

Листинг 2.9. Фрагмент кода после генерации заготовок документационных комментариев

```
/**
 *
 */
public int a;

/**
 *
 * @param x
 * @return
 */
public int f1(int x) {
    return 2 * x;
}

/**
 *
 * @param x
 * @param y
 * @return
 */
public float f2(float x, float y) {
    return x + y;
}
```

Программисту остается только дополнить заготовки соответствующей информацией.

Генерация документации по документационным комментариям производится с помощью выбора пункта главного меню: **Run/Generate Javadoc for** *"Имя проекта"*. После генерации головная страница документации автоматически запускается в браузере (программе просмотра HTML-документов). Пример документации, созданной для пакета, из которого взят содержащийся в листинге 2.7 фрагмент кода, приведен на рис. 2.62—2.64.

Для просмотра документации следует зайти в папку 02_13_EnumExample\dist\javadoc и двойным щелчком запустить на просмотр файл index.html. Если необходимо распространить документацию о ваших классах, то ее следует копировать целиком.

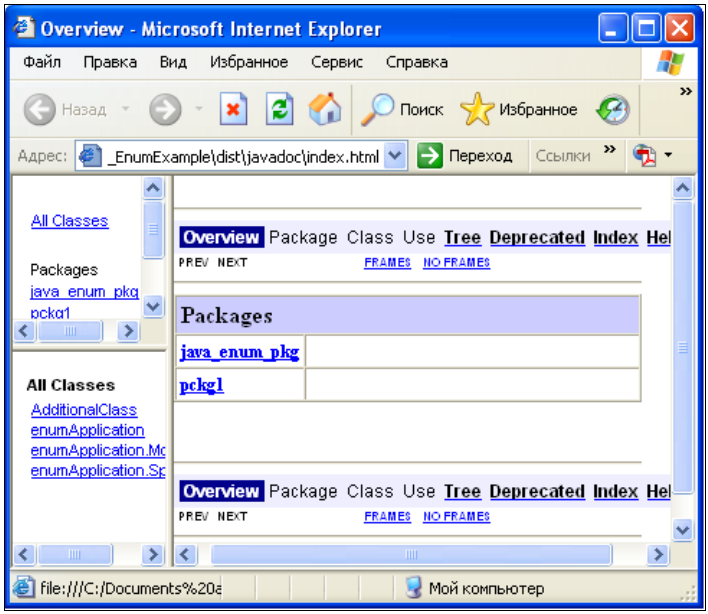


Рис. 2.62. Головная страница файлов документации

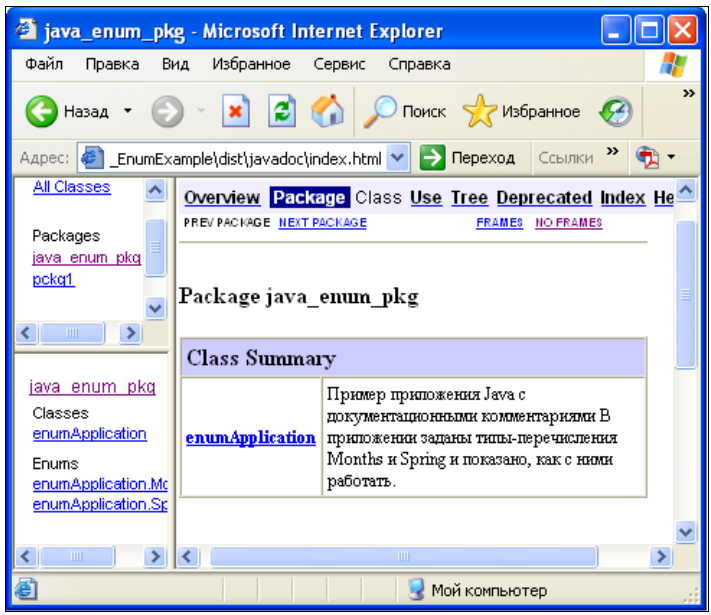


Рис. 2.63. Страница описания элементов пакета `java_enum_pkg`

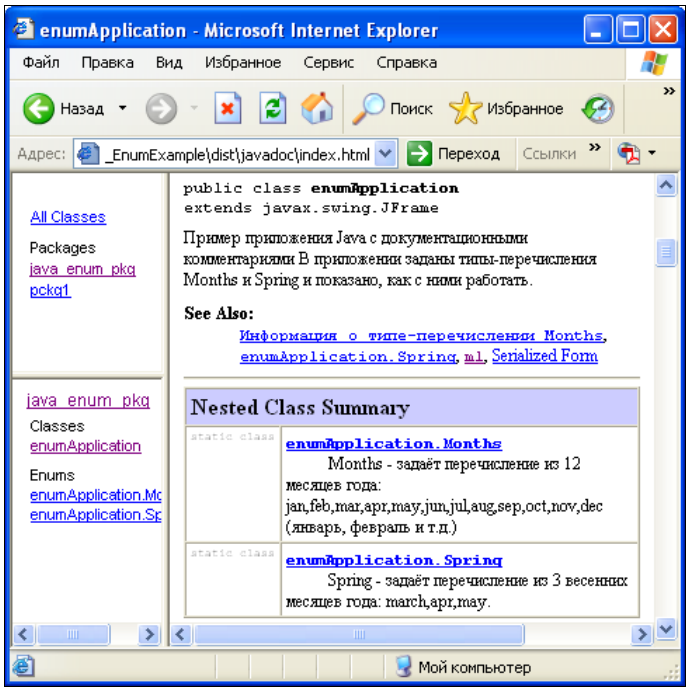


Рис. 2.64. Страница описания класса `enumApplication`

Обратите внимание, что в краткой сводке (summary) приводятся только начальные строки соответствующей информации по элементам пакета или класса. Полную информацию можно прочитать после перехода по гиперссылке в описании соответствующего элемента. Поэтому важно, чтобы первые 2—3 строки текста содержали самые важные сведения.

Следует подчеркнуть, что приведенный проект не очень прост для понимания. Он служит исключительно в качестве примера кода сложной структуры — чтобы было понятно, как использовать документационные комментарии и как выглядит сгенерированная документация.

При щелчке правой кнопкой мыши по имени проекта и выборе в появившемся контекстном меню пункта **Properties** или выборе в главном меню NetBeans пункта **Run/Set Main Project Configuration/Customize** появляется диалог, позволяющий дополнительно настроить генератор документации, если в появившейся форме в окне **Categories** (категории) выбрать пункт **Documenting** (документирование) — рис. 2.65.

В пункте **Browser Window Title** (заголовок окна браузера) необходимо ввести текст, который будет отображаться в заголовке окна браузера (в верхней синей полосе, находящейся в самом верху) при показе HTML-файлов документации. По умолчанию в документацию включается только информация об общедоступных (public) и защищенных (protected) полях и методах. А для показа в сгенерированной документации элементов с областями видимости private и пакетной (видимости по умолчанию) требуется проставить соответствующий флажок (пункт **Include Private and Package Private Members**). Естественно, данные опции будут относиться только к заново сгенерированной документации. Кроме того, следует помнить, что вызов пункта **Clean and Build Project** сти-

рает сгенерированную документацию, следовательно, после этого варианта компиляции необходимо заново генерировать документацию по проекту.

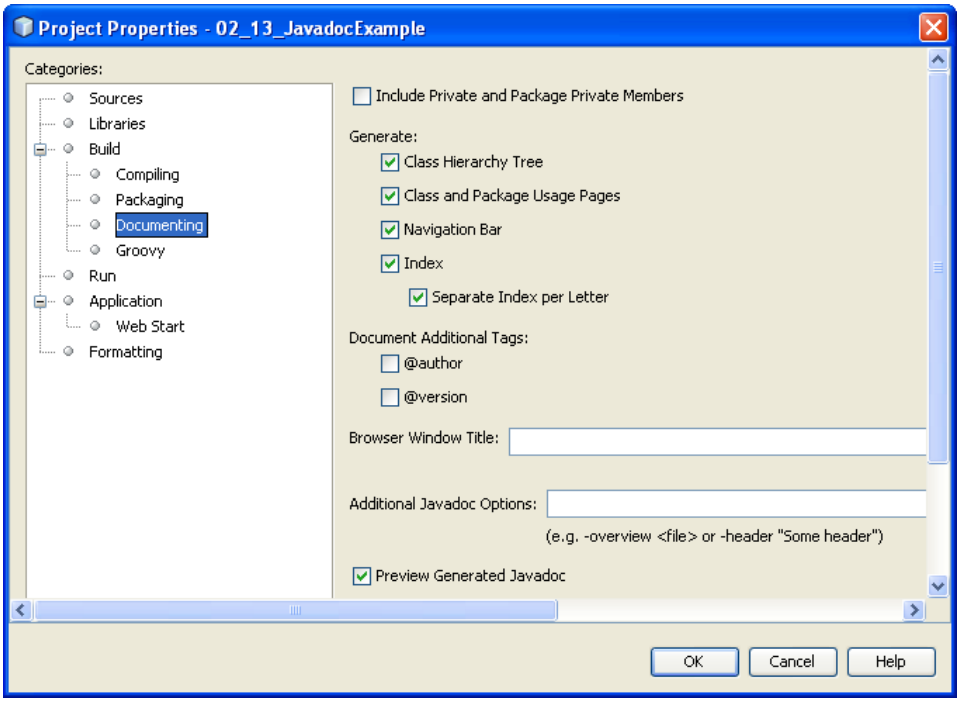


Рис. 2.65. Настройка дополнительных возможностей документирования проекта

После того как документация сгенерирована, она будет использоваться в проекте, и мы получим для наших пакетов и классов те же возможности, что и для стандартных библиотечных. Например, при выборе пункта меню **Source/Show Documentation** (либо комбинации клавиш <Shift>+<Ctrl>+<Space>) вызывается контекстная справка по элементу, на который установлен курсор в редакторе исходного кода (рис. 2.66).

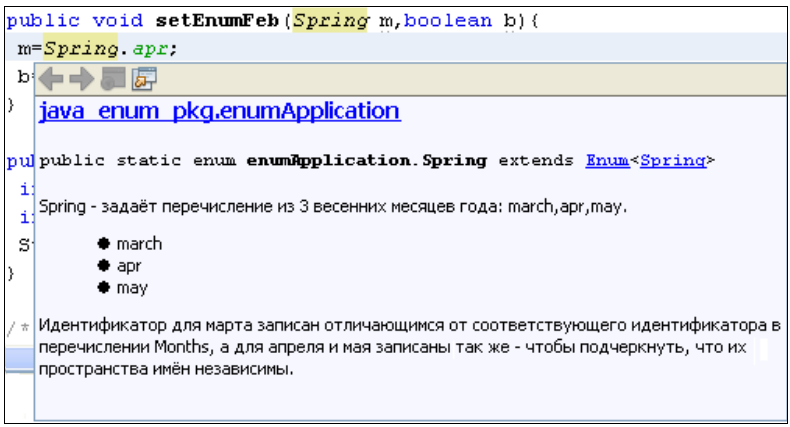


Рис. 2.66. Контекстная справка по элементу, на который установлен курсор

А при редактировании исходного кода этого приложения или другого (импортирующего пакет из этого приложения) возникает "умная подсказка" после набора с клавиатуры имени элемента, за которым следует точка. Например, если мы наберем `pckg1.`, то появится список элементов пакета, в который входит только класс `AdditionalClass` (поэтому в списке будет присутствовать только один элемент, он будет выделен, и по нему покажется справка, сгенерированная по этому классу (рис. 2.67)).

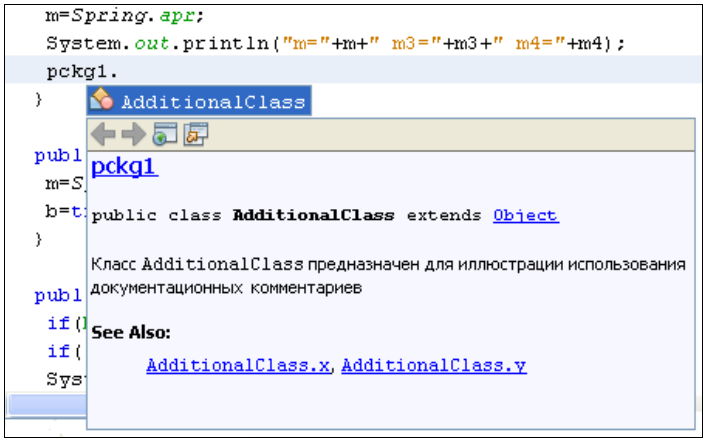


Рис. 2.67. "Умная" подсказка при редактировании исходного кода

2.14. Основные компоненты пакетов *swing* и *awt*

Пока мы научились работать только с формами, кнопками и контекстными меню. Перечислим еще ряд полезных компонентов.

Во-первых, следует остановиться на том, что в палитре компонентов NetBeans предлагается семь категорий компонентов: четыре категории для библиотеки Swing (пакет `swing`), одна для библиотеки AWT (пакет `awt`), а также категории Beans и Java Persistence.

Библиотека Swing — основная для большинства современных графических приложений Java. В табл. 2.3 перечислены компоненты Swing в том порядке, в каком они возникают в палитре компонентов, но сами палитры расположены в порядке их важности при начальном освоении.

Таблица 2.3. *Компоненты библиотеки Swing*

№	Компонент	Назначение компонента
Swing Controls (управляющие элементы)		
1	`JLabel`	Метка — вывод однострочного неформатированного текста
2	`JButton`	Кнопка — кнопка с текстом и/или с картинкой

Таблица 2.3 *(продолжение)*

№	Компонент	Назначение компонента
3	JToggleButton	Защелкивающаяся кнопка — кнопка с фиксацией. Может быть одной из нескольких таких кнопок в группе, в этом случае нажатие одной кнопки вызывает отпускание другой. Работа группы обеспечивается компонентом ButtonGroup, который должен быть перетащен на форму, а затем назначен свойству buttonGroup
4	JCheckBox	Пункт выбора с независимой фиксацией
5	JRadioButton	Радиокнопка — пункт выбора с зависимой фиксацией. Должен быть одним из нескольких в группе. Работа группы обеспечивается компонентом ButtonGroup
6	ButtonGroup	Обеспечивает работу групп компонентов JToggleButton или JradioButton
7	JComboBox	Выпадающий список
8	JList	Прокручивающийся список
9	JTextField	Текстовое поле — однострочный пункт ввода и редактирования текста
10	JTextArea	Текстовая область — многострочный пункт ввода и редактирования текста
11	JScrollBar	Независимая полоса прокрутки. Используется редко — для программно-управляемой прокрутки содержимого компонентов, не имеющих встроенных полос прокрутки
12	JSlider	Ползунок. Служит для плавной регулировки числовых величин, а также связанных с ними программно-регулируемых изменений
13	JProgressBar	Полоса показа доли выполнения задачи. Показывает уровень, отражающий долю выполнения задачи
14	JFormattedTextField	Поле ввода форматированного текста
15	JPasswordField	Поле ввода пароля — вводимый текст отображается звездочками
16	JSpinner	Поле ввода числа с кнопками увеличения/уменьшения
17	JSeparator	Разделительная линия. Используется в декоративных целях для разделения рабочих областей формы и других группирующих компонентов
18	JTextPane	Текстовая панель. По умолчанию автоматически переносит текст на новую строку, а не располагает его в одну строку с показом горизонтального скроллера, как это делает JTextArea
19	JEditorPane	Панель текстового редактора
20	JTree	Дерево — показывает дерево, в котором каждая ветвь может быть с иконками и текстом, а узлы разворачиваются и сворачиваются
21	JTable	Таблица — показ текстовой таблицы. Имеет возможность заполнения значениями по умолчанию на этапе проектирования

Таблица 2.3 *(продолжение)*

№	Компонент	Назначение компонента
Swing Menus (элементы меню)		
1	JMenuBar	Меню формы — предназначено для расположения в нем компонентов типа JMenu (заголовков меню), показываемых в меню
2	JMenu	Заголовок меню (в него можно добавлять пункты меню JMenuItem, JCheckBoxMenuItem и JradioButtonMenuItem)
3	JMenuItem	Пункт меню. Может добавляться к заголовкам меню JMenu и в контекстное меню JPopupMenu
4	JCheckBoxMenuItem	Пункт меню с элементом CheckBox — чтобы этот пункт меню можно было отмечать флажком. Может добавляться к заголовкам меню JMenu и в контекстное меню JPopupMenu
5	JRadioButtonMenuItem	Пункт меню с радиокнопкой (чтобы этот пункт меню можно было отмечать точкой в кружке). Может добавляться к заголовкам меню JMenu и в контекстное меню JPopupMenu
6	JPopupMenu	Контекстное меню — предназначено для показа по нажатию правой кнопки мыши в области компонента, которому назначено свойство componentPopupMenu. Пункты меню состоят из компонентов типа: JMenuItem, JCheckBoxMenuItem и JRadioButtonMenuItem
7	JSeparator	Разделитель. Добавляется в меню для группировки пунктов меню (разделяя их горизонтальной линией)
Swing Containers (контейнеры)		
1	JPanel	Панель — группирующий компонент, позволяющий располагать на себе другие компоненты. Передвижение панели перемещает вместе с ней все расположенные на ней компоненты. По умолчанию свойство layout (размещение) установлено как FlowLayout (в виде потока). Для простых задач вместо этого компонента лучше использовать JLayeredPane
2	JTabbedPane	Панель с вкладками — каждый положенный на нее компонент показывается в отдельной вкладке. Чтобы разместить на одной вкладке несколько компонентов, сначала положите на панель с вкладками обычную панель. Для того чтобы создать последующие вкладки, выделите панель с вкладками, вызовите правой кнопкой мыши контекстное меню, запустите команду **Add From Palette** (добавить из палитры) и добавьте еще одну панель или другой компонент
3	JSplitPane	Панель с разделителем — панель, состоящая из двух частей, между ними имеется линия разделителя, которую можно перетаскивать мышью, меняя взаимный размер частей
4	JScrollPane	Панель с полосами прокрутки
5	JToolBar	Панель инструментов. Обычно на ней размещают кнопки JToggleButton, для которых назначены иконки
6	JDesktopPane	Панель — рабочий стол. Еще один тип панели с абсолютным позиционированием элементов

Таблица 2.3 *(продолжение)*

№	Компонент	Назначение компонента
7	`JInternalFrame`	Дочернее окно — окно многооконного приложения. Его можно перемещать в пределах родительского окна (главного окна приложения). В настоящее время такой стиль приложений практически не встречается
8	`JLayeredPane`	Панель с абсолютным позиционированием элементов
Swing Windows (окна, диалоговые формы)		
1	`JDialog`	Диалоговая форма. Показывается вызовом вида `jDialog1.setVisible (true)`
2	`JFrame`	Экранная форма. Показывается вызовом вида `jFrame1.setVisible (true)`
3	`JColorChooser`	Диалог выбора цвета — предназначен для выбора пользователем цвета
4	`JFileChooser`	Диалог выбора файла — предназначен для выбора пользователем файлов. Перед использованием требуется положить его на какую-нибудь диалоговую форму (`JDialog`, `JFrame`) или какой-нибудь группирующий компонент формы

5	`JOptionPane`	Диалоговая панель — предназначена для показа диалоговых форм. В отличие от большинства других компонентов работа идет с помощью методов класса. Имеются вызовы диалогов:
		С сообщением:
		`javax.swing.JOptionPane.showMessageDialog(null, "Кнопку нажали");`
		`JOptionPane.showMessageDialog(null,"Привет!", "Заголовок сообщения", JOptionPane.INFORMATION_MESSAGE);`
		С подтверждением:
		`int option=javax.swing.JOptionPane.showConfirmDialog( null,"Продолжить?");`

		С предложением ввести значение: `String input=javax.swing.JOptionPane.showInputDialog( null,"Введите значение:");` Проверка, какую кнопку нажали или диалог закрыли без осуществления выбора, осуществляется сравнением с константами `javax.swing.JOptionPane.NO_OPTION`, `CANCEL_OPTION`, `CLOSED_OPTION`, `OK_OPTION`, `YES_OPTION` При отказе от ввода возвращается `null`. Первый параметр — имя формы, в которой показывается диалог. Если он имеет значение `null`, то используется форма по умолчанию (главная форма приложения). Существуют варианты указанных методов, позволяющие при вызове задавать дополнительные параметры диалога (заголовки, надписи на кнопках и др.)

Таблица 2.3 *(окончание)*

№	Компонент	Назначение компонента
AWT (управляющие элементы пакета AWT)		
1—13	`Label`, `Button`, `TextField`, `TextArea`, `CheckBox`, `Choice`, `List`, `ScrollBar`, `ScrollPane`, `Panel`, `Canvas`, `MenuBar`, `PopupMenu`	Устаревший набор компонентов. Может представлять интерес только для тех разработчиков, кому необходимо писать приложения Java, работающие с самыми древними версиями Java-машин
Beans (компоненты, установленные пользователем)		
1	`Coose Bean`	Выберите компонент. Это не компонент, а диалог добавления компонента в палитру на страницу `Beans`. О том, как добавить компонент на нужную страницу палитры, рассказывается в *главе 15*
Java Persistence (компоненты технологии Java Persistence)		
1	`EntityManager`	Менеджер сущностей. Используется в технологии Java Persistence для управления сущностями (`entities`) — объектами, которые могут сохраняться после окончания работы приложения и загружаться "из внешнего мира" (например, из базы данных)
2	`Query`	Запрос в технологии Java Persistence
3	`Query Result`	Результат запроса в технологии Java Persistence (имеет тип `List`)

Очень часто в приложении требуется вывести служебную информацию. В старых версиях Java для этого служил вызов **System.out.println** (текст сообщения). В учебных проектах и при выводе отладочной информации этот метод до сих пор удобен. Но предоставлять таким образом информацию конечному пользователю — анахронизм. Для выдачи пользователю информационного сообщения больше подходит следующий вызов:

```
JOptionPane.showMessageDialog(null,"Привет!","Заголовок сообщения",
JOptionPane.INFORMATION_MESSAGE);
```

Если требуется вывести предупреждение об опасной ошибке (error), то последний параметр должен иметь значение:

```
JOptionPane.ERROR_MESSAGE,
```

простое предупреждение (warning):

```
JOptionPane.WARNING_MESSAGE ,
```

вопрос:

```
JOptionPane.QUESTION_MESSAGE.
```

Наконец, если не требуется сопровождать вопрос иконкой на диалоговой панели, параметр должен быть:

```
JOptionPane.PLAIN_MESSAGE.
```

(Пример 02_14_OptionPane.)

Библиотека компонентов AWT (Abstract Window Toolkit — абстрактный инструментарий графического окна) устаревшая по сравнению с библиотекой Swing, хотя сам пакет `awt` до сих пор является основой графики Java. В библиотеке AWT имеются практически те же компоненты, что и в Swing, но в меньшем количестве и в более примитивном варианте (с худшим дизайном и меньшей функциональностью).

Единственный компонент AWT, у которого нет прямого аналога в Swing, — это компонент типа `Canvas` (холст для рисования). Он обеспечивал вывод графических примитивов, например, следующим образом:

```
java.awt.Graphics g=canvas1.getGraphics();
g.drawLine(10,10,100,100);
```

В Swing для этих целей можно рисовать по любому компоненту, например по панели или даже по кнопке:

```
java.awt.Graphics g=jPanel1.getGraphics();
g.drawLine(10,10,100,100);
g=jButton3.getGraphics();
g.drawLine(10,10,100,100);
```

2.15. Менеджеры размещения (*Layout*) и якоря (*Anchor*)

Еще один вопрос, на котором следует остановиться при обсуждении внешнего вида приложений с графическим пользовательским интерфейсом (GUI), — это компоненты `Layout` — *менеджеры размещения*. Разработчики Java предложили оригинальную, но очень неоднозначную идею организации расположения компонентов на форме. Вместо того чтобы явно указывать позиции компонентов на этапе проектирования или работы программы и использовать якоря (anchors) для привязки краев компонентов к краям группирующего компонента (как это делается в других языках программирования), предлагается задействовать тот или иной менеджер размещения. При изменении размера формы взаимное расположение компонентов будет меняться в зависимости от типа менеджера.

Например, обычное размещение с фиксированным положением достигается с помощью размещения на форме менеджера `AbsoluteLayout`. В NetBeans это делается через пункт **Set Layout** контекстного меню (рис. 2.68). По умолчанию действует режим `Free Design` (свободный дизайн). Если установить менеджер размещения `AbsoluteLayout`, то в редакторе свойств компонентов оказываются доступны свойства `x` и `y` — координаты компонентов.

Применять якоря можно, но с ограниченными возможностями и только в менеджере размещения `Free Design` — в других менеджерах они не работают. Для использования якоря следует щелкнуть правой кнопкой мыши по компоненту, расположенному на форме (например, кнопке), и в появившемся меню выбрать пункт **Anchor**. Якорь привязывает компонент к соответствующей стороне формы (рис. 2.69) (`Left` — привязка к левому краю формы, `Right` — к правому, `Top` — к верхнему, `Bottom` — к нижнему). По умолчанию менеджер сам выбирает варианты привязки, показывая их пунктирными линиями.

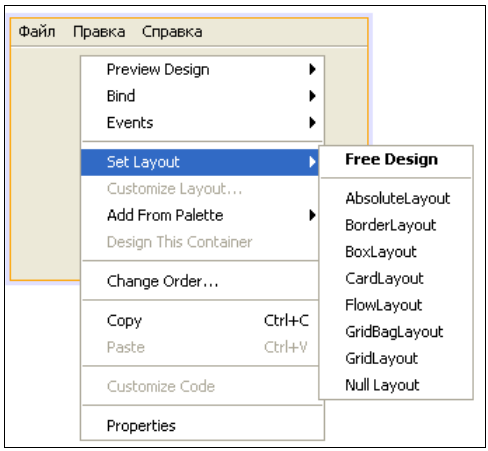


Рис. 2.68. Выбор менеджера размещения

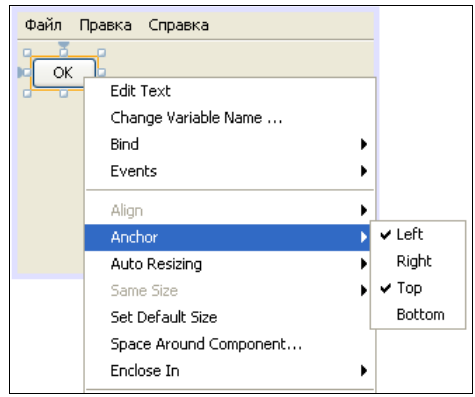


Рис. 2.69. Установка привязки к краям формы якорей

Для того чтобы привязать левый край компонента к левому краю формы, а правый край компонента к правому краю формы, следует в окне **Properties** в секции **Layout** установить для компонента флаг **Horisontal Resizable** ("Изменяемый размер по горизонтали"). В этом случае при изменении ширины формы вместе с ней будет меняться и ширина компонента — в пределах, ограничиваемых свойствами `minimumSize` и `maximumSize`.

Аналогично, для привязки верхнего края компонента к верхнему краю формы, а нижнего края компонента к нижнему краю формы, следует установить флаг **Vertical Resizable** ("Изменяемый размер по вертикали").

С точки зрения автора, `AbsoluteLayout` и `Free Design` — это наиболее удачные менеджеры размещения. Если зафиксировать размер формы, менеджер `AbsoluteLayout` дает возможность расположить все элементы на фиксированных местах. А при необходимости изменять размер формы следует воспользоваться менеджером `Free Design`. Хотя излишняя самостоятельность этого менеджера порой раздражает: менеджер сам меняет привязки краев компонентов друг к другу и краям формы, и при этом зачастую не очень удачно меняет размеры компонентов. В том числе не только того компонента, над расположением которого вы работаете, но и тех, к краям которых менеджер решил осуществить привязку.

2.16. Создание приложения Desktop Application

Как уже говорилось, в NetBeans 6.5 имеется заготовка GUI-приложения, которая называется Desktop Application (приложение рабочего стола, экранное приложение). Для создания такого приложения следует выбрать в главном меню команду **File/New Project...**, после чего в открывшейся форме выбрать категорию **Java**, а в ней вид проекта **Java Desktop Application** и нажать кнопку **Next >** (рис. 2.70). Появится следующая форма (рис. 2.71), в которой следует выбрать вариант **Basic Application** (простое приложение) и нажать кнопку **Finish**. В результате на экране появится исходный код

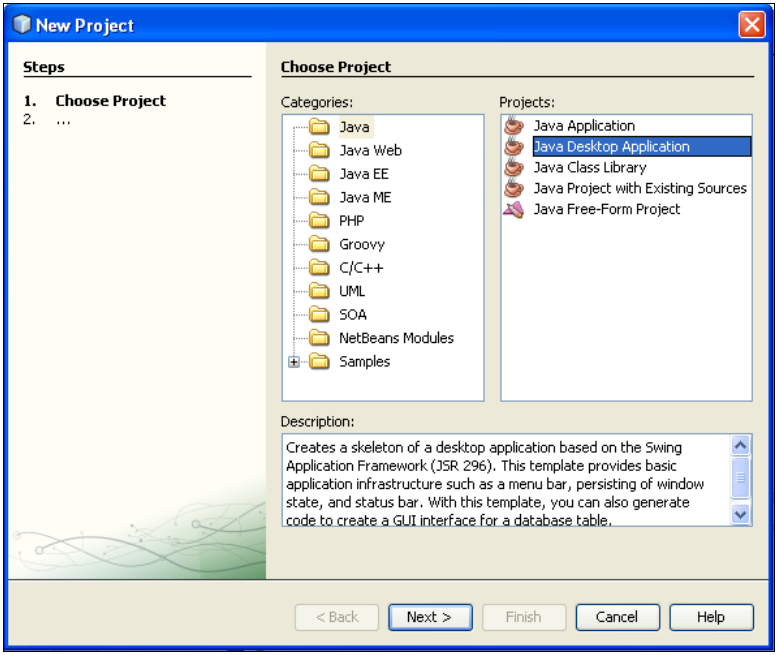


Рис. 2.70. Создание приложения Desktop Application. Шаг 1

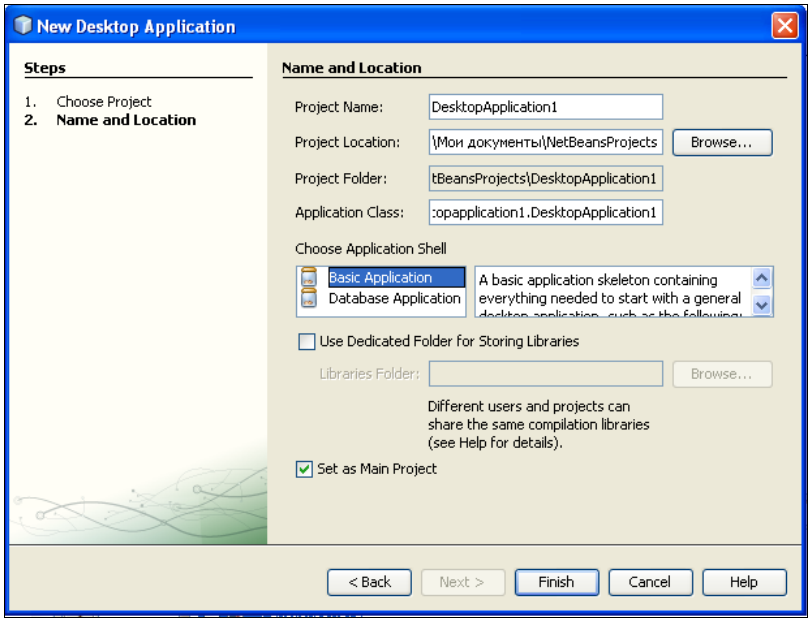


Рис. 2.71. Создание приложения Desktop Application. Шаг 2

одного из классов приложения в режиме проектирования пользовательского интерфейса (дизайна) (рис. 2.72). После запуска оно выглядит так, как показано на рис. 2.73.

В нижней части приложения находится прямоугольная область, предназначенная для вывода служебной информации во время работы приложения. Подробнее об этом будет рассказано в *главе 13*, посвященной созданию полноценного приложения с графическим пользовательским интерфейсом.

Структура исходного кода данного приложения более сложная, чем изучавшаяся нами ранее. Она также будет подробно рассмотрена в *главе 13*.

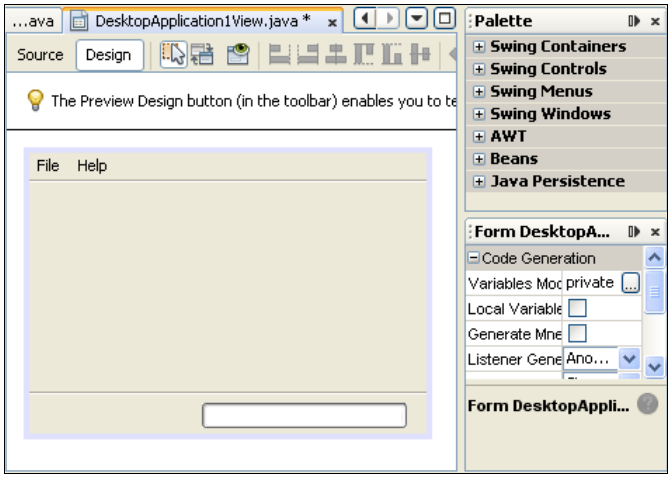


Рис. 2.72. Создание приложения Desktop Application. Заготовка диалоговой формы

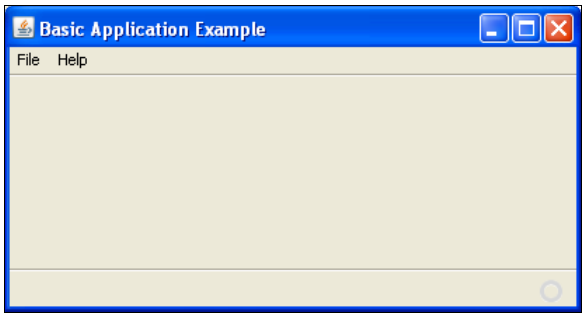


Рис. 2.73. Внешний вид приложения Desktop Application

Краткие итоги по главе

- Визуальное проектирование приложения с графическим интерфейсом пользователя (GUI) происходит в режиме **Design**. Как правило, основой для построения такого интерфейса служат компоненты `Swing`.

- *Горячие клавиши* — это отдельные клавиши или их комбинации, которые срабатывают непосредственно при нажатии данной комбинации и предназначены для дублирования пунктов меню без навигации по меню. Обычно используются функциональные клавиши <F1>, <F2> и т. д., а также сочетание обычных клавиш с <Ctrl>, <Alt> или <Shift>.
- *Акселераторы* — это клавиши, ускоряющие навигацию по пунктам меню. Они никак не связаны с горячими клавишами и могут задаваться независимо. Обычно в качестве акселератора задают одну из букв пункта меню, при этом во время ускоренной навигации клавиши-акселераторы оказываются подчеркнутыми.
- Документирование исходного кода в Java осуществляется с помощью специальных документационных комментариев `/** Текст комментария в формате HTML */`. Также имеется ряд команд документации, начинающихся с символа `@`. Утилита javadoc позволяет по документационным комментариям создавать систему HTML-страниц с документацией о пакетах и классах.
- Для выдачи пользователю информационного сообщения следует использовать вызов вида:

```
JOptionPane.showMessageDialog(null," Текст сообщения ", "Заголовок сообщения",
JOptionPane.INFORMATION_MESSAGE).
```

- При изменении размера формы взаимное расположение компонентов меняется в зависимости от типа менеджера размещения (`Layout`).
- Для привязки компонентов к краям формы или краям группирующих компонентов (контейнеров) следует применять якоря (Anchors). Они обеспечивают привязку компонентов и их сторон к нужным сторонам формы, что обеспечивает движение этих компонентов или изменение их размеров при изменении размеров формы.
- Менеджер размещения `Free Design` (установленный по умолчанию) позволяет пользоваться якорями. В других менеджерах размещения они не работают.
- Для создания полноценного приложения с графическим пользовательским интерфейсом следует выбрать заготовку Desktop Application.

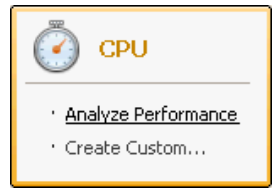
Типичные ошибки

- При работе с радиокнопками `JRadioButton` или кнопками `JToggleButton` забывают назначить им группу `ButtonGroup`.

Задания

- Создать новый проект NetBeans как Java Application. Импортировать классы пакета `swing`. В методе `main` вызвать диалоговую панель `JOptionPane` с каким-либо сообщением (например, "Привет всем!").
- На основе проекта с имеющимся кодом (заготовки приложения с графическим интерфейсом) создать приложение с кнопкой **OK**, закрывающей приложение, и кнопкой **Нажми меня**, вызывающей при нажатии диалоговую панель с сообщением **Меня нажали**.

- Переименовать пункты меню приложения, переведя их на русский язык.
- Добавить в класс приложения общедоступное числовое поле `x`, инициализированное неким значением, и поле `y` того же типа, но без инициализации. Добавить кнопку, вызывающую диалоговую панель с сообщением о значениях полей `x` и `y`.
- Создать документационные комментарии для поля `x` и методов (обработчиков событий нажатия на кнопки). Вызвать генерацию документации для проекта, просмотреть в ней созданные комментарии.



ГЛАВА 3

Примитивные типы данных и операторы для работы с ними

3.1. Булевый (логический) тип

Величины типа `boolean` принимают значения `true` или `false`.

Объявление булевых переменных:

```
boolean a;
boolean b;
```

Примеры выражений присваивания:

```
a = true;
b = a;
```

Булевы величины обычно используются в логических операторах (табл. 3.1 и 3.2) и в операциях отношения (табл. 3.3).

Таблица 3.1. *Логические операторы*

Оператор	Название	Пример
`&&`	`and` (логическое И)	`a&&b`
`||`	`or` (логическое ИЛИ)	`a||b`
`^`	`xor` (логическое исключающее ИЛИ)	`a^b`
`!`	`not` (логическое НЕ)	`!a`

Логическое выражение принимает значения `true` или `false`, например:

если `a = true`, `b = true`, то `a && b` имеет значение `true`;

если `a = false` или `b = false`, то выражение `a && b` принимает значение `false`.

Для работы с логическими выражениями часто применяют так называемые таблицы истинности. В них вместо логической единицы (`true`) пишут 1, а вместо логического нуля (`false`) пишут 0. В табл. 3.2 указаны значения логических выражений при всех возможных комбинациях значений `a` и `b`.

Выполнение булевых операторов происходит на аппаратном уровне, а значит, очень быстро. Реально процессор оперирует числами 0 и 1, хотя в Java это осуществляется скрытым от программиста образом.

Таблица 3.2. *Таблица истинности для булевых операторов*

Выражение	Значения			
`a`	0	0	1	1
`b`	0	1	0	1
`a&&b`	0	0	0	1
`a||b`	0	1	1	1
`a^b`	0	1	1	0
`!a`	1	1	0	0

Логические выражения в Java вычисляются в соответствии с так называемым укороченным оцениванием: из табл. 3.2 видно, что если `a` имеет значение `false`, то значение оператора `a&&b` будет равно `false` независимо от значения `b`. Поэтому, если `b` является булевым выражением, его можно не вычислять. Аналогично, если `a` имеет значение `true`, то значение оператора `a||b` будет равно `true` независимо от значения `b`.

Операции отношения — это операторы сравнения и принадлежности (табл. 3.3). Они имеют результат типа `boolean`, применимы к любым величинам `a` и `b` одного типа, а также к произвольным числовым величинам `a` и `b`, не обязательно одинакового типа.

Таблица 3.3. *Операторы сравнения и принадлежности*

Оператор	Название	Пример
`==`	Равно	`a==b`
`!=`	Не равно	`a!=b`
`>`	Больше	`a>b`
`<`	Меньше	`a<b`
`>=`	Больше или равно	`a>=b`
`<=`	Меньше или не равно	`a<=b`
`instanceof`	Принадлежность объекта классу *(см. разд. 6.12 и 9.3)*	`obj instanceof MyClass`

3.2. Целые типы, переменные, константы

В табл. 3.4 приведены основные характеристики целых типов, используемых в Java.

Для задания в тексте программы численных литерных констант типа `long`, выходящих за пределы диапазона чисел типа `int`, после написания числа следует ставить постфикс — букву `L`, например `600000000000000L`.

Допустима и строчная `l`, но ее хуже видно, особенно на распечатках программы (можно перепутать с единицей). В остальных случаях для всех целочисленных типов значение указывается в обычном виде, оно считается имеющим тип `int`, но при присваивании число типа `int` автоматически преобразуется в значение типа `long`.

Таблица 3.4. *Целочисленные типы*

Тип	Число байтов	Диапазон значений	Описание
`byte`	1	`-128..127`	Однобайтовое целое число (8-битовое целое со знаком)
`short`	2	$-2^{15}..2^{15}-1$ = `-32768.. 32767`	Короткое целое число (16-битовое целое со знаком)
`char`	2	`\u0000..\uFFFF=0.. 65535`	Символьный тип (беззнаковое 16-битовое целое)
`int`	4	$-2^{31}..2^{31}-1$ = $-2.147483648 \cdot 10^{9}..$ $2.147483647 \cdot 10^{9}$	Целое число (32-битовое целое со знаком)
`long`	8	$-2^{63}..2^{63}-1$ = $-9.22337203685478 \cdot 10^{18}..$ $9.22337203685478 \cdot 10^{18}$	Длинное целое число (64-битовое целое со знаком)

Замечание

Язык Java имеет общий с языками C и C++ синтаксис управляющих конструкций (простейших операторов языка), а также названия стандартных примитивных типов данных. Кажущееся сходство может вызвать опасные ошибки при переходе с одного языка на другой, т. к. в C и C++ тип `int` эквивалентен типам `short` или `int` языка Java в зависимости от разрядности процессора, для которого компилируется программа. Тип `long` языков C и C++ эквивалентен типу `int` языка Java. Аналога типа `long` языка Java в стандарте языков C и C++ нет, но во многих реализациях имеется нестандартный тип с названиями `long long`, `int64`, `_int64` или `__int64`. Для компилятора GNU g++ этот тип называется `long long`, а `int` эквивалентен `long`.

Как уже говорилось, переменные можно объявить либо в классе, либо в методе.

При объявлении в классе значение целочисленной переменной можно не указывать, при этом по умолчанию оно равно нулю. Если объявляется несколько переменных одного типа, то после указания имени типа их разрешается перечислять и присваивать им начальные значения.

В методе все переменные перед использованием обязательно нужно инициализировать — автоматической инициализации для них не происходит. Это следует делать либо при объявлении, либо в любом месте до первого обращения к переменной в подпрограмме (иначе возникнет ошибка во время компиляции). В листинге 3.1 приведен пример задания переменных в классе.

Листинг 3.1. Пример задания переменных в классе

```
int   i,j,k;
int   j1;
byte i1,i2=-5;
short i3=-15600;
long m1=1,m2,m3=-100;
```

Заметим, что после указанных объявлений переменные `i`, `j`, `k`, `j1`, `i1`, `m2` не инициализированы, т. е. не имеют никакого значения. До первого использования в качестве параметра или в выражении в правой части присваивания переменная обязательно должна быть инициализирована путем присваивания ей значения, иначе компилятор выдаст сообщение об ошибке вида:

```
Variable имяПеременной might not have been initialized —
(Переменная имяПеременной, возможно, не была инициализирована).
```

Для `i1` это неочевидно — можно подумать, что обе переменные инициализируются значением `-5`. Поэтому лучше писать так:

```
byte i1=0,i2=-5;
```

После задания переменных их можно использовать в выражениях (листинг 3.2).

Листинг 3.2. Использование переменных в выражениях

```
i=5;
j=i*i + 1
m1=j
m2=255;
m1=m1 + m2*2;
```

Тип `char` в Java, как и в C/C++, *числовой*, хотя и предназначен для хранения отдельных символов. Переменной символьного типа можно присваивать один символ, заключенный в одинарные кавычки, либо кодирующую символ управляющую последовательность Unicode, либо можно присваивать числовой код символа Unicode (номер символа в кодовой таблице):

```
char  c1='a';
char  c2='\u0061';
char  c3=97;
```

Все три приведенные декларации переменных присваивают переменным десятичное значение `97`, соответствующее латинской букве `a`.

Существуют различные кодовые таблицы для сопоставления числам символов, которые могут быть отображены на экране или другом устройстве. В силу исторических причин самая распространенная кодовая таблица — ASCII для символов латиницы, цифр и стандартных специальных символов. Поэтому в таблице Unicode им были даны те же номера, и для них коды Unicode и ASCII совпадают (листинг 3.3).

Листинг 3.3. Примеры символов таблиц Unicode и ASCII с совпадающими кодами

'A' имеет код 65, '9' — 57;
'B' — 66; ':' — 58;
'Z' — 90; ';' — 59;
'a' — 97; '<' — 60;
'z' — 122; '=' — 61;
'0' — 48; '>' — 62
'1' — 49;

и т. д.

К сожалению, с переменными и значениями символьного типа можно совершать все действия, которые разрешено совершать с целыми числами. Поэтому символьные значения можно складывать и вычитать, умножать и делить не только на обычные целые величины, но и друг на друга.

Таким образом, выражение

```
c1 = 'a' * 'a' + 1000 / 'b'
```

вполне допустимо, несмотря на явную логическую абсурдность.

Константами называются именованные ячейки памяти с неизменяемым содержимым. Объявление констант осуществляется в классе, при этом перед именем типа константы ставится комбинация зарезервированных слов `public` и `final`:

```
public final int MAX1=255;
public final int MILLENIUM=1000;
```

Константами можно пользоваться как переменными, доступными только для чтения. Попытка присвоить константе значение с помощью оператора присваивания вызывает ошибку компиляции.

Для того чтобы имена констант были хорошо видны в тексте программы, их обычно пишут в верхнем регистре (заглавными буквами).

Существует правило хорошего тона: никогда не использовать одно и то же числовое литерное значение в разных местах программы — вместо этого следует задать константу и обращаться в соответствующих местах именно к константе. Например, мы пишем программу обработки текста, в которой во многих местах встречается литерная константа 26 (число букв в английском алфавите). Если у нас возникнет задача модифицировать эту константу для работы с алфавитом, в котором другое число букв (например, с русским), то программу придется корректировать во многих местах. При этом нет никакой гарантии, что не будет забыто какое-либо из необходимых исправлений или случайно не будет задано новое значение литерному выражению 26, не имеющему никакого отношения к числу букв алфавита. Оптимистам, считающим, что проблема решается поиском числа 26 по файлам проекта (в некоторых средах разработки такое возможно), приведу пример, когда это не поможет: символ подчеркивания "_" иногда (но не всегда) считается буквой. Поэтому в одних местах программы будет фигурировать число 27, а в других — 26 (а исправления нужно будет вносить и туда, и туда). По-

этому работа перестает быть чисто технической — при изменениях потребуется вникать в смысл участка программы, и всегда есть шанс ошибиться.

Именованные константы полностью решают эту проблему. Если мы введем константу `CHARS_COUNT = 26`, то вместо `27` будет записано:

```
CHARS_COUNT + 1,
```

и смена значения `CHARS_COUNT` повлияет правильным образом и на это место программы (т. е. достаточно внести одно изменение, и можно гарантированно получить правильный результат для всех мест программы, где необходимы данные исправления).

3.3. Основные операторы для работы с целочисленными величинами

Перечисленные в табл. 3.5 операторы предназначены для работы с целыми величинами. Они действуют на две целочисленные величины или выражения, которые называются *операндами*. В результате действия оператора возвращается целочисленный результат. Во всех перечисленных далее примерах `i` и `j` обозначают целочисленные выражения, а `v` — целочисленную переменную.

Таблица 3.5. *Основные операторы для работы с целочисленными величинами*

Оператор	Название	Пример	Примечание
`+`	Оператор сложения	`i+j`	Если операнды `i` и `j` имеют разные типы или типы `byte`, `short` или `char`, то действуют правила автоматического преобразования типов
`-`	Оператор вычитания	`i-j`	
`*`	Оператор умножения	`i*j`	
`/`	Оператор деления	`i/j`	Результат округляется до целого путем отбрасывания дробной части как для положительных, так и для отрицательных чисел
`%`	Оператор остатка от целочисленного деления	`i%j`	Возвращается остаток от целочисленного деления
`=`	Оператор присваивания	`v=i`	Сначала вычисляется выражение `i`, после чего полученный результат копируется в ячейку `v`

`++`	Оператор инкремента (увеличения на 1)	`v++`
		`++v`
			`v++` эквивалентно `v=v+1`
			`++v` эквивалентно `v=v+1`
			О различии префиксной и постфиксной форм *см. разд. 5.5*

`--`	Оператор декремента (уменьшения на 1)	`v--`
		`--v`
			`v--` эквивалентно `v=v-1`
			`--v` эквивалентно `v=v-1`
			О различии префиксной и постфиксной форм *см. разд. 5.5*

`+=`		`v+=i`	`v+=i` эквивалентно `v=v+i`

***Таблица 3.5** (окончание)*

Оператор	Название	Пример	Примечание
-=		v-=i	v-=i эквивалентно v=v-i
*=		v*=i	v*=i эквивалентно v=v*i
/=		v/=i	v/=i эквивалентно v=v/i
%=		v%=i	v%=i эквивалентно v=v%i

Также предусмотрены важные методы классов `Integer` и `Long`, обеспечивающие преобразование строкового представления числа в целое значение:

```
Integer.parseInt(строка)
Long.parseLong(строка)
```

Например, если в экранной форме имеются текстовые пункты ввода `jTextField1` и `jTextField2`, то преобразование введенного в них текста в число может проводиться так:

```
int n=Integer.parseInt(jTextField1.getText());
long n1=Long.parseLong(jTextField2.getText());
```

Функции `Integer.signum(число)` и `Long.signum(число)` возвращают знак числа (т. е. `1` — если число положительное, `0` — если оно равно `0`, `−1` — если число отрицательное).

Кроме того, в классах `Integer` и `Long` есть ряд операторов для работы с числами на уровне их битового представления.

Классы `Integer` и `Long` являются так называемыми оболочечными классами (wrappers), о них речь пойдет чуть позже.

3.4. Вещественные типы и класс *Math*

В табл. 3.6 приведены основные характеристики вещественных типов Java, а в табл. 3.7 — основные операторы для работы с ними.

В классе `java.lang.Math` заданы константы $\pi = 3{,}14...$ (`Math.PI`) и $e = 2{,}71...$ (`Math.E`), а также математические функции (табл. 3.8).

***Таблица 3.6.** Вещественные типы*

Тип	Размер	Диапазон	Описание
`float`	4 байта (32 бита)	$1.5 \cdot 10^{-45}..3.4 \cdot 10^{38}$	Одинарная точность, 7—8 значащих десятичных цифр мантиссы. Тип `real*4` стандарта IEEE754
`double`	8 байт (64 бита)	$5 \cdot 10^{-324}..1.7 \cdot 10^{308}$	Двойная точность, 15—16 значащих цифр мантиссы. Тип `real*8` стандарта IEEE754

Таблица 3.7. *Основные операторы для работы с вещественными типами*

Оператор	Название	Пример	Примечание
`+`	Оператор сложения	`x+y`	В случае, когда операнды `x` и `y` имеют разные типы, действуют правила автоматического преобразования типов
`-`	Оператор вычитания	`x-y`	
`*`	Оператор умножения	`x*y`	
`/`	Оператор деления	`x/y`	Результат является вещественным. В случае, когда операнды `x` и `y` имеют разные типы, действуют правила автоматического преобразования типов
`%`	Оператор остатка от целочисленного деления	`x%y`	Возвращается остаток от целочисленного деления `x` на `y`. В случае, когда операнды `x` и `y` имеют разные типы, действуют правила автоматического преобразования типов
`=`	Оператор присваивания	`v=x`	Сначала вычисляется выражение `x`, после чего полученный результат копируется в ячейку `v`
`++`	Оператор инкремента (увеличения на 1)	`v++` `++v`	Эквивалентно `v=v+1`
`--`	Оператор декремента (уменьшения на 1)	`v--` `--v`	Эквивалентно `v=v-1`
`+=`		`v+=x`	Эквивалентно `v=v+x`
`-=`		`v-=x`	Эквивалентно `v=v-x`
`*=`		`v*=x`	Эквивалентно `v=v*x`
`/=`		`v/=x`	Эквивалентно `v=v/x`
`%=`		`v%=x`	Эквивалентно `v=v%x`

Таблица 3.8. *Функции, заданные в классе* `Math`

Функция	Примечание
Тригонометрические и обратные тригонометрические функции	
`sin(x)`	sin(*x*) — синус
`cos(x)`	cos(*x*) — косинус
`tan(x)`	tg(*x*) — тангенс
`asin(x)`	arcsin(*x*) — арксинус
`acos(x)`	arccos(*x*) — арккосинус
`atan(x)`	arctg(*x*) — арктангенс
`atan2(y, x)`	Возвращает угол, соответствующий точке с координатами `x,y`, лежащий в пределах от –π до π (–π не входит в интервал допустимых значений)

Таблица 3.8 *(продолжение)*

Функция	Примечание
`toRadians(angdeg)`	angdeg / 180.0 · *PI*; — перевод углов из градусов в радианы
`toDegrees(angrad)`	angrad · 180.0 / *PI*; — перевод углов из радиан в градусы
Степени, экспоненты, логарифмы	
`exp(x)`	e^x — экспонента
`expm1(x)`	$e^x - 1$. При x, близком к 0, дает гораздо более точные значения, чем exp(x) –1
`log(x)`	ln(x) — натуральный логарифм
`log10(x)`	$\log_{10}(x)$ — десятичный логарифм
`log1p(x)`	ln(1 + x). При x, близком к 0, дает гораздо более точные значения, чем log(1 + x)
`sqrt(x)`	$\sqrt{x}$ — квадратный корень
`cbrt(x)`	$\sqrt[3]{x}$ — кубический корень
`hypot(x,y)`	$\sqrt{x^2+y^2}$ — вычисление длины гипотенузы по двум катетам
`pow(x, y)`	x^y — возведение x в степень y
`sinh(x)`	$\mathrm{sh}(x)=\frac{e^x-e^{-x}}{2}$ — гиперболический синус
`cosh(x)`	$\mathrm{ch}(x)=\frac{e^x+e^{-x}}{2}$ — гиперболический косинус
`tanh(x)`	$\mathrm{th}(x)=\frac{e^x-e^{-x}}{e^x+e^{-x}}$ — гиперболический тангенс
Модуль, знак, минимальное, максимальное число	
`abs(m), abs(x)`	Абсолютное значение числа. Аргумент типа `int`, `long`, `float` или `double`. Результат того же типа, что аргумент
`signum(a), signum(x)`	Знак числа. Аргумент типа `float` или `double`. Результат того же типа, что аргумент
`min(m,n), min(x,y)`	Минимальное из двух чисел. Аргументы одного типа. Возможны типы: `int`, `long`, `float`, `double`. Результат того же типа, что аргумент
`max(m,n), max(x,y)`	Максимальное из двух чисел. Аргументы одного типа. Возможны типы: `int`, `long`, `float`, `double`. Результат того же типа, что аргумент
Округления	
`ceil(x)`	Ближайшее к x целое, большее или равное x
`floor(x)`	Ближайшее к x целое, меньшее или равное x

Таблица 3.8 *(окончание)*

Функция	Примечание
`round(a)` `round(x)`	Ближайшее к *x* целое. Аргумент типа `float` или `double`. Результат типа `long`, если аргумент `double`, и типа `int` — если `float`. То же, что `(int)floor(x + 0.5)`
`rint(x)`	Ближайшее к *x* целое
`ulp(a), ulp(x)`	Расстояние до ближайшего большего чем аргумент значения того же типа (дискретность изменения чисел в формате с плавающей точкой вблизи данного значения). Аргумент типа `float` или `double`. Результат того же типа, что аргумент
Случайное число, остаток	
`random()`	Псевдослучайное число в диапазоне от 0,0 до 1,0. При этом: `0 <= Math.random() < 1`
`IEEEremainder(x,y)`	Погрешность целочисленного деления *x* на *y*, т. е. разность: *x — y * n*, где *n* — результат целочисленного деления

Замечание

В табл. 3.8 величины `x`, `y`, `angdeg`, `angrad` имеют тип `double`, величина `a` — тип `float`, величины `m`, `n` — целые типов `long` или `int`. Математические функции возвращают значения типа `double`, если в примечании не указано иное.

При обращении к математическим функциям указывают имя функции (или константы), квалифицированное впереди именем класса `Math`.

Например, возможны вызовы `Math.PI` или `Math.sin(x)`. При этом имя пакета `java.lang` указывать не нужно — он импортируется автоматически. Напомним, что указывать математические функции без квалификатора `Math` в старых версиях Java было нельзя, т. к. это методы класса. Но начиная с jdk 1.5, для этих целей можно воспользоваться статическим импортом:

```
import static java.lang.Math.*;
```

Константы в классе `Math` заданы так:

```
public static final double E = 2.7182818284590452354;
public static final double PI = 3.14159265358979323846;
```

Модификатор `static` означает, что это переменная класса, `final` — что изменять это значение нельзя (в том числе переопределять в классе-наследнике).

При работе с числами кроме математических функций часто применяются методы оболочечных классов `Float` и `Double`, обеспечивающие преобразование строкового представления числа в значение типа `Float` или `Double`:

```
Float.parseFloat(строка)
Double.parseDouble(строка)
```

Например, если в экранной форме имеются текстовые пункты ввода `jTextField1` и `jTextField2`, то преобразовать введенный в них текст в число можно так:

```
float f1= Float.parseFloat(jTextField1.getText());
double d1= Double.parseDouble(jTextField2.getText());
```

Иногда вместо класса `Math` ошибочно пытаются использовать пакет `java.math`, в котором содержатся классы `BigDecimal`, `BigInteger`, `MathContext`, `RoundingMode`.

Класс `BigDecimal` обеспечивает работу с десятичными числами с произвольным числом значащих цифр (в том числе с произвольным числом цифр после десятичной точки).

Класс `BigInteger` обеспечивает работу с целыми числами произвольной длины.

Классы `MathContext` и `RoundingMode` вспомогательные, обеспечивающие настройки для работы с некоторыми методами классов `BigDecimal` и `BigInteger`.

3.5. Правила явного и автоматического преобразования типа при работе с числовыми величинами

Виртуальный процессор Java-машины может проводить на аппаратном уровне целочисленные математические вычисления только с величинами типа `int` или `long`. При этом операнды в основных математических операциях (`+`, `–`, `*`, `/`, `%`) должны быть одного типа.

В тех случаях, когда операнды имеют разные типы или типы `byte`, `short` или `char`, действуют правила автоматического преобразования типов: для величин типа `byte`, `short` или `char` сначала происходит преобразование в тип `int`, после чего производится их подстановка в качестве операндов. Если же один из операндов имеет тип `long`, то действия производятся с числами типа `long`, поскольку второй операнд автоматически преобразуется к этому типу.

Аналогично при работе с вещественными величинами в интерпретаторе Java возможна работа на аппаратном уровне (математический сопроцессор) только с операндами типов `float` и `double`. При этом операнды в основных математических операциях должны быть одного типа. Если один из операндов имеет тип `double`, а другой `float`, то действия производятся с числами типа `double`, поскольку операнд типа `float` автоматически преобразуется к типу `double`.

Если один из операндов целочисленный, а другой вещественный, то сначала идет преобразование целочисленного операнда к такому же вещественному типу, а уже потом выполняется оператор.

Рассмотрим теперь правила совместимости типов по присваиванию. Они просты: диапазон значений типа левой части не должен быть у́же, чем диапазон типа правой. Поэтому в присваиваниях, где тип в правой части не умещается в диапазон левой части, требуется указывать *явное преобразование типа*. Иначе компилятор выдаст сообщение об ошибке с не очень адекватной диагностикой:

```
possible loss of precision (возможная потеря точности).
```

Для явного преобразования типа в круглых скобках перед преобразуемой величиной ставят имя того типа, к которому нужно преобразовать.

Например, пусть имеется:

```
double d=1.5;
```

Величину `d` можно преобразовать к типу `float` таким образом:

```
(float)d
```

Аналогично, если имеется величина `f` типа `float`, то ее можно преобразовать в величину типа `double` так:

```
(double)f
```

Во многих случаях к явному преобразованию типов прибегать не нужно, т. к. действует автоматическое преобразование, если переменной, имеющей тип с более широким диапазоном изменения, присваивается выражение, имеющее тип с более узким диапазоном изменения.

Для вещественных типов разрешено присваивание только в том случае, когда слева стоит вещественная переменная более широкого диапазона, чем целочисленная или вещественная справа. Например, переменной типа `double` можно присвоить значение типа `float`, но не наоборот. Но можно преобразовать типы, чтобы выполнить такое присваивание:

```
double d=1.5;
float f=(float)d;
```

Листинг 3.4 иллюстрирует примеры допустимых присваиваний.

Листинг 3.4. Примеры допустимых присваиваний переменным числовых типов

```
byte byte0 = 1;  //-128..127
short short0 = 1;//-32768.. 32767
char char0 = 1;//0.. 65535
int int0 = 1;    //-2.147483648E9.. 2.147483647E9
long long0 = 1;//-9.223E18.. 9.223E18
float float0 = 1;// ±(1.4E-45..3.402E38)
double double0 = 1;// ±(4.9E-324..1.797E308 )
short0 = byte0;
byte0 = (byte)short0;
char0 = (char)short0;
int0 = short0;
int0 = char0;
char0 = (char)int0;
short0 = (short)int0;
long0 = byte0;
byte0 = (byte)long0;
long0 = char0;
long0 = int0;
int0 = (int)long0;
float0 = byte0;
float0 = int0;
float0 = (float)double0;
double0 = float0;
```

3.6. Оболочечные классы. Упаковка (*boxing*) и распаковка (*unboxing*)

В ряде случаев, например для работы со списками объектов, вместо значения примитивного типа требуется объект. Это связано с тем, что работа с объектами в Java может быть унифицирована, поскольку все классы Java являются наследниками класса `Object`, а для примитивных типов этого сделать нельзя.

Для таких целей в Java каждому примитивному типу сопоставляется объектный тип, т. е. класс. Такие классы называются оболочечными (class wrappers). В общем случае они имеют те же имена, что и примитивные типы, но начинающиеся не со строчной, а с заглавной буквы. Исключение почему-то составляют типы `int` и `char`, для которых имена оболочечных классов `Integer` и `Character`. Сопоставление примитивных типов и оболочечных классов приведено в табл. 3.9.

***Таблица 3.9.** Сопоставление примитивных типов и оболочечных классов*

Примитивный тип	Оболочечный класс
`byte`	`Byte`
`short`	`Short`
`char`	`Character`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Внимание!

Класс `Character` несколько отличается от остальных оболочечных числовых классов, потому что тип `char` не вполне числовой.

Основное назначение оболочечных классов — создание объектов, являющихся оболочками над значениями примитивных типов. Процесс создания такого объекта (коробки — box) из значения примитивного типа называется упаковкой (boxing), а обратное преобразование из объекта в величину примитивного типа — распаковкой (unboxing).

Оболочечные объекты ("обертки" для значения примитивного типа) хранят это значение в поле соответствующего примитивного типа, доступном по чтению с помощью функции `имяТипаValue()` (например, метода `byteValue()` для объекта типа `Byte`). Но во многих случаях можно вообще не обращать внимания на отличие переменных с типом оболочечных классов от переменных примитивных типов, т. к. упаковка и распаковка при подстановке такого объекта в выражение происходит автоматически, и объект оболочечного типа в этих случаях внешне ведет себя как число. Таким образом, если нам необходимо хранить в объекте числовое значение, то следует создать объект соответствующего оболочечного типа. Листинг 3.5 содержит примеры подобных фрагментов кода.

Листинг 3.5. Упаковка и распаковка при присваиваниях и в выражениях

```
Integer obj1 = 10; //упаковка
int i1 = obj1*2;   //распаковка при вычислении выражения
Byte b = 1;        //упаковка
obj1 = i1/10;      //упаковка
b = 2;             //упаковка
```

Но не следует забывать, что при операциях упаковки/распаковки происходят внешне невидимые дополнительные операции копирования значений в промежуточные буферные ячейки, поэтому соответствующие вычисления несколько медленнее операций с примитивными типами и требуют несколько больше памяти. Поэтому в критичных к быстродействию и занимаемым ресурсам местах программы их использование нежелательно.

С другой стороны, автоматическая упаковка/распаковка (она появилась в пятой версии JDK) во многих случаях заметно упрощает программирование и делает текст программы более читаемым. Так что для участков программы, некритичных к быстродействию и созданию мусора в памяти, ею вполне можно пользоваться.

Помимо создания объектов, оболочечные классы имеют ряд других полезных применений. Например, в числовых оболочечных классах хранятся константы, с помощью которых можно получить максимальные и минимальные значения:

```
Byte.MIN_VALUE
Byte.MAX_VALUE
Float.MIN_VALUE
Float.MAX_VALUE
Double.MIN_VALUE
Double.MAX_VALUE
```

и т. п.

В оболочечных классах также имеются методы классов, т. е. такие, которые могут работать в отсутствии объекта соответствующего типа, для их вызова можно пользоваться именем типа. Ниже приведены методы, преобразующие строку в число соответствующего типа:

```
Byte.parseByte(строка)
Short.parseShort(строка)
Integer.parseInt(строка)
Long.parseLong(строка)
Float.parseFloat(строка)
Double.parseDouble(строка)
```

Вызовы, приведенные далее, аналогичны им, но возвращают не числовые значения, а объекты соответствующих оболочечных типов:

```
Byte.valueOf(строка)
Short.valueOf(строка)
Integer.valueOf(строка)
Long.valueOf (строка)
Float.valueOf (строка)
Double.valueOf (строка)
```

Примеры использования оболочечных классов:

```
int n1=Integer.MAX_VALUE;
double d1= Double.MIN_VALUE;
```

Отметим, что присваивание

```
double d2= Double.parseDouble(jTextField1.getText());
```

функционально будет работать совершенно так же, как:

```
double d2= Double.valueOf(jTextField1.getText());
```

несмотря на то, что во втором случае методом `valueOf` создается объект оболочечного типа `Double`.

Но скорость работы первого варианта будет больше, т. к. отсутствуют дополнительные промежуточные операции, и объем задействованной памяти будет меньше. Поскольку в левой части присваивания стоит переменная типа `double`, то происходит автоматическая распаковка, и переменной `d2` присваивается распакованное значение. Сам объект при этом становится мусором, программная связь с ним теряется, и он через некоторое время удаляется из памяти системой сборки мусора.

Но в данном случае ни быстродействие, ни объем памяти некритичны, поскольку операции взаимодействия с пользователем по компьютерным меркам очень медленные, а один объект оболочечного типа занимает пренебрежимо мало места в памяти (около сотни байтов). Так что с потерями ресурсов в этом случае можно не считаться, обращая внимание только на читаемость текста программы. Поэтому автор предпочитает второй вариант присваивания, хотя он и неоптимальный по затрате ресурсов (но удобнее для восприятия).

3.7. Приоритет операторов

При вычислении выражений важен приоритет операторов. Для операторов сложения, вычитания, умножения и деления он естественный: умножение и деление обладают одинаковым наиболее высоким приоритетом, а сложение и вычитание — одинаковым приоритетом, который ниже. Таким образом, например, выражения

`a*b/c+d` и `((a*b)/c)+d`

эквивалентны.

Круглые скобки позволяют группировать элементы выражений, при этом выражение в скобках вычисляется до того, как участвует в вычислении остальной части выражения (т. е. скобки обеспечивают больший приоритет, чем все остальные операторы).

Поэтому `(a+b)*c` будет вычисляться так: сначала будет вычислена сумма `a+b`, после чего полученный результат умножен на значение `c`.

Кроме перечисленных, в Java существует множество других правил, определяющих приоритеты различных операторов. Автор считает их изучение не только нецелесообразным, но даже вредным, поскольку программу следует писать так, чтобы все последовательности действий были очевидны и не могли вызывать сложности в понимании текста программы (что в конечном итоге может привести к логической ошибке). Поэтому рекомендуется расставлять скобки даже в тех случаях, когда они теоретически не

нужны, но, тем не менее, делают очевидной последовательность действий. Отметим, что такой подход часто помогает заодно решить гораздо более сложные проблемы, связанные с арифметическим переполнением.

В справочных целях приведена табл. 3.10 с перечислением приоритета операторов, которой имеет смысл пользоваться при анализе плохо написанных программ (когда из текста программы неясна последовательность операторов).

Таблица 3.10. *Таблица приоритета операторов*

Приоритет	Группа операторов	Операторы				
1 *высший*	Постфиксные	`( )`	`[ ]`	`.`		
2	Унарные	`++` операнд операнд `++`	`--`операнд операнд`--`	`~`	`!`	`+` операнд `-` операнд
3	Создания объектов и преобразования типа	`new`	(тип) операнд			
4	Мультипликативные	`*`	`/`	`%`		
5	Аддитивные	`+`	`-`			
6	Сдвиги битов	`>>`	`>>>`	`<<`		
7	Отношения	`>`	`>=`	`<`	`<=`	`instanceof`
8	Эквивалентности	`==`	`!=`			
9	Побитовое И	`&`				
10	Побитовое исключающее ИЛИ	`^`				
11	Побитовое ИЛИ	`|`				
12	Логическое И	`&&`				
13	Логическое ИЛИ	`||`				
14	Условный	`? :`				
15 *низший*	Присваивания	`=`	Оператор `=` (`+=`, `-=`, `*=`, `/=` и т. п.)			

Краткие итоги по главе

- Величины типа `boolean` принимают значения `true` или `false`.
- Логические операторы `&&` (И), `||` (ИЛИ), `^` (исключающее ИЛИ), ! (НЕ) применимы к величинам булевого типа. Логические выражения в Java вычисляются в соответствии с укороченным оцениванием.
- Операторы сравнения применимы к любым величинам `a` и `b` одного типа, а также к произвольным числовым величинам `a` и `b`, не обязательно имеющим один тип.

В качестве оператора сравнения на равенство используется составной символ, состоящий из двух подряд идущих символов равенства ==.

- В Java имеются встроенные примитивные целые типы byte, short, int, long и символьный тип char, в некотором смысле также являющийся целочисленным. При этом только тип char беззнаковый, все остальные — знаковые.
- Для задания в тексте программы численных литерных констант типа long, выходящих за пределы диапазона чисел типа int, после написания числа следует ставить постфикс — букву L.
- Константами называются именованные ячейки памяти с неизменяемым содержимым. Объявление констант осуществляется в классе, при этом перед именем типа константы ставится комбинация зарезервированных слов public и final.
- В Java имеются два встроенных примитивных вещественных типа float и double (точнее, типы чисел в формате с плавающей точкой).
- Математические функции, а также константы PI (Math.PI) и E (Math.E) заданы в классе Math, находящемся в пакете java.lang.
- Возможен статический импорт элементов класса Math в виде:

```
import static java.lang.Math.*;
```

что позволяет избежать префикса Math. при обращении к константам и функциям класса Math в местах их вызова.

- Целочисленные математические вычисления проводятся на аппаратном уровне только с величинами типа int или long. Для величин типа byte, short или char сначала происходит преобразование в тип int, после чего производится их подстановка в качестве операндов. Если же один из операндов имеет тип long, то действия производятся с числами типа long, поскольку второй операнд автоматически преобразуется к этому типу.
- При работе с вещественными величинами в Java возможна работа на аппаратном уровне только с операндами типов float и double. Если один из операндов имеет тип double, а другой float, то действия производятся с числами типа double, поскольку операнд типа float автоматически преобразуется к типу double.
- Если один из операндов целочисленный, а другой вещественный, то сначала целочисленный операнд преобразуется к такому же вещественному типу, а уже потом выполняется оператор.
- В Java каждому примитивному типу сопоставляется объектный тип, т. е. класс. Такие классы называются оболочечными (class wrappers). В общем случае они имеют те же имена, что и примитивные типы, но начинающиеся не со строчной, а с заглавной буквы. Исключение составляют типы int и char, для которых имена оболочечных классов Integer и Character.
- Основное назначение оболочечных классов — создание объектов, являющихся оболочками над значениями примитивных типов. Процесс создания такого объекта (коробки — box) из значения примитивного типа называется *упаковкой* (boxing), а обратное преобразование из объекта в величину примитивного типа — *распаковкой* (unboxing). Упаковка и распаковка для числовых классов осуществляется автоматически.

- В оболочечных классах есть ряд полезных методов и констант. Например, минимальное по модулю не равное нулю и максимальное значение числового типа можно получить с помощью констант, вызываемых через имя оболочечного типа: `Integer.MIN_VALUE`, `Integer.MAX_VALUE`, `Float.MIN_VALUE`, `Float.MAX_VALUE`, `Double.MIN_VALUE`, `Double.MAX_VALUE` и т. п.
- В Java имеются 15 уровней приоритета операторов. В хорошо написанной программе следует предусматривать скобки, повышающие читаемость программы, даже если они не нужны с точки зрения таблицы приоритетов.

Типичные ошибки

- Очень часто встречается ошибка, когда вместо сравнения численных значений вида `a==b` программист пишет `a=b`. Чаще всего такая ошибка возникает в условных операторах: вместо `if(a==b){...}` пишут `if(a=b){...}`.

  В Java подобная ошибка диагностируется на этапе компиляции для всех типов, кроме булевого. Для булевых `a` и `b` вызов `if(a=b){...}` приведет к тому, что величине `a` будет присвоено значение `b`, а результат присваивания окажется равным этому значению. Диагностики ошибки, к сожалению, выдано не будет.
- Начинающие программисты, изучавшие ранее язык Basic, пытаются писать выражение вида `a^b` для возведения в степень, т. е. для вычисления выражения a^b. В Java для такой операции предназначено выражение `Math.pow(a,b)`.
- Путают класс `java.lang.Math` и пакет `java.math`.
- Очень часто встречается ошибка, когда забывают, что величины типа `byte`, `short` или `char` сначала преобразуются в тип `int`, и только после этого подставляются в качестве операндов. Поэтому математические операции с величинами этих типов всегда дают те же результаты, что и при работе с величинами типа `int`.
- Не ставят скобки, группирующие операнды в длинных выражениях, где стороннему программисту неочевидна последовательность выполнения операндов.
- Элементы перечисления задают как строковые, или же пытаются предварительно задать числовые переменные с именами элементов перечисления.

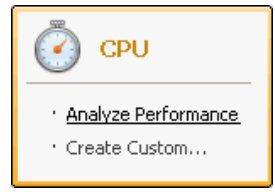
Задания

- На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем для каждого из целочисленных и вещественных типов задать переменные и кнопки с соответствующими надписями. При нажатии на кнопки должны появляться диалоговые панели с сообщением об имени и значении соответствующей переменной.
- На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем сделать два пункта ввода, метку и кнопки **Сложить**, **Умножить**, **Разделить**, **sin**. При нажатии кнопок **Сложить**, **Умножить**, **Разделить** в метку должен выводиться результат. Действия проводить с величинами типа `double`. При нажатии кнопки **sin** в метку должен выводиться синус значения, показывавшегося до того в метке.

- На основе проекта с графическим пользовательским интерфейсом создать новый проект. В нем сделать пункт ввода и радиогруппу с выбором варианта для каждого из целочисленных типов, а также кнопку `JButton` с надписью **Преобразовать в число**. При выборе соответствующего варианта в пункте ввода должно возникать случайным образом генерируемое число, лежащее в пределах допустимых значений для этого типа. При нажатии на кнопку содержимое пункта ввода должно быть преобразовано в число соответствующего типа, и это значение показано с помощью диалоговой панели с сообщением.

  Выбор вариантов реализовать следующим образом:

```
if(jRadioButton1.isSelected())
  оператор1;
if(jRadioButton2.isSelected())
  оператор2;
if(jRadioButton3.isSelected())
  оператор3;
```



ГЛАВА 4

Работа с числами в языке Java

Данная глава посвящена изучению работы с числами на более глубоком уровне. В ней рассматривается машинное представление целых и вещественных чисел, эффективное для аппаратной реализации, а также объясняются особенности и проблемы, к которым это приводит.

4.1. Двоичное представление целых чисел

На начальной стадии обучения достаточно иметь самые поверхностные знания о компьютерном представлении чисел, однако при переходе к профессиональному программированию необходимо хорошо разобраться в этом вопросе. Надежность и быстродействие расчетных алгоритмов в значительной мере зависит от правильности выбора программистом числовых типов и понимания их особенностей.

Позиционные и непозиционные системы счисления

Позиционная система счисления — это такой способ записи числа, при котором вес цифры зависит от занимаемой позиции и пропорционален степени некоторого числа. Основание степени называется основанием системы счисления.

Например, число 2006 в десятичной системе счисления представляется в виде суммы тысяч, сотен, десятков и единиц:

$2006 = 2 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 6 \cdot 10^0$,

т. е. слагаемых с различными степенями числа 10. По основанию степени (числу десять) система называется десятичной. Другие позиционные системы счисления отличаются только числом в основании степени.

При написании программ чаще всего используют десятичную, шестнадцатеричную (основание шестнадцать), восьмеричную (основание восемь) и двоичную (основание два) системы. Число различных знаков (цифр) для записи чисел в каждой системе равно основанию данной системы счисления:

- 0, 1 — цифры двоичной системы;
- 0, 1, 2, 3, 4, 5, 6, 7 — цифры восьмеричной системы;

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 — цифры десятичной системы;
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *B*, *C*, *D*, *E*, *F* — цифры шестнадцатеричной системы.

В шестнадцатеричной системе обычных десятичных цифр недостаточно, и цифры, большие девяти, обозначают заглавными латинскими буквами *A*, *B*, *C*, *D*, *E*, *F*.

В дальнейшем везде, где это необходимо, мы будем указывать основание системы счисления индексом рядом с числом, например:

- 95_{10} — в десятичной системе;
- $27F_{16}$ — в шестнадцатеричной системе;
- 6752_8 — в восьмеричной системе;
- 1000111_2 — в двоичной системе.

Помимо этого существует множество непозиционных систем счисления, в которых числа изображаются и называются по своим правилам.

Для римской системы чисел характерно:

- сопоставление отдельного знака каждому большому числу (*V* — 5, *X* — 10, *L* — 50, *C* — 100, *M* — 1000);
- повторение знака столько раз, сколько таких чисел во всем числе (III — 3, XX — 20);
- отдельные правила для предшествующих и последующих чисел (IV — 4, VI — 6, IX — 9).

Некоторые непозиционные системы счисления связаны с традиционными способами измерения конкретных величин:

- времени (секунда, минута, час, сутки, неделя, месяц, год);
- длины (дюйм, фут, ярд, миля, аршин, сажень);
- массы (унция, фунт);
- денежных единиц.

Выполнение арифметических действий в таких системах представляет собой крайне сложную задачу.

Приведем пример самой простой из возможных систем счисления — унарную, в которой имеется всего одна цифра 1. В унарной системе счисления число 1 изображается как 1, число 2 изображается как 11, число 3 как 111, число 4 как 1111, и т. д. Первоначально вместо единицы использовались палочки (помните детский сад?), поэтому такую систему счисления иногда называют палочковой. Как ни странно, она позиционная.

Позиционные системы счисления с основанием 2 более удобны для алгоритмизации математических операций с числами (вспомните способ сложения и умножения столбиком). Двоичная система является естественным способом кодирования информации в компьютере, когда сообщение представляется набором нулей (0 — нет сигнала на линии) и единиц (1 — есть сигнал на линии). Для обозначения двоичных цифр применяется термин *бит* — сокращение английского словосочетания BInary digiT (двоичная цифра).

Архитектура компьютера накладывает существенное ограничение на длину информации, обрабатываемой за одну операцию. Эта длина измеряется количеством двоичных

разрядов и называется разрядностью. С помощью восьми двоичных разрядов можно представить $2^8 = 256$ целых чисел.

Порция информации размером 8 битов (8-битовое число) служит основной единицей измерения компьютерной информации и называется байтом (byte). Как правило, передача информацией внутри компьютера и между компьютерами идет порциями, кратными целому числу байтов.

Машинным словом называют порцию данных, которую процессор компьютера может обработать за одну операцию (микрокоманду). Первые персональные компьютеры были 16-разрядными, т. е. работали с 16-битовыми (двухбайтовыми) словами. Поэтому операционные системы для этих компьютеров также были 16-разрядными (например, MS-DOS). Операционные системы для персональных компьютеров следующих поколений были 32-разрядными (Windows ’95 /’98 /NT /ME /2000 /XP, Linux, MacOS), т. к. предназначались для 32-разрядных процессоров. Современные операционные системы либо 32-разрядные, либо даже 64-разрядные (версии для 64-разрядных процессоров).

Представление чисел в двоичной и шестнадцатеричной системах счисления, а также преобразование из одной системы в другую часто необходимо при программировании аппаратуры для измерений, контроля и управления с помощью портов ввода/вывода, цифроаналоговых и аналого-цифровых преобразователей.

Двоичное представление положительных целых чисел

Целые числа в компьютере обычно кодируются в двоичном коде, т. е. в двоичной системе счисления. Например, число 5 можно представить в виде:

$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2$

Показатель системы счисления принято записывать справа снизу около числа.

Аналогично:

$6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 110_2$

$2 = 10_2$

$4 = 100_2$

$8 = 1000_2$

$9 = 1001_2$

и т. д.

Все очень похоже на обозначение чисел в десятичной системе счисления, но только в качестве основания системы счисления используется число $2 = 10_2$:

$153 = 1 \cdot 10^2 + 5 \cdot 10^1 + 3 \cdot 10^0$

У чисел, записанных в десятичной системе счисления, индекс 10 обычно опускают, но его можно писать:

$2 = 2_{10}$

$10 = 10_{10}$

и т. д.

В двоичной арифметике всего две цифры — 0 и 1. Двоичный код положительного целого числа — это коэффициенты разложения числа по степеням двойки.

Умножение числа на двоичное десять (т. е. на $10_2 = 2$) приводит к дописыванию справа нуля в двоичном представлении числа. Умножение на двоичное сто (т. е. на $100_2 = 4$) приводит к дописыванию двух нулей и т. д.

Целочисленное деление на 10_2 с отбрасыванием остатка выполняется отбрасыванием последнего (младшего) бита, деление на 100_2 — отбрасыванием двух последних битов, и т. д.

Обычно такие операции называют побитовыми сдвигами на n битов влево (умножение на 2^n) или вправо (целочисленное деление на 2^n).

Двоичные числа можно складывать в столбик по полной аналогии со сложением десятичных чисел. Единственное отличие — то, что в двоичной арифметике только две цифры (0 и 1), а не десять (от 0 до 9) как в десятичной. Поэтому если в десятичной арифметике единицу более старшего разряда дает, к примеру, сложение 1 и 9, то в двоичной арифметике ее даст сложение 1 и 1:

$1_2 + 1_2 = 10_2$

(в десятичной системе это равенство выглядит как 1 + 1 = 2).

Аналогично:

$10_2 + 10_2 = 100_2$

и т. д.

Примеры *сложения* в столбик:

$$\begin{array}{r} 0110_2 \\ +\ 1011_2 \\ \hline 10001_2 \end{array} \qquad \begin{array}{r} 1100_2 \\ +\ 0010_2 \\ \hline 1110_2 \end{array} \qquad \begin{array}{r} 111_2 \\ +\ 001_2 \\ \hline 1000_2 \end{array}$$

Совершенно аналогично выполняется *умножение*:

$$\begin{array}{r} 101_2 \\ \times\ 11_2 \\ \hline 101 \\ +\ 101 \\ \hline 1111_2 \end{array}$$

В машинной реализации целочисленного умножения используют побитовые сдвиги влево и сложение. Поскольку эти алгоритмы очень просты, они реализуются аппаратно.

Двоичное представление отрицательных целых чисел. Дополнительный код

Старший бит в целых числах без знака имеет обычный смысл, в целых со знаком для положительных чисел всегда равен 0. В отрицательных числах старший бит всегда равен 1. В примерах для простоты мы будем рассматривать четырехбитовую арифметику. Тогда в качестве примера целого положительного числа можно привести 0110_2.

Для хранения отрицательных чисел используется дополнительный код. Машинное представление положительного числа n переводится в машинное представление числа $n2 = -n$ по следующему алгоритму:

1. Число n преобразуется в число $n1$ путем преобразования $n \to n1$, во время которого все единицы числа n заменяются нулями, а нули единицами (т. е. $1 \to 0$, $0 \to 1$).
2. Перевод $n1 \to n2 = n1 + 1$ (т. е. к получившемуся числу $n1$ добавляется единица младшего разряда).

Отметим, что дополнительный код отрицательных чисел зависит от разрядности. Например, код числа (–1) в четырехразрядной арифметике будет 1111_2, а в 8-разрядной арифметике будет 11111111_2. Коды числа (–2) будут: 1110_2 и 11111110_2 и т. д.

Чтобы понять преимущества дополнительного кода, рассмотрим сложение чисел, представленных в дополнительном коде.

Сложение положительного и отрицательного чисел

Рассмотрим, чему равна сумма числа 1 и числа –1, представленного в дополнительном коде. Сначала переведем в дополнительный код число –1.

При этом $n = 1_{10}$, $n_2 = -1_{10}$.

Этап 1: $n = 1_{10} = 0001_2 \to n1 = 1110_2$;

Этап 2: $n2 = 1110_2 + 1 = 1111_2$.

Таким образом, в четырехбитовом представлении $-1_{10} = 1111_2$.

Проверка:

$n2 + n = 10000_2$.

Получившийся пятый разряд, выходящий за пределы четырехбитовой ячейки, отбрасывается, поэтому в рамках четырехбитовой арифметики получается:

$n2 + n = 0000_2 = 0$.

Аналогично:

$n = 2_{10} = 0010_2 \to n1 = 1101_2$; $n2 = 1110_2$;

$n = 3_{10} = 0011_2 \to n1 = 1100_2$; $n2 = 1101_2$;

$n = 4_{10} = 0100_2 \to n1 = 1011_2$; $n2 = 1100_2$.

Очевидно, во всех этих случаях $n2 + n = 0$.

Что будет, если мы сложим 3_{10} и -2_{10} (равное 1110_2, как мы уже знаем)?

$$\begin{array}{rr} & 0011_2 \\ + & 1110_2 \\ \hline & 10001_2 \end{array}$$

После отбрасывания старшего бита, выходящего за пределы нашей четырехбитовой ячейки, получаем:

$0011_2 + 1110_2 = 0001_2$

(т. е. $3_{10} + (-2_{10}) = 1_{10}$, как и должно быть).

Сложение отрицательных чисел

$(-1) + (-1) = 1111_2 + 1111_2 = 11110_2 \rightarrow 1110_2$ (из-за отбрасывания лишнего старшего бита, выходящего за пределы ячейки).

Поэтому:

$(-1) + (-1) = 1110_2 = -2$

Вычитание положительных чисел осуществляется путем сложения положительного числа с отрицательным, получившимся из вычитаемого в результате его перевода в дополнительный код.

Приведенные примеры иллюстрируют тот факт, что сложение положительного числа с отрицательным (хранящимся в дополнительном коде) или двух отрицательных может происходить аппаратно (что значит очень эффективно) на основе крайне простых электронных устройств.

Проблемы целочисленной машинной арифметики. Целочисленное переполнение

Несмотря на достоинства, в двоичной машинной (аппаратной) арифметике имеются очень неприятные особенности, возникающие из-за конечной разрядности машинной ячейки. Будем для простоты рассматривать четырехбитовые ячейки, хотя реально разрядность чисел, конечно, гораздо больше.

Сложение положительных чисел

Пусть:

$a = 3_{10} = 0011_2$;

$b = 2_{10} = 0010_2$;

$a + b = 0101_2 = 5_{10}$;

т. е. все в порядке.

Пусть теперь:

$a = 6_{10} = 0110_2$;

$b = 5_{10} = 0101_2$.

Тогда $a + b = 1011_2 = -5_{10}$.

То есть сложение двух положительных чисел может дать отрицательное число, если результат сложения превышает максимальное положительное число, выделяемое под целое со знаком для данной разрядности ячеек! В любом случае при выходе за разрешенный диапазон значений результат оказывается неверным.

Если у нас беззнаковые целые, проблема остается в несколько измененном виде. Сложим $8_{10} + 8_{10}$ в двоичном представлении.

Поскольку $8_{10} = 1000_2$,

то $8_{10} + 8_{10} = 1000_2 + 1000_2 = 10000_2$.

Но лишний бит отбрасывается и получаем 0.

Аналогично в четырехбитовой арифметике:

$8_{10} + 9_{10} = 1_{10}$ и т. д.

Умножение положительных чисел

Как уже говорилось, умножение двоичных чисел осуществляется путем сложения и сдвигов по алгоритму, напоминающему умножение в столбик, но гораздо более простому, т. к. умножать надо только на 0 или 1. При целочисленном умножении выход за пределы разрядности ячейки происходит гораздо чаще, чем при сложении или вычитании.

Например:

$110_2 \cdot 101_2 = 110_2 \cdot 100_2 + 110_2 \cdot 1_2 = 11000_2 + 110_2 = 11110_2$.

Так как наша ячейка четырехразрядная, то произойдет выход за ее пределы, и мы получим после отбрасывания лишнего бита $1110_2 = -2_{10} < 0$.

Таким образом, умножение целых чисел легко может дать неправильный результат (в том числе даже отрицательное число!). Поэтому при работе с целочисленными типами данных следует обращать особое внимание на то, чтобы в программе не возникало ситуаций арифметического переполнения. Повышение разрядности целочисленных переменных позволяет смягчить проблему, хотя полностью ее не устраняет.

Пусть имеются величины типа `byte`:

```
m=10, n=10, k=10
```

Тогда при проведении их умножения в рамках однобайтовой арифметики значения:

`m·n`, `m·k` и `n·k`

будут лежать в разрешенном диапазоне от –128 до 127.

А вот `m·n + m·k` из этого диапазона выйдет (не говоря уж о `m·n·k`).

Замечание

Мы рассматриваем перемножение величин, имеющих тип `byte`, используя однобайтовую арифметику. Следует помнить, что в Java перед выполнением арифметических действий происходит автоматическое преобразование таких величин к типу `int`. Поэтому, чтобы увидеть проблемы однобайтовой арифметики, в Java следует преобразовывать результат к типу `byte`, например: `(byte)(m*n)`, `(byte)(m*k)` и т. п.

Если мы зададим:

```
short m=10,n=10,k=10;
```

то переполнения не возникнет даже для `(short)(m*n*k)`.

Однако при `m=n=k=100` значение `m*n*k` будет равно 10^6, что заметно выходит за пределы разрешенного диапазона для величин типа `short` (от –32 768 до 32 767).

Хотя `m*n`, `m*k` и `n*k` не будут за него выходить (но уже `4*m*n` за него выйдет). Использование типа `int` поможет и в этом случае.

Но уже значения `m=n=k=2000` (не такие уж большие!) опять приведут к выходу `m*n*k` за пределы диапазона, теперь уже без всякого преобразования типа с нашей стороны (хотя для `m*n` выход произойдет только при значениях около 50 000).

Использование типа `long` позволяет в подавляющем большинстве случаев решить проблему переполнения (но не всегда). Например, вычисление факториала с помощью целочисленной арифметики (даже для типа `long`) даст удивительные результаты! Факториал числа 100 окажется равным нулю (читателю предлагается подумать, что произойдет при перемножении 50 четных чисел, 25 из которых делятся на 4. Рекомендуется эти числа представить в двоичной форме и подсчитать количество нулей в конце числа в получившемся результате. А потом сравнить с числом битов числа типа `long`). В таких случаях предпочтительнее числа с плавающей точкой.

Таким образом, за скорость целочисленных вычислений нам приходится платить высокую цену — резко понижается надежность программ в ситуациях, когда возможно целочисленное переполнение. Поэтому там, где не важны небольшие отличия от целочисленных значений, лучше задавать числа в формате с плавающей точкой, а там, где важны — использовать тип `long` (т. е. наши `m`, `n` и `k` лучше задать как `double`, если, конечно, это не счетчики циклов).

В противном случае решить проблему можно, например, так:

вместо `m*n*k` написать `(double)m*(double)n*k`.

На первый взгляд можно было бы написать `1.0*m*n*k`, т. к. при перемножении чисел `double` и `int` (или `double` и `long`) результат будет иметь тип `double`.

Сначала вычислится `1.0*m` и результат будет иметь тип `double`.

Затем будет вычислено `(1.0*m)*n` (при этом сначала `n` будет преобразовано в значение типа `double`, а затем перемножены два числа типа `double`).

Наконец, аналогичным образом вычислится произведение полученного значения `(1.0*m)*n` на множитель `k`.

Если мы проверим в примере 04_1_integerCalc каков результат, то обнаружим, что вроде бы все хорошо — результат правильный. Но это кажущаяся правильность. Оптимизирующий компилятор может переставлять порядок операндов, имеющих одинаковый приоритет. Поэтому вполне возможно (хотя и не обязательно), что сначала будут перемножены `m*n*k`, и только потом результат будет умножен на 1.0. Если вы скомпилируете ваш проект другим компилятором или программа изменится, не затрагивая строк с вычислением `1.0*m*n*k`, то может оказаться, что компилятор изменит порядок вычисления выражений, и вместо `1.0E10` мы получим `1.410065408E9`. Так что для гарантированности результата нужно писать `((1.0*m)*n)*k`.

Следует подчеркнуть еще один момент — от перемены мест сомножителей или слагаемых в целочисленной машинной арифметике результат может измениться, и очень значительно:

`1.0*m*n*k=1.0E10`,

а

`m*n*k*1.0=1.410065408E9`.

Но так бывает только при переполнениях.

Таким образом, при написании алгоритмов с использованием целых типов следует внимательно следить, чтобы не возникало выражений, в которых возможно целочис-

ленное переполнение. Для гарантии, что не произойдет переполнения, в соответствующих местах должны быть переменные типа `long` или `double`. Для получения необходимого типа промежуточных результатов следует явно преобразовывать типы и расставлять скобки в тех случаях, когда требуется задать гарантированную последовательность выполнения операций.

Пример:

```
byte i=127, j=1, k;
k=(byte)(i+j);
System.out.println(k);
```

В результате получим число (–128).

Если бы мы попробовали написать

```
byte i=127,j=1,k;
System.out.println(i+j);
```

то получили бы +128.

Напомним, что значения величин типа `byte` перед проведением сложения преобразуются в значения типа `int`.

Шестнадцатеричное представление целых чисел и перевод из одной системы счисления в другую

При программировании аппаратуры (различного рода внешних устройств, регистров процессора и т. п.), работе с битовыми масками и при обработке изображений приходится иметь дело с кодами беззнаковых целых чисел. При этом десятичные числа крайне неудобны из-за невозможности быстрого сопоставления числа в десятичном виде и его двоичных битов. А двоичная кодировка весьма громоздка — получаются слишком длинные последовательности нулей и единиц. Программисты пришли к компромиссному решению — шестнадцатеричной кодировке чисел, где в качестве основания системы счисления выступает число 16 ($16_{10} = 10_{16}$).

В десятичной системе счисления имеется только 10 цифр: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

А в 16-ричной системе счисления должно быть 16 цифр. Как уже упоминалось, недостающие цифры принято обозначать заглавными буквами латинского алфавита: *A*, *B*, *C*, *D*, *E*, *F*:

$10_{10} = A = A_{16}$

$11_{10} = B$

$12_{10} = C$

$13_{10} = D$

$14_{10} = E$

$15_{10} = F$

Таким образом, к примеру:

$FF_{16} = 15 \cdot 16^1 + 15 \cdot 16^0 = 255$.

В Java, для того чтобы отличать 16-ричные числа, как мы уже знаем, перед ними ставят префикс 0x:

0x*FF* обозначает FF_{16}, а 0x10 — это 10_{16}, т. е. 16_{10}.

Число *N* может быть записано с помощью разных систем счисления:

- в десятичной:

  $N = A_n \cdot 10^n + ... + A_2 \cdot 10^2 + A_1 \cdot 10^1 + A_0 \cdot 10^0$ (A_n = 0—9);

- в двоичной:

  $N = B_n \cdot 2^n + ... + B_2 \cdot 2^2 + B_1 \cdot 2^1 + B_0 \cdot 2^0$ (B_n = 0 или 1);

- в шестнадцатеричной:

  $N = C_n \cdot 16^n + ... + C_2 \cdot 16^2 + C_1 \cdot 16^1 + C_0 \cdot 16^0$ (C_n = 0—*F*).

Преобразование в другую систему счисления сводится к нахождению соответствующих коэффициентов (например, при переводе из десятичной системы в двоичную — коэффициентов B_n по известным коэффициентам A_n; при переводе из двоичной системы в десятичную — коэффициентов A_n по коэффициентам B_n).

Преобразование чисел из системы с меньшим основанием в систему с большим основанием

Рассмотрим преобразование из двоичной системы в десятичную.

Запишем число *N* в виде:

$N = B_n \cdot 2^n + ... + B_2 \cdot 2^2 + B_1 \cdot 2^1 + B_0 \cdot 2^0$ (B_n = 0 или 1)

и будем рассматривать его как алгебраическое выражение в десятичной системе. Выполним арифметические действия по правилам десятичной системы. Полученный результат даст десятичное представление числа *N*.

Пример.

Преобразуем 01011110_2 к десятичному виду:

$01011110_2 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$

$= 0 + 64 + 0 + 16 + 8 + 4 + 2 + 0 = 94_{10}$

Преобразование чисел из системы с большим основанием в систему с меньшим основанием

Для известного числа N_{10} найдем коэффициенты в выражении:

$N = B_n \cdot 2^n + ... + B_2 \cdot 2^2 + B_1 \cdot 2^1 + B_0 \cdot 2^0$ (B_n = 0 или 1).

Воспользуемся следующим алгоритмом:

1. В десятичной системе разделим число *N* на 2 с остатком.
2. Остаток от деления (он не превосходит делителя) даст коэффициент B_0 при младшей степени 2^0.
3. Далее делим на 2 частное, полученное от предыдущего деления. Остаток от деления будет следующим коэффициентом B_1 двоичной записи числа *N*.

4. Повторим эту процедуру до тех пор, пока частное не станет равным 0, в результате чего получим последовательность коэффициентов B_n.

Например, преобразуем 345_{10} к двоичному виду (табл. 4.1).

Таблица 4.1. *Преобразование числа 345 к двоичному виду*

Операция	Частное	Остаток	Коэффициент B_i
345 / 2	172	1	B_0
172 / 2	86	0	B_1
86 / 2	43	0	B_2
43 / 2	21	1	B_3
21 / 2	10	1	B_4
10 / 2	5	0	B_5
5 / 2	2	1	B_6
2 / 2	1	0	B_7
1 / 2	0	1	B_8

Таким образом $345_{10} = 101011001_2$.

Преобразование чисел в системах счисления с кратными основаниями

Рассмотрим число N в двоичном и шестнадцатеричном представлениях:

$N = B_n \cdot 2^n + ... + B_2 \cdot 2^2 + B_1 \cdot 2^1 + B_0 \cdot 2^0$ ($B_i = 0$ или 1);

$N = H_n \cdot 16^n + ... + H_2 \cdot 16^2 + H_1 \times \cdot 16^1 + H_0 \cdot 16^0$ ($H_i = 0$—F_{16}, где $F_{16} = 15_{10}$).

Заметим, что $16 = 2^4$. Объединим цифры в двоичной записи числа группами по четыре. Каждая группа из четырех двоичных цифр представляет число от 0 до F_{16} (т. е. от 0 до 15_{10}). От группы к группе вес цифры изменяется в $2^4 = 16$ раз (основание шестнадцатеричной системы). Таким образом, перевод чисел из двоичного представления в шестнадцатеричное и обратно осуществляется простой заменой всех групп из четырех двоичных цифр на шестнадцатеричные (по одному на каждую группу) и обратно:

$0000_2 = 0_{16}$
$0001_2 = 1_{16}$
$0010_2 = 2_{16}$
$0011_2 = 3_{16}$
$0100_2 = 4_{16}$
$0101_2 = 5_{16}$
$0110_2 = 6_{16}$
$0111_2 = 7_{16}$
$1000_2 = 8_{16}$
$1001_2 = 9_{16}$
$1010_2 = A_{16}$
$1011_2 = B_{16}$
$1100_2 = C_{16}$
$1101_2 = D_{16}$
$1110_2 = E_{16}$
$1111_2 = F_{16}$

Например, преобразуем 1011010111_2 к шестнадцатеричному виду:

$1011010111_2 = 0010\ 1101\ 0111_2 = 2D7_{16}$.

4.2. Побитовые маски и сдвиги

Побитовые операции — операции, в которых целые числа рассматриваются как наборы битов, где 0 и 1 играют роли логического нуля и логической единицы. При этом все логические операции для двух чисел осуществляются поразрядно — *k*-й разряд первого числа с *k*-м разрядом второго. В табл. 4.2 приведены побитовые операторы Java.

Таблица 4.2. *Побитовые операторы*

Оператор	Название	Пример
~	Оператор побитового дополнения (побитовое НЕ, побитовое отрицание)	~i
^	Оператор XOR (побитовое исключающее ИЛИ)	i^j
&	Оператор AND (побитовое И)	i&j
|	Оператор OR (побитовое ИЛИ)	i|j
<<	Оператор левого побитового сдвига	m<<n
>>>	Оператор беззнакового правого побитового сдвига	m>>>n
>>	Оператор правого побитового сдвига с сохранением знака отрицательного числа	m>>n
&=	y&=x эквивалентно y=y&x	m&=n
|=	y|=x эквивалентно y=y|x	m|=n
^=	y^=x эквивалентно y=y^x	m^=n
>>=	y>>=x эквивалентно y= y>>x	m>>=n
>>>=	y>>>=x эквивалентно y= y>>>x	m>>>=n
<<=	y<<=x эквивалентно y= y<<x	m<<=n

Остановимся подробнее на побитовых операциях (посмотрим, как производятся действия с битовыми масками). Для простоты возьмем четырехбитовые ячейки, хотя реально самая малая по размеру ячейка восьмибитовая и соответствует типу byte.

- Установка в числе a нужных битов в 1 с помощью маски m операцией a|m (арифметический или, что то же самое, побитовый оператор OR) (табл. 4.3).

  Пусть $a = a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$, где значения a_i — содержание соответствующих битов числа (т. е. либо 0, либо 1).

Таблица 4.3. *Операция OR с битовой маской*

a	a_3	a_2	a_1	a_0
m	0	1	0	1
a|m	a_3	1	a_1	1

Из табл. 4.3 видно, что независимо от начального значения в числе `a` в итоге нулевой и второй бит установились в единицу. Таким образом, операцию `OR` с маской можно использовать для установки нужных битов переменной в 1 (если нужные биты маски установлены в 1, а остальные — в 0).

- Установка в числе `a` нужных битов в 0 с помощью маски `m` операцией `a&m` (арифметический или побитовый оператор `AND`) (табл. 4.4).

Таблица 4.4. *Операция `AND` с битовой маской*

a	a_3	a_2	a_1	a_0
m	0	1	0	1
a&m	0	a_2	0	a_0

Из табл. 4.4 видно, что независимо от начального значения в числе `a` в конечном итоге первый и третий бит установились в 0. Таким образом операцию `AND` с маской можно использовать для установки нужных битов переменной в 0 (если нужные биты маски установлены в 0, а остальные — в 1).

- Инверсия (замена единиц на нули, а нулей на единицы) в битах числа `a` (стоящих на задаваемых маской `m` местах) операцией `a^m` (арифметический или побитовый оператор `XOR`) (табл. 4.5).

Из табл. 4.5 видно, что в бите, где маска `m` имеет единицу, у числа `a` происходит инверсия: если стоит 1 — в результате будет 0, а если 0 — в результате будет 1. В остальных битах значение не меняется.

Таблица 4.5. *Операция `XOR` с битовой маской*

a	1	1	0	0
m	0	1	0	1
a^m	1	0	0	1

Восстановление первоначального значения после операции `XOR` — повторное `XOR` с той же битовой маской (табл. 4.6).

Таблица 4.6. *Повторная операция `XOR` с битовой маской*

a^m	1	0	0	1
m	0	1	0	1
(a^m)^m	1	1	0	0

Из табл. 4.6 видно, что содержание ячейки приняло то же значение, что было первоначально в ячейке `a`.

Очевидно, что всегда `(a ^ m) ^ m = a`, т. к. повторная инверсия возвращает первоначальные значения в битах числа.

Операция XOR часто применяется в программировании для инверсии цветов частей экрана с сохранением в памяти только информации о маске. Повторное XOR с той же маской восстанавливает первоначальное изображение. Имеется команда перевода вывода графики в режим XOR при рисовании — `graphics.setXORMode` (цвет).

Еще одна область, где часто используется эта операция, — криптография.

Инверсия всех битов числа осуществляется с помощью побитового отрицания `~a`.

Побитовые сдвиги <<, >> и >>> приводят к перемещению всех битов ячейки, к которой применяется оператор, на указанное число битов влево или вправо.

Сначала рассмотрим действие операторов на положительные целые числа.

Побитовый сдвиг на `n` битов влево `m<<n` эквивалентен быстрому целочисленному умножению числа `m` на 2^n. Младшие биты (находящиеся справа), освобождающиеся после сдвигов, заполняются нулями. Следует учитывать, что старшие биты (находящиеся слева), выходящие за пределы ячейки, теряются, как и при обычном целочисленном переполнении.

Побитовый сдвиг на `n` битов вправо `m>>n` или `m>>>n` эквивалентен быстрому целочисленному делению числа `m` на 2^n. При этом для положительных `m` разницы между операторами >> и >>> нет.

Рассмотрим теперь операции побитовых сдвигов для отрицательных чисел `m`. Поскольку они хранятся в дополнительном коде, то их действие нетривиально. Как и раньше, для простоты будем считать, что ячейки четырехбитовые, хотя на самом деле побитовые операции проводятся только для ячеек типа `int` или `long`, т. е. для 32- или 64-битовых чисел.

Пусть `m = -1`. В этом случае $m = 1111_2$.

Оператор `m << 1` даст $m = 11110_2$, но из-за ограниченности ячейки четырьмя битами старший бит теряется, и мы получаем $m = 1110_2 = -2$ (т. е. также получается полная эквивалентность умножению `m` на 2^n).

Иная ситуация возникает при побитовых сдвигах вправо.

Оператор правого сдвига >> для положительных чисел заполняет освободившиеся биты нулями, а для отрицательных — единицами. Легко заметить, что этот оператор эквивалентен быстрому целочисленному делению числа `m` на 2^n как для положительных, так и для четных отрицательных чисел.

Оператор `m>>>n`, заполняющий нулями освободившиеся после сдвигов биты, переводит отрицательные числа в положительные. Поэтому он не может быть эквивалентен быстрому делению числа на 2^n. Но иногда такой оператор бывает нужен для манипуляции с наборами битов, хранящихся в числовой ячейке. Само значение числа в этом случае значения не имеет, а ячейка служит буфером соответствующего размера.

Например, можно преобразовать последовательность битов, образующих некое целое значение, в число типа `float` методом `Float.intBitsToFloat` (целое значение) или типа `double` методом `Double.intBitsToDouble` (целое значение). Так, `Float.intBitsToFloat(0x7F7FFFFF)` даст максимальное значение типа `float`.

4.3. Двоичное представление вещественных чисел

Для действий с вещественными числами в Java, как и в большинстве других языков программирования, предусмотрен формат хранения чисел в виде floating point (с плавающей точкой). В англоязычной традиции дробная часть числа отделяется от целой части точкой, а не запятой как в России, Китае, Голландии, Дании или Испании. В компьютерном представлении используются не десятичные дроби, а двоичные, причем с конечным числом двоичных цифр после точки.

Двоичные дроби

Целое число 0101_2 можно представить в виде:

$0101_2 = 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$.

Аналогично можно записать двоичную дробь:

$11.0101_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$.

Заметим, что сдвиг двоичной точки на `n` разрядов вправо (чаще говорят о сдвиге самого числа влево) эквивалентен умножению числа на $(10_2)^n = 2^n$. Сдвиг точки влево (т. е. сдвиг самого числа вправо) — делению на 2^n.

Мантисса и порядок числа

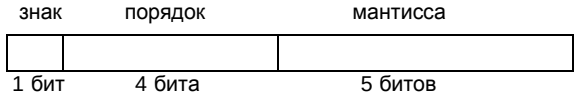
Рассмотрим сначала упрощенную схему хранения чисел в формате с плавающей точкой (floating point), несколько отличающуюся от реальной.

Число x с плавающей точкой можно представить в виде:

$x = s \cdot m \cdot 2^p$.

Здесь множитель s — знак числа, второй множитель m называется мантиссой, а число p — порядком числа.

Для простоты рассмотрим 10-битовую ячейку, состоящую из трех независимых частей:



Первым идет знаковый бит. Если он равен 0, то число положительное, если равен 1 — то отрицательное. Набор битов, хранящийся в мантиссе, задает положительное число m, лежащее в пределах $1 <= m < 2$. Оно получается из нашего двоичного числа путем переноса двоичной точки на место после первой значащей цифры числа.

Например, числа 1.0101_2, 10.101_2 и 0.10101_2 имеют одну и ту же мантиссу, равную 1.0101_2. При этом следующая за ведущей единицей точка в ячейке, выделяемой под мантиссу, не хранится, а подразумевается (т. е. мантиссы приведенных чисел будут храниться в виде 10101_2).

Число сдвигов двоичной точки (с учетом знака) хранится в части ячейки, выделяемой под порядок числа. В нашем примере числа 1.0101_2, 10.101_2 и 0.10101_2 будут иметь порядки 0, 1 и –1 соответственно.

При перемножении чисел их мантиссы перемножаются, а порядки складываются.

При делении мантиссы делятся, а порядки вычитаются.

И умножение, и деление мантисс происходит по тем же алгоритмам, что и для целых чисел. Но при выходе за размеры ячейки отбрасываются не старшие, а младшие байты.

В итоге каждая операция умножения или деления дает результат, отличающийся от точного на несколько значений младшего бита мантиссы. Аналогичная ситуация с потерей младших битов возникает при умножениях и делениях. Ведь если в ячейках для чисел данного типа хранится k значащих цифр числа, то при умножении двух чисел точный результат будет иметь $2k$ значащих цифр, последние k из которых при записи в ячейку будут отброшены даже в том случае, если они сохранялись при вычислениях. А в случае деления при точных вычислениях должна получаться бесконечная периодическая двоичная дробь, так что даже теоретически невозможно провести эти вычисления без округлений. С этим связана конечная точность вычислений на компьютерах при использовании формата с плавающей точкой. При этом чем больше двоичных разрядов выделяется под мантиссу числа, тем меньше погрешность в такого рода операциях.

Замечания

При использовании в Java типов `BigInteger` и `BigDecimal`, позволяющих работать с целыми и вещественными числами произвольной разрядности, скорость вычислений падает на порядки, а потребление оперативной памяти возрастает на порядки по сравнению с типами `long` и `double`. Поэтому их применение в расчетах очень ограничено.

Системы символьных вычислений (или аналитических вычислений, или, что то же самое, системы компьютерной алгебры) позволяют проводить точные численные расчеты с получением результатов в виде формул. Однако они выполняют вычисления еще на много порядков медленнее, требуют значительно больше ресурсов и не могут работать без громоздкой среды разработки. Поэтому для решения задач, требующих большого количества численных расчетов, они неприменимы.

При сложении или вычитании сначала происходит приведение чисел к одному порядку: мантисса числа с меньшим порядком делится на 2^n, а порядок увеличивается на n, где n — разница в порядке чисел. При этом деление на 2^n осуществляется сдвигом мантиссы на n битов вправо, с заполнением освобождающихся слева битов нулями. Младшие биты мантиссы, выходящие за пределы отведенной под нее части ячейки, теряются.

Пример.

Сложим числа 11.011_2 и 0.11011_2.

Для первого числа мантисса 1.1011_2, порядок 1, т. к. $11.011_2 = 1.1011_2 \cdot (10_2)^1$.

Для второго числа мантисса 1.1011_2, порядок –1, т. к. $0.11011_2 = 1.1011_2 \cdot (10_2)^{-1}$.

Приводим порядок второго числа к значению 1, сдвигая мантиссу на 2 места вправо, т. к. разница порядков равна 2:

$0.11011_2 = 0.011011_2 \cdot (10_2)^1$.

Но при таком сдвиге теряются два последних значащих бита мантиссы (напомним, хранится 5 битов), поэтому получаем приближенное значение $0.0110_2 \cdot (10_2)^1$, из-за чего в машинной арифметике получается:

$1.1011_2 \cdot (10_2)^1 + 0.011011_2 \cdot (10_2)^1 =$

$(1.1011_2 + 0.011011_2) \cdot (10_2)^1 \approx$

$(1.1011_2 + 0.0110_2) \cdot (10_2)^1 =$

$10.0001_2 \cdot (10_2)^1 \approx 1.0000_2 \cdot (10_2)^2$

вместо точного значения $1.0000111_2 \cdot (10_2)^2$.

Таким образом, числа в описанном формате являются на деле рациональными, а не вещественными. При этом операции сложения, вычитания, умножения и деления выполняются с погрешностями тем меньшими, чем больше разрядность мантиссы. Число двоичных разрядов, отводимых под порядок числа, влияет лишь на допустимый диапазон значений чисел и не влияет на точность вычислений.

Научная нотация записи вещественных чисел

При записи программы в текстовом файле или выдачи результатов в виде плоского текста (plain text) невозможна запись выражений типа $1.5 \cdot 10^{14}$.

В этом случае прибегают к так называемой научной нотации, когда основание 10 заменяют латинской буквой *E* (сокращение от Exponent — экспонента).

Таким образом:

$1.5 \cdot 10^{14}$ запишется как 1.5*E*14,

а $0.31 \cdot 10^{-7}$ как 0.31*E*–7.

Первоначально буква *E* писалась заглавной, что не вызывало проблем. Однако с появлением возможности набора текста программы в нижнем регистре стали употреблять строчную букву *e*, которая в математике обозначает основание натуральных логарифмов.

Запись вида 3*e*2 легко воспринять как $3e^2$, а не $3 \cdot 10^2$, поэтому заглавная буква предпочтительнее.

Литерные константы для вещественных типов по умолчанию имеют тип `double` (например, 1.5, –17*E*2, 0.0). Если требуется ввести литерную константу типа `float`, то после записи числа добавляют постфикс `f` (сокращение от float):

`1.5f`, `-17E2f`, `0.0f`.

Минимальное по модулю не равное нулю и максимальное значение типа `float` можно получить с помощью констант:

`Float.MIN_VALUE` — равна 2^{-149};

`Float.MAX_VALUE` — равна $(2–2^{-23})\cdot 2^{127} \approx 2\cdot 2^{127}$.

Аналогичные значения для типа `double` с помощью констант:

`Double.MIN_VALUE` — равна 2^{-1074};

`Double.MAX_VALUE` — равна $(2–2^{-52})\cdot 2^{1023} \approx 2\cdot 2^{1023}$.

Стандарт IEEE 754 представления чисел в формате с плавающей точкой

В каком виде на самом деле хранятся числа в формате с плавающей точкой? Ответ дает стандарт IEEE 754 (другой вариант названия IEC 60559:1989), разработанный для электронных счетных устройств. В соответствии с ним предусмотрены три типа чисел в формате с плавающей точкой, с которыми могут работать процессоры: `real*4`, `real*8` и `real*10`. Эти числа занимают 4, 8 и 10 байтов соответственно. В Java типу `real*4` соответствует `float`, а типу `real*8` — `double`. Тип `real*10` из распространенных языков программирования встречается только в диалектах языка Pascal, в Java он не применяется.

Число r представляется в виде произведения знака s, мантиссы m и экспоненты 2^{p-d}:

$r = s \cdot m \cdot 2^{p-d}$.

Число p называется порядком. Оно может меняться для разных чисел.

Значение d (называемое сдвигом порядка) постоянно для всех чисел заданного типа. Оно примерно равно половине максимального числа p_{max}, которое можно закодировать битами порядка. Точнее:

`d = (pmax+1)/2-1`.

- для чисел `real*4`: `pmax = 255`, `d = 127`.
- для чисел `real*8`: `pmax = 2047`, `d = 1023`.
- для чисел `real*10`: `pmax = 32767`, `d = 16383`.

Число называется *нормализованным* в случае, когда мантисса лежит в пределах $1 <= m < 2$. В этом случае первый бит числа всегда равен единице. Максимальное значение мантиссы достигается в случае, когда все ее биты равны 1. Оно меньше 2 на единицу младшего разряда мантиссы (т. е. с практически важной точностью может считаться равным 2).

Согласно стандарту IEEE 754 все числа формата с плавающей точкой при значениях порядка в диапазоне от 1 до $p_{max} - 1$ хранятся в нормализованном виде. Такое представление чисел будем называть *базовым*.

Когда порядок равен нулю, применяется несколько другой формат хранения чисел. Будем называть его *особым*.

Порядок p_{max} резервируется для кодировки нечисловых значений, соответствующее представление будем называть *нечисловым*. Об особом и нечисловом представлениях чисел будет сказано чуть позже.

В табл. 4.7 показано размещение чисел в формате с плавающей точкой в ячейках памяти.

Здесь:

- `s` — знаковый бит;
- `p` — биты двоичного представления порядка;
- `m` — биты двоичного представления мантиссы.

Примечание

В табл. 4.7 биты записаны в две строки для экономии места; подразумевается их запись в одну строчку.

Таблица 4.7. *Размещение чисел в ячейках памяти*

Тип	Байт1	Байт2	Байт3	Байт4	Байт5	...	Байт10
real*4	sppp pppp	pmmm mmmm	mmmm mmmm	mmmm mmmm		...	
real*8	sppp pppp	pppp mmmm	mmmm mmmm	...	mmmm mmmm		
real*10	sppp pppp	pppp pppp	1mmm mmmm	...	mmmm mmmm	mmmm mmmm	mmmm mmmm

Если знаковый бит равен 0, то число положительное, если равен 1 — то отрицательное.

В числах `real*4 (float)` и `real*8` (`double`) при базовом представлении ведущая единица мантиссы подразумевается, но не хранится, поэтому реально можно считать, что у них под мантиссу отведено не 23 и 52 бита (которые реально хранятся), а 24 и 53 бита.

В числах `real*10` ведущая единица мантиссы реально хранится, и мантисса занимает 64 бита.

Под порядок в числах `real*4` отведено 8 битов, в числах `real*8` отведено 11 битов, а в числах `real*10` отведено 15 битов.

В табл. 4.8 представлены параметры хранения чисел в соответствии со стандартом IEEE 754.

Таблица 4.8. *Стандарт IEEE 754 представления чисел в формате с плавающей точкой*

Тип IEEE 754	Тип Java	Число битов мантиссы	Число битов порядка	Сдвиг порядка
real*4	float	23+ подразумевается 1 ведущий бит	8	127
real*8	double	52+ подразумевается 1 ведущий бит	11	1023
real*10	-	64	15	16383

Чему равны минимальное и максимальное по модулю числа при их базовом представлении?

Минимальное значение достигается при минимальном порядке и всех нулевых битах мантиссы (за исключением ведущего), т. е. при $m = 1$ и $p = 1$. Значит, минимальное значение равно 2^{1-d}.

Максимальное значение достигается при максимальном порядке и всех единичных битах мантиссы, т. е. при $m \approx 2$ и $p = p_{\max} - 1$.

Значит, максимальное значение примерно равно:

$$2 \cdot 2^{P\max - 1 - d} = 2^{P\max - d}$$

При значениях порядка в диапазоне от 1 до $p_{\max} - 1$ базовое представление позволяет закодировать:

- числа `real*4` примерно от 2.350989*E*–38 до 3.402824*E*38;
- числа `real*8` примерно от 2.225074*E*–308 до 1.797693*E*308;
- числа `real*10` примерно от 3.362103*E*–4932 до 1.189731*E*4932.

Когда порядок равен нулю или p_{max}, используется *особое* представление чисел, несколько отличающееся от базового.

Если все биты порядка равны нулю, но мантисса отлична от нуля, то порядок считается равным 1 (а не 0), а вместо 1 в качестве подразумеваемой ведущей цифры служит 0. Это *ненормализованное* представление чисел. Максимальное значение мантиссы в этом случае чуть меньше единицы (на значение младшего бита мантиссы).

Так что максимальное значение числа в особой форме представления равно:

(`1-значение_младшего_бита_мантиссы`) $\cdot 2^{1-d}$,

т. е. верхний предел диапазона изменения чисел в этом представлении смыкается с нижним диапазоном изменения чисел в базовом представлении.

Минимальное ненулевое значение мантиссы в особом представлении равно 2^{-n}, где n — число битов мантиссы после двоичной точки.

Минимальное отличное от 0 положительное число для некоторого типа чисел с плавающей точкой равно 2^{1-d-n}.

Таким образом, *особое представление* позволяет закодировать:

- числа `real*4` примерно от 1.401298*E*–45 до 2.350989*E*–38;
- числа `real*8` примерно от 4.940656*E*–324 до 2.225074*E*–308;
- числа `real*10` примерно от 3.6451995*E*–4951 до 3.362103*E*–4932.

Специальный случай особого представления — когда и порядок, и мантисса равны нулю. Это значение обозначает машинный нуль. В соответствии со стандартом IEEE 754 имеются +0 и –0. Но во всех известных автору языках программирования +0 и –0 при вводе/выводе и сравнении чисел отождествляются.

Нечисловое представление соответствует случаю, когда $p = p_{max}$, т. е. все биты порядка равны единице. Такое число в зависимости от значения мантиссы соответствует одному из трех специальных нечисловых значений, которые обозначаются:

- `Inf` (Infinity — бесконечность);
- `NaN` (Not a Number — не число);
- `Ind` (Indeterminate — неопределенность).

Эти значения появляются при переполнениях и неопределенностях в вычислениях. Например, при делении 1 на 0 получается `Inf`, а при делении 0 на 0 получается `Ind`. Значение `NaN` может получаться при преобразовании строки в число, взятии логарифма от отрицательного числа, тригонометрической функции от бесконечности и т. п.

Значение `Inf` соответствует нулевым битам мантиссы. Согласно IEEE 754 бесконечность имеет знак. Если знаковый бит равен нулю, то это `+Inf`, если знаковый бит равен единице, то это `-Inf`.

Значение `Ind` кодируется единицей в знаковом бите и битами мантиссы, равными нулю, во всех разрядах, кроме старшего (реального, а не подразумеваемого), где стоит единица. Все остальные сочетания знакового бита и мантиссы отведены под величины `NaN`. Значения `NaN` бывают двух типов — вызывающие возбуждение сигнала о переполнении (`Signaling NaN`) и не вызывающие (`Quiet NaN`). Значения обоих этих типов могут быть положительными (знаковый бит равен 0) и отрицательными (знаковый бит равен 1).

В современных языках программирования поддерживается только часть возможностей, реализованных в процессорах в соответствии со стандартом IEEE 754. Например, в Java значения бесконечности различаются как по знаку, так и по типу. Имеются:

`Float.NEGATIVE_INFINITY`,

`Float.POSITIVE_INFINITY`,

`Double.NEGATIVE_INFINITY`,

`Double.POSITIVE_INFINITY`.

Но значение `Ind` вообще не употребляется и отождествляется с `NaN`, хотя `Float.NaN` и `Double.NaN` различаются.

Числа в формате с плавающей точкой занимают диапазоны значений битов, указанные в табл. 4.9.

Таблица 4.9. *Диапазоны значений чисел в формате с плавающей точкой*

Название значения	**s (знак)**	**p (порядок)**	**m (мантисса)**
–NaN (отрицательное не число)	1	11..11	11..11 : 10..01
Indeterminate (неопределенность)	1	11..11	10..00
Signaling –NaN (отрицательное не число, возбуждающее ошибку)	1	11..11	01..11 : 00..01
–Infinity (отрицательное переполнение плюс бесконечность)	1	11..11	00..00
Отрицательное нормализованное	1	11..10 : 00..01	11..11 : 00..00
Отрицательное ненормализованное	1	00..00	11..11 : 00..01
–0	1	00..00	00..00
+0	0	00..00	00..00
Положительное ненормализованное	0	00..00	00..01 : 11..11
Положительное нормализованное	0	00..01 : 11..10	00..00 : 11..11
+Infinity (положительное переполнение плюс бесконечность)	0	11..11	00..00

Таблица 4.9 *(окончание)*

Название значения	s (знак)	p (порядок)	m (мантисса)
Signaling +NaN (положительное не число, возбуждающее ошибку)	0	11..11	00..01 : 01..11
Quiet + NaN (положительное не число)	0	11..11	10..00 : 11..11

Имеются методы оболочечных классов, позволяющие преобразовывать наборы битов, хранящихся в ячейках типа `int`, в значения `float`, и наоборот — значения типа `float` в их битовое представление типа `int`. При этом содержание ячеек не меняется — просто содержащиеся в ячейках наборы битов начинают по-другому трактоваться.

Аналогичные операции существуют и для значений типа `long` и `double`:

`Float.intBitsToFloat(`*значение типа int*`)`
`Double.longBitsToDouble(`*значение типа long*`)`
`Float.floatToIntBits(`*значение типа float*`)`
`Double.doubleToLongBits(`*значение типа double*`)`

Например:

`Float.intBitsToFloat(0x7F7FFFFF)` даст значение, равное `Float.MAX_VALUE`,

`Float.intBitsToFloat(0x7F800000)` — значение `Float.POSITIVE_INFINITY`,

`Float.intBitsToFloat(0xFF800000)` — значение `Float.NEGATIVE_INFINITY`.

Если аргумент метода `Float.intBitsToFloat` находится в пределах от `0x7F800001` до `0x7FFFFFFF` либо от `0xFF800001` до `0xFFFFFFFF`, то результатом будет `Float.NaN`.

Следует подчеркнуть, что данные операции принципиально отличаются от обычных преобразований типов (например, из `int` в `float` или из `double` в `long`). При обычных преобразованиях значение числа не меняется, просто меняется форма хранения этого значения и, соответственно, наборы битов, которыми кодируется это значение. Причем может измениться размер ячейки (скажем, при преобразовании значений `int` в значения `double`). А при рассматриваемых в данном разделе операциях сохраняется набор битов и размер ячейки, но меняется тип, который приписывается этому набору. Такая операция называется приведением типа.

Краткие итоги по главе

- Отрицательные целые числа кодируются в дополнительном коде.
- При сложении, вычитании и особенно умножении целых чисел может возникать выход за пределы допустимого диапазона, из-за чего у результата может получиться значение, имеющее противоположный (по сравнению с правильным результатом) знак, или нуль при сложении чисел одного знака.

- Побитовая маска AND (оператор `&`) служит для сбрасывания в нуль тех битов числа, где в маске стоит 0, остальные биты числа не меняются.
- Побитовая маска OR (оператор `|`) служит для установки в единицу тех битов числа, где в маске стоит 1, остальные биты не меняются.
- Побитовая маска XOR (оператор `^`) служит для инверсии тех битов числа, где в маске стоит 1 (единицы переходят в нули, а нули — в единицы), остальные биты числа не меняются. Имеется команда перевода вывода графики в режим XOR при рисовании, для этого используется команда `graphics.setXORMode` *(цвет)*.
- Инверсия всех битов числа осуществляется с помощью побитового отрицания `~a`.
- Побитовые сдвиги `<<`, `>>` и `>>>` приводят к перемещению всех битов ячейки, к которой применяется оператор, на указанное число битов влево или вправо. Причем `m<<n` является очень быстрым вариантом операции $m*2^n$, а `m>>n` соответствует целочисленному делению `m` на 2^n как для положительных, так и для четных отрицательных чисел.
- Литерные константы для вещественных типов по умолчанию имеют тип `double` (например: `1.5`, `−17E2`, `0.0`). Если требуется ввести литерную константу типа `float`, то после записи числа добавляют постфикс `f` (сокращение от float): `1.5f`, `−17E2f`, `0.0f`.
- Число в формате с плавающей точкой состоит из знакового бита, мантиссы и порядка. Все операции с числами такого формата сопровождаются накоплением ошибки порядка нескольких младших битов мантиссы ненормализованного результата.
- При проведении умножения и деления чисел в формате с плавающей точкой не происходит катастрофической потери точности результата. А при сложениях и вычитаниях такая потеря может происходить в случае, когда вычисляется разность двух значений, к которым прибавляется малая относительно этих значений добавка. Число порядков при потере точности равно числу порядков, на которые отличаются эти значения от малой добавки.

Типичные ошибки

- При работе с целыми типами не знают или забывают, что в случае выхода за пределы диапазона соответствующие биты отбрасываются без всякой диагностики переполнения, и что получившийся результат может быть неправильного знака или равен нулю. Особенно часто переполнение возникает при умножении нескольких целых величин друг на друга.
- Написание численных алгоритмов с катастрофической потерей точности.
- Сравнение двух чисел в формате с плавающей точкой на *равенство* или *неравенство* в случае, когда в точной арифметике они должны быть равны.

Задания

- Создать проект с графическим пользовательским интерфейсом, в котором имеются два пункта ввода и радиогруппа с выбором варианта для каждого из целочисленных типов, а также кнопки `JButton` с надписями **Сложить** и **Умножить**. Кроме того,

должна быть кнопка, при нажатии на которую в пунктах ввода должны возникать случайным образом генерируемые числа, лежащие в пределах допустимых значений для выбранного типа. При нажатии на кнопку **Сложить** или **Умножить** содержимое пунктов ввода и результат действий должны быть преобразованы в значения соответствующего типа. Эти значения нужно показать в отдельных метках.

Проверить наличие проблемы арифметического переполнения для разных значений каждого типа. Выяснить, в каком случае чаще происходит переполнение — при сложении или умножении. Определить насколько часто встречаются такие значения одного знака, чтобы получился результат противоположного знака. Вручную ввести в пункты ввода такие значения одного знака, чтобы в итоге получилось нулевое значение.

- Написать приложение с графическим пользовательским интерфейсом, в котором вычисляется выражение `1 + x` и присваивается переменной `float f`, а также вычисляется выражение `1 + y` и присваивается переменной `double d`.

  Величины `x` типа `float` и `y` типа `double` вводятся пользователем с помощью пунктов ввода.

  Вывести в метку `jLabel1` разность `f − 1`, а в метку `jLabel2` разность `d − 1`.

  Вывести в метку `jLabel3` отношение `(f − 1) / x + 2`, а в метку `jLabel4` отношение `(d − 1) / y + 2`.

  Провести вычисления для `x` и `y`, меняющихся в пределах от `1E-3` до `1E-18`. Объяснить результаты.

- * *Для студентов технических вузов и физико-математических факультетов университетов*. Написать приложение с графическим пользовательским интерфейсом, в котором вычисляются выражения $f1(x)$ и $f2(x)$,

  где $f1(x) = a(x) - b(x)$, $a(x) = \frac{e^x}{x^2}$, $b(x) = \frac{1+x}{x^2}$, $f2(x) = \frac{1}{2} + \frac{1}{6}x + \frac{1}{24}x^2$.

  Поскольку $f2(x)$ состоит из первых членов разложения $f1(x)$ в ряд, то $f1(x)$ и $f2(x)$ должны быть примерно равны.

  Требуется сравнить значения выражения $f1(x)$ и $f2(x)$ при различных x.

  Все вычисления сначала проводить для переменных и функций типа `float`, а затем для переменных и функций типа `double`.

  Величину x вводит пользователь.

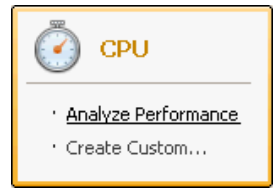
  Вывести в метки значения:

  `f1_double(x)`, `a_double(x)`, `b_double(x)`, `f1_float(x)`, `a_float(x)`, `b_float(x)`,

  а также разности:

  `f1_float(x) − f2_float(x)` и `f1_double(x) − f2_double(x)`.

Провести такое сравнение для аргументов `x`, меняющихся в пределах от `1E-8` до `0.1`, и объяснить результаты.



ГЛАВА 5

Управляющие конструкции

5.1. Составной оператор

Согласно синтаксису языка Java во многих конструкциях возможен только один оператор, но часто встречается ситуация, когда необходима последовательность из нескольких операторов.

Тогда на помощь приходит составной оператор — блок кода между фигурными скобками `{}`.

Имеются два общепринятых способа форматирования текста с использованием фигурных скобок.

В первом из них скобки пишут друг под другом, а текст, находящийся между ними, сдвигают на 1—2 символа вправо (изредка — больше):

```
оператор
{
  последовательность простых или составных операторов
}
```

Во втором способе (более распространенном) открывающую фигурную скобку пишут на той же строке, где должен начинаться составной оператор (без переноса его на следующую строку), а закрывающую скобку располагают на уровне первого слова:

```
оператор{
   последовательность простых или составных операторов
}
```

Второй способ установлен по умолчанию в среде NetBeans, и именно он используется в приводимых в данной книге фрагментах программного кода. Чтобы изменить установленный по умолчанию способ форматирования, следует перейти в меню **Tools/Options**, выбрать пункт **Editor/Indentation**, после чего в появившейся диалоговой форме отметить флажком пункт **Add New Line Before Brace**.

После фигурных скобок по правилам Java (как и в C/C++) точку с запятой (`;`) не ставят.

5.2. Условный оператор *if*

У условного оператора `if` имеются две формы: `if` и `if... else`.

По-английски if означает "если", else — "в противном случае". Таким образом, эти операторы можно перевести как: "если...то..." и "если...то... в противном случае...".

Первая форма:

```
if(условие)
  оператор1;
```

В этом случае если условие равно `true`, то выполняется `оператор1`. Если же условие равно `false`, то в операторе не выполняется никаких действий.

Вторая форма:

```
if(условие)
  оператор1;
else
  оператор2;
```

В этом варианте оператора `if` если условие равно `false`, то выполняется `оператор2`.

Обратите особое внимание на форматирование текста. Не располагайте все части оператора `if` на одной строке — это характерно для новичков!

Пример правильного форматирования

```
if(a<b)
  a=a+1;
else if(a==b)
       a=a+1;
     else{
       a=a+1;
       b=b+1;
     }
```

Из этого правила имеется исключение: если подряд идет большое количество операторов `if`, умещающихся в одну строку, то для повышения читаемости программ бывает целесообразно не переносить другие части операторов на отдельные строки.

Отметим, что в операторе `if` в области выполнения, которая следует после условия, а также в области `else` должен стоять только один оператор, а не последовательность операторов. Поэтому запись оператора в виде:

```
if(условие)
  оператор1;
  оператор2;
else
  оператор3;
```

недопустима.

В таких случаях применяют составной оператор, ограниченный фигурными скобками. Между ними, как мы знаем, может стоять произвольное число операторов:

```
if(условие){
  оператор1;
  оператор2;
}else
  оператор3;
```

Если же мы напишем:

```
if(условие)
  оператор1;
else
  оператор2;
  оператор3;
```

никакой диагностики ошибки компилятор не выдаст!

`Оператор3` в данном случае никакого отношения к условию `else` иметь не будет — подобное форматирование текста будет подталкивать к логической ошибке.

При следующем форматировании текста программы, эквивалентном предыдущему при компиляции, уже более очевидно, что `оператор3` не относится к части `else`:

```
if(условие)
  оператор1;
else
  оператор2;
оператор3;
```

Для того чтобы `оператор3` относился к части `else`, следует использовать составной оператор:

```
if(условие)
  оператор1;
else{
  оператор2;
  оператор3;
}
```

В последовательности операторов

```
if(условие1) if(условие2) оператор1 else оператор2;
```

имеющийся `else` относится к последнему `if`, поэтому лучше отформатировать текст так:

```
 if(условие1)
   if(условие2)
     оператор1;
   else
     оператор2;
```

Таким образом, если писать соответствующие `if` и `else` друг под другом, логика работы программы становится очевидной.

Пример неправильного стиля оформления

```
if(условие1)
   if(условие2)
     оператор1;
else
   оператор2;
```

Этот стиль подталкивает к логической ошибке при чтении программы. Человек, читающий такую программу (как и тот, кто ее писал), будет полагать, что `оператор2` выполняется, если `условие1==false`, т. к. кажется, что он относится к первому `if`, а не ко второму. Нужно отметить, что сама возможность такой ошибки связана с непродуманностью синтаксиса языка Java.

Для правильной работы предыдущий пример следует оформить иначе.

Пример правильного оформления

```
if(условие1){
  if(условие2)
     оператор1;
}else
  оператор2;
```

Чтобы избегать такого рода проблем, используйте команду **Format** (Форматировать исходный код) из пункта **Source** главного или контекстного меню либо комбинацию клавиш <Alt>+<Shift>+<F>. Среда разработки не только переформатирует исходный код в соответствии с принятым в мире стандартом, но и автоматически добавит фигурные скобки везде, где они необходимы, исходя из логики безопасного написания программы, исключающего описанные логические ошибки.

Типична ситуация с забытым объединением последовательности операторов в составной с помощью фигурных скобок.

Например, пишут:

```
if(условие)
   оператор1;
   оператор2;
```

вместо:

```
if(условие){
   оператор1;
   оператор2;
}
```

Причем такую ошибку время от времени допускают даже опытные программисты в случае, когда они работают в не очень развитых средах разработки. Аналог команды **Format** в развитых средах разработки также не помогает, а только усугубляет ситуацию, поскольку фигурные скобки в этом случае расставляются в соответствии с синтаксисом языка Java, а не в соответствии с форматированием исходного кода. Причем

обычно такая команда вызывается после того, как программист далеко ушел от проблемного места, из-за чего неправильную расстановку фигурных скобок после выполнения автоматического форматирования он не замечает.

Для того чтобы гарантированно избежать такой ошибки, следует *всегда* добавлять фигурные скобки в тех местах операторов Java, где по синтаксису может стоять только один оператор. Конечно, было бы лучше, чтобы разработчики Java предусмотрели такую конструкцию, как часть синтаксиса языка. Среда NetBeans во многих случаях автоматически создает такие скобки, но не для оператора `if`. Тем не менее, в версии 6.5 было сделано важное усовершенствование — после ввода оператора `оператор1;` и нажатия клавиши <Enter> среда автоматически форматирует исходный код так, что вводимый `оператор2;` оказывался правильным образом сдвинут:

```
if(условие)
    оператор1;
оператор2;
```

В результате логическая ошибка при чтении программистом данного фрагмента исключается, и если была забыта фигурная скобка, программист сразу это видит. Заметим, что при вызове команды **Format** (или <Alt>+<Shift>+<F>) фигурные скобки автоматически генерируются во всех необходимых местах.

Следует отметить разные правила записи точки с запятой при наличии части `else` с фигурными скобками и без них.

Пример 1

```
  if(i==0)
       i++;
  else
       i--;
```

Пример 2

```
if(i==0){
       i++;
} else{
     i--;
}
```

Обратите внимание, что во втором примере после скобок (перед `else`) точка с запятой не ставится.

Неприятная проблема, унаследованная в Java от языка C, — возможность ошибочной записи оператора присваивания `=` вместо оператора сравнения `==`.

Например, если мы используем булевы переменные `b1` и `b2`, и вместо

```
if(b1==b2)
    i=1;
else
    i=2;
```

напишем:

```
if(b1=b2)
    i=1;
else
    i=2;
```

то никакой диагностики ошибки (или даже предупреждения в редакторе исходного кода) выдано не будет — см. пример 05_2_if_switch на прилагаемом к книге DVD.

Дело в том, что по правилам C и Java любое присваивание рассматривается как функция, возвращающая в качестве результата присваиваемое значение. Поэтому присваивание `b1=b2` возвратит значение, хранящееся в переменной `b2`. В результате оператор будет работать, но совсем не так, как ожидалось. Более того, будет испорчено значение, хранящееся в переменной `b1`.

Проблемы с ошибочным написанием `=` вместо `==` в Java гораздо менее серьезны, чем в C/C++, поскольку они возникают только при сравнении булевой переменной с булевым значением. Если же оператор `=` ставится вместо `==` при сравнении числовой переменной, то происходит диагностика ошибки на этапе компиляции, т. к. такое присваивание должно возвращать число, а не булево значение. Жесткая типизация Java обеспечивает повышение надежности программ.

5.3. Оператор выбора *switch*

Этот оператор аналогичен `if` для нескольких условий выбора. Синтаксис оператора иллюстрирует листинг 5.1.

Листинг 5.1. Синтаксис оператора `switch`

```
switch(выражение){
 case значение1: операторы1;
 ...
 case значениеN: операторыN;
 default: операторы;
}
```

Правда, крайне неудобно, что нельзя ни указывать диапазон значений, ни перечислять через запятую значения, которым соответствуют одинаковые операторы.

Тип выражения должен быть одним из целых типов (`byte`, `short`, `char`, `int`) либо одним из соответствующих им оболочечных типов (см. *разд. 3.6*), либо типом-перечислением. В частности, недопустимы тип `long` и вещественные типы.

Работает оператор следующим образом: сначала вычисляется выражение, затем вычисленное значение сравнивается со значениями вариантов, которые должны быть определены еще на этапе компиляции программы. Если найден вариант, которому удовлетворяет значение выражения, то выполняется соответствующая этому варианту последовательность операторов, после чего *не происходит* выхода из оператора `case` (что было бы естественно), а выполняются по очереди все операторы, соответствующие следующим пунктам `case` (без всякой проверки условия соответствующего `case`) и пункту `default` —

см. пример 05_2_if_switch на прилагаемом к книге DVD. Для выхода нужно поставить оператор `break`. Эта неприятная особенность Java унаследована от языка C.

Часть с `default` необязательная и выполняется в том случае, если ни один вариант не найден. Листинг 5.2 иллюстрирует пример использования оператора `switch`.

Листинг 5.2. Пример использования оператора `switch`

```
switch(i/j){
    case 1:
        i=0;
        break;
    case 2:
        i=2;
        break;
    case 10:
        i=3;
        j=j/10;
        break;
    default:
        i=4;
}
```

Подчеркнем две особенности, которые имеются у оператора `switch`:

- можно писать произвольное число операторов для каждого варианта `case`, что весьма удобно, но полностью выпадает из логики операторов языка Java;
- выход из выполнения последовательности операторов осуществляется с помощью оператора `break`. Если он отсутствует, то происходит "провал" в блок операторов, соответствующих варианту, следующему за тем, с которым совпало значение выражения. При этом соответствие очередному значению не проверяется. И так продолжается до тех пор, пока не встретится оператор `break` или не кончатся все операторы в вариантах выбора (в том числе вариант default). Такие правила проверки порождают типичную ошибку, называемую "забытый break".

5.4. Условное выражение ...?... : ...

Эта не очень удачная по синтаксису конструкция унаследована из языка C.

Ее синтаксис таков:

```
условие?значение1:значение2
```

В выражениях для условий и значений можно использовать пробелы, в том числе отделять ими символы `?` и `:`, но эти пробелы игнорируются.

Когда условие имеет значение `true`, функция возвращает `значение1`, в противном случае — `значение2`.

Например, мы хотим присвоить переменной `j` значение, равное `i+1` при `i<5` и `i+2` в других случаях. Это можно сделать так:

```
j=i<5?i+1:i+2
```

К данному оператору приходится прибегать при вычислении громоздких выражений, поскольку без него программа оказывается еще менее прозрачной. Приоритет разделителей `?` и `:` очень низкий — ниже только приоритет оператора присваивания (в любых его формах). Поэтому в принципе можно обойтись без скобок, но лучше их все-таки добавить:

```
j=(i<5)?(i+1):(i+2).
```

5.5. Операторы инкремента ++ и декремента --

Оператор `++` называется инкрементным (увеличивающим), а оператор `--` — декрементным (уменьшающим). У этих операторов имеются две формы: постфиксная (наиболее распространенная, когда оператор ставится после операнда) и префиксная (встречается очень редко, в ней оператор ставится перед операндом).

Для любой числовой величины `x` выражение `x++` или `++x` означает увеличение `x` на 1, а выражение `x--` или `--x` означает уменьшение `x` на 1.

Различие двух форм связано со временем изменения величины: для постфиксной формы — после, а для префиксной до вычисления выражения, в котором используется оператор.

Например, присваивания `j=i++` и `j=++i` дадут разные результаты (см. пример 05_5_consol_example на прилагаемом DVD). Если первоначально `i=0`, то первое присваивание даст 0, т. к. `i` увеличится на 1 после выполнения присваивания. А второе присваивание даст 1, т. к. сначала выполнится инкремент, и только потом будет вычисляться выражение и выполняться присваивание. При этом в обоих случаях после выполнения присваивания `i` станет равно 1.

5.6. Оператор цикла *for*

Синтаксис:

```
for(блок инициализации; условие выполнения тела цикла;
    блок изменения счетчиков)
оператор;
```

В блоке инициализации через запятую перечисляют операторы задания локальных переменных, область существования которых ограничивается оператором `for`. Также могут быть присвоены значения переменным, заданным вне цикла. Но инициализация может происходить только для переменных одного типа.

В блоке условия продолжения цикла проверяется выполнение условия, и если оно выполняется, то идет выполнение тела цикла, в качестве которого выступает оператор. В противном случае цикл прекращается и осуществляется переход к оператору программы, следующему за оператором `for`.

После каждого выполнения тела цикла (очередного шага цикла) выполняются операторы блока изменения счетчиков. Они должны разделяться запятыми.

Пример:

```
for(int i=1,j=5; i+j<100; i++,j=i+2*j){
 ...
}
```

Каждый из блоков оператора `for` является необязательным, но при этом разделительный символ `;` (точку с запятой) все равно требуется писать.

Чаще всего оператор `for` служит для перебора значений некоторых переменных, увеличивающихся или уменьшающихся на 1, и выполнения последовательности операторов, использующих эти значения. Переменная в этом случае называется *счетчиком цикла*, а последовательность операторов — *телом цикла*.

Напишем цикл, в котором суммируются все числа от 1 до 100. Результат будем хранить в переменной `result` (листинг 5.3).

Листинг 5.3. Вычисление суммы последовательно идущих чисел

```
int result=0;
for(int i=1; i<=100; i++){
   result=result+i;
}
```

Цикл (повторное выполнение одних и тех же действий) выполняется следующим образом:

1. До начала цикла создается переменная `result`, в которой мы будем хранить результат. Одновременно выполняется инициализация — присваивается начальное значение 0.
2. Начинается цикл. Сначала выполняется блок инициализации — счетчику цикла `i` присваивается значение 1. Блок инициализации выполняется только один раз в самом начале цикла.
3. Начинается первый шаг цикла. Проверяется условие выполнения цикла. Значение `i` сравнивается с 100.
4. Поскольку сравнение 1<=100, то возвращает `true`, выполняется тело цикла. В переменной `result` хранится 0, а значение `i` равно 1, поэтому присваивание `result=result+i` эквивалентно `result=1`. Таким образом, после первого шага цикла в переменной `result` будет храниться значение 1.
5. После выполнения тела цикла изменяется счетчик цикла (т. е. оператор `i++`, увеличивающий `i` на 1). Значение `i` становится равным 2.
6. Начинается второй шаг цикла. Проверяется условие выполнения тела цикла. Поскольку сравнение 2<=100 возвращает `true`, то идет очередное выполнение тела цикла, а затем — увеличение счетчика цикла.
7. Шаги цикла продолжаются до тех пор, пока счетчик цикла не станет равным 101. В этом случае условие выполнения тела цикла 101<=100 возвращает `false`, и происходит выход из цикла. Последнее присваивание `result=result+i`, проведенное в цикле, — это `result=result+100`.

Если бы нам требовалось просуммировать числа от 55 до 1234, то в блоке инициализации `i` нужно было присвоить значение 55, а в условии проверки поставить 1234 вместо 100. В листинге 5.4 приведен пример вычисления факториала.

Листинг 5.4. Вычисление факториала

```
double x=1;
for(i=1;i<=n;i++){
   x=x*i;
}
```

Заметим, что в приведенных примерах можно сделать некоторые усовершенствования — операторы присваивания записать следующим образом:

для первого примера: `result+=i;` вместо `result=result+i;`

и

для второго примера: `x*=i;` вместо `x=x*i;`

На начальных стадиях обучения так лучше не делать, поскольку текст программы должен быть ясен (все алгоритмы должны быть прозрачны для понимания).

В Java версии JDK 5 появилась еще одна форма оператора `for`, которую принято называть for-each (для каждого). Она позволяет перебирать все элементы массива или коллекции (неиндексированного набора данных, во многом похожего на массив). Описание работы с этим оператором дано в *разд. 8.3*.

5.7. Ошибки при использовании вещественного счетчика цикла

Наиболее распространенная ошибка при работе с циклами (в том числе с циклом `for`) — это использование вещественного счетчика цикла. Разберем эту ошибку на примере вычисления площади под кривой.

Пусть требуется вычислить площадь s под кривой, задаваемой функцией $f(x)$, на промежутке от a до b (провести интегрирование). Разобьем этот промежуток на n интервалов, при этом длина каждого интервала будет $h = (b - a) / n$.

Мы предполагаем, что величины a, b, n и функция $f(x)$ заданы. Площадь под кривой на интервале с номером j будем считать равной значению функции на левом конце интервала. Такой способ численного нахождения интеграла называется методом левых прямоугольников.

На первый взгляд можно записать алгоритм вычисления этой площади так, как в листинге 5.5.

Листинг 5.5. Цикл с неустойчивостью из-за вещественного счетчика

```
double s=0;
double h=(b-a)/n;
```

```
for(double x=a; x<b; x=x+h){
  s=s+f(x)*h;
}
```

И действительно, иногда такой алгоритм правильно работает. Но изменение числа интервалов *n* или границ *a* или *b* может привести к тому, что будет учтен лишний интервал, находящийся справа от точки *b*. Это связано с тем, что в циклах с вещественным счетчиком всегда проявляется нестабильность, если последняя точка попадает на границу интервала. Что будет происходить на последних шагах нашего цикла?

На предпоследнем шаге мы попадаем в точку $x = b - h$, при этом условие $x < b$ всегда выполняется, и никаких проблем не возникает.

Следующей точкой должна быть $x = b$. Проверяется условие $x < b$, и в идеальном случае должен происходить выход из цикла, поскольку $x == b$ (т. е. условие не выполнено). Но ведь все операции в компьютере для чисел в формате с плавающей точкой проводятся с конечной точностью. Поэтому практически всегда значение *x* для этого шага окажется либо чуть меньше *b*, либо чуть больше. Отличие будет только в последних битах мантиссы, но этого окажется достаточно, чтобы условие $x < b$ иногда давало `true`, хотя для заданных *a*, *b* и *n* результат будет прекрасным образом воспроизводиться, в том числе и на других компьютерах.

Более того, при увеличении числа разбиений *n* погрешность вычисления площади даже для "неправильного" варианта будет убывать, хотя и гораздо медленнее, чем для "правильного". Это один из самых неприятных типов ошибок, когда алгоритм вроде бы работает правильно, но для получения нужной точности требуется гораздо больше времени или ресурсов.

Рассмотрим теперь правильную реализацию алгоритма. Для этого воспользуемся целочисленным счетчиком цикла. Нам потребуется чуть больше рассуждений, алгоритм окажется немного менее прозрачным, но зато гарантируется его устойчивость.

Значение функции в начале интервала с номером *j* будет равно $f(a + j * h)$. Первый интервал будет иметь номер 0, второй — номер 1 и т. д. Последний интервал будет иметь номер $n–1$.

Листинг 5.6 содержит правильно работающий код.

Листинг 5.6. Цикл без неустойчивости

```
double s=0;
double h=(b-a)/n;
for(int j=0; j<=n-1; j++){
  s=s+f(a+j*h)*h;
}
```

Проверить неустойчивость первого алгоритма и устойчивость второго можно на примере 05_7_integral, приведенном на прилагаемом к книге DVD. В нем вычисляется интеграл от 0 до 1 от функции, представляющей собой неизменную величину, равную единице. Правильное значение такого интеграла равно 1. Для указанной функции ис-

пользуемый в алгоритмах метод левых прямоугольников должен давать верный результат для любого числа интервалов, начиная с единицы. Но при заданном в примере числе интервалов 10 неустойчивый алгоритм дает неверный результат. Читателю предлагается поэкспериментировать с тем, как будут меняться результаты вычисления при изменении числа шагов. Заметим, что отличие правильного результата от 1 в последнем разряде мантиссы связано с ограниченной точностью вычислений в формате с плавающей точкой.

Приведем более сложный пример неустойчивого поведения. Пусть

$f(x) = x^2$, где $a = 0$, $b = 1$ (табл. 5.1).

В таблице подчеркнуты первые значащие цифры неправильных значений, получающихся в результате неустойчивости алгоритма при $n = 10$, $n = 13$, $n = 102$, $n = 103$, $n = 1001$.

Таблица 5.1. *Проверка неустойчивости первого алгоритма и устойчивости второго*

n	s для первого алгоритма	s для второго алгоритма
8	0.2734375	0.2734375
9	0.2798353909465021	0.279835390946502
10	0.3849999999999999	0.2850000000000001
11	0.28925619834710753	0.28925619834710750
12	0.29282407407407407	0.292824074074074
13	0.3727810650887573	0.2958579881656805
...	...	...
100	0.32835000000000036	0.32835000000000014
101	0.32839917655131895	0.3283991765513185
102	0.33825131359733385	0.3284473920287069
103	0.3382034121971908	0.3284946743331133
...	...	...
1000	0.33283350000000095	0.33283350000000034
1001	0.3338330001666631	0.33283399916766554
1002	0.3328344973393309	0.33283449733932

Если оба алгоритма устойчивы, то при всех n они должны давать одинаковые результаты (с точностью до нескольких младших битов мантиссы). Очень характерная особенность такого рода неустойчивости — скачкообразное изменение результата при плавном изменении какого-либо параметра. В табл. 5.1 меняется число n, но такая же ситуация будет наблюдаться и при плавном изменении чисел a или b.

Например, при $n = 10$ и $a = 0$ в случае очень малых изменений b получим результаты, приведенные в табл. 5.2.

Таблица 5.2. *Проверка неустойчивости первого алгоритма и устойчивости второго при плавном изменении числа b*

b	s для первого алгоритма	s для второго алгоритма
1.00000	0.3849999999999999	0.2850000000000001
1.00001	0.28500855008550036	0.28500855008550036
1.00002	0.2850171003420023	0.2850171003420022
1.00003	0.38503465103951023	0.28502565076950764
1.00004	0.2850342013680182	0.2850342013680184
1.00005	0.2850427521375357	0.2850427521375357

Вещественный счетчик цикла не всегда приводит к проблемам. Рассмотрим вариант численного нахождения интеграла методом средних прямоугольников. Для этого площадь под кривой на интервале с номером *j* будем считать равной значению функции в середине интервала. Алгоритм идентичен описанному ранее для метода левых прямоугольников, за исключением выбора начальной точки ($a + h / 2$ вместо a).

Листинг 5.7. Цикл с вещественным счетчиком, но без неустойчивости

```
double s=0;
h=(b-a)/n;
for(double x=a+h/2; x<b; x=x+h){
  s=s+f(x)*h;
}
```

Для данного алгоритма (листинг 5.7) проблем не возникает благодаря тому, что все точки x, в которых производится сравнение $x < b$, отстоят достаточно далеко от значения b (по крайней мере на $h / 2$).

Замечания

Метод средних прямоугольников гораздо точнее метода левых прямоугольников, и поэтому предпочтительнее в реальных вычислениях. Еще точнее метод Симпсона, однако он несколько сложнее.

Тестировать работу алгоритмов следует на самых простых примерах, для которых результаты хорошо известны и могут быть легко проверены. В нашем случае начинать проверку правильности интегрирования следует для функции-константы при небольшом числе интервалов разбиения и простейших границах интегрирования. И только потом необходимо проверить алгоритм при других границах интегрирования, числе интервалов интегрирования и для более сложных функций. Подробнее об этом говорится в *разд. 17.4*.

5.8. Эффективная организация циклов при вычислениях в формате с плавающей точкой

Циклы наиболее критичны к скорости выполнения. Для их эффективной организации следует делать как можно меньше вычислений внутри циклов. Каждое такое вычисление повторяется столько раз, сколько циклов будет выполнено, но если это вычисление можно вынести из цикла, то оно выполнится всего один раз.

Предложенные ранее реализации циклов не самые эффективные по скорости, поскольку в них много раз повторяется достаточно медленная операция — вещественное умножение (умножение в формате с плавающей точкой). Лучше написать алгоритм метода средних прямоугольников так, как показано в листинге 5.8.

Листинг 5.8. Вычисление интеграла методом средних прямоугольников

```
double s=0;
double h=(b-a)/n;
for(double x=a+h/2; x<b; x=x+h){
   s=s+f(x);
}
s=s*h;
```

При этом суммируются значения $f(x)$ (а не $f(x) * h$), а само умножение выполняется только один раз (уже после выхода из цикла для всей получившейся суммы).

Если время операции вещественного умножения $t_{умнож}$, то экономия времени работы программы по сравнению с первоначальным вариантом будет:

$(n - 1) \cdot t_{умнож}$.

Для больших значений n экономия времени может быть существенной даже для мощных компьютеров.

При написании программ с циклами (особенно при большом количестве шагов) следует выносить из тела цикла все операции, которые могут быть проделаны однократно вне цикла. Однако при таких усовершенствованиях часто теряется прозрачность алгоритмов. Поэтому полезно сначала написать реализацию алгоритма "один к одному" по формулам (без всяких усовершенствований) и убедиться в том, что все работает правильно, а уже потом вносить исправления, повышающие скорость работы программы в наиболее критичных местах. Не следует пытаться сразу написать программу, которая максимально эффективна по всем параметрам — это обычно приводит к гораздо более длительному поиску неочевидных ошибок.

Прозрачность алгоритмов и их максимальная близость к первоначальным "бумажным" вариантам формул при работе с Java особенно важна, поскольку правила с вынесением действий из циклов и другие варианты оптимизации вручную в Java не всегда работают — часто на основе неоптимального исходного кода компилятор и виртуальная Java-машина самостоятельно создают достаточно оптимизированный скомпилированный код.

Скорость работы программы можно проверить с помощью примера 05_8_effectiveCalc (он иллюстрирует перечисленные черты программирования вычислений в формате с плавающей точкой в Java). В этом примере рассматриваются следующие варианты вычисления интеграла (в первых трех функция f передается через список параметров с помощью объекта `fnc`):

- оптимизированный (`optimisedIntegral`) — из цикла вынесено все, что можно;
- обычный (`integral`) — из цикла вынесено только вычисление шага h;
- медленный (`slowIntegral`) — значение $(b - a) / n$ используется непосредственно во всех местах, где необходима величина h;

- встроенный (`inlineIntegralX`) — вычисляется интеграл от функции *x*, встроенной в алгоритм вычисления интеграла. Алгоритм служит для проверки того, насколько вызов внешней функции влияет на скорость вычислений — нельзя ли сэкономить время на таких вызовах, поместив функцию непосредственно в алгоритм вычисления.

Для интеграла со встроенной функцией в табл. 5.3 приведен пример только для обычного варианта функции, но проверялись все три варианта (оптимизированный, обычный и медленный), как и для внешних функций, заданных вне алгоритма интегрирования в классе `MiddleRectangles` (листинг 5.9).

Листинг 5.9. Разные варианты подпрограмм вычисления интеграла

```
public static double optimisedIntegral(double a,double b,double n,Fnc fnc) {
        double r=0, h=(b-a)/n, a1=a+h/2;
        for(int i=0;i<n;i++)
            r+=fnc.f(a1+i*h);
        return r*h;
}
public static double integral(double a,double b,double n,Fnc fnc) {
        double r=0, h=(b-a)/n;
        for(int i=0;i<n;i++)
            r=r+fnc.f(a+i*h+h/2)*h;
        return r;
}
public static double slowIntegral(double a,double b,double n,Fnc fnc) {
        double r=0;
        for(int i=0;i<n;i++)
            r=r+fnc.f( a+i*(b-a)/n+(b-a)/n/2 )*(b-a)/n;
        return r;
}
public static double inlineIntegralX(double a,double b, double n) {
        double r=0, h=(b-a)/n;
        for(int i=0;i<n;i++)
            r=r+(a+i*h+h/2)*h;
        return r;
}
```

Время вычислений определялось следующим образом:

```
long t0=System.currentTimeMillis();
result=MiddleRectangles.slowIntegral(a,b,n,fnc);
long t=System.currentTimeMillis()-t0;
```

Функция `System.currentTimeMillis()` возвращает текущее время в миллисекундах (отсчет времени идет от 1 января 1970 года).

В переменной `t0` сохраняется значение времени перед началом вычислений. Сразу после проведения вычислений эта функция вызывается еще раз, и время вычислений рассчитывается так:

```
System.currentTimeMillis() – t0.
```

В табл. 5.3 приведено время вычисления интеграла от различных функций *f*(*x*) (на компьютере с процессором Intel Core 2 Duo с тактовой частотой 1,73 ГГц в однопроцессорном режиме под операционной системой Windows XP SP2) для 100 миллионов интервалов (циклов суммирования).

Таблица 5.3. *Время вычислений в формате с плавающей точкой double — интегрирование функций*

Функция f(x)	Время вычисления интеграла, секунд				
	Оптимизированный вариант	Обычный вариант	Медленный вариант	Delphi	C++
`1`	1,5	1,5	6,0	0,8	0,5
`x`	1,5	1,5	6.0	0,8	0,5
`x (встроенная)`	1,4	1,4	6.0	0,8	0,5
`x*x`	1,5	1,5	6,0	1,1	0,6
`1/x`	2,3	2,3	7,7	2,3	2,1
`sqrt(x)`	3,8	3,8	9,3	4,2	4,0
`sin(x)`	13	13	19	7,9	6,2
`cos(x)`	13	13	19	7,9	5,7
`tan(x)`	26	26	32	8,9	12,4
`pow(x,0.5)`	109	109	115	19	60
`pow(x,2)`	6,1	6,1	12,9	10	5,8
`pow(x,2.0)`	6,1	6,1	12,9	10	5,8
`pow(x,2.5)`	60	60	64	19	36
`x*x*sqrt(x)`	4,7	3,9	9,3	4,0	3,9

На основе приведенных результатов можно сделать следующие выводы: при вычислениях в формате с плавающей точкой в языке Java попытки вручную обеспечить наивысшую степень оптимизации реально не влияют на скорость выполнения программы. Разброс по времени при повторных вычислениях заметно больше, чем эффект от такой оптимизации. Точно так же встраивание функции в алгоритм интегрирования не приводит к сколько-нибудь заметному эффекту.

Это может удивить тех, кто программирует на Fortran, C/C++ или Object Pascal (Delphi, Lazarus). Но, как уже говорилось, в Java реализована очень эффективная оптимизация кода как на этапе компиляции в байт-код средой NetBeans, так и на этапе выполнения байт-кода виртуальной Java-машиной (особенно в JDK 6). Поэтому на начальном этапе написания программы лучше игнорировать проблемы скорости расчетов, лучше постараться написать максимально понятные программы (с прозрачным исходным кодом).

Тем не менее, совсем неоптимизированный вариант выполняется заметно медленнее (для простых функций в четыре раза). Поэтому совсем уж игнорировать ручную оптимизацию вычислений не следует. Но оптимизация должна быть разумной, и ее нужно

проводить только там, где без нее не обойтись. В *разд. 17.5—17.7* будет рассказано, как находить "горячие точки" (hot spots) — участки программы, потребляющие максимальные вычислительные ресурсы.

Из табл. 5.3 также следует, что вид исходной функции, в отличие от "экстремистских" вариантов оптимизации, очень заметно влияет на время вычислений. Больше всего времени требует операция возведения в вещественную степень.

Часто встречающаяся ошибка — попытка использования функции `pow` вместо последовательности из нескольких действий умножения, либо деления, либо вызова функции извлечения квадратного корня:

`x*x`, `x*x*x`, `1/x`, `sqrt(x)` и т. п.

Как можно заметить, функция `sqrt(x)` вычисляется почти в 30 раз быстрее, чем `pow(x,0.5)`.

Аналогично, функция `x*x* sqrt(x)` вычисляется в 15 раз быстрее, чем `pow(x,2.5)`. И если вместо `pow(x,2.5)` написать `x*x* sqrt(x)`, то скорость вычислений возрастет на порядок и более. Вот за такую оптимизацию стоит побороться, но только в том случае, если быстродействие в данном месте программы действительно важно для вашей задачи!

На следующем месте (по времени вычисления) после функции `pow` стоят тригонометрические функции, а из них — тангенс (напомним, что в англоязычной традиции эта функция записывается как `tan`), затем — извлечение квадратного корня с помощью функции `sqrt`. Наконец, заметное время занимает вещественное деление. А вот сложение, вычитание и умножение занимают пренебрежимо малое время, которое можно не учитывать по сравнению со временем вызова функций.

Заметим, что замена оператора `s=s+f(x)` на `s+=f(x)` или `s=s*h` на `s*=h` при вычислениях в формате с плавающей точкой никогда не приведет к выигрышу в скорости, и целесообразна только в случаях очень длинных выражений в левой части.

Например, запись выражения:

```
object1.myObjField1.myObjField2.myObjField3.x+=5;
```

выглядит проще и понятнее, чем:

```
object1.myObjField1.myObjField2.myObjField3.x=
object1.myObjField1.myObjField2.myObjField3.x+5;
```

Но использование комбинированных операторов с целью повышения быстродействия совершенно неоправданно.

Существует мнение, что программы, написанные на Java (как и на других интерпретируемых языках), выполняются медленно. Чтобы опровергнуть это заблуждение, в последних двух столбцах табл. 5.3 приведено время выполнения максимально оптимизированных вариантов вычислений на языке программирования Object Pascal (среда программирования Delphi) и языке C++ (компилятор g++ от GNU, оптимизация `-O3`).

Object Pascal — компилируемый язык, но скорость вычислений в формате с плавающей точкой для Java и Delphi оказывается близкой или даже в точности равной (например, при вычислении интеграла от функции `1/x`). Иногда Java обеспечивает даже бóльшую скорость (например, при расчете интеграла от квадратного корня с помощью функции

sqrt или интеграла от x^2 с помощью функции pow(x,2) в Java и power(x,2) в Delphi). Хотя при использовании той же функции pow для других значений степени (0.5 или 2.5) Java заметно проигрывала по скорости. Очевидно, что отличия по скорости скорее связаны с особенностями реализации вычислительных алгоритмов, нежели со скоростью работы Java-кода как такового.

Результаты для C++ показывают, что программа, написанная на C++, обычно выполняет расчеты немного быстрее, чем написанная на Delphi, хотя заметных отличий по скорости нет. Аналогичные результаты получаются и для Fortran (компилятор f77, поставляемый в составе комплектов Cygwin и MGW, дает очень медленный код: для получения сопоставимых с C++ результатов необходимы компиляторы gfortran от GNU, ifortran от Intel или Fortran из Sun Studio от Sun Microsystems). Отличие по скорости хорошо оптимизированного кода Fortran от хорошо оптимизированного кода C++ составляет 2—3 процента. Поэтому сделанные ранее выводы остаются в силе и для C++, и для Fortran. Причем хотелось бы отметить, что для функции sqrt(x) вычисления на Java происходят даже немного быстрее, чем на C++!

В рассмотренной программе C++ отсутствовала оптимизация под конкретную аппаратную платформу (тип процессора), при такой оптимизации удалось бы увеличить быстродействие еще примерно на 30%. Но даже при этих условиях превосходство C++ или Fortran над Java в скорости незначительно по сравнению с теми преимуществами, которые может дать разумная ручная оптимизация выражений. Поэтому можно сделать вывод, что легенды об абсолютном превосходстве C++ и Fortran над Java по скорости в области вычислений в формате с плавающей точкой — не более чем вымыслы — отголоски эпохи первых JDK. JIT-компиляция, применяемая в современных вариантах Java-машины, позволяет Java вплотную приблизиться по быстродействию к компилируемым языкам программирования.

Таким образом, при вычислениях с плавающей точкой программы на Java уступают по скорости всего в 1,1—2 раза программам, написанным на обычных компилируемых языках, не использующих интерпретацию кода виртуальной машиной. При этом вызов подпрограмм занимает пренебрежимо малое время по сравнению со временем вычисления выражений.

5.9. Особенности целочисленных вычислений — организация циклов, приоритет операторов и арифметическое переполнение

Для целочисленных расчетов (в отличие от вычислений в формате с плавающей точкой) выигрыш от некоторых вариантов оптимизации может оказаться значительным. Это можно показать с помощью примера 05_9_effectiveCalc, который иллюстрирует особенности эффективных целочисленных алгоритмов.

Проверялись три варианта — оптимизированный, обычный и медленный для 100 миллионов суммирований, как и в предыдущем варианте (табл. 5.4).

В медленном варианте функция *f* передавалась в подпрограмму через список параметров с помощью объекта fnc, как в основных вариантах вычислений в формате с плавающей точкой (листинг 5.10).

Таблица 5.4. *Время целочисленных вычислений — суммирование выражений*

Суммируемая величина	Время, секунд						
	Оптимизированный вариант (`long`)	Обычный вариант (`long`)	Медленный вариант (`long`)	Оптимизированный вариант (`int`)	Delphi (`int`)	C++ (`long` — аналог `int` в Java)	C++ (`long long` — аналог `long` в Java)
`1`	0.4	0,4	1,6	0,12	0,07	0,06	0,23
`i`	0,4	0,4	1,6	0,15*	0,12*	0,1*	0,24
`i*i`	0,7	0,7	1,6	0,23*	0,17*	0,1*	0,62
`i/3`	3,4	3,4	5,6	2,1*	2,0*	0,27*	2,1
`i & 0xAF`	0,6	0,6	1,7	0,24*	0,15*	0,07*	0,24
`i | 0xAF`	0,6	0,6	1,7	0,22*	0,15*	0,07*	0,25
`i ^ 0xAF`	0,6	0,6	1,7	0,22*	0,15*	0,07*	0,24
`i<<3`	0,6	0,6	1,7	0,21*	0,15*	0,06*	0,24
`i*8`	0,7	0,7	1,7	0,21*	0,15*	0,06*	0,24
`i>>3`	0,6	0,6	1,7	0,21*	0,15*	0,07*	0,29
`i/8`	3,4	3,4	5,6	2,1*	0,18*	0,11*	0,33

* Звездочкой помечены результаты, для которых наблюдается арифметическое переполнение.

Листинг 5.10. Медленный вариант вычисления суммы

```
public static long slowSum(long i1,long i2,Fnc fnc) {

    long r=0;
    for(long i=i1; i<=i2; i++)
        r=r+fnc.f(i);
    return r;
}
```

В оптимизированном и обычном вариантах вычисляемая функция встраивалась в подпрограмму расчета интеграла, причем в оптимизированном варианте применялись комбинированные операторы вида `+=` (листинг 5.11).

Листинг 5.11. Различные варианты вычисления сумм для целочисленных функций

```
public static long fastSum_1(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r+=1;
    return r;
}
```

```
public static long fastSum_i(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r+=i;
    return r;
}
public static long fastSum_and(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r+=i&0xAF;
    return r;
}
public static long fastSum_shl(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r+=i<<3;
    return r;
}
public static long usualSum_1(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+1;
    return r;
}
public static long usualSum_i(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r= r+i; // i+r;-работает медленнее!
    return r;
}
public static long usualSum_i_3(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+i/3;
    return r;
}
public static long usualSum_and(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+(i&0xAF);
    return r;
}
public static long usualSum_or(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+(i|0xAF);
    return r;
}
```

```
public static long usualSum_xor(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+(i^0xAF);
    return r;
}
public static long usualSum_shl(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+(i<<3);
    return r;
}
public static long usualSum_shr(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=r+(i>>3);
    return r;
}
```

Как видно из табл. 5.4, в случае целочисленных вычислений встраивание функции в подпрограмму вычисления суммы заметно повышает скорость расчетов даже для величин типа `long`, не говоря уже о величинах типа `int`. Для простейших выражений время вычислений удается уменьшить в несколько раз, что в ряде случаев может оказаться важным. Но при этом комбинированные операторы не дают никакого выигрыша.

Правда, если в подпрограмме `usualSum_i` (см. листинг 5.11) заменить алгоритм вычисления суммы (листинг 5.12), то во время первых 2—3 запусков вычислений (пример 05_9_effectiveCalc) получится время 0,7 с, а при последующих вычислениях данное время уменьшается до 0,4 с. Это означает, что оптимизацией с заменой оператора `r=i+r` на `r+=i` занимается виртуальная Java-машина JRE 6, а не компилятор NetBeans (в более старых версиях JDK такого эффекта не наблюдается).

Листинг 5.12. Алгоритм с заменой присваивания `r+=i` на `r=i+r`

```
public static long usualSum_i(long i1,long i2) {
    long r=0;
    for(long i=i1;i<=i2;i++)
        r=i+r;//r+=i;
    return r;
}
```

Кроме того, становится понятным, что быстродействие одного и того же алгоритма в Java зависит как от версии JDK (как правило, улучшаясь с выходом новых версий), так и от истории вызова кода. В JRE 6 оптимизация идет динамически в фоновом режиме параллельно с работой программы, с постепенной заменой наиболее активно работающих участков на JIT-скомпилированный код. Поэтому эффект от оптимизации в нашем случае становится заметным примерно через секунду-другую после начала активной работы вычислительных алгоритмов программы.

Следует отметить, что скорость вычислений с 64-битовыми числами типа `long` примерно в 2—3 раза меньше, чем с 32-битовыми числами типа `int`. Тем не менее уже в приведенном простейшем примере мы убеждаемся в опасности такого выигрыша в скорости — за исключением суммирования единиц, все полученные ответы неправильны, т. к. возникает арифметическое переполнение. Таким образом, в большинстве случаев там, где скорость вычислений некритична, лучше выбрать тип `long`.

Тем не менее в некоторых алгоритмах, для которых скорость чрезвычайно важна и где можно гарантировать, что переполнения не будет, приходится использовать тип `int`. Во многих случаях его лучше использовать и в многопоточных программах, так как при работе с величинами типа `long` в них возникают дополнительные проблемы — *см. главу 10*.

Еще одна причина, по которой могут потребоваться числа типа `int`, — работа с массивами. Ячейки типа `long` занимают в два раза больше места, чем ячейки типа `int`, поэтому хранить большой объем данных в ряде случаев бывает полезно в данных именно этого типа.

Из табл. 5.4 видно, что скорость работы с целыми числами программ, написанных на языке Java, все-таки несколько меньше, чем у аналогичных программ, разработанных с помощью Delphi (примерно в полтора раза), не говоря уже о C/C++. Для большинства целочисленных операций можно оценить проигрыш Java в быстродействии по сравнению с C/C++ как 1,5—2 раза, хотя иногда проигрыш доходит до 10 раз. Причем в приведенных примерах для Java реализован неструктурный, но зато максимально быстрый вариант вызовов, когда алгоритм вычисления функций не выносился в отдельную подпрограмму, а встраивался непосредственно внутрь цикла. Отметим, что вызов функции в C/C++ осуществляется очень быстро, и временем этого вызова можно пренебречь, а в Java вызов функции происходит в несколько раз медленнее, чем целочисленные операции.

В большинстве случаев приведенные различия в скорости не имеют принципиального значения — это не те порядки, которые первые интерпретируемые языки программирования проигрывали в скорости компилируемым. Но некоторые особо ресурсоемкие алгоритмы (например, для обработки изображений) приходится писать на C или C++ и подключать к программам Java с помощью встраивания native-кода (платформозависимого кода, скомпилированного для конкретного типа процессора и операционной системы). Это, конечно, крайний случай, когда от такого быстродействия зависит слишком много, ведь тогда приходится жертвовать всеми преимуществами Java: надежностью, переносимостью на уровне исполняемого кода (независимостью от аппаратной и программной платформ), модифицируемостью исходного кода. Про применение native-кода будет более подробно изложено в *главе 16*.

При анализе целочисленных вычислений следует остановиться еще на двух важных моментах.

Во-первых, это приоритет операторов.

Обратите внимание на форму записи операторов: `&` (И), `|` (ИЛИ), `^` (исключающее ИЛИ) в следующих примерах:

```
r=r+(i&0xAF);
r=r+(i|0xAF);
r=r+(i^0xAF);
```

Если бы мы написали соответствующие выражения без круглых скобок, то диагностики ошибок не было, но ответ в вычислении сумм оказался совсем другим.

Так, в операторе

```
r=r+i&0xAF;
```

сначала выполняется сложение `r+i`, после чего вычисляется арифметическое `И` и полученный результат записывается в переменную `r`.

То есть он эквивалентен следующему:

```
r=(r+i)& 0xAF;
```

К сожалению, никаких предупреждений компилятор не выдает, не говоря уже о синтаксической ошибке. Это одна из немногих опасных конструкций Java, где на уровне синтаксиса заложена возможность недиагностируемых ошибок.

Во-вторых, из табл. 5.4 видно, что время операции целочисленного деления очень велико (оно почти на порядок больше времени других простейших целочисленных операций и примерно равно времени вещественного деления). Таким образом, в критичных по времени алгоритмах следует по мере возможности избегать деления.

Можно заметить, что в Delphi и C++ операция деления на 8 выполняется на порядок быстрее, чем в Java, хотя деление на 3 происходит с такой же скоростью, как и в Java. Это связано с тем, что оптимизирующий компилятор Delphi и C++ заменяет деление на 2 операцией правого сдвига на 1 бит, деление на 4 — операцией правого сдвига на 2 бита, деление на 8 — операцией правого сдвига на 3 бита и т. д.

В Java компилятор пока не делает такой замены. Будем надеяться, что в ближайшем будущем эти простейшие и достаточно очевидные варианты оптимизации будут в него добавлены на этапе компиляции, а не при анализе кода виртуальной машиной, точно так же, как замена кода вида:

`r=r+выражение` или `r= выражение+r` на `r+=выражение`,

`r=r*выражение` или `r= выражение*r` на `r*=выражение`,

и т. п.

Вообще-то следовало бы встраивать в компилятор Java два режима — обычную (быструю, рабочую) компиляцию и окончательную (более длительную, но значительно более тщательно оптимизирующую), как это сделано в C и C++.

5.10. Оператор цикла *while* — цикл с предусловием

Цикл на основе оператора `while` выглядит следующим образом:

```
while(условие)
    оператор;
```

Пока условие сохраняет значение `true`, в цикле выполняется оператор, иначе действие цикла прекращается. Если условие с самого начала `false`, то цикл сразу прекращается, и тело цикла не выполнится ни разу.

Конструкцию while обычно применяют вместо цикла for тогда, когда условия продолжения достаточно сложные. В отличие от цикла for в этом случае нет формально заданного счетчика цикла, и он не изменяется автоматически (за это отвечает программист). Хотя вполне возможна замена цикла for на while и наоборот. Многие программисты предпочитают пользоваться только циклом for, считая его более универсальным.

Пример цикла while

```
i=1;
x=0;
while(i<=n){
  x+=i;//эквивалентно x=x+i;
  i*=2;//эквивалентно i=2*i;
}
```

В операторе while очень распространены ошибки, приводящие к неустойчивости алгоритмов из-за сравнения чисел с плавающей точкой на неравенство. Это бывает даже чаще, чем попытки использовать цикл for с вещественным счетчиком цикла. Как мы знаем, проверять числа double или float на равенство в подавляющем большинстве случаев некорректно (из-за ошибок представления таких чисел в компьютере). Но большинство программистов почему-то считают, что при проверке на неравенство проблем не возникает, хотя это не так. Приведем пример типичной ошибки в организации такого цикла (листинг 5.13).

Листинг 5.13. Цикл while с неустойчивостью

```
double a=...;
double b=...;
double dx=...;
double x=a;
while(x<=b){
  ...
  x=x+dx;
}
```

Как мы уже знаем, данный цикл будет обладать неустойчивостью в случае, когда на интервале от a до b укладывается целое число шагов.

Например, при a=0, b=10, dx=0.1 тело цикла будет выполняться,

если x=0, x=0.1, ..., x=9.9.

А вот при x=10 тело цикла может либо выполниться, либо не выполниться (как повезет!). Причина связана с конечной точностью операций с числами в формате с плавающей точкой. Величина шага dx в двоичном представлении чуть-чуть отличается от значения 0.1, и при каждом цикле систематическая погрешность в значении x накапливается. Поэтому точное значение x=10 достигнуто не будет. Величина x будет либо чуть-чуть меньше, либо чуть-чуть больше. В первом случае тело цикла выполнится, во втором — нет, т. е. пройдет либо 100, либо 101 итерация (число выполнений тела цикла).

5.11. Оператор цикла *do...while* — цикл с постусловием

Цикл с оператором `do...while` выглядит следующим образом:

```
do
  оператор;
while(условие);
```

Если условие принимает значение `false`, цикл прекращается. Тело цикла выполняется до проверки условия, поэтому оно всегда выполнится хотя бы один раз.

Пример цикла `do...while`

```
int i=0;
double x=1;
do{
 i++;  // i=i+1;
 x*=i; // x=x*i;
}
while(i<n);
```

Если с помощью оператора `do...while` организуется цикл с вещественным счетчиком или другой проверкой на равенство/неравенство чисел типа `float` или `double`, то возникают точно такие же проблемы, как описанные для циклов `for` и `while`.

При необходимости организовать бесконечный цикл (с выходом из тела цикла с помощью оператора прерывания) часто используют следующий вариант:

```
do{
 ...
}
while(true);
```

5.12. Операторы прерывания *continue*, *break*, *return*, *System.exit*

Довольно часто требуется при выполнении какого-либо условия прервать цикл или подпрограмму и перейти к выполнению другого алгоритма или очередной итерации цикла. При неструктурном программировании для этих целей служил оператор `goto`. В Java имеются более гибкие структурные средства для решения подобных проблем — это операторы `continue`, `break`, `return`, `System.exit`:

- `continue` — прерывание выполнения тела цикла и переход к следующей итерации (проверке условия) текущего цикла;
- `continue` *`имя_метки`* — прерывание выполнения тела цикла и переход к следующей итерации (проверке условия) цикла, помеченного меткой;

- `break` — выход из текущего цикла;
- `break` *`имя_метки`* — выход из цикла, помеченного меткой;
- `return` — выход из текущей подпрограммы (в том числе из тела цикла) без возврата значения;
- `return` *`значение`* — выход из текущей подпрограммы (в том числе из тела цикла) с возвратом значения;
- `System.exit(n)` — выход из приложения с кодом завершения `n`. Целое число `n` произвольно задается программистом. Если `n=0`, то выход считается нормальным, в других случаях — аварийным. Приложение перед завершением сообщает значение числа `n` операционной системе, для того чтобы программист мог установить, по какой причине произошел аварийный выход.

Существует два варианта операторов `continue` и `break`: без меток для выхода из текущего (самого внутреннего по вложенности) цикла и с меткой — для выхода из помеченного ею цикла. Меткой является идентификатор, после которого стоит двоеточие. Метку можно ставить непосредственно перед ключевым словом, начинающим задание цикла (`for`, `while`, `do`), а также перед любым составным оператором. В последнем случае оператор `break имя_метки` завершает выполнение составного оператора.

Листинг 5.14 иллюстрирует пример вызова оператора `continue` без метки.

Листинг 5.14. Пример вызова оператора `continue` без метки

```
        for(int i=1;i<=10;i++){
            if(i==(i/2)*2){
                continue;
            }
            System.out.println("i="+i);
        }
```

В данном цикле не будут печататься все значения `i`, для которых `i==(i/2)*2` (т. е. выводиться в окно консоли будут только нечетные значения `i`).

Листинг 5.15 содержит еще один пример использования `continue` без метки.

Листинг 5.15. Вызов оператора `continue` без метки во вложенном цикле

```
for(int i=1;i<=20;i++){
for(int j=1;j<=20;j++){
    if(i*j==(i*j/2)*2){
        continue;
    }
    System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+ (1.0/(i*j-20)));
}
}
```

Из листинга 5.16 ясно, что в этом случае будут выводиться значения `i`, `j` и `1.0/(i*j-20)` для всех нечетных `i` и `j` от 1 до 19 (т. е. будут пропущены значения для всех четных `i` и `j`).

Листинг 5.16. Результаты вывода в консоль

```
i=1 j=1 1.0/(i*j-20)=-0.05263157894736842
i=1 j=3 1.0/(i*j-20)=-0.058823529411764705
i=1 j=5 1.0/(i*j-20)=-0.06666666666666667
i=1 j=7 1.0/(i*j-20)=-0.07692307692307693
i=1 j=9 1.0/(i*j-20)=-0.09090909090909091
i=1 j=11 1.0/(i*j-20)=-0.1111111111111111
i=1 j=13 1.0/(i*j-20)=-0.14285714285714285
i=1 j=15 1.0/(i*j-20)=-0.2
i=1 j=17 1.0/(i*j-20)=-0.3333333333333333
i=1 j=19 1.0/(i*j-20)=-1.0
i=3 j=1 1.0/(i*j-20)=-0.058823529411764705
i=3 j=3 1.0/(i*j-20)=-0.09090909090909091
i=3 j=5 1.0/(i*j-20)=-0.2
i=3 j=7 1.0/(i*j-20)=1.0
...
i=19 j=9 1.0/(i*j-20)=0.006622516556291391
i=19 j=11 1.0/(i*j-20)=0.005291005291005291
i=19 j=13 1.0/(i*j-20)=0.004405286343612335
i=19 j=15 1.0/(i*j-20)=0.0037735849056603774
i=19 j=17 1.0/(i*j-20)=0.0033003300330033004
i=19 j=19 1.0/(i*j-20)=0.002932551319648094
```

Листинг 5.17 иллюстрирует пример вызова оператора `continue` с меткой `label1`.

Листинг 5.17. Пример вызова оператора `continue` с меткой

```
label1:
  for(int i=1;i<=20;i++){
  for(int j=1;j<=20;j++){
      if(i*j==(i*j/2)*2){
          continue label1;
      }
      System.out.println("i=" + i + " j=" + j + " 1.0/(i*j-20)=" +
                                                    (1.0/(i*j-20)) );
  }
  }
```

В отличие от предыдущего случая, после каждого достижения равенства `i*j==(i*j/2)*2` будет производиться выход из внутреннего цикла (по `j`), и все последующие `j` для таких значений `i` будут пропущены. Поэтому будут выведены только значения (листинг 5.18).

Листинг 5.18. Результаты вывода в консоль

```
i=1 j=1 1.0/(i*j-20)=-0.05263157894736842
i=3 j=1 1.0/(i*j-20)=-0.058823529411764705
i=5 j=1 1.0/(i*j-20)=-0.06666666666666667
i=7 j=1 1.0/(i*j-20)=-0.07692307692307693
i=9 j=1 1.0/(i*j-20)=-0.09090909090909091
```

```
i=11 j=1 1.0/(i*j-20)=-0.1111111111111111
i=13 j=1 1.0/(i*j-20)=-0.14285714285714285
i=15 j=1 1.0/(i*j-20)=-0.2
i=17 j=1 1.0/(i*j-20)=-0.3333333333333333
i=19 j=1 1.0/(i*j-20)=-1.0
```

В листинге 5.19 приведен пример использования оператора `break` без метки.

Листинг 5.19. Пример вызова оператора `break` без метки

```
        for(int i=1;i<=10;i++){
            if( i+6== i*i ){
                break;
            }
            System.out.println("i="+i);
        }
```

Данный цикл остановится при выполнении условия `i+6== i*i` (т. е. вывод в окно консоли будет только для значений `i`, равных 1 и 2).

Еще один пример использования оператора `break` без метки:

```
  for(int i=1;i<=20;i++){
  for(int j=1;j<=20;j++){
      if(i*j==(i*j/2)*2){
          break;
      }
      System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+
                                              (1.0/(i*j-20)) );
  }
  }
```

В этом случае будут выводиться все значения `i` и `j` до тех пор, пока не найдется пара `i` и `j`, для которых `i*j==(i*j/2)*2`, после чего внутренний цикл прекращается — значения `i`, `j` и `1.0/(i*j-20)` для данного и последующих значений `j` при соответствующем `i` не будут выводиться в окно консоли. Но внешний цикл (по `i`) будет продолжаться, и вывод продолжится для новых `i` и `j`. Результат будет таким же, как для `continue` с меткой для внешнего цикла.

Листинг 5.20 иллюстрирует пример использования `break` с меткой `label_for1`.

Листинг 5.20. Вызов оператора `break` с меткой во вложенном цикле

```
label_for1:
  for(int i=1;i<=20;i++){
  for(int j=1;j<=20;j++){
      if(i*j==(i*j/2)*2){
          break label_for1;
      }
      System.out.println("i="+i+" j="+j+ " 1.0/(i*j-20)="+
                                                    (1.0/(i*j-20)) );
  }
  }
```

В данном случае также будут выводиться все значения `i` и `j` до тех пор, пока не найдется пара `i` и `j`, для которой `i*j==(i*j/2)*2`, после чего прекращается внешний цикл, а значит, и внутренний тоже (так что вывод в окно консоли прекратится). Поэтому вывод будет только для `i=1`, `j=1`.

Краткие итоги по главе

- Среди управляющих конструкций в Java имеются:
  - условный оператор `if`;
  - оператор выбора `switch`;
  - условное выражение `...?... : ...`;
  - операторы инкремента `++` и декремента `--`.
- В Java есть следующие операторы цикла:
  - `for` — цикл со счетчиками;
  - `while` — цикл с предусловием;
  - `do...while` — цикл с постусловием;
  - `for each` — цикл перебора коллекции.

  А также операторы прерывания циклов `continue` и `break`, прерывания подпрограмм `return`, программы — `System.exit`.
- Сравнение чисел в формате с плавающей точкой практически всегда некорректно. Но даже проверку на неравенство таких чисел нельзя выполнять в случае, когда в точной арифметике эти числа должны быть равны. Поэтому циклы с вещественным счетчиком в случаях, когда на интервале изменения счетчика укладывается целое число шагов счетчика, недопустимы.
- Циклы относятся к управляющим конструкциям, наиболее критичным к скорости выполнения.
- Не следует делать попытки оптимизации циклов, некритичных к скорости выполнения.
- Как при вычислениях с плавающей точкой, так и при целочисленных вычислениях программы, написанные на Java, уступают по скорости всего в 1,5—2 раза программам, написанным на обычных компилируемых языках, не использующих интерпретацию кода виртуальной машиной.
- Вызов подпрограмм занимает пренебрежимо малое время по сравнению со временем вычисления выражений в формате с плавающей точкой.
- Комбинированные операторы вида `+=`, `*=` и т. п. практически никогда не дают выигрыша в скорости работы Java-кода.
- При целочисленных вычислениях время вызова подпрограмм значительно превышает время вычисления простых целочисленных выражений. Поэтому внутри циклов, критичных к скорости выполнения, следует минимизировать число вызовов функций.

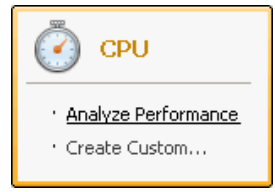
- Скорость вычислений с 64-битовыми числами типа `long` примерно в 2—3 раза меньше, чем с 32-битовыми числами типа `int`.
- В критичных по времени целочисленных алгоритмах следует по мере возможности избегать целочисленного деления.
- На участках кода, критичных к скорости, целочисленное деление `x` на 2^n лучше заменять на оператор `x>>n`, поскольку пока компилятор Java не делает такой замены автоматически.
- Умножение `x` на 2^n выполняется практически с той же скоростью, что оператор `x<<n`. Поэтому нет необходимости заменять оператор умножения левым сдвигом.
- Скорость целочисленных вычислений в Java в несколько раз меньше по сравнению с C/C++. При этом в C/C++ временем вызова подпрограмм в большинстве случаев можно пренебречь.

Типичные ошибки

- "Забытый break" в операторе `switch`.
- Проверка на равенство значений в формате с плавающей точкой.
- Проверка на неравенство значений в формате с плавающей точкой, которые при точных вычислениях должны быть равны.
- Использование вещественного счетчика цикла.
- Попытка оптимизации циклов, некритичных к скорости выполнения.
- Наличие внутри циклов, критичных к скорости выполнения, выражений, которые вычисляются каждый раз в очередном цикле, хотя могут быть вычислены один раз вне цикла.

Задания

- Написать приложение с графическим пользовательским интерфейсом, в котором при нажатии кнопки иллюстрируется действие операторов цикла `for`, `while`, `do...while` в зависимости от того, какая из кнопок `JRadioButton` нажата. С помощью этих циклов должна вычисляться площадь под кривой, задаваемой функцией $f(x) = ax^2 + bx + c$, при x, меняющемся от x_1 до x_2, где величины a, b, c, x_1 и x_2 должен вводить пользователь в соответствующих пунктах ввода.
- Добавить в это приложение вычисление с вещественным счетчиком цикла для какого-либо из операторов (`for`, `while` или `do...while`) в случае, когда отмечена соответствующая опция с помощью кнопки `JCheckBox`. Показать наличие неустойчивости при очень малом изменении входных параметров (какой-либо из величин a, b, c, x_1, x_2). Объяснить, почему изменение одних параметров приводит к неустойчивости, а других — нет.
- Написать приложение с графическим пользовательским интерфейсом, в котором при нажатии кнопки `JButton` иллюстрируется действие операторов прерывания `continue`, `break`, `return`, `System.exit` в зависимости от того, какая из кнопок `JToggleButton` нажата.



ГЛАВА 6

Начальные сведения об объектном программировании

В *главе 1* мы уже познакомились с первым принципом объектного программирования — *инкапсуляцией*. Затем научились пользоваться уже готовыми классами. Это начальная стадия изучения объектного программирования. Но для того чтобы овладеть основными возможностями объектного программирования, требуется научиться создавать собственные классы, изменяющие и усложняющие поведение существующих классов. Важнейшими элементами такого умения является использование *наследования*, *полиморфизма* и *композиции*.

6.1. Наследование и полиморфизм

Наследование опирается на инкапсуляцию. Оно позволяет строить на основе первоначального класса другие, добавляя в классы новые поля данных и методы. Первоначальный класс называется *прародителем* (ancestor), новые классы — его *потомками* (descendants). От потомков, в свою очередь, тоже можно наследовать, получая очередных потомков и т. д.

Набор классов, связанных отношением наследования, называется *иерархией классов*. А класс, стоящий во главе иерархии, от которого унаследованы все остальные (прямо или опосредованно), называется *базовым классом иерархии*. В Java все классы являются потомками класса `Object` (т. е. он базовый для всех классов). Тем не менее, если рассматривается поведение, характерное для объектов какого-то класса, а также всех его потомков, то говорят об иерархии, начинающейся с этого класса. В этом случае именно он будет базовым классом иерархии.

Полиморфизм опирается как на инкапсуляцию, так и на наследование. Как показывает опыт преподавания автором курса объектного программирования, это наиболее сложный для понимания принцип. Слово *полиморфизм* в переводе с греческого означает "имеющий много форм". В объектном программировании под полиморфизмом подразумевается наличие кода, написанного с использованием ссылок, имеющих тип базового класса иерархии. При этом такой код должен правильно работать для любого объекта, являющегося экземпляром класса из данной иерархии, независимо от того, где этот класс расположен в иерархии. Такой код и называется *полиморфным*. При написании полиморфного кода заранее неизвестно, для объектов какого типа он будет работать — один и тот же метод будет исполняться по-разному в зависимости от типа объекта.

Пусть, например, у нас имеется класс `Figure` (фигура), в котором заданы методы `show()` (показать фигуру на экране) и `hide()` (скрыть ее). Тогда переменная `figure` может ссылаться на объект, принадлежащий любому из классов-наследников `Figure`. А поскольку для типа `Figure` определены методы `show()` и `hide()`, то вызовы `figure.show()` и `figure.hide()` будут отображать или скрывать объект, на который ссылается эта переменная. Причем сам объект знает, как себя показывать или скрывать, а код пишется на уровне абстракций этих действий.

Основное преимущество объектного программирования (по сравнению с процедурным) как раз и заключается в возможности написания полиморфного кода. Именно для этого создается иерархия классов. Полиморфизм позволяет резко увеличить *коэффициент повторного использования* программного кода и *модифицируемость* этого кода по сравнению с процедурным программированием.

В качестве примера того, как строится иерархия, рассмотрим иерархию фигур, отрисовываемых на экране (рис. 6.1). Базовый класс в ней — `Figure`, от него наследуются `Dot` (точка), `Triangle` (треугольник) и `Square` (квадрат). От `Dot` наследуется класс `Circle` (окружность), а от `Circle` наследуется `Ellipse` (эллипс). И, наконец, от `Square` наследуется `Rectangle` (прямоугольник).

Отметим, что в иерархии принято рисовать стрелки в направлении от наследника к прародителю. Такое направление называется Generalization (обобщение, генерализация). Оно противоположно направлению наследования, которое принято называть Specialization (специализация). Стрелки символизируют направление в сторону упрощения.

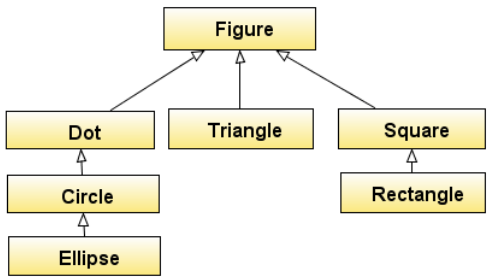


Рис. 6.1. Иерархия фигур, отрисовываемых на экране

Часто класс-прародитель называют *суперклассом* (superclass), а класс-наследник — *субклассом* (subclass). Но такая терминология подталкивает начинающих программистов к неверной логике: суперкласс пытаются сделать суперсложным, так, чтобы его подклассы (это неверно воспринимается синонимом выражению упрощенные разновидности) обладали упрощенным по сравнению с ним поведением. На деле же *потомки должны обладать более сложным устройством и поведением по сравнению с прародителем*. Поэтому в данном учебном пособии предпочтение отдается терминам "прародитель" и "наследник".

Чем ближе к основанию иерархии находится класс, тем он более общий и универсальный (general) (и одновременно более простой). Класс, который лежит в основе иерархии, называется *базовым классом* этой иерархии. Базовый класс всегда называют име-

нем, которое характеризует все объекты (экземпляры классов-наследников) и которое выражает наиболее общую абстракцию, применимую к таким объектам. В нашем случае это класс `Figure`. Любая фигура будет иметь поля данных *x* и *y* — координаты фигуры на экране.

Класс `Dot` (точка) — наследник `Figure` — будет иметь поля данных `x` и `y`, наследуемые от `Figure` (т. е. в самом классе `Dot` задавать эти поля не надо). От `Dot` мы наследуем класс `Circle` (окружность), поэтому в нем также имеются поля `x` и `y`, наследуемые от `Figure`, но одновременно появляются и дополнительные поля данных.

У `Circle` это поле, соответствующее радиусу. Мы назовем его `r`. Кроме того, для окружности возможна операция изменения радиуса, поэтому в ней может появиться новый метод, обеспечивающий это действие, назовем его `setSize` (установить размер).

Класс `Ellipse` имеет те же поля данных и обеспечивает то же поведение, что и `Circle`, но в этом классе появляется дополнительное поле данных `r2` (длина второй полуоси эллипса), значение которого можно регулировать.

Возможен и другой подход (в некотором роде более логичный) — считать эллипс сплюснутой или растянутой окружностью. В этом случае необходимо ввести коэффициент растяжения (aspect ratio), назовем его `k`. Тогда эллипс будет характеризоваться радиусом `r` и коэффициентом растяжения `k`. Метод, обеспечивающий изменение `k`, назовем `stretch` (растянуть). Обратим внимание, что исходя из выбранной логики действий метод `scale` должен приводить к изменению поля `r` и не затрагивать поле `k`, чтобы эллипс масштабировался без изменения формы.

Каждый из классов этой ветви иерархии фигур можно считать описанием усложненной точки. При этом важно, что любой объект такого типа можно считать точкой, которую усложнили. Грубо говоря, круг или эллипс — это такая "жирная" точка.

Аналогично `Ellipse` — усложненная окружность.

Класс `Square` наследует поля `x` и `y`, но в нем добавляется поле, соответствующее стороне квадрата (назовем его `a`). У `Triangle` в качестве новых, не унаследованных полей данных, могут выступать либо координаты вершин треугольника, либо координаты одной из вершин, либо длины прилегающих к ней сторон и угол между ними и т. д.

Как располагать классы иерархии — базовый класс внизу, а наследники вверху, образуя ветви дерева наследования, или наоборот, базовый класс вверху, а наследники внизу, образуя корни дерева наследования — принципиального значения не имеет. По-видимому, на начальном этапе развития объектного программирования применялся первый вариант, почему базовый класс, лежащий в основе иерархии, и получил такое название. В настоящее время чаще применяется второй способ, когда базовый класс располагают сверху. Такой вариант выбран и в данном учебном пособии.

В литературе по объектному программированию часто встречается следующий критерий: "если имеются классы `A1` и `A2` и можно сказать, что `A2` является частным случаем `A1`, то `A2` должен описываться как потомок `A1`". Данная формулировка не совсем корректна.

Очень часто встречающийся вариант ошибочных рассуждений, основанный на этом критерии и приводящий к *неправильному построению иерархии*, выглядит так: "по-

скольку Circle — частный случай Ellipse (при равных длинах полуосей), а Dot — частный случай Circle (при нулевом радиусе), то класс Ellipse более общий, чем Circle, а Circle — более общий, чем Dot. Поэтому Ellipse должен являться прародителем для Circle, а Circle — прародителем для Dot". Ошибка заключается в неправильном понимании идей общности и специализации, а также в характерной путанице, когда объекты не отличают от классов.

В концепции наследования основное внимание уделяется *поведению объектов*. Объекты с разным поведением имеют другой тип, а значения полей данных характеризуют *состояние объекта*, но не его тип. Поведение задается методами, и это делается в классах. А состояние относится к конкретному объекту, причем в конкретный момент выполнения программы.

Каждый объект класса-потомка *при любых значениях полей* нужно рассматривать как экземпляр класса-прародителя. Он должен обладать той же абстракцией поведения, что и прародитель, но только с некоторыми изменениями на уровне реализации этого поведения. Мы говорим про абстракцию поведения как о тех характерных действиях, которые могут быть описаны на уровне полиморфного кода, безотносительно к конкретной реализации в конкретном классе.

По своему поведению любой объект-эллипс вполне может рассматриваться как экземпляр типа Окружность и даже вести себя в точности как окружность, но не наоборот: объект типа Окружность не обладает поведением Эллипса (мы намеренно используем для имен классов заглавные буквы, чтобы не путать их с объектами). Например, если для эллипса можно изменить значение aspectRatio (вызвать метод setAspectRatio(новое значение)), то для окружности такая операция не имеет смысла — в классе Окружность отсутствуют соответствующие поле и метод.

Аналогично и для эллипса, и для окружности имеет смысл операция установки нового размера setSize (новое значение), а для точки она не имеет смысла или запрещена. И даже если построить неправильную иерархию Ellipse-Circle-Dot и унаследовать от Ellipse эти методы в Circle и Dot, то возникнет проблема с их переопределением: если setAspectRatio будет менять отношение полуосей нашей окружности, она перестанет быть окружностью. А если не будет менять — метод окажется бессмысленным, вводящим в заблуждение. И не сможет обеспечить соответствующую абстракцию поведения.

Аналогично, если setSize изменит размер точки, та перестанет быть точкой. Если же сделать эти методы ничего не делающими заглушками, то экземпляры таких потомков не смогут обладать поведением прародителя.

Например, мы не сможем вписать окружность в прямоугольник, установив нужное значение aspectRatio — найдутся только две точки, общие для окружности и сторон прямоугольника (а не четыре, как для объекта типа Ellipse). То есть объект типа Circle на уровне абстракции поведения во многих случаях не сможет обладать всеми особенностями поведения объекта типа Ellipse. А значит, Circle не может быть потомком Ellipse.

Можно привести нескончаемое число примеров того, какие ситуации окажутся нереализуемыми для объектов таких неправильных иерархий. А отдельные хитрости, позволяющие выпутываться из некоторых подобных ситуаций, обычно бывают крайне ис-

кусственными, не позволяющими решить очередную внезапно возникшую проблему, и только усложняют программу.

Сформулируем *критерий применимости* наследования одного класса от другого более корректно.

Критерий применимости наследования

> Если имеются классы `A1` и `A2` и можно считать, что `A2` является модифицированным (усложненным или измененным) вариантом `A1` с сохранением всех особенностей поведения `A1`, то `A2` должен описываться как потомок `A1`. На уровне абстракции, описывающей поведение, объект типа `A2` должен вести себя, как объект типа `A1` *при любых значениях полей данных*.

Специализированный класс, вообще говоря, должен быть устроен *более сложно* (расширенно — extended) по сравнению с прародительским. У него должны иметься дополнительные поля данных и/или дополнительные методы. С этой точки зрения очевидно, что класс `Окружность` более специализированный, чем `Точка`, а `Эллипс` более специализированный, чем `Окружность`.

Иногда встречаются ситуации, когда потомок отличается от прародителя только своим поведением. У него не добавляется новых полей или методов, а только переопределяется часть методов (возможно, только один). Отметим, что поля или методы, имеющиеся в прародителе, не могут отсутствовать в наследнике — они наследуются из прародителя, даже если доступ к ним в классе-наследнике закрыт (так бывает в случае, когда поле или метод объявлены с модификатором видимости `private` (закрытый, частный)).

Когда про класс-потомок можно сказать, что он является специализированной разновидностью класса-прародителя (`B` есть `A`), то все очевидно. Но в объектном программировании иногда приходится использовать отношение "Класс `B` *похож* на класс `A` — имеет те же поля данных (возможно и дополнительные), но обладает несколько иным поведением". Рассмотрим подобную неоднозначность и способы решения данной проблемы на примере иерархии фигур.

Любой наш объект мы можем назвать фигурой. Поэтому то, что базовый класс нашей иерархии называется `Figure`, естественно и однозначно. И однозначно то, что все классы нашей иерархии должны быть его наследниками. А вот остальные элементы иерархии можно было бы устроить совсем по-другому. Например, как показано на рис. 6.2.

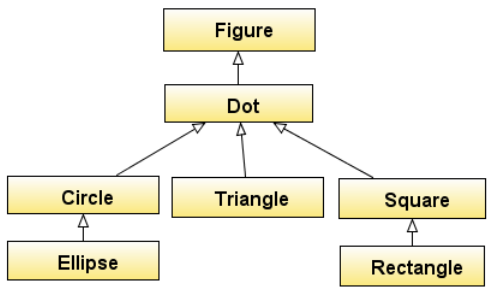


Рис. 6.2. Альтернативный вариант иерархии фигур

Возможно и такое решение: все указанные классы сделать наследниками `Figure` и расположить на одном уровне наследования (рис. 6.3).

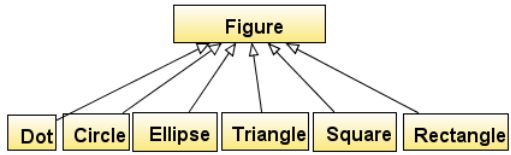


Рис. 6.3. Еще один вариант иерархии фигур

Возможны и другие варианты, не менее логичные. Какой вариант выбрать?

Уже на этом простейшем примере мы убеждаемся, что проектирование иерархии — весьма многовариантная задача. Требуется большой опыт, чтобы грамотно построить иерархию, в противном случае при написании кода классов не удается в полной мере обеспечить их функциональность, а код классов становится неуправляемым — внесение исправления в одном месте приводит к возникновению ошибок совсем в других местах (причем ошибок возникает больше, чем исправляется).

Один из важных принципов при построении таких иерархий — *соответствие представлений из предметной области строящейся иерархии*. В примере, приведенном на рис. 6.1, мы имеем вполне логичную с точки зрения идеологии наследования иерархию. С точки зрения общности/специализации такая иерархия безупречна. По этой причине она хорошо подходит для написания учебных программ, демонстрирующих работу с классами-наследниками, удобна для иллюстрации правил совместимости объектных типов, а также при написании полиморфного кода.

Но в геометрии, из которой мы знаем о свойствах этих фигур, считается, что окружность является частным случаем эллипса, а точка — частным случаем окружности (а значит, и эллипса). Так как значения полей данных объекта задают его состояние, то в некоторых случаях объекты, относящиеся к `Эллипсам` по типу (внутреннему устройству), окажутся в состоянии, когда с точки зрения предметной области они будут окружностями, хотя по внутреннему устройству и будут отличаться от объектов `Окружностей`.

Приведенная иерархия может вызывать внутренний протест у многих людей, особенно учитывая сложность разграничения классов и объектов в обычной речи при не очень строгих рассуждениях. Поэтому такое решение может привести к логическим ошибкам в рассуждениях. Вот почему последний из предложенных вариантов иерархий, когда все классы наследуются непосредственно от `Figure`, во многих случаях предпочтительнее, тем более что никакого выигрыша при создании программного кода увеличение числа поколений наследования не дает — код, написанный для класса `Dot`, вряд ли будет необходим для объектов классов `Circle` и `Ellipse`. А ведь наследование нужно не само по себе — это инструмент для написания более экономного и прозрачного для понимания полиморфного кода. Более того, увеличение числа поколений приводит к снижению надежности кода. Так что этим не следует злоупотреблять.

На выбор варианта иерархии заметно влияют соображения повторного использования кода. Если бы класс `Ellipse` активно работал с частью кода, написанного для класса `Circle`, а тот, в свою очередь, активно пользовался бы кодом класса `Dot`, то выбор первого варианта мог стать предпочтительным по сравнению с третьим, даже несмотря на некоторый конфликт с обыденными (не принципиальными!) представлениями предметной области.

Правда, иерархия `Dot – Circle – Ellipse` и в этом случае была бы устроена не совсем правильно. Дело в том, что наследование от классов, у которых бывают экземпляры *(неабстрактных)*, нежелательно.

Классы, у которых нет экземпляров, называются *абстрактными*. В этих классах имеются и поля данных, и методы. Но некоторые методы (по крайней мере один) не имеют реализации — помечены как абстрактные (имеют модификатор abstract). В абстрактных классах все методы обеспечивают абстракцию поведения экземпляров классов-наследников. Таким образом, абстрактные классы предназначены исключительно для порождения от них наследников. При модификации кода какого-либо метода абстрактного класса это изменение будет осуществляться в соответствии с абстракцией поведения данного класса — а значит, и в соответствии с абстракцией поведения любого из наследников. Ведь по определению наследования у них должна быть одна и та же абстракция поведения!

С неабстрактными классами дело обстоит совсем иначе. Реализация методов в неабстрактных классах зависит от особенностей устройства данного класса и конкретной реализации его методов. Данный класс (назовем его `Класс1`), конечно, реализует некоторую абстракцию поведения, но кроме данной абстракции в реализации его методов присутствуют элементы конкретного варианта внутреннего устройства класса. Предположим, мы создали класс-наследник (назовем его `Класс2`), и все его методы работают правильно. Вроде бы, все хорошо. Но практически никогда не бывает, чтобы удалось создать класс, не нуждающийся в изменении. Так что реализация нашего родительского класса может измениться. Мы будем рассматривать только случай, когда абстракция поведения класса сохраняется, а изменили только особенность внутреннего устройства класса или вариант реализации каких-либо его методов. При этом оказывается, что нет никакой гарантии правильной работы наследников данного класса, хотя их код не менялся! Ведь они наследуют от прародителя код, который опирается не на абстракцию поведения, а на конкретную реализацию класса — теперь уже претерпевшую изменения.

Правильное построение иерархии должно быть таким: вместо наследования `Класса2` от `Класса1` следует ввести класс `АбстрактныйКласс1`, обеспечивающий абстракцию поведения `Класса1`. Значит, он должен быть устроен таким образом, чтобы не было зависимости от конкретной реализации этого поведения. Затем следует унаследовать `Класс1` и `Класс2` от данного абстрактного класса. Если предполагается, что у `Класса2` должны быть наследники, необходимо аналогичным образом ввести `АбстрактныйКласс2`, обеспечивающий абстракцию поведения `Класса2`. И наследовать `АбстрактныйКласс2` от `АбстрактногоКласса1`, а `Класс2` от `АбстрактногоКласса2`. Иными словами, строить всю цепочку наследования через абстрактные классы, и только на самом последнем уровне наследования иметь неабстрактные классы.

Попробуем построить иерархию на основе изложенных идей. Мы пришли к выводу, что фигуры могут быть масштабируемые (без изменения формы, оставаясь подобными), а также растягиваемые. Поэтому можно ввести классы `ScalableFigure` (масштабируемая фигура) и `StretchableFigure` (растягиваемая фигура). Точка `Dot` не является ни масштабируемой, ни растягиваемой. Очевидно, что любая растягиваемая фигура должна быть масштабируемой. Окружность `Circle` и квадрат `Square` масштабируемы, но не растягиваемы. А прямоугольник `Rectangle`, эллипс `Ellipse` и треугольник `Triangle` как

масштабируемы, так и растягиваемы. Поэтому нашу иерархию можно представить на рис. 6.4.

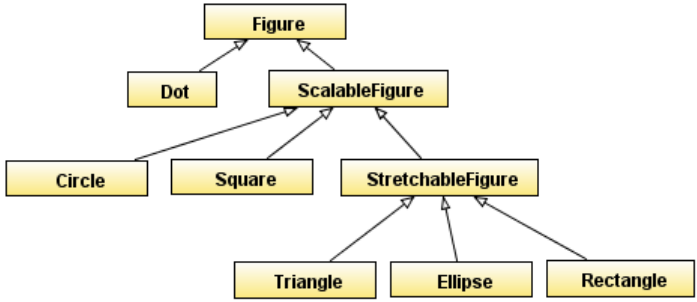


Рис. 6.4. Итоговый вариант иерархии фигур

Основное ее преимущество по сравнению с предыдущими — возможность писать полиморфный код для наиболее общих разновидностей фигур. Введение промежуточных уровней наследования, отвечающих соответствующим абстракциям, — характерная черта объектного программирования. При этом классы `Figure`, `ScalableFigure` и `StretchableFigure` будут абстрактными — экземпляры такого типа создавать не предполагается, поскольку не бывает фигуры (масштабируемой или растягиваемой) в общем виде, без указания ее конкретной формы. Точно так же методы `show` и `hide` для этих классов тоже будут абстрактными.

Продумывание того, как устроены классы, т. е. какие в них должны быть поля и методы (без уточнения конкретной реализации этих методов), а также описание того, какая должна быть иерархия наследования, называется *проектированием*. Это сложный процесс, который гораздо важнее написания конкретных операторов в реализации *(кодирования)*.

Автор предлагает пять основных критериев правильности построения иерархии:

1. В процессе наследования должно идти расширение (усложнение, специализация, конкретизация) классов, а не наоборот.
2. Наследование должно идти только от абстрактных классов (или интерфейсов как варианта полностью абстрактных классов — *см. главу 9*).
3. Названия всех методов должны давать точное представление о том, что делает метод. Причем эти имена должны восприниматься как команды (установить что-то, прочитать что-то, показать что-то и т. д.). Заметим, что названия методов при построении иерархии очень важны, так как именно они отражают абстракцию поведения объектов.
4. Недопустимы названия методов, содержащие связку "и" ("and"). Например, `readAndShowSpeed`, `calculateIntegralAndWriteToFile` и т. п. Такого рода гибриды следует разделять на два и более независимых метода — `readSpeed` и `ShowSpeed`; `calculateIntegral` и `writeToFile`, и т. д.
5. Названия всех классов должны давать четкое представление о соответствующих абстракциях поведения, в том числе для неабстрактных классов.

Последнее требование кажется слишком расплывчатым, поэтому можно рекомендовать следующий простой способ его проверки — завести объектную переменную с соответствующим классу именем. Например, для класса `Figure` нужно задать переменную `Figure figure`, для класса `Dot` — переменную `Dot dot`, для класса `ScalableFigure` — переменную `scalableFigure` и т. д. В этом случае вызов метода `figure.show()` будет означать "показать фигуру", и эта расшифровка смысла вызова должна быть правильной по логике независимо от того, на объект какого типа ссылается переменная `figure`. Точно так же вызов `dot.show()` будет означать "показать точку", вызов `scalableFigure.show()`— "показать масштабируемую фигуру" и т. д. На первый взгляд кажется, что все слишком просто и очевидно, и по-другому быть не может. А теперь представьте, что прародительским классом иерархии фигур сделали класс `Dot`. Тогда вызов `dot.show()` будет означать "показать точку", даже если переменная `dot` ссылается на совершенно другую фигуру! Или в неправильной иерархии `Figure – Dot – Circle – Ellipse` для переменной `circle`, ссылающейся на эллипс, вызов `circle.show()` будет означать "показать окружность", хотя будет отображаться эллипс. Налицо ошибка в логике поведения, хотя с точки зрения синтаксиса никаких ошибок не будет.

Иногда подобного рода ошибки пытаются исправить с помощью введения промежуточных абстрактных классов. И, например, неправильную иерархию `Figure – Dot – Circle – Ellipse`, нарушающую второй критерий, преобразуют в иерархию `Figure – AbstractDot – AbstractCircle – AbstractEllipse`, соответствующую этому критерию. Причем наследуют `Dot` от `AbstractDot`, `Circle` от `AbstractCircle` и `Ellipse` от `AbstractEllipse`. На первый взгляд, в такой иерархии не к чему придраться. Но попробуем ввести переменную `abstractDot`, ссылающуюся на окружность или эллипс, или `abstractCircle`, ссылающуюся на эллипс — и сразу станет понятно, что иерархия неправильная. При вызове любого метода станет видно несоответствие имени переменной типу объекта, на который она ссылается.

В данном примере может возникнуть вопрос о префиксе `abstract` в имени переменных. Сам по себе префикс `Abstract` в имени абстрактного класса выглядит достаточно логично, но в имени переменной совершенно неуместен. Например, мы могли бы назвать наш базовый класс `AbstractFigure`, и это вполне правильное название. Но вот вызов `abstractFigure.show()` по своей сути абсурден. Абстрактная фигура не может быть показана! В то же время ведь весь полиморфный код основан на использовании переменных абстрактного типа. Поэтому в рекомендацию по именам переменных, соответствующих абстрактным классам, следует внести коррективы: в имени переменной следует убирать префикс `abstract`. Поэтому для класса `AbstractFigure` следует вводить переменную с именем `figure`, для класса `AbstractName` — переменную с именем `name` и т. д.

Конечно, приведенных критериев недостаточно для того, чтобы в полной мере гарантировать правильность вашей иерархии. Но если у вас имеется иерархия, не удовлетворяющая этим критериям, сразу встает вопрос о необходимости ее доработки или отказа от нее.

6.2. Функции. Модификаторы. Передача примитивных типов в функции

В построении иерархии особую роль играет задание методов, поскольку именно они задают как абстракцию поведения, так и реальное поведение объектов. В Java методы всегда являются функциями.

Формат объявления в классе метода (т. е. функции) следующий:

```
Модификаторы Тип Имя(список параметров){
   Тело функции
}
```

Это формат простой функции, не содержащей операторов, возбуждающих исключительные ситуации. Про исключительные ситуации и формат объявления функции, которая может порождать такие ситуации, речь пойдет немного позже.

Комбинация элементов декларации метода:

```
Модификаторы Тип Имя(список параметров)
```

называется *заголовком* метода.

Модификаторы — это зарезервированные слова, задающие:

- правила доступа к методу (`private`, `protected`, `public`). Если модификатор не задан, действует доступ по умолчанию (так называемый пакетный);
- принадлежность к методам класса (`static`). Если модификатор отсутствует, считается, что это метод объекта;
- невозможность переопределения метода в потомках (`final`). Если модификатор не указан, считается, что этот метод можно переопределять в классах-потомках;
- способ реализации метода (`native` — метод, заданный во внешней библиотеке DLL, написанной на языке C или C++; `abstract` — абстрактный, не имеющий реализации; `strictfp` — метод, работающий в режиме полной совместимости операций с плавающей точкой с первоначальной версией Java). Если модификатор не задан, считается, что это обычный метод;
- синхронизацию при работе с потоками (`synchronized`) — *см. главу 10*.
- особенности работы с переменными и полями данных (`transient` — значение поля данных, помеченное этим модификатором, не нужно сохранять, если объект помещается в постоянное хранилище (например, на диск) — т. е. при *сериализации* объекта; `volatile` — указывает компилятору, что переменную или поле данных можно неявно изменить из другого потока выполнения *(см. главу 10)*, и поэтому для нее не разрешены некоторые оптимизации, например, заменяющие значение этой переменной константой в случае, когда явный код создает видимость неизменности значения переменной);
- особенности реализации класса (`abstract` — абстрактный класс, т. е. имеющий по крайней мере один абстрактный метод; `strictfp` — класс, в котором все методы работают в режиме полной совместимости операций с плавающей точкой с первоначальной версией Java).

В качестве *типа* следует указать тип результата, возвращаемого методом. В Java, как мы уже знаем, все методы являются функциями, возвращающими значение какого-либо типа. Если требуется метод, не возвращающий никакого значения (т. е. процедура), то он объявляется с типом `void`. Возврат значения осуществляется в теле функции с помощью зарезервированного слова `return`.

Для выхода без возврата значения требуется написать:

```
return;
```

Выход из метода с возвратом значения обеспечивает запись:

```
return выражение;
```

Выражение будет вычислено, после чего полученное значение возвратится как результат работы функции.

Оператор `return` осуществляет прерывание выполнения подпрограммы, поэтому он обычно располагается в ветвях операторов `if...else` или `switch...case` в случаях, когда необходимо возвращать тот или иной результат в зависимости от различных условий. Если в подпрограмме-функции в какой-либо из ветвей опустить оператор `return`, то будет выдана ошибка компиляции с диагностикой `missing return statement` (ошибочное высказывание с `return`).

Список параметров — это объявление через запятую переменных, с помощью которых можно принимать значения и объекты в данную подпрограмму из внешнего мира, а также передавать значения и объекты из подпрограммы во внешний мир.

Объявление параметров имеет вид:

тип1 имя1, тип2 имя2, ..., типN имяN

Если список параметров пуст, то пишут круглые скобки без параметров.

Тело функции представляет собой последовательность операторов, реализующую необходимый алгоритм. Эта последовательность может быть пустой, в этом случае говорят о *заглушке* (т. е. заготовке метода, имеющей только имя и список параметров, но с отсутствующей реализацией). Иногда в такой заглушке вместо правильной реализации временно пишут операторы служебного вывода в консольное окно или в файл.

В теле функции в произвольном месте могут задаваться переменные. Они доступны только внутри данной подпрограммы и поэтому называются *локальными*. Переменные, заданные на уровне класса (поля данных класса или объекта), называются *глобальными*.

Данные (значения или объекты) можно передавать в подпрограмму либо через список параметров, либо через глобальные переменные.

Сначала рассмотрим передачу в подпрограмму через список параметров значений примитивного типа. Предположим, что мы написали в классе `MyMath` метод `mult1` умножения двух чисел, каждое из которых перед этим увеличивается на единицу (листинг 6.1).

Листинг 6.1. Задание метода с передачей значений через список параметров

```
double mult1(double x, double y){
 x++;
 y++;
 return x*y;
}
```

Листинг 6.2 иллюстрирует пример вызова данного метода.

Листинг 6.2. Вызов метода с передачей значений через список параметров

```
double a,b,c;
...
MyMath obj1=new MyMath();//создали объект типа MyMath
...
c=obj1.mult1(a+0.5,b);
```

Параметры, указанные в заголовке функции при ее декларации, называются *формальными*, а подставляемые во время вызова функции — *фактическими*. Формальные параметры нужны для того, чтобы указать последовательность действий с фактическими параметрами после их передачи в подпрограмму во время вызова. Это не что иное, как особый вид локальных переменных, которые служат для обмена данными с внешним миром.

В нашем случае `x` и `y` — формальные параметры функции `mult1`, а выражения `a+0.5` и `b` — фактические параметры этой функции. При вызове сначала вычисляются выражения, переданные в качестве фактических параметров, после чего получившийся результат копируется в локальные переменные, используемые в качестве формальных параметров (т. е. в локальную переменную `x` будет скопировано значение, получившееся в результате вычисления `a+0.5`, а в локальную переменную `y` — значение, хранящееся в переменной `b`). После этого с локальными переменными происходят все те действия, которые указаны в реализации метода.

Соответствие фактических и формальных параметров идет в порядке перечисления (т. е. первый фактический параметр соответствует первому формальному, второй фактический — второму формальному и т. д.). Фактические параметры должны быть совместимы с формальными, при этом действуют все правила, относящиеся к совместимости примитивных типов по присваиванию, в том числе и к автоматическому преобразованию типов. Например, для `mult1` вместо параметров типа `double` в качестве фактических возможны значения типа `int` или `float`. А если бы формальные параметры имели тип `float`, то фактические параметры типа `int` были бы допустимы, а типа `double` — нет.

Влияет ли как-нибудь увеличение переменной `y` на единицу, происходящее благодаря оператору `y++`, на значение, хранящееся в переменной `b`? Конечно, нет! Ведь действия происходят с локальной переменной `y`, в которую при начале вызова было скопировано значение из переменной `b`. С самой переменной `b` в результате вызова ничего не происходит.

А можно ли сделать так, чтобы подпрограмма изменяла значение внутри передаваемой в нее переменной? Нет, нельзя! В Java значения примитивного типа наружу, к сожалению, передавать нельзя (в отличие от подавляющего большинства других языков программирования). Применяемый в Java способ передачи параметров называется передачей *по значению*.

Иногда необходимо передать в подпрограмму неизменяемую константу, т. е. гарантировать, что значение, переданное в качестве параметра, не будет изменено. Конечно, можно проверить, нет ли где-нибудь оператора, изменяющего соответствующую переменную. Но надежней будет проверка на уровне синтаксических конструкций. В этих целях используют модификатор `final`. Предположим, что увеличивать на единицу нужно только первый параметр, а второй должен оставаться неизменным (листинг 6.3).

Листинг 6.3. Параметр, передаваемый с модификатором `final`

```
double mult1(double x, final double y){
 x++;
 return x*y;
}
```

А вот при компиляции кода листинга 6.4 будет выдано сообщение об ошибке.

Листинг 6.4. Параметр, передаваемый с модификатором `final` — ошибка в коде

```
double mult1(double x, final double y){
 x++;
 y++;
 return x*y;
}
```

6.3. Локальные и глобальные переменные. Модификаторы доступа и правила видимости. Ссылка *this*

Как уже говорилось, данные в подпрограмму можно передавать через глобальные переменные. Это могут быть поля данных объекта, в методе которого осуществляется вызов, поля данных соответствующего класса, поля данных другого объекта или класса. Глобальные переменные не рекомендуется применять по двум причинам:

- при вызове в списке параметров не видно, что идет обращение к соответствующим переменным, и программа становится непрозрачной для программиста, что делает ее неструктурной;
- при изменении внутри подпрограммы-функции глобальной переменной возникает *побочный эффект*, связанный с тем, что функция не только возвращает вычисленное значение, но и меняет состояние окружения незаметным для программиста образом. Это может быть причиной плохо обнаруживаемых *логических* ошибок, не отслеживаемых компилятором.

Конечно, бывают случаи, когда глобальные переменные не только желательны, а просто необходимы, иначе их не стали бы вводить как конструкцию языков программирования! Например, при написании метода в каком-либо классе обычно требуется доступ к полям и методам этого класса. В Java такой доступ осуществляется напрямую, без указания имени объекта или класса.

Правила доступа (иначе их называют правилами видимости) к методам и полям данных (переменным) из других пакетов, классов и объектов задаются с помощью модификаторов `private`, `protected`, `public`. Если переменная или подпрограмма невидимы в некой области программы, то доступ к ним запрещен.

Модификатором `private` отмечают элементы (поля данных или методы), которые должны быть доступны только в методах данного класса. Доступа из объектов к private-элементам нет (т. е. если мы создали объект, у которого имеется поле или метод `private`, то получить доступ к этому полю или методу из объекта нельзя).

Если модификатор не задан, то действует доступ по умолчанию (так называемый пакетный, когда соответствующий элемент доступен только из классов своего пакета). Иногда (по аналогии с C++) этот тип доступа называют дружественным.

Модификатором `protected` отмечают элементы, которые должны быть доступны только в методах данного класса, данного пакета, а также классах-наследниках (они могут располагаться в других пакетах).

Модификатором `public` отмечают элементы, которые должны быть доступны из любых классов и объектов. Если соответствующий класс не импортирован, то для доступа к элементу требуется указывать полное имя, включающее имя пакета.

Пример на задание элементов с различными модификаторами видимости иллюстрирует листинг 6.5.

Листинг 6.5. Элементы с различными модификаторами видимости

```
class Vis1 {
    private int x=10,y=10;
    int p1=1;
    protected int p2=1;
    public int p3=1;
}
```

В классе, приведенном в листинге 6.5, заданы переменные `x`, `y`, `p1`, `p2`, `p3`, причем `x` и `y` обладают уровнем доступа `private`, `p1` — пакетным, `p2` — `protected`, `p3` — `public`. Перечисление однотипных переменных через запятую позволяет однократно, без повторений задавать для нескольких переменных имя типа и модификатор.

Как уже говорилось, локальные переменные можно вводить в любом месте подпрограммы. Их можно использовать в данном методе только после того места, где они заданы. Область существования и видимости локальной переменной — это часть программного кода от места объявления переменной до окончания блока, в котором она объявлена (обычно до окончания метода).

А вот переменные, заданные на уровне класса (глобальные переменные), формируются при создании объекта для методов объекта и при первом вызове класса для переменных класса. Их можно использовать в методах данного класса как глобальные, независимо от того, заданы они до метода или после.

Еще одна важная особенность локальных переменных — время их существования: память под них выделяется в момент вызова, а высвобождается сразу после окончания вызова.

Рассмотрим функцию (листинг 6.6), вычисляющую сумму чисел от 1 до `n`.

Листинг 6.6. Локальные переменные `i` и `r` в методе `sum1`

```
double sum1(int n){
 int i;
 double r=0;
 for(i=1;i<=n;i++){
  r+=i;
 }
 return r;
}
```

Вызов данного метода может выглядеть так:

```
c=obj1.sum1(1000);
```

При этом переменные `i` и `r` существуют только во время вызова `obj1.sum1(1000)`. При повторном аналогичном вызове будет создан, а затем высвобожден из памяти следующий комплект `i` и `r`.

Все сказанное про локальные переменные также относится и к объектным переменным. Но не следует путать переменные и объекты — время жизни объектов гораздо больше, даже если объект создается во время вызова подпрограммы, а после окончания этого вызова связь с ним прекращается. Уничтожением неиспользуемых объектов занимается сборщик мусора (garbage collector). Если же объект создан в подпрограмме и ссылка на него передана какой-либо глобальной переменной, то он будет существовать после выхода из подпрограммы столько времени, сколько необходимо для работы с ним.

Остановимся на области видимости локальной переменной. Имеются следующие уровни видимости:

- на уровне метода. Переменная видна от места декларации до конца метода;
- на уровне блока. Если переменная задана внутри блока `{...}`, то она видна от места декларации до конца блока. Блоки могут быть вложены один в другой с произвольным уровнем вложенности;
- на уровне цикла `for`. Переменная видна от места декларации в секции инициализации до конца тела цикла.

Глобальные переменные видны во всей подпрограмме.

Каждый объект имеет поле данных с именем `this` (этот) (не слишком удачное название унаследовано из C++), в котором хранится ссылка на сам объект. Поэтому доступ в методе объекта к полям и методам этого объекта может осуществляться либо напрямую, либо через ссылку `this` на этот объект.

Например, если у объекта имеется поле `x` и метод `show()`, то:

`this.x` означает то же самое, что `x`,

а `this.show()` — то же самое, что `show()`.

Но при перекрытии области видимости (о чем речь пойдет чуть позже) доступ по короткому имени оказывается невозможен, и приходится задействовать ссылку `this`. Отметим, что ссылка `this` позволяет обойтись без указания имени объектной переменной, что делает код с такой ссылкой более универсальным (например, в методах того класса, экземпляром которого является объект).

Ссылка `this` не может быть использована в методах класса (т. е. заданных с модификатором `static`), поскольку они могут вызываться без существующего объекта.

Возникает вопрос о том, что произойдет, если на разных уровнях будут заданы две переменные с одним именем. Имеются следующие варианты ситуаций:

- в классе имеется поле с некоторым именем (глобальная переменная) и в списке параметров задается локальная переменная с тем же именем. Такая проблема часто возникает в конструкторах при инициализации полей данных и в методах установки значений полей данных `setИмяПоля`. Это разрешено, и доступ к параметру идет по имени, как обычно. Но при этом видимость поля данных (доступ к полю данных по его имени) перекрывается и требуется ссылка на объект `this`. Например, если имя

поля данных x и имя параметра в методе тоже x, то установка значения поля выглядит так:

```
void setX(double x){
  this.x=x
}
```

- в классе есть поле с некоторым именем (глобальная переменная) и в методе задается локальная переменная с тем же именем. Ситуация разрешена и аналогична заданию локальной переменной в списке параметров. Доступ к полю идет через ссылку this;
- в классе имеется поле с некоторым именем (глобальная переменная) и в секции инициализации цикла for или внутри какого-нибудь блока, ограниченного фигурными скобками {...}, задается локальная переменная с тем же именем. В Java такая ситуация разрешена. При этом внутри цикла или блока доступна заданная в нем локальная переменная, а глобальная переменная видна через ссылку this;
- существует локальная переменная (возможно, заданная как элемент списка параметров) и в секции инициализации цикла for или внутри какого-нибудь блока, ограниченного фигурными скобками {...}, задается локальная переменная с тем же именем. В Java такая ситуация запрещена. При этом выдается ошибка компиляции с информацией, что переменная с таким именем уже задана (is already defined);
- есть метод с неким именем, и в другом методе данного класса задается локальная переменная с тем же именем. В Java такая ситуация разрешена и не вызывает проблем, т. к. компилятор отличает вызов метода от обращения к полю данных по наличию после имени метода круглых скобок.

6.4. Передача ссылочных типов в функции. Проблема изменения ссылки внутри подпрограммы

При передаче в подпрограмму ссылочной переменной имеется ряд отличий по сравнению со случаем примитивных типов, т. к. в локальную переменную, с которой идет работа в подпрограмме, копируется не сам объект, а его адрес. Поэтому глобальная переменная, ссылающаяся на этот объект, будет получать доступ к тем же самым полям данных, что и локальная. В результате изменение полей данных объекта внутри метода приведет к тому, что мы увидим эти изменения после выхода из метода (причем неважно, будем ли менять поля непосредственно или с помощью вызова каких-либо методов).

Для примера создадим в нашем пакете класс Location (листинг 6.7). Он послужит для задания объекта соответствующего типа, который будет передаваться через список параметров в метод ml, вызываемый из нашего приложения. А в классе приложения напишем код, приведенный в листинге 6.8.

Листинг 6.7. Код класса Location

```
public class Location {
     public int x=0,y=0;
```

```
    public Location (int x, int y) {
      this.x=x;
      this.y=y;
    }
}
```

Листинг 6.8. Фрагмент приложения с созданием и вызовом объекта типа `Location`

```
Location locat1=new Location(10,20);

 public static void m1(Location obj){
     obj.x++;
     obj.y++;
 }
```

Мы задали переменную `locat1` типа `Location`, инициализировав ее поля `x` и `y` значениями 10 и 20. А в методе `m1` происходит увеличение на единицу значения полей `x` и `y` объекта, связанного с формальным параметром `obj`.

Создадим две кнопки с обработчиками событий. Нажатие на первую кнопку будет приводить к выводу информации о значениях полей `x` и `y` объекта, связанного с переменной `locat1`, а нажатие на вторую — к вызову метода `m1` (листинг 6.9).

Листинг 6.9. Вывод информации о полях и вызов метода в обработчиках событий

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    System.out.println("locat1.x="+locat1.x);
    System.out.println("locat1.y="+locat1.y);
}
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    m1(locat1);
    System.out.println("Прошел вызов m1(locat1)";
}
```

Легко проверить, что вызов `m1(locat1)` приводит к увеличению значений полей `locat1.x` и `locat1.y`.

При передаче в подпрограмму ссылочной переменной есть особенность, которая часто приводит к ошибкам — потеря связи с первоначальным объектом при изменении ссылки. Модифицируем наш метод `m1` (листинг 6.10).

Листинг 6.10. Проблема изменения ссылки на объект внутри метода

```
 public static void m1(Location obj){
     obj.x++;
     obj.y++;
     obj=new Location(4,4);
     obj.x++;
     obj.y++;
 }
```

После первых двух строк, которые приводили к инкременту (увеличению) значений полей передаваемого объекта, появилось создание нового объекта и перещелкивание на него локальной переменной obj, а затем две точно такие же строчки, как в начале метода. Какие значения полей x и y объекта, связанного с переменной locat1, покажет нажатие на первую кнопку после вызова модифицированного варианта метода? Первоначальный и модифицированный вариант метода дадут одинаковые результаты!

Присваивание obj=new Location(4,4); приводит к тому, что переменная obj становится связанной с новым только что созданным объектом. И изменение полей данных в операторах obj.x++ и obj.y++ происходит уже для этого объекта, а вовсе не для того объекта, ссылку на который передали через список параметров.

Следует обратить внимание на то, какая терминология используется для описания программы. Понятия "ссылочная переменная" и "объект, связанный со ссылочной переменной" не отождествляются, как это часто делают программисты при описании программы. И именно строгая терминология позволяет разобраться в происходящем, иначе трудно понять, почему оператор obj.x++ в одном месте метода дает совсем не тот эффект, что в другом месте. Если бы мы сказали "изменение поля x объекта obj", то было бы невозможно понять, что объекты-то разные! А правильная фраза "изменение поля x объекта, связанного со ссылочной переменной obj", подталкивает к мысли, что эти объекты в различных местах программы могут быть разными.

Способ передачи данных (ячейки памяти) в подпрограмму, позволяющий изменять содержимое внешней ячейки памяти с помощью использования ссылки на эту ячейку, называется передачей *по ссылке*. И хотя в Java объект передается по ссылке, объектная переменная, в которой хранится адрес объекта, передается по значению. Ведь этот адрес копируется в другую ячейку, локальную переменную, и именно переменная является параметром, а не связанный с ней объект (т. е. *параметры в Java всегда передаются по значению*). Передачи параметров по ссылке в языке Java нет.

Упомянем теперь нетривиальные ситуации, которые часто возникают при передаче ссылочных переменных в качестве параметров.

Мы уже упоминали о проблемах, возникающих при работе со строками. Рассмотрим подпрограмму, которая по идее должна бы возвращать с помощью переменной s3 сумму строк, хранящихся в переменных s1 и s2:

```
void strAdd1(String s1,s2,s3){
 s3=s1+s2;
}
```

Строки в Java — это объекты, а строковые переменные ссылочные. Поэтому можно было бы рассчитывать на возврат измененного состояния строкового объекта, с которым связана переменная s3. Но это неверное предположение.

При вызове

```
obj1.strAdd1(t1,t2,t3);
```

значение строковой переменной t3 *не изменится*.

Дело в том, что в Java строки типа String являются неизменяемыми объектами, и вместо изменения состояния прежнего объекта в результате вычисления выражения s1+s2 создается новый объект. Поэтому присваивание s3=s1+s2 приводит к перещелкиванию ссылки s3 на новый объект. А мы уже знаем, что это ведет к тому, что новый объект

оказывается недоступен вне подпрограммы — внешняя переменная `t3` будет ссылаться на прежний объект-строку. В данном случае, конечно, лучше сделать функцию `strAdd1` строковой, и возвращать получившийся строковый объект как результат вычисления этой функции.

Еще пример: пусть нам необходимо внутри подпрограммы обработать некоторую строку и вернуть измененное значение. Допустим, в качестве входного параметра передается имя и мы хотим добавить в конец этого имени порядковый номер (примерно так, как это делает среда разработки при создании нового компонента). Следует отметить, что для этих целей имеет смысл создавать подпрограмму, хотя на первый взгляд достаточно выражения `name+count`. Ведь на следующем этапе мы, возможно, захотим проверить, является ли входное значение идентификатором (начинающимся с буквы и содержащим только буквы и цифры), либо захотим проверить, нет ли уже в списке такого имени.

Напишем в классе нашего приложения код, приведенный в листинге 6.11.

Листинг 6.11. Проблема при передаче в метод строки в качестве параметра

```
String componentName="myComponent";
int count=0;
public void calcName1(String name) {
  count++;
  name+=count;
  System.out.println("Новое значение="+name);
}
```

Создадим в нашем приложении кнопку, при нажатии на которую срабатывает следующий обработчик события:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
   calcName1(componentName);
   System.out.println("componentName="+componentName);
}
```

Многие начинающие программисты считают, что раз строки являются объектами, то при первом нажатии на кнопку значение `componentName` станет `"myComponent1"`, при втором — `"myComponent2"` и т. д. Но значение `myComponent` остается неизменным, хотя в методе `calcName1` новое значение выводится именно таким, каким должно быть. В чем причина такого поведения программы и как добиться правильного результата?

Если мы меняем в подпрограмме значения полей у объекта, а ссылка на объект не меняется, то изменение значения полей оказывается наблюдаемым с помощью доступа к тому же объекту через внешнюю переменную. А вот присваивание строковой переменной внутри подпрограммы нового значения приводит к созданию нового объекта-строки и перещелкивания на него ссылки, хранящейся в локальной переменной `name`. Причем глобальная переменная `componentName` остается связанной с первоначальным объектом-строкой `"myComponent"`.

Как бороться с данной проблемой? Существуют несколько вариантов решения.

Во-первых, в данном случае наиболее разумно вместо подпрограммы-процедуры, не возвращающей никакого значения, написать подпрограмму-функцию, возвращающую значение типа `String` (листинг 6.12).

Листинг 6.12. Возврат из метода измененной строки

```
    public String calcName2(String name) {
      count++;
      name+=count;
      return name;
    }
```

В этом случае не возникает никаких проблем с возвратом значения, и следующий обработчик нажатия на кнопку это демонстрирует:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
   componentName=calcName2(componentName);
   System.out.println("componentName="+componentName);
}
```

К сожалению, если необходимо возвращать более одного значения, то данный способ решения проблемы не подходит. А ведь часто из подпрограммы требуется возвращать два или более измененных или вычисленных значения.

Во-вторых, можно воспользоваться глобальной строковой переменной, но это плохой стиль программирования. Даже наличие глобальной переменной `count` в предыдущем примере не очень хорошо (но мы это сделали только лишь для того, чтобы не усложнять пример).

В-третьих, возможно создание оболочечного объекта (wrapper), у которого имеется поле строкового типа. Такой объект передается по ссылке в подпрограмму, и у него внутри подпрограммы меняется значение строкового поля. При этом, конечно, это поле будет ссылаться на новый объект-строку. Но поскольку ссылка на оболочечный объект внутри подпрограммы не меняется, то связь с новой строкой через оболочечный объект сохранится и снаружи. Такой подход, в отличие от использования подпрограммы-функции строкового типа, позволяет возвращать произвольное количество значений одновременно, причем любого типа, а не только строкового. Но у него тоже есть недостаток — требуется создавать специальные классы для формирования возвращаемых объектов.

В-четвертых, можно применить классы `StringBuffer` или `StringBuilder`. Это наиболее адекватный способ при необходимости возврата более чем одного значения, поскольку в этой ситуации он и самый простой, и весьма эффективный по быстродействию и потребляемым ресурсам. Листинг 6.13 содержит соответствующий код.

Листинг 6.13. Передача строки в метод с помощью объекта типа `StringBuffer`

```
  public void calcName3(StringBuffer name) {
      count++;
      name.append(count);
      System.out.println("Новое значение="+name);
  }
  StringBuffer sbComponentName=new StringBuffer();
  {
    sbComponentName.append("myComponent");
  }
```

```
private void jButton8ActionPerformed(java.awt.event.ActionEvent evt){
    calcName3(sbComponentName);
    System.out.println("sbComponentName="+sbComponentName);
}
```

Строковое поле `componentName` мы теперь заменяем полем `sbComponentName` типа `StringBuffer`. Почему-то разработчики этого класса не догадались сделать в нем конструктор с параметром строкового типа, поэтому приходится добавлять блок инициализации, где переменной `sbComponentName` присваивается нетривиальное начальное значение:

```
{
  sbComponentName.append("myComponent");
}
```

Более подробную информацию о блоках инициализации можно найти в *разд. 6.9*.

В остальном код очевиден. Принципиальное отличие от использования переменной типа `String` в том, что изменение значения строки, хранящейся в переменной `StringBuffer`, не приводит к созданию нового объекта, связанного с этой переменной.

Вообще говоря, с этой точки зрения для работы со строками переменные типа `StringBuffer` и `StringBuilder` подходят гораздо больше, чем типа `String`. Но метода `toStringBuffer()` в классах не предусмотрено. Поэтому при работе с переменными типа `StringBuffer` обычно приходится применять конструкции вида `sb.append(выражение)`.

В методы `append` и `insert` можно передавать выражения произвольных примитивных или объектных типов. Правда, массивы преобразуются в строку весьма своеобразно, так что их следует преобразовывать не методом `toString`, а через вызов `Arrays.toString(массив)` *(см. разд. 8.1)*.

Например, при выполнении фрагмента

```
int[] a=new int[]{10,11,12};
System.out.println("a="+a);
```

из-за того, что для массива `a` автоматически вызывался метод `a.toString()`, был получен следующий результат:

```
a=[I@15fea60
```

И выводимое значение не зависело ни от значений элементов массива, ни от их числа. Дело в том, что для массивов метод `toString` сначала выдает квадратную скобку, являющуюся признаком массива. Затем выдается в верхнем регистре первая буква базового типа для массива (в нашем случае `I` как сокращение от `int`), после чего идет символ `@`, за которым следует хэш-код объекта (нашего массива).

Хэш-кодом называется условно-уникальный числовой идентификатор, сопоставляемый какому-либо элементу. Из соображений безопасности выдавать адрес объекта прикладной программе нельзя. Поэтому в Java хэш-код заменяет адрес объекта в тех случаях, когда для каких-либо целей нужно хранить таблицы адресов объектов. Хэш-код остается уникальным только на время сеанса Java-машины, поэтому его и называют условно-уникальным.

Наличие автоматической упаковки/распаковки также приводит к проблемам.

Пусть у нас имеется случай, когда в списке параметров указана объектная переменная:

```
void m1(Double d){
 d++;
}
```

Несмотря на то, что переменная `d` объектная, изменение значения `d` внутри подпрограммы не приведет к изменению передаваемого через нее объекта снаружи подпрограммы по той же причине, что и для переменных типа `String`. При инкременте сначала производится распаковка в тип `double`, для которого выполняется оператор ++, после чего выполняется упаковка в новый объект типа `Double`, с которым становится связанной переменная `d`.

Приведем еще один аналогичный пример:

```
    public void proc1(Double d1,Double d2,Double d3){
        d3=d1+sin(d2);
    }
```

Надежда на то, что в объект, передаваемый через параметр `d3`, возвратится вычисленное значение `d3=d1+sin(d2)`, ошибочна, т. к. при упаковке вычисленного результата создается новый объект.

Таким образом, объекты стандартных оболочечных числовых классов не позволяют возвращать измененное числовое значение из подпрограмм, что во многих случаях вызывает проблемы. Для этих целей приходится писать собственные оболочечные классы (листинг 6.14).

Листинг 6.14. Пользовательский класс, оболочечный для типа `double`

```
public class UsableDouble{
   Double value=0;
   UsableDouble(Double value){
      this.value=value;
   }
}
```

Объект `UsableDouble d` можно передавать в подпрограмму по ссылке и без проблем получать возвращенное измененное значение. Аналогичного рода оболочечные классы легко написать для всех примитивных типов.

Если бы в стандартных оболочечных классах были методы, позволяющие изменить числовое значение, связанное с объектом, без изменения адреса объекта, то в такого рода деятельности не было бы необходимости.

Заканчивая рассуждения о проблемах передачи параметров в подпрограмму, автор хочет выразить надежду, что разработчики Java либо добавят в стандартные оболочечные классы такого рода методы, либо обеспечат возможность передачи переменных в подпрограммы по ссылке, как, к примеру, это было сделано в Java-образном языке C#.

6.5. Наследование. Суперклассы и подклассы. Переопределение методов

В объектном программировании принято использовать имеющиеся классы в качестве заготовок для создания новых классов, которые на них похожи, но обладают более сложной структурой и/или отличающимся поведением. Такие заготовки называются прародителями *(ancestors)*, а основанные на них новые классы — потомками *(descendants)* или наследниками. Классы-потомки получают в пользование поля и методы, заданные в классах-прародителях, это называется наследованием *(inheritance)* полей и методов.

В C++ и Java термины "прародители" и "потомки" чаще заменяют неудачными названиями *суперклассы (superclasses)* и *подклассы (subclasses)*. Как уже говорилось, суперклассы должны быть примитивнее подклассов, но приставка "супер" подталкивает программиста к прямо противоположным действиям.

При задании класса-потомка сначала идут модификаторы, затем после ключевого слова `class` идет имя декларируемого класса, потом идет зарезервированное слово `extends` (расширяет), после чего требуется указать имя класса-родителя (непосредственного прародителя). Если не указывается, от какого класса идет наследование, то родителем считается класс `Object`. Сам класс-потомок называется наследником, или *дочерним*.

В синтаксисе Java словом `extends` подчеркивается, что потомок расширяет то, что задано в прародителе — добавляет новые поля, методы, усложняет поведение (но все это делает класс более специализированным, менее общим).

Далее в фигурных скобках идет реализация класса — описание его полей и методов. При этом поля данных и методы, имеющиеся в прародителе, в потомке описывать не нужно — они наследуются. Однако в случае, если реализация прародительского метода нас не устраивает, в классе-потомке его можно реализовать по-другому. Тогда метод необходимо продекларировать и реализовать в классе-потомке. Кроме того, в потомке можно задавать новые поля данных и методы, отсутствующие в прародителях.

Перечислим возможные модификаторы:

- `public` — модификатор, задающий публичный (общедоступный) уровень видимости. Если он отсутствует, то действует пакетный уровень доступа — класс доступен только элементам того же пакета;
- `abstract` — модификатор, указывающий, что класс абстрактный, т. е. у него не бывает экземпляров (объектов). Обязательно объявлять класс абстрактным в случае, если какой-либо метод объявлен как абстрактный;
- `final` — модификатор, указывающий, что класс окончательный (final), т. е. что у него не может быть потомков.

Таким образом, задание класса-наследника имеет следующий формат:

```
Модификаторы class ИмяКласса extends ИмяРодителя {
 Задание полей;
 Задание подпрограмм – методов класса, методов объекта, конструкторов
}
```

Данный формат относится к классам, не реализующим интерфейсы (interfaces). Работе с интерфейсами будет посвящена *глава 9*.

Рассмотрим в качестве примера наследование для классов описанной ранее иерархии фигур. Для простоты выберем вариант, в котором Figure — это класс-прародитель иерархии, Dot его потомок, а Circle — потомок Dot (т. е. является жирной точкой). Напомним, что имена классов принято начинать с заглавной буквы.

Класс Figure опишем как абстрактный. Объектов такого типа создавать не предполагается, т. к. фигура без указания конкретного вида — это, действительно, чистая абстракция. По той же причине методы show (показать) и hide (скрыть) объявлены как абстрактные. Напомним также, что если в классе хоть один метод является абстрактным, то этот класс должен быть объявлен как абстрактный. Пример такого класса иллюстрирует листинг 6.15.

Листинг 6.15. Задание абстрактного класса Figure

```
public abstract class Figure {   //это абстрактный класс
    int x=0;
    int y=0;
    java.awt.Color color;
    java.awt.Graphics graphics;
    java.awt.Color bgColor;
    public abstract void show();  //это абстрактный метод
    public abstract void hide();  //это абстрактный метод

    public void moveTo(int x, int y){
        hide();
        this.x= x;
        this.y= y;
        show();
    }
}
```

Поля x и y задают координаты фигуры, а color — ее цвет. Соответствующий тип задан в пакете java.awt. Поле graphics задает ссылку на графическую поверхность, по которой будет идти отрисовка фигуры. Соответствующий тип также задан в пакете java.awt. В отличие от полей x, y и color, для этого поля при написании класса невозможно задать начальное значение, оно будет присвоено при создании объекта. То же относится к полю bgColor (от "background color") — в нем мы будем хранить ссылку на цвет фона графической поверхности. Цветом фона мы будем выводить фигуру в методе hide, чтобы она перестала отображаться на экране. Это не самый лучший, но зато самый простой способ скрыть фигуру. В дальнейшем, при желании, реализацию метода можно изменить — это никак не коснется остальных частей программы. Например, в классе FilledCircle можно применить более совершенный способ отрисовки и сокрытия фигур, основанный на режиме рисования XOR (исключающее ИЛИ), установка которого производится методом setXORMode. Такой режим можно использовать при отображении на экране фигур нашей иерархии *(см. разд. 6.9)*.

Метод moveTo имеет реализацию, несмотря на то что класс абстрактный, и в этой реализации присутствуют имена абстрактных методов show и hide. Этот вопрос будет подробно обсуждаться в *разд. 6.7*, посвященном полиморфизму.

Рассмотрим теперь, как задается потомок класса `Figure` — класс `Dot` (точка). Для `Dot` классы `Object` и `Figure` будут прародителями (суперклассами), причем `Figure` будет непосредственным прародителем. Соответственно, для них класс `Dot` будет потомком (подклассом), причем для класса `Figure` — непосредственным потомком. Класс `Dot` расширяет (extends) функциональность класса `Figure` (хотя в нем и не появляется новых полей, но зато пишется реализация для методов `show` и `hide`, которые в прародительском классе были абстрактными). В классе `Figure` мы использовали классы пакета `java.awt` без импорта этого пакета. В классе `Dot` задействован импорт — обычно это удобнее, т. к. не придется много раз писать длинные имена. Листинг 6.16 иллюстрирует окончательный код.

Листинг 6.16. Задание класса `Dot` — наследника `Figure`

```
package java_gui_example;

import java.awt.*;
/**
 * @author В.В.Монахов
 */
public class Dot extends Figure{

    /** Создает новый экземпляр типа Dot */
    public Dot(Graphics graphics,Color bgColor) {
        this.graphics=graphics;
        this.bgColor=bgColor;
    }

    public void show(){
       Color oldC=graphics.getColor();
       graphics.setColor(Color.BLACK);
       graphics.drawLine(x,y,x,y);
       graphics.setColor(oldC);
    }
    public void hide(){
       Color oldC=graphics.getColor();
       graphics.setColor(bgColor);
       graphics.drawLine(x,y,x,y);
       graphics.setColor(oldC);
    }
}
```

Отметим, что в классе `Dot` не задаются поля `x`, `y`, `graphics` и метод `moveTo` — они наследуются из класса `Figure`. А методы `show` и `hide` *переопределяются* (override) — для них пишется реализация, соответствующая тому, каким именно образом точка появляется и скрывается на экране.

Конструктор `Dot` (`Graphics graphics, Color bgColor`) занимается созданием объекта типа `Dot` и инициализацией его полей. В методах `show` и `hide` используются методы объекта `graphics`. В методе `show` сначала во временной переменной `oldC` сохраняется информация

о текущем цвете рисования. Далее в качестве текущего цвета устанавливается черный цвет (константа `java.awt. Color.BLACK`). Затем вызывается метод, рисующий точку (линия рисуется с совпадающими началом и концом), после чего восстанавливается первоначальный цвет рисования. Это необходимо для того, чтобы не повлиять на поведение других объектов, пользующихся для каких-либо целей текущим цветом. Такого рода действия очень характерны при пользовании *разделяемыми* (shared) ресурсами. Если вам при работе какого-либо метода потребуется изменить состояние разделяемых внешних данных, то сначала сохраните информацию о текущем состоянии, а в конце вызова восстановите это состояние.

Термин override (переопределить) на русский язык часто переводят как "перекрыть". Это может вводить в заблуждение, т. к. имеется еще одно понятие — перекрытие области видимости (hiding). Такое перекрытие возникает в случае, когда в классе-потомке задается поле с тем же именем, что и в прародителе (но, возможно, другого типа). Для методов совпадение имен разрешено, в том числе с именами глобальных и локальных переменных.

Имя метода в сочетании с числом параметров и их типами называется его *сигнатурой*. А сигнатура метода в сочетании с типом возвращаемого значения называется *контрактом* метода. В контракт также входит перечисление типов исключений, возбуждаемых методом, о соответствующих правилах будет говориться в *разд. 8.6*.

Если контракты задаваемого и прародительского методов совпадают, то говорят, что метод переопределен. Если у двух методов имена совпадают, но сигнатуры различны, то говорят, что производится *перегрузка* (overloading) методов. Перегрузке методов далее будет посвящен *разд. 6.11*. Если же в одном классе два метода имеют одинаковые сигнатуры, то, даже если их контракты отличаются, компилятор выдает сообщение об ошибке.

В классе нашего приложения создадим на экранной форме панель и будем вести отрисовку на ней. Зададим с помощью редактора свойств белый (или какой-нибудь другой) цвет панели (свойство `background`). Затем зададим переменную `dot`, которой назначим объект в обработчике нажатия на кнопку:

```
Dot dot=new Dot(jPanel1.getGraphics(),jPanel1.getBackground());
```

После создания объекта-точки с помощью переменной `dot` можно вызывать методы `show` и `hide`:

```
dot.show();
dot.hide();
```

Создадим на форме пункты ввода/редактирования текста `jTextField1` и `jTextField2`. В этом случае можно будет вызывать метод `moveTo`, следующим образом задавая координаты, куда должна перемещаться точка:

```
int newX=Integer.parseInt(jTextField1.getText());
int newY=Integer.parseInt(jTextField2.getText());
dot.moveTo(newX,newY);
```

Наш пример оказывается достаточно функциональным для того, чтобы увидеть работу с простейшим объектом.

Рассмотрим теперь класс `ScalableFigure` (масштабируемая фигура), расширяющий класс `Figure`. Он очень прост (листинг 6.17).

Листинг 6.17. Задание абстрактного класса `ScalableFigure` — наследника `Figure`

```
package java_gui_example;
public abstract class ScalableFigure extends Figure{
    int size;

    public void setSize(int size) {
        hide();
        this.size=size;
        show();
    }
    public int getSize() {
        return size;
    }

}
```

Класс `ScalableFigure` абстрактный — объектов такого типа создавать не предполагается, т. к. масштабируемая фигура без указания конкретного вида — это абстракция. По этой же причине в классе не заданы реализации методов `show` и `hide`.

Зато появилось поле `size` (размер) и методы `setSize` (установить размер) и `getSize` (получить размер), расширяющие этот класс по сравнению с прародителем. Для того чтобы изменить размер фигуры, отрисовываемой на экране, нужно не только присвоить полю `size` новое значение, но и правильно перерисовать фигуру. Сначала ее необходимо скрыть, затем изменить значение `size`, после чего показать на экране (уже нового размера). Следует обратить внимание, что мы пишем данный код на уровне абстракций, для нас не имеет значения, какого типа будет фигура — главное, чтобы она была масштабируемая, т. е. являлась экземпляром класса-потомка `ScalableFigure`. О механизме, позволяющем такому коду правильно работать, будет рассказано в *разд. 6.7* и *6.12*.

Опишем класс `Circle` (окружность), расширяющий класс `ScalableFigure` (листинг 6.18).

Листинг 6.18. Задание класса `Circle` — наследника `ScalableFigure`

```
package java_gui_example;
import java.awt.*;

public class Circle extends ScalableFigure {

    Circle(Graphics g,Color bgColor, int r){  //это конструктор
        graphics=g;
        this.bgColor=bgColor;
        size=r;
    }

    public void show(){
        Color oldC=graphics.getColor();
        graphics.setColor(Color.BLACK);
        graphics.drawOval(x,y,size,size);
        graphics.setColor(oldC);
    }
```

```
    public void hide(){
        Color oldC=graphics.getColor();
        graphics.setColor(bgColor);
        graphics.drawOval(x,y,size,size);
        graphics.setColor(oldC);
    }
}
```

В классе `Circle` не задается новых полей — в качестве радиуса окружности используется поле `size`, унаследованное от класса `ScalableFigure`. Зато введен конструктор, позволяющий задавать радиус при создании окружности.

Кроме того, написаны новые реализации для методов `show` и `hide`, поскольку окружность показывается, скрывается и движется по экрану не так, как точка.

Таким образом, усложнение структуры `Circle` по сравнению со `ScalableFigure` в основном связано с появлением реализации у методов, которые до этого были абстрактными. Очевидно, что класс `Circle` более специализированный по сравнению со `ScalableFigure`, не говоря уже о `Figure`.

Поля `x`, `y`, `color`, `bgColor`, `graphics` и метод `moveTo` наследуются в `Circle` из класса `Figure`. А из `ScalableFigure` наследуются поле `size` и метод `setSize`.

Следует особо подчеркнуть, что наследование относится к классам, а не к объектам. Можно говорить, что один класс — наследник другого, но категорически нельзя, что один объект — наследник другого объекта. Иногда говорят фразы вроде "объект `circle` является наследником `Figure`". Это не страшно, если подразумевается, что "объект `circle` является экземпляром класса-наследника `Figure`". Правильную фразу слишком долго произносить, но нужно четко понимать, что имеется в виду, и злоупотреблять такими оборотами не следует.

Класс `Circle` — непосредственный (прямой) потомок `ScalableFigure`, а `ScalableFigure` — непосредственный (прямой) прародитель класса `Circle` (т. е. для `ScalableFigure` класс `Circle` является подклассом, а для `Circle` класс `ScalableFigure` является суперклассом). Аналогично для `Figure` подклассами будут и `ScalableFigure`, и `Circle`. А для `Circle` суперклассами будут и `ScalableFigure`, и `Figure`.

Поскольку в Java все классы — это потомки класса `Object`, то `Object` прародитель и для `Figure`, и для `ScalableFigure`, и для `Circle`. Но непосредственным прародителем он будет только для `Figure`.

6.6. Наследование и правила видимости. Зарезервированное слово *super*

В данном разделе рассматривается ряд нетривиальных ситуаций, связанных с правилами видимости при наследовании.

Поля и методы, помеченные как `private` (закрытый, частный), наследуются, но в классах-наследниках недоступны. Это сделано в целях обеспечения безопасности. Пусть, например, некий класс `Password1` обеспечивает проверку правильности пароля, и у него есть строковое поле `password` (пароль), в котором хранится пароль и с которым сравни-

вается пароль, введенный пользователем. Если это поле имеет тип `public`, то такое поле общедоступно и сохранить его в тайне мы не сможем. При отсутствии модификатора видимости или при модификаторе `protected` на первый взгляд имеется необходимое ограничение доступа, но если мы напишем класс `Password2`, являющийся наследником от `Password1`, то в нем легко будет написать метод, вскрывающий пароль:

```
public String getPass(){
  return password;
}
```

Если же поставить модификатор `private`, то в потомке до прародительского поля `password` не добраться!

То, что private-поля наследуются, проверить достаточно просто.

В листинге 6.19 зададим класс.

Листинг 6.19. Класс с private-полем

```
public class TestPrivate1 {
  private String s="Значение поля private";

  public String get_s(){
    return s;
  }
}
```

Назначим его потомка, имеющего другое имя, но больше ничего не делающего:

```
public class TestPrivate2 extends TestPrivate1 {
}
```

Если из объекта, являющегося экземпляром `TestPrivate2`, вызвать функцию `get_s()`, то в качестве ее результата получим строку: "`Значение поля private`":

```
TestPrivate2 tst=new TestPrivate2();
System.out.println(tst.get_s());
```

Таким образом, поле `s` наследуется, и его значение можно прочитать с помощью метода, которому разрешен доступ к этому полю. Но если в классе, где задано private-поле, не предусмотрен доступ к нему с помощью каких-либо методов, доступных в наследнике, извлечь информацию из этого поля оказывается невозможным.

Модификатор `protected` предназначен для использования соответствующих полей и методов разработчиками классов-наследников. Он дает несколько большую открытость, чем пакетный вид доступа (доступ по умолчанию, если модификатор не указан), поскольку в дополнение к видимости из текущего пакета позволяет обеспечить доступ к protected-членам из классов-наследников, находящихся в других пакетах. Модификатором `protected` полезно помечать различного рода служебные методы, ненужные пользователям класса, но необходимые для функциональности этого класса.

Существует правило хорошего тона: поля данных помечать модификатором `private`, а доступ к этим полям обеспечивать с помощью методов с тем же именем, но с префиксом `get` (*получить*, т. е. обеспечить доступ к полю по чтению) и `set` (*установить*, т. е. обеспечить доступ к полю по записи). Эти методы называют *геттерами* и *сеттерами*.

Такие правила основаны на том, что прямой доступ по записи к полям данных может разрушить целостность объекта.

Рассмотрим следующий пример. Пусть у нас имеется фигура, отрисовываемая на экране. Изменение ее координат должно сопровождаться отрисовкой на новом месте. Но если мы напрямую изменили поле `x` или `y`, то фигура останется на прежнем месте, хотя поля имеют новые значения! Если же доступ к полю осуществляется через методы `setX` и `setY`, то кроме изменения значений полей будут вызваны необходимые методы, обеспечивающие перерисовку фигуры в новом месте. Также можно обеспечить проверку вводимых значений на допустимость.

Возможен и гораздо худший случай доступа к полям напрямую. Пусть у нас есть объект-прямоугольник, у которого заданы поля: `x1`, `y1` — координаты левого верхнего угла, `x2`, `y2` — координаты правого нижнего угла, `w` — ширина, `h` — высота, `s` — площадь прямоугольника.

Они взаимосвязаны: `w=x2-x1`, `h=y2-y1`, `s=w*h`. Поэтому изменение какого-либо из этих полей должно приводить к изменению других. Если же, скажем, изменить только `x2` без изменения `w` и `s`, то части объекта станут несогласованными. Предсказать, как поведет себя в таких случаях программа, окажется невозможно!

Еще хуже обстоит дело при наличии наследования в тех случаях, когда в потомке задано поле с тем же именем, что и в прародителе, да к тому же имеющее совместимый с прародительским полем тип. Поскольку для полей данных полиморфизм не работает, то возможны очень неприятные ошибки.

Указанные правила хорошего тона программирования нашли выражение в среде NetBeans 6.5. В ней при разработке UML-диаграмм добавление в класс поля автоматически приводит к установке ему модификатора `private` и созданию двух public-методов с тем же именем, но с префиксами `get` и `set` (*геттер и сеттер*). Модификаторы видимости в дальнейшем, конечно, можно менять, как и удалять ненужные методы.

Иногда возникает необходимость вызвать поле или метод из прародительского класса. Обычно это бывает в тех случаях, когда в классе-потомке задано поле с таким же именем (но, обычно, с другим типом) или переопределен метод. В результате видимость прародительского поля данных или метода в классе-потомке утеряна. Говорят, что поле или метод *затеняются* в потомке. В этих случаях используют вызов `super.имяПоля` или `super.имяМетода(список параметров)`. Слово `super` здесь означает сокращение от superclass. Если метод или поле заданы не в непосредственном прародителе, а унаследованы от более далекого прародителя, то соответствующие вызовы все равно будут работать, но комбинации вида `super.super.имя` не разрешены.

Вызовы с помощью слова `super` разрешены только для методов и полей данных объектов. Для методов и переменных класса (т. е. объявленных с модификатором `static`) вызовы с помощью ссылки `super` запрещены.

6.7. Статическое и динамическое связывание методов. Полиморфизм

Данный раздел, несмотря на его краткость, очень важен — практически все профессиональное программирование в Java основано на использовании полиморфизма. В то же

время эта тема одна из наиболее сложных для понимания, поэтому рекомендуется внимательно перечитать этот раздел несколько раз.

Методы классов помечаются модификатором `static` неслучайно — для них при компиляции программного кода действует *статическое связывание*. Это значит, что если в контексте какого-либо класса указано имя метода в исходном коде, то на метод того же класса в скомпилированном коде ставится ссылка (т. е. осуществляется *связывание имени метода* в месте вызова *с исполняемым кодом* этого метода). Иногда статическое связывание называют *ранним связыванием*, т. к. оно происходит на этапе компиляции программы. Статическое связывание в Java используется еще в одном случае — когда класс объявлен с модификатором `final` (финальный, окончательный). Но при этом поля и методы остаются полями и методами объектов, в то время как модификатор `static` помечает члены класса как переменные класса и методы класса соответственно.

Методы объектов в Java динамические, т. е. для них действует *динамическое связывание*. Оно происходит на этапе выполнения программы непосредственно во время вызова метода, причем на этапе написания данного метода заранее неизвестно, из какого класса будет произведен вызов. Это определяется типом объекта, для которого работает данный код (какому классу принадлежит объект, из того класса и вызывается метод). Такое связывание происходит гораздо позже того, как был скомпилирован код метода, поэтому такой тип связывания часто называют *поздним связыванием*.

Программный код, основанный на вызове динамических методов, обладает свойством *полиморфизма* — один и тот же код работает по-разному в зависимости от того, объект какого типа его вызывает, но делает одни и те же вещи на уровне абстракции, относящейся к исходному коду метода.

Для пояснения этих не очень понятных при первом чтении слов рассмотрим пример из предыдущего раздела — работу метода `moveTo`. Неопытным программистам кажется, что этот метод следует переопределять в каждом классе-наследнике. Это действительно можно сделать, и все заработает правильно. Но такой код окажется крайне избыточным — ведь реализация метода будет во всех классах-наследниках `Figure` совершенно одинаковой (листинг 6.20).

Листинг 6.20. Код метода `moveTo`, одинаковый во всех потомках `Figure`

```
public void moveTo(int x, int y){
 hide();
 this.x = x;
 this.y = y;
 show();
}
```

Кроме того, в этом случае не используются преимущества полиморфизма, поэтому мы не будем так делать.

Еще часто возникает вопрос — зачем в абстрактном классе `Figure` писать реализацию данного метода. Ведь используемые в нем вызовы методов `hide` и `show` на первый взгляд должны быть вызовами абстрактных методов, т. е., кажется, вообще не могут работать!

Но методы `hide` и `show` динамические, а это, как мы уже знаем, означает, что связывание имени метода и его исполняемого кода производится на этапе выполнения программы.

Поэтому то, что данные методы указаны в контексте класса `Figure`, вовсе не означает, что они будут вызываться из класса `Figure`! Более того, можно гарантировать, что методы `hide` и `show` никогда не будут вызываться из этого класса.

Пусть у нас имеются переменные `dot1` типа `Dot` и `circle1` типа `Circle`, и им назначены ссылки на объекты соответствующих типов. Рассмотрим, как поведут себя вызовы `dot1.moveTo(x1,y1)` и `circle1.moveTo(x2,y2)`.

При вызове `dot1.moveTo(x1,y1)` происходит вызов из класса `Figure` метода `moveTo`. Действительно, этот метод в классе `Dot` не переопределен, а значит, он наследуется из `Figure`. В методе `moveTo` первый оператор — вызов динамического метода `hide`. Реализация этого метода берется из того класса, экземпляром которого является объект `dot1`, вызывающий данный метод (т. е. из класса `Dot`). Таким образом, скрывается точка. Затем изменяются координаты объекта, после чего вызывается динамический метод `show`. Реализация этого метода берется из того класса, экземпляром которого служит объект `dot1`, вызывающий данный метод (т. е. из класса `Dot`). Таким образом, на новом месте показывается точка.

Для вызова `circle1.moveTo(x2,y2)` все абсолютно аналогично — динамические методы `hide` и `show` вызываются из того класса, экземпляром которого является объект `circle1`, т. е. из класса `Circle`. Таким образом, скрывается на старом месте и показывается на новом именно окружность.

Поэтому если объект является точкой, то перемещается точка, а если объект является окружностью, то перемещается окружность. Более того, если когда-нибудь кто-нибудь напишет, например, класс `FilledCircle` (заполненная окружность — т. е. окружность с заливкой цветом ее внутренней области), являющийся наследником `Circle`, и создаст объект `FilledCircle filledCircle =new FilledCircle(...)`, то вызов `filledCircle.moveTo(...)` приведет к перемещению на новое место такой окружности. И происходить это будет в соответствии с тем, каким образом в классе `FilledCircle` реализуют методы `hide` и `show`. Заметим, что работать будет давным-давно скомпилированный полиморфный код класса `Figure`. Полиморфизм обеспечивается тем, что ссылки на эти методы в код метода `moveTo` в момент компиляции не ставятся — они настраиваются на методы с такими именами из класса вызывающего объекта непосредственно в момент вызова метода `moveTo`.

В объектно-ориентированных языках программирования различают две разновидности динамических методов — собственно динамические и *виртуальные*. По принципу работы они совершенно аналогичны и отличаются только особенностями реализации.

6.8. Базовый класс *Object*

Класс `Object` базовый для всех классов Java, поэтому все его поля и методы наследуются и содержатся во всех классах. В классе `Object` содержатся следующие методы:

- `public Boolean equals(Object obj)` — возвращает `true` в случае, когда равны значения объекта, из которого вызывается метод, и значения объекта, передаваемого через ссылку `obj` в списке параметров. Если объекты не равны, то возвращается `false`. В классе `Object` равенство рассматривается как равенство ссылок и эквивалентно оператору сравнения `==`. Но в потомках этот метод может быть переопределен, и

может сравнивать объекты по их содержимому (например, так происходит для объектов оболочечных числовых классов). Это легко проверить с помощью такого кода:

```
Double d1=1.0,d2=1.0;
System.out.println("d1==d2 ="+(d1==d2));
System.out.println("d1.equals(d2) ="+(d1.equals(d2)));
```

Первая строка вывода даст `d1==d2 =false`, а вторая — `d1.equals(d2) =true`.

При переопределении метода `equals` обязательно переопределение метода `hashCode` — эта проблема подробно описана в книге Джошуа Блоха [11];

- `public int hashCode()` — выдает *хэш-код* объекта. Как уже говорилось, хэш-кодом называется условно-уникальный числовой идентификатор, сопоставляемый какому-либо элементу. Из соображений безопасности выдавать адрес объекта прикладной программе нельзя. Поэтому в Java хэш-код заменяет адрес объекта в тех случаях, когда для каких-либо целей необходимо хранить таблицы адресов объектов;

- `protected Object clone() throws CloneNotSupportedException` — метод занимается копированием объекта и возвращает ссылку на созданный *клон* (дубликат) объекта. В наследниках класса `Object` его обязательно нужно переопределить, а также указать, что класс реализует интерфейс `Clonable`. Попытка вызова метода из объекта, не поддерживающего *клонирования*, вызывает возбуждение исключительной ситуации `CloneNotSupportedException` (клонирование не поддерживается). Про интерфейсы и исключительные ситуации будет рассказано в дальнейшем.

  Различают два вида клонирования: *мелкое* (shallow), когда в клон один к одному копируются значения полей оригинального объекта, и *глубокое* (deep), при котором для полей ссылочного типа создаются новые объекты, клонирующие объекты, на которые ссылаются поля оригинала. При мелком клонировании и оригинал, и клон будут ссылаться на одни и те же объекты. Если объект имеет поля только примитивных типов, то различия между мелким и глубоким клонированием нет. Реализацией клонирования занимается программист, разрабатывающий класс, автоматического механизма клонирования нет. И именно на этапе разработки класса следует решить, какой вариант клонирования выбрать. В подавляющем большинстве случаев требуется глубокое клонирование.

  К клонированию следует относиться очень осторожно — различные аспекты этой проблемы подробно описаны в книге Джошуа Блоха [11];

- `public final Class getClass()` — возвращает ссылку на метаобъект типа `класс`. С его помощью можно получать информацию о классе, к которому принадлежит объект, и вызывать его методы класса и поля класса;

- `protected void finalize() throws Throwable` — вызывается перед уничтожением объекта. Должен быть переопределен в тех потомках `Object`, в которых требуется совершать какие-либо вспомогательные действия перед уничтожением объекта (закрыть файл, вывести сообщение, отрисовать что-либо на экране и т. п.). Вообще говоря, данный метод не следует использовать. Подробнее об этом методе говорится в *разд. 6.10*;

- `public String toString()` — возвращает строковое представление объекта (настолько адекватно, насколько это возможно). В классе `Object` этот метод реализует выдачу в

строку полного имени объекта (с именем пакета), после которого следует символ @, а затем в шестнадцатеричном виде хэш-код объекта. В большинстве стандартных классов этот метод переопределен. Для числовых классов возвращается строковое представление числа, для строковых — содержимое строки, для символьного — сам символ (а не строковое представление его кода!).

Листинг 6.21. Примеры вызова метода `toString()` для объектов разных типов

```
Object obj=new Object();
System.out.println(" obj.toString() дает "+obj.toString());
Double d=new Double(1.0);
System.out.println(" d.toString()дает "+d.toString());
Character c='A';
System.out.println("c.toString() дает "+c.toString());
```

Например, фрагмент кода из листинга 6.21 обеспечит такой вывод:

```
obj.toString() дает java.lang.Object@fa9cf
d.toString()дает 1.0
c.toString()дает A
```

Также имеются методы `notify()`, `notifyAll()` и несколько перегруженных вариантов метода `wait`, предназначенных для работы с потоками (threads). О них говорится в *разд. 10.3*.

6.9. Конструкторы. Зарезервированные слова *super* и *this*. Блоки инициализации

Как уже говорилось, объекты в Java создаются с помощью зарезервированного слова `new`, после которого идет конструктор — специальная подпрограмма, занимающаяся созданием объекта и инициализацией его полей.

Имя конструктора в Java всегда совпадает с именем класса. Хотя для конструктора не указывается тип возвращаемого значения, на самом деле он в сочетании с оператором `new` возвращает ссылку на создаваемый объект. Конструктор не является ни методом объекта (вызывается через имя класса, когда объекта еще нет), ни методом класса (в конструкторе доступен объект и его поля через ссылку `this`). Поэтому конструкторы могут считаться особым видом методов, соединяющих в себе черты как методов класса, так и методов объекта.

Такая специфика приводит к тому, что для работы с конструкторами в Java имеются специальные правила, о которых будет говориться далее.

Если в объекте при создании не нужна никакая дополнительная инициализация, то можно использовать конструктор, который по умолчанию присутствует для каждого класса. Это имя класса, после которого ставятся пустые круглые скобки (без списка параметров). Такой конструктор при разработке класса задавать не надо, он присутствует автоматически.

Если требуется инициализация, обычно применяют конструкторы со списком параметров. Примеры таких конструкторов рассматривались нами для классов `Dot` и `Circle`.

Классы `Dot` и `Circle` были унаследованы от абстрактных классов, в которых не было конструкторов. Если же идет наследование от неабстрактного класса, т. е. такого, в котором уже имеется конструктор (пусть даже и конструктор по умолчанию), то возникает некоторая специфика. Первым оператором в конструкторе должен быть вызов конструктора из суперкласса. Но его делают не через имя этого класса, а с помощью зарезервированного слова `super` (от superclass), после которого идет необходимый для прародительского конструктора список параметров. Этот конструктор инициализирует поля данных, которые наследуются от суперкласса (в том числе и от всех более ранних прародителей). Например, напишем класс `FilledCircle` — наследник от `Circle`, экземпляр которого будет отрисовываться как цветной круг (листинг 6.22).

Листинг 6.22. Использование прародительского конструктора в классе `FilledCircle`

```
package java_gui_example;
import java.awt.*;

public class FilledCircle extends Circle{

    /** Creates a new instance of FilledCircle */
    public FilledCircle(Graphics g,Color bgColor, int r,Color color) {
        super(g,bgColor,r);
        this.color=color;
    }

    public void show(){
        Color oldC=graphics.getColor();
        graphics.setColor(color);
        graphics.setXORMode(bgColor);
        graphics.fillOval(x,y,size,size);
        graphics.setColor(oldC);
        graphics.setPaintMode();
    }

    public void hide(){
        Color oldC=graphics.getColor();
        graphics.setColor(color);
        graphics.setXORMode(bgColor);
        graphics.fillOval(x,y,size,size);
        graphics.setColor(oldC);
        graphics.setPaintMode();
    }
}
```

Последовательность создания сложно устроенных объектов определяется иерархией наследования: родительская часть объекта создается и инициализируется первой, начиная от части, доставшейся от класса `Object`, и далее по иерархии, заканчивая частью, относящейся к самому классу. Именно поэтому обычно первый оператор конструктора — вызов прародительского конструктора `super(список_параметров)`, т. к. обращение к

неинициализированной части объекта, относящейся к ведению прародительского класса, может привести к непредсказуемым последствиям.

В данном классе мы применяем более совершенный способ отрисовки и сокрытия фигур по сравнению с предыдущими классами. Он основан на использовании режима рисования XOR (исключающее ИЛИ). Этот режим устанавливается методом setXORMode. При этом повторный вывод фигуры на то же место приводит к восстановлению первоначального изображения в области вывода. Переход в обычный режим рисования осуществляет метод setPaintMode.

В конструкторах очень часто встречается зарезервированное слово this, служащее для доступа к полям объекта, видимость имен которых перекрыта переменными из списка параметров конструктора. Но в конструкторах оно имеет еще одно применение — для обращения из одного варианта конструктора к другому, имеющему другой список параметров. Напомним, что наличие таких вариантов называется *перегрузкой* конструкторов. Например, пусть мы первоначально задали в классе Circle конструктор, в котором значение полей x, y и r задается случайным образом (листинг 6.23).

Листинг 6.23. Первый вариант конструктора в классе Circle

```
    Circle(Graphics g, Color bgColor){
        graphics=g;
        this.bgColor=bgColor;
        size=(int)Math.round(Math.random()*40);
    }
```

Тогда конструктор, в котором случайным образом задаются значения полей x и y, а значение size задается через список параметров конструктора, можно написать так:

```
Circle(Graphics g, Color bgColor, int r){
  this(g, bgColor);
  size=r;
}
```

При вызове конструктора с помощью слова this требуется, чтобы вызов this был первым оператором в реализации вызывающего конструктора.

Порядок вызовов при создании объекта некого класса (будем называть его дочерним классом):

1. Создается объект, в котором все поля данных имеют значения по умолчанию (нули на двоичном уровне представления).
2. Вызывается конструктор дочернего класса.
3. Конструктор дочернего класса вызывает конструктор родителя (непосредственного прародителя), а также по цепочке все прародительские конструкторы и инициализацию полей, заданных в этих классах, вплоть до класса Object.
4. Поля родительской части объекта инициализируются значениями, заданными в декларации родительского класса.
5. Выполняется тело конструктора родительского класса.

6. Поля дочерней части объекта инициализируются значениями, заданными в декларации дочернего класса.
7. Выполняется тело конструктора дочернего класса.

Знание данного порядка важно в случаях, когда в конструкторе вызываются какие-либо методы объекта, и надо быть уверенным, что к моменту вызова этих методов объект получит правильные значения полей данных.

Как правило, для инициализации полей сложно устроенных объектов используют конструкторы. Но кроме них в Java, в отличие от большинства других языков программирования, для этих целей могут также служить блоки инициализации класса и блоки инициализации объекта. Синтаксис задания классов с блоками инициализации иллюстрирует листинг 6.24.

Листинг 6.24. Синтаксис задания классов с блоками инициализации

```
Модификаторы class ИмяКласса extends ИмяРодителя {
  Задание полей;
  static {
   тело блока инициализации класса
  }
  {
    тело блока инициализации объекта
  }
  Задание подпрограмм – методов класса, методов объекта конструкторов
}
```

Блоков инициализации класса и блоков инициализации объекта может быть несколько.

Порядок выполнения операторов при наличии блоков инициализации главного класса приложения (содержащего метод `main`):

1. Инициализация полей данных и выполнение блоков инициализации класса (в порядке записи в декларации класса).
2. Метод `main`.
3. Выполнение блоков инициализации объекта.
4. Выполнение тела конструктора класса.

Для других классов порядок аналогичен, но без вызова метода `main`:

1. Инициализация полей данных и выполнение блоков инициализации класса (в порядке записи в декларации класса).
2. Выполнение блоков инициализации объекта.
3. Выполнение тела конструктора класса.

Чем лучше пользоваться — блоками инициализации или конструкторами? Ответ, конечно, неоднозначен: в одних ситуациях — конструкторами, в других — блоками инициализации. Для придания начальных значений переменным класса в случаях, когда для этого требуются сложные алгоритмы, можно пользоваться только статическими блоками инициализации. Для инициализации полей объектов в общем случае предпочтительнее конструкторы, но если необходимо выполнить какой-либо код инициализа-

ции до вызова унаследованного конструктора, можно воспользоваться блоком динамической инициализации.

6.10. Удаление неиспользуемых объектов и метод *finalize*. Проблема деструкторов для сложно устроенных объектов

Как мы знаем, конструктор занимается созданием и рядом дополнительных действий, связанных с инициализацией объекта. Уничтожение объекта также может требовать дополнительных действий. В таких языках программирования, как C++ или Object Pascal, для этих целей предусмотрены *деструкторы* — методы, которые уничтожают объект и совершают все сопутствующие этому действия.

Например, у нас имеется список фигур, отрисовываемых на экране, и мы хотим удалить из этого списка какую-нибудь фигуру. Перед уничтожением фигура должна исключить себя из списка, затем дать команду списку заново отрисовать содержащиеся в нем фигуры, и только после этого "умереть". Именно такого рода действия характерны для деструкторов. Заметим, что возможна другая логика работы: дать списку команду исключить из него фигуру, после чего перерисовать фигуры, содержащиеся в списке. Но желательно, чтобы язык программирования поддерживал возможность реализации обоих подходов.

В Java имеется метод `finalize()`. Если в классе, который производит завершающие действия перед уничтожением объекта сборщиком мусора, переопределить этот метод, то он, казалось бы, может служить некой заменой деструктора. Но поскольку момент уничтожения объекта не определен и часто отстоит по времени очень далеко от момента потери ссылки на объект, то метод `finalize` не может служить реальной заменой деструктору. Даже явный вызов сборщика мусора `System.gc()` сразу после обращения к методу `finalize()` не слишком удачное решение, т. к. и в этом случае нет гарантии правильности порядка высвобождения ресурсов, а также гарантии вызова всех методов `finalize()`. Кроме того, сборщик мусора потребляет много ресурсов и в ряде случаев может приостановить работу программы на заметное время.

Гораздо более простое и правильное решение — написать в базовом классе разрабатываемой вами иерархии метод `destroy()` (уничтожить, разрушить), который будет заниматься выполнением всех необходимых вспомогательных действий. Можно назвать метод `dispose()` (избавиться, отделаться), можно назвать `free()` (освободить). Причем при необходимости потребуется переопределение этого метода в классах-наследниках. В случае, когда нужно вызывать прародительский деструктор, следует делать вызов `super.destroy()`. При этом желательно, чтобы он был последним оператором в деструкторе класса, в противном случае логика работы деструктора может оказаться неправильной. Например, произойдет попытка обращения к объекту, исключенному из списка, или попытка записи в уже закрытый файл.

Логика разрушения объектов обратна используемой при их создании. Сначала разрушается часть, относящаяся к самому классу, затем — часть, относящаяся к непосредственному прародителю, и далее по иерархии, заканчивая частью, относящейся к базово-

му классу. Поэтому последним оператором деструктора бывает вызов прародительского деструктора `super.destroy()`.

6.11. Перегрузка методов

Напомним, что имя функции в сочетании с числом параметров и их типами называется *сигнатурой* функции. Тип возвращаемого значения и имена параметров в сигнатуру не входят. Понятие сигнатуры важно при задании подпрограмм с одинаковыми именами, но разными списками параметров — *перегрузке* (overloading) подпрограмм. Методы, имеющие одинаковое имя, но разные сигнатуры, разрешается перегружать. Если же сигнатуры совпадают, то перегрузка запрещена. Для задания перегруженных методов в Java не требуется никаких дополнительных действий по сравнению с заданием обычных методов. Если же перегрузка запрещена, то компилятор выдаст сообщение об ошибке.

Чаще всего перегружают конструкторы при желании иметь разные их варианты, т. к. имя конструктора определяется именем класса.

Например, рассмотренные ранее конструкторы:

```
Circle(Graphics g, Color bgColor){
...
    }
```

и

```
Circle(Graphics g, Color bgColor, int r){
...
}
```

отличаются числом параметров, поэтому перегрузка разрешена.

Вызовы обычных и перегруженных методов синтаксически не различаются, но все-таки в ряде случаев возникает некоторая специфика из-за неочевидности того, какой вариант метода будет вызван. При разном числе параметров такой проблемы, очевидно, нет. Если же два варианта методов имеют одинаковое число параметров, и отличие только в типе одного или более параметров, то возможны логические ошибки.

Напишем класс `Math1`, в котором имеется подпрограмма-функция `product`, вычисляющая произведение двух чисел и имеющая варианты с разными целыми типами параметров. Пример полезен как для иллюстрации проблем, связанных с вызовом перегруженных методов, так и для исследования проблем арифметического переполнения (листинг 6.25).

Листинг 6.25. Пример перегрузки методов в классе

```
public class Math1 {

public static byte product(byte x, byte y){
   return x*y;
}
```

```
public static short product(short x, short y){
   return x*y;
}

public static int product(int x, int y){
   return x*y;
}

public static char product(char x, char y){
   return x*y;
}

public static long product(long x, long y){
   return x*y;
}

}
```

Такое задание методов разрешено, т. к. сигнатуры перегружаемых вариантов различны. Обратим внимание на типы возвращаемых значений — они могут задаваться по желанию программиста. Подпрограммы заданы как методы класса (`static`), чтобы при их использовании не пришлось создавать объект.

Листинг 6.26. Пример ошибочного задания перегруженных методов

```
public static byte product(byte x, byte y){
   return x*y;
}

public static int product(byte a, byte b){
   return a*b;
}
```

Если бы мы попытались задать варианты методов из листинга 6.26, то компилятор выдал бы сообщение об ошибке, т. к. у данных вариантов одинаковая сигнатура. Ни тип возвращаемого значения, ни имена параметров на сигнатуру не влияют.

Если при вызове метода `product` параметры имеют типы, совпадающие с заданными в одном из перегруженных вариантов, все просто. Но что произойдет, когда в качестве параметра будут переданы значения типов `byte` и `int`? Какой вариант будет вызван? Проверка идет при компиляции программы, при этом перебираются все допустимые варианты. В нашем случае это `product(int x, int y)` и `product(long x, long y)`. Остальные варианты не подходят из-за типа второго параметра — тип подставляемого значения должен иметь диапазон значений, вписывающийся в диапазон вызываемого метода. Из допустимых вариантов выбирается тот, который ближе по типу параметров, т. е. в нашем случае `product(int x, int y)`.

Если для перегруженных методов среди разрешенных вариантов не удается найти предпочтительный, то при компиляции класса, где делается вызов, выдается диагностика ошибки.

Так бы случилось при вызове `Math1.product(b1,b2)`, если бы мы имели реализацию класса `Math2` (листинг 6.27) и в качестве параметров были переданы переменные `byte b1, b2`.

Листинг 6.27. Пример задания перегруженных методов — с проблемой при вызове

```
public class Math2 {

public static int product(int x, byte y){
   return x*y;
}

public static int product(byte x, int y){
   return x*y;
}

}
```

Оба варианта перегруженного метода подходят, и выбрать более подходящий невозможно. Отметим, что класс `Math2` при этом компилируется без проблем — в нем самом ошибок нет. Проблема в том классе, который его использует.

Самая неприятная особенность перегрузки — вызов не того варианта метода, на который рассчитывал программист. Особо опасные ситуации при этом возникают в случае, когда перегруженные методы отличаются типом параметров, и в качестве таких параметров выступают объектные переменные. В этом случае близость совместимых типов определяется по близости в иерархии наследования — по числу этапов наследования. Отметим, что выбор перегруженного варианта проводится статически, на этапе компиляции. Поэтому тип, используемый для этого выбора, определяется типом объектной переменной, передаваемой в качестве параметра, а не типом объекта, который этой переменной назначен.

6.12. Правила совместимости ссылочных типов как основа использования полиморфного кода. Приведение и проверка типов

Мы уже говорили, что полиморфный код обеспечивает основные преимущества объектного программирования. Но как им воспользоваться? Ведь тип объектных переменных задается на этапе компиляции.

Правило 1

Переменной некоторого объектного типа можно присваивать выражение, имеющее тот же тип или тип класса-наследника.

Аналогичное правило действует при передаче фактического параметра в подпрограмму.

Правило 2

В качестве фактического параметра вместо формального параметра некоторого объектного типа можно подставлять выражение, имеющее тот же тип или тип класса-наследника.

В качестве выражения может выступать переменная объектного типа, оператор создания нового объекта (слово `new`, за которым следует конструктор), функция объектного типа (в том числе приведения объектного типа).

Поэтому если мы создадим переменную базового типа, для которой можно писать полиморфный код, то этой переменной можно назначить ссылку на объект, имеющий тип любого из классов-потомков (в том числе еще не написанных на момент компиляции базового класса).

Пусть, например, мы хотим написать подпрограмму, позволяющую перемещать фигуры из нашей иерархии не в точку с новыми координатами (как метод `moveTo`), а на необходимую величину `dx` и `dy` по соответствующим осям. При этом у нас отсутствуют исходные коды базового класса нашей иерархии (либо их запрещено менять). Для этих целей создадим класс `FiguresUtil` (сокращение от `Utilities` (утилиты, служебные программы)), а в нем зададим метод `moveFigureBy` (переместить фигуру на) (листинг 6.28).

Листинг 6.28. Метод с параметром базового типа и полиморфным кодом

```
public class FiguresUtil{

   public static void moveFigureBy(Figure figure,int dx, int dy){
      figure.moveTo(figure.x+dx, figure.y+dy);
   }
}
```

В качестве фактического параметра такой подпрограммы вместо `figure` можно подставлять выражение, имеющее тип любого класса из иерархии фигур. Пусть, например, новая фигура создается по нажатию на кнопку в зависимости от того, какой выбор сделал пользователь во время работы программы: если в радиогруппе отмечен пункт **Точка**, то создается объект типа `Dot`. Если в радиогруппе отмечен пункт **Окружность**, то создается объект типа `Circle`. Если же отмечен пункт **Круг**, то создается объект типа `FilledCircle`. Отметим также, что класс `FilledCircle` был написан уже после компиляции классов `Figure`, `Dot` и `Circle`.

Листинг 6.29 содержит фрагмент кода для класса нашего приложения.

Листинг 6.29. Иллюстрация полиморфизма с помощью вызова метода `moveFigureBy`

```
Figure figure;
java.awt.Graphics g=jPanel1.getGraphics();

//обработчик кнопки создания фигуры
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
  if(jRadioButton1.isSelected() )
    figure=new Dot(g,jPanel1.getBackground());
  if(jRadioButton2.isSelected())
    figure=new Circle(g,jPanel1.getBackground());
  if(jRadioButton3.isSelected())
    figure=new FilledCircle(g,jPanel1.getBackground(),20,
                                    java.awt.Color.BLUE);
  figure.show();
}
//обработчик кнопки передвижения фигуры
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
```

```
  int dx= Integer.parseInt(jTextField1.getText());
  int dy= Integer.parseInt(jTextField2.getText());
  FiguresUtil.moveFigureBy(figure,dx,dy);
}
```

При написании программы неизвестно, ни какого типа будет передвигаемый объект, ни насколько его передвинут — все зависит от решения пользователя во время работы программы. Именно назначение ссылки на объект класса-потомка обеспечивает возможность использования полиморфного кода.

Следует обратить внимание еще на один момент — стиль написания вызова `FiguresUtil.moveFigureBy(figure,dx,dy);`.

Можно было бы написать его так, как в листинге 6.30.

Листинг 6.30. Нежелательный стиль — громоздкие выражения в качестве параметров

```
FiguresUtil.moveFigureBy(
   figure,
   Integer.parseInt(jTextField1.getText()),
   Integer.parseInt(jTextField2.getText())
);
```

При этом экономились бы две локальные переменные (целых 8 байтов памяти!), но читаемость, восприятие и отладка кода стали бы гораздо хуже.

Часто встречающаяся ошибка: пытаются присвоить переменной типа `наследник` выражение типа `прародитель` (листинг 6.31).

Листинг 6.31. Правильное и ошибочное присваивание ссылки на объекты

```
Figure figure;
Circle circle;
...
figure =new Circle (); //так можно
...
circle= figure; /* Так нельзя! Выдастся ошибка компиляции. Несмотря на то что
                   переменной figure назначен объект типа Circle — ведь проверка на
                   допустимость присваивания делается на этапе компиляции,
                   а не динамически.*/
```

Если программист уверен, что объект имеет тип класса-потомка, в таких случаях требуется *приведение типа*. Для этого перед выражением или именем переменной в круглых скобках ставят имя того типа, к которому нужно осуществить приведение (листинг 6.32).

Листинг 6.32. Приведение типа при присваивании ссылок

```
Figure figure;
Circle circle;
Dot dot;
...
```

```
figure =new Circle (); //так можно
...
circle= (Circle)figure; //так можно!
dot=(Dot) figure; //так тоже можно!
```

Отметим, что приведение типа принципиально отличается от преобразования типа, хотя синтаксически записывается так же. Преобразование типа приводит к изменению содержимого ячейки памяти и может привести к изменению ее размера. А вот приведение типа не меняет ни размера, ни содержимого никаких ячеек памяти — оно изменяет только тип, сопоставляемый ячейке памяти. В Java приведение типа применяется к ссылочным типам, а преобразование — к примитивным. Это связано с тем, что изменение типа ссылочной переменной не приводит к изменению той ячейки, на которую она ссылается (т. е. в случае приведения тип объекта остается прежним — меняется тип ссылки на объект).

Приводить тип можно как в сторону генерализации, так и в сторону специализации.

Приведение в сторону генерализации безопасно, т. к. объект класса-потомка всегда является экземпляром прародителя, хоть и усложненным. А вот приведение в сторону специализации опасно — вполне допустимо, что во время выполнения программы окажется, что объект, назначенный переменной, не является экземпляром нужного класса. Например, при приведении (`Circle`)`figure` может оказаться, что переменной `figure` назначен объект типа `Dot`, который не может быть приведен к типу `Circle`. В этом случае возникает исключительная ситуация приведения типа (`typecast`).

Возможна программная проверка того, что объект является экземпляром заданного класса:

```
if(figure instanceof Circle)
           System.out.println("figure instanceof Circle");
```

Иногда вместо работы с самими классами бывают удобны ссылки на класс. Они получаются с помощью доступа к полю `.class` из любого класса.

Возможно создание переменных типа `ссылка на класс`:

```
Class c=Circle.class;
```

С их помощью можно обращаться к переменным класса и методам класса. Кроме того, переменные типа `ссылка на класс` позволяют создать экземпляры этого класса с помощью метода `newInstance()`:

```
Circle circle=(Circle)c.newInstance();
```

Возможна программная проверка соответствия объекта нужному типу с помощью ссылки на класс:

```
if(figure.getClass()==Circle.class)
        circle= (Circle)figure;
 ...;
```

Но следует учитывать, что при этом идет точное сравнение классов, а не проверка на допустимость приведения типов. А вот метод `isInstance` (его не следует путать с оператором `instanceof`) позволяет проверять, является ли объект `figure` экземпляром класса, на который указывает ссылка на класс `c`:

```
if(c.isInstance(figure))
  System.out.println("figure isInstance of Circle");
```

Краткие итоги по главе

- *Наследование* опирается на инкапсуляцию. Оно позволяет строить на основе первоначального класса новые, добавляя в классы новые поля данных и методы. Первоначальный класс называется *прародителем* (ancestor), новые классы — его *потомками* (descendants). От потомков можно наследовать, получая очередных потомков. Набор классов, связанных отношением наследования, называется *иерархией классов*. Класс, стоящий во главе иерархии, от которого унаследованы все остальные (прямо или опосредованно), называется *базовым классом иерархии*.
- Иерархия нужна для того, чтобы писать *полиморфный код*, который и обеспечивает основные преимущества объектного программирования.
- Поля отражают состояние объекта, а методы задают его поведение.
- Чем ближе к основанию иерархии лежит класс, тем более *общим*, универсальным (general) он является. Чем дальше от базового класса иерархии стоит класс, тем он более *специализированный* (specialized).
- Каждый объект класса-потомка при любых значениях его полей данных должен рассматриваться как экземпляр класса-прародителя на уровне абстракций поведения, но с некоторыми изменениями в реализации этого поведения.
- В Java подпрограммы задаются только как методы в каком-либо классе и называются *функциями*. Объявление в классе функции состоит из задания *заголовка* и *тела* функции (ее *реализации*).
- Параметры, указанные в заголовке функции при ее декларации, — *формальные*, а подставляемые во время вызова функции — *фактические*. Формальные параметры нужны для указания последовательности действий с фактическими параметрами после того, как те будут переданы в подпрограмму во время вызова. Это не что иное, как особый вид локальных переменных, которые служат для обмена данными с внешним миром.
- Параметры в Java передаются в функцию всегда *по значению*. Передача *по ссылке* отсутствует. В частности, ссылки передаются в функции по значению.
- В Java имеются уровни видимости `private` (частный, закрытый), пакетный (без указания модификатора), `protected` (защищенный) и `public` (общедоступный). Пакетный — это уровень видимости по умолчанию (default) для полей и методов, для задания остальных уровней предназначены модификаторы `private`, `protected` и `public`.
- Ссылка `this` обеспечивает ссылку на объект из метода объекта. Чаще всего она используется при перекрытии области видимости имени поля объекта формальным параметром функции.
- В конструкторах зарезервированное слово `this` применяется также для обращения из одного варианта конструктора к другому, имеющему другой список параметров — в формате `this(`*параметры*`)`.

- В классе-наследнике методы можно *переопределять*. При этом у них должен сохраняться *контракт* — в который входит весь заголовок метода за исключением имен формальных параметров.
- Можно задавать *перегруженные* (overloaded) варианты методов, отличающиеся *сигнатурой*. В сигнатуру входит только часть заголовка метода — имя функции, а также число, порядок и тип ее параметров.
- Переменной некоторого объектного типа можно присваивать выражение, имеющее тот же тип или тип класса-наследника. В качестве фактического параметра функции вместо формального параметра некоторого объектного типа можно подставлять выражение, имеющее тот же тип или тип класса-наследника. Именно это правило обеспечивает возможность работы с полиморфным кодом.
- Для приведения типа используется имя типа, заключенное в круглые скобки (как и для преобразования типа). Но при преобразовании типа могут меняться содержимое и размер ячейки, к которой применяется данный оператор, а при приведении типа ячейка и ее содержимое остаются теми же, просто начинают считать, что у ячейки другой тип.
- Проверка на то, что объект является экземпляром заданного класса, осуществляется оператором `instanceof`: `if(figure instanceof Circle)...`
- Возможна программная проверка точного соответствия объекта нужному типу с помощью ссылки на класс: `if(figure.getClass()==Circle.class)...` При этом для экземпляра класса-наследника `Circle` сравнение даст значение `false`, в отличие от экземпляра `Circle`.
- Метод `isInstance` позволяет проверять совместимость типа объекта с классом, на который задана ссылка `c`: `if(c.isInstance(figure))...` Если класс, экземпляром которого является объект `figure`, является наследником класса `c` или совпадает с ним, то сравнение даст значение `true`.

Типичные ошибки

- Очень часто ошибочно считают более общим объект, из которого можно получить другой при каких-либо *конкретных значениях полей данных*. Например, окружность считают частным значением эллипса, а точку — частным значением окружности. Но в объектном программировании отношения общности и специализации объектов определяются по сложности их устройства (наличию дополнительных полей данных и методов), а также их поведению при *произвольных* значениях их полей данных. Чем сложнее объект, тем он более специализирован (менее общий).
- Очень часто возникают ошибки при попытке вернуть из функции измененные значения ссылочных переменных, переданных через список параметров. Для такого возврата требуется либо глобальная переменная, либо передача ссылки на объект, полем которого является изменяемая переменная.
- Неправильно приводят типы. Например, если переменной `Object object` присвоена ссылка на объект типа `Dot`, пытаются сделать приведение (`Circle`) `object`. Такая ошибка на этапе компиляции не может быть распознана и возникает исключитель-

ная ситуация неправильного приведения типа (invalid typecast) во время выполнения программы.

- При использовании оператора `instanceof` пытаются написать `instanceOf`.
- Оператор `instanceof` ошибочно пытаются применить для распознавания типов объектов, считая, что он обеспечивает проверку на совпадение с указанным типом, а не на совместимость с ним.
- В родительском классе выполняют проверку на то, не принадлежит ли экземпляр одному из классов-наследников. Например, в каком-либо методе в классе `Figure` пишут что-нибудь вроде

```
if( this instanceof Dot ) op1;
else  if( this instanceof Circle ) op2;
else  if( this instanceof Rectangle ) op3;
```

или

```
if( this.getClass() == Dot.class ) op1;
else  if( this.getClass() == Circle.class ) op2;
else  if( this.getClass() == Rectangle.class ) op3;
```

Такого рода проверки — вопиющее нарушение идеологии наследования. *Прародительский класс не должен ничего знать о своих потомках!* Он должен реализовывать абстракцию поведения, выраженную в названиях методов. И ни в коем случае не должен задействовать частные случаи реализации этой логики, иначе это не будет абстракцией.

Задания

- В классе `MathUtil` написать подпрограмму вычисления факториала `public static double factorial(int n)`.

  Модификатор `static` помечает подпрограмму как метод класса (т. е. позволяет вызывать метод через имя класса без создания объекта).

  Напомним, что факториал натурального числа `n` — это произведение всех натуральных чисел от `1` до `n`:

  `n!=1·2·...·(n-1)·n`

  Кроме того, `0!` считается равным `1`. Обозначение факториала в виде `n!` — математическое, в Java символ `!` зарезервирован для других целей. Также написать подпрограммы вычисления факториала с другими типами возвращаемых значений:

  `public static long factorial_long(int n)`

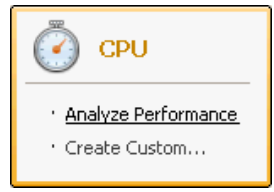
  и

  `public static int  factorial_int(int n)`

  Сравнить работу подпрограмм при `n=0,1,5,10,20,50,100`. Объяснить результаты.
- Разработать в пакете приложения библиотеку классов для иерархии фигур. Реализация должна быть с заглушками при реализации методов `show` и `hide`. Вместо отобра-

жения на экране эти методы должны выводить в консольное окно вывода имя класса фигуры и слово `show` или `hide`, а также координаты `x` и `y` фигуры.

- Разработать приложение, в котором используются эти классы — создается фигура нужного типа по нажатию кнопки и показывается. Создать документационные комментарии для полей и методов разработанных классов. Вызвать генерацию документации для проекта, просмотреть в ней созданные комментарии.
- Проверить в исходном коде проекта справку, возникающую при вызовах `figure.` и `dot.`, где эти переменные заданы как `Figure figure` и `Dot dot`. Предварительно для классов `Figure` и `Dot` должны быть заданы документационные комментарии и сгенерирована документация.



ГЛАВА 7

UML-диаграммы. Прямое и обратное проектирование. Рефакторинг

Современные методы разработки программного обеспечения принципиально отличаются от того, что было принято 10—20 лет тому назад. На смену компиляторам командной строки пришли интегрированные среды разработки (IDE), позволяющие заметно упростить разработку приложений, их отладку, нахождение ошибок (в том числе с подчеркиванием ошибочных конструкций прямо во время редактирования исходного кода, еще до этапа компиляции, как это делается в среде NetBeans). Кроме того, в последние годы появилось несколько дополнительных возможностей, в частности средства, позволяющие проектировать классы, и инструменты, облегчающие модификацию проектов.

7.1. UML-диаграммы

В языке Java, к сожалению, отсутствуют адекватные средства для проектирования классов. Более того, в этом отношении он заметно уступает таким языкам, как C++ или Object Pascal, поскольку в Java нет разделения декларации класса (описание полей и заголовков методов) и реализации методов. Но в NetBeans имеется способ решения этой проблемы — создание UML-диаграмм. Для этого достаточно установить UML Pack. UML расшифровывается как Unified Modeling Language (унифицированный язык моделирования). Этот язык предназначен для моделирования классов и их связей друг с другом на уровне абстракций, т. е. для решения задач объектно-ориентированного проектирования (Object-Oriented Architecture, OOA). Приведенные в предыдущей главе рисунки иерархий классов — это UML-диаграммы, сделанные с помощью NetBeans, но только уменьшенные по размерам, для того чтобы скрыть ненужные особенности классов (имена полей и методов, их типы и списки параметров).

Технология проектирования, позволяющая при изменении UML-диаграмм автоматически менять исходный код проекта и наоборот (при изменении исходного кода проекта автоматически менять UML-диаграммы), называется Software Engineering (проектирование программного обеспечения).

В рамках Software Engineering различают два направления: прямое проектирование (построение исходного кода классов по UML-диаграммам, Forward Engineering) и обратное проектирование (построение UML-диаграмм по исходному коду классов, Reverse Engineering).

Изучение технологии проектирования начинающими программистами — важная и полезная задача, которая может быть успешно решена с помощью NetBeans 6.5. Гораздо правильнее с самого начала приучать начинающих программистов продумывать логику устройства классов и их взаимодействия, нежели писать последовательности операторов, не связанных единой логикой и непонятно для чего предназначенных.

Среда NetBeans предоставляет весьма развитые возможности прямого и обратного проектирования, причем работа с UML-диаграммами в NetBeans 6.5 заметно усовершенствована по сравнению с предыдущими версиями. Правда, все еще присутствует ряд серьезных проблем, не позволяющих считать средства UML-проектирования среды NetBeans основой для профессиональной разработки классов. Например, при UML-проектировании пока нет возможности совершать откаты (Undo/Redo). UML-проекты невозможно переименовывать. Приходится избегать копирования элементов и диаграмм, поскольку при этом возникают неразрешимые проблемы. Ни в коем случае нельзя одновременно открывать два и более проектов, т. к. в итоге можно потерять все спроектированные классы. Сигнатуру метода проще сначала задавать пустой, и только потом в соответствующем диалоге добавлять параметры в список параметров метода, и т. д.

Тем не менее среду NetBeans можно рассматривать как современный инструмент, пригодный для учебного UML-проектирования.

Из-за перечисленных недостатков модуль UML pack, обеспечивающий поддержку UML-проектирования, не входит в состав дистрибутива NetBeans 6.5, и его необходимо загружать и устанавливать через Интернет как плагин (установка плагинов с локального диска или с компакт-диска не предусмотрена). Но эту проблему можно обойти — для подключения к среде NetBeans 6.5 данного модуля следует открыть архив uml6.zip на прилагаемом к книге DVD, извлечь из него папку uml6 и скопировать ее в папку NetBeans 6.5 (т. е. внутрь папки C:\Program Files\NetBeans 6.5, если при установке среды использовались значения по умолчанию). При запуске среды NetBeans этот модуль автоматически подключится.

Рассмотрим создание UML-диаграмм в среде NetBeans.

Для этого с помощью меню **File/New Project...** откроем диалог создания нового проекта и в окне **Categories** (Категории) выберем **UML**, а в окне **Projects** (Проекты) выберем **Java-Platform Model** и нажмем кнопку **Next >** (далее) (рис. 7.1).

Появится диалоговая форма, в которой предлагается ввести имя проекта и место его расположения (рис. 7.2).

После нажатия кнопки **Finish** (Окончание) создается UML-проект и сразу появляется форма, где предлагается выбрать вид создаваемой диаграммы, которую следует добавить в этот проект. Нам будут нужны диаграммы классов, поэтому выберем пункт **Class Diagram** и нажмем кнопку **Finish** (рис. 7.3).

После этого в окне палитры **Palette**, находящемся в правом верхнем углу, выберем мышью пункт **Class** и перетащим его в основное окно проекта — окно редактирования диаграммы. На том месте, где сделаем щелчок мышью, появится элемент-класс (рис. 7.4). Для создания на диаграмме второго элемента-класса необходимо еще раз перетащить элемент **Class** из палитры элементов в окно редактирования.

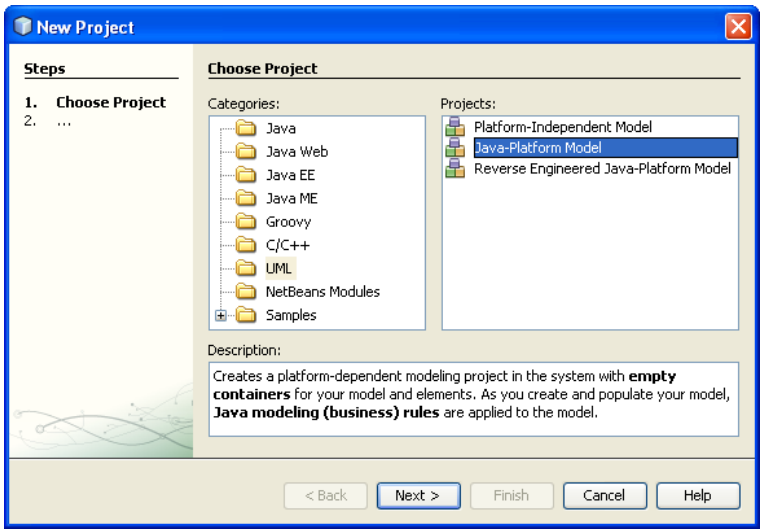


Рис. 7.1. Создание проекта с UML-диаграммами. Шаг 1

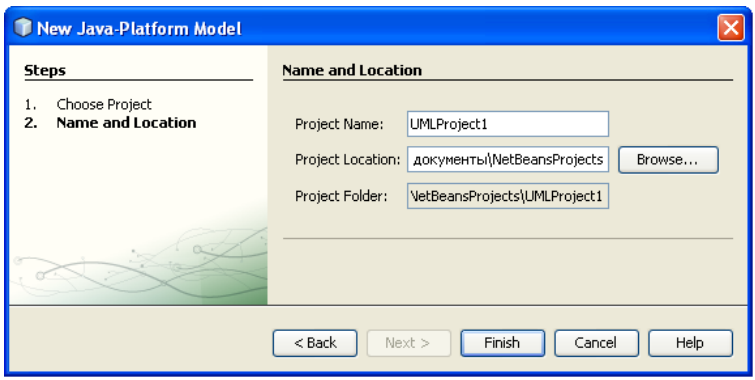


Рис. 7.2. Создание проекта с UML-диаграммами. Шаг 2

По умолчанию создается диаграмма с именем **Class Diagram1**. Желательно переименовать нашу диаграмму, дав ей какое-нибудь осмысленное имя. Мы будем проектировать иерархию фигур, поэтому назовем диаграмму **Figures Diagram**. В принципе можно писать и по-русски, но пока NetBeans не всегда правильно работает с национальными языками, поэтому во избежание проблем при переносе проекта на другие компьютеры рекомендуется (как и для всех остальных идентификаторов) использовать буквы латинского алфавита, символ подчеркивания, цифры и символ пробела.

Для переименования диаграммы имеется несколько способов. Первый из них — выделить имя **Class Diagram 1** в окне **Projects** (левый столбец с деревом проекта) в пункте **Diagrams**. Повторный щелчок (а не двойной!), когда имя уже выделено, переводит в режим редактирования имени элемента дерева. Редактирование оканчивается нажатием клавиши <Enter>, при этом появится диалоговая форма с предупреждением (рис. 7.5), в переводе на русский язык означающее: "Диаграмма Class Diagram 1 была изменена. Она должна быть сохранена перед переименованием. Сохранить ее?"

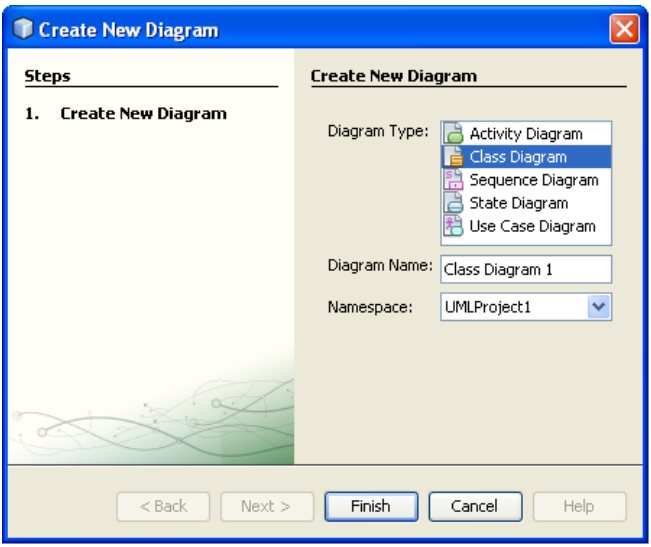


Рис. 7.3. Создание UML-диаграммы класса. Шаг 1

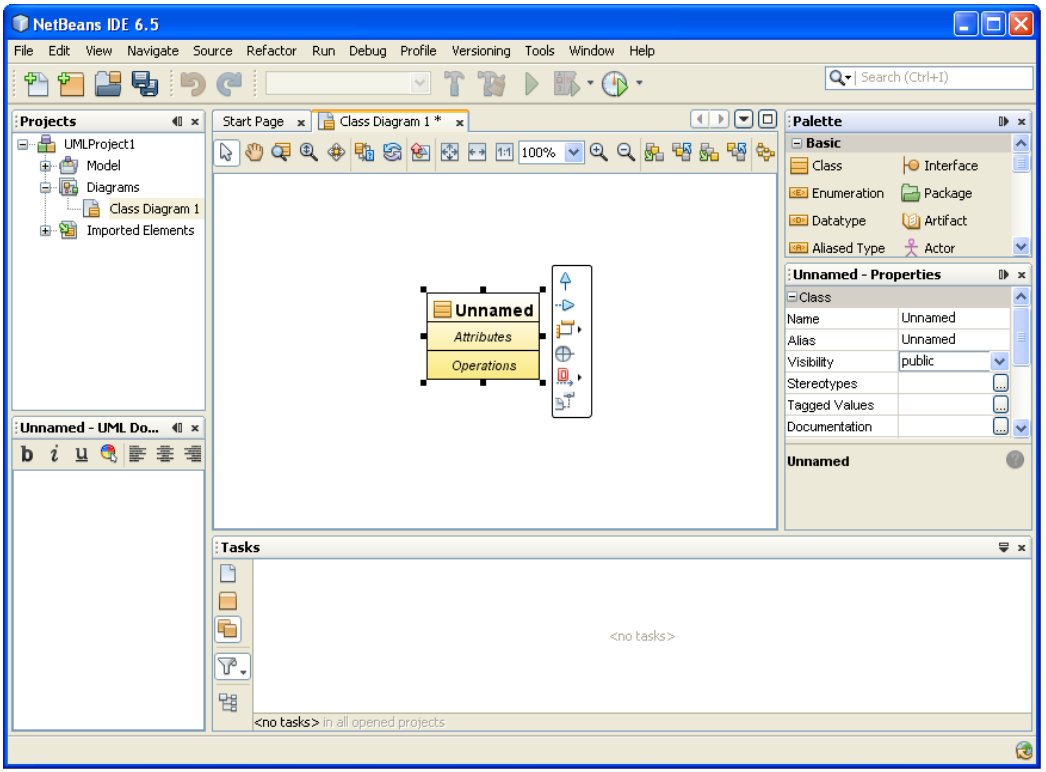


Рис. 7.4. Создание UML-диаграммы класса. Шаг 2

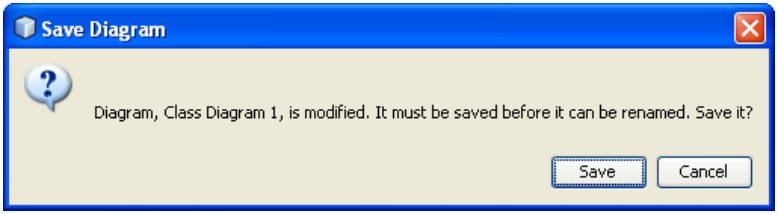


Рис. 7.5. Диалоговая форма с предупреждением

При нажатии на кнопку **Save** осуществляется переименование. Нажатие на кнопку **Cancel** (Прервать), как обычно, приводит к отмене операции (в нашем случае — переименования).

Второй способ переименования — щелкнуть по имени диаграммы правой кнопкой мыши и в появившемся контекстном меню выбрать пункт **Rename...** (Переименовать).

Третий способ — изменить имя диаграммы в окне **Properties** (Свойства). Либо щелкнуть мышью по пустой области диаграммы, выбрать в появившемся контекстном меню пункт **Properties** и изменить имя в появившейся форме установки свойств.

Аналогично можно переименовывать классы. Правда, их удобнее переименовывать непосредственно в диаграмме класса — двойной щелчок по имени какого-либо поля на диаграмме приводит к режиму редактирования этого поля.

По умолчанию новые классы создаются с именем `Unnamed` (без имени). При этом атрибуты (attributes) и методы (operations — операции) в заготовках отсутствуют. Дадим нашему первому классу имя `Figure`, а второму — `Dot`. На диаграммах классов в области **Operations** (находится в нижней части прямоугольника, изображающего класс) и в окне **Projects** в дереве элементов автоматически появятся заголовки конструкторов `public Figure()` и `public Dot()`.

Пока наши классы `Figure` и `Dot` никак не связаны друг с другом. Для того чтобы указать, что `Dot` является потомком `Figure`, требуется переместить изображение класса `Dot` под изображение класса `Figure`. Вообще говоря, это не обязательно, но в практике объектного программирования сложилась традиция, что на диаграммах базовый класс изображается на самом верху, ниже показываются его непосредственные наследники, еще ниже — их наследники и т. д.

Если мы выделим на диаграмме какой-нибудь элемент, справа от него появится панель с пиктограммами в виде вертикальной стрелки (**Generalization** — генерализация), горизонтальной стрелки и т. д. (см. рис. 7.4). Перечислим подробнее эти пиктограммы и назначение связанных с ними действий.

- **Generalization** (генерализация, общность) — треугольная стрелка, указывающая направление, противоположное направлению наследования. Если мы перетащим пиктограмму на свободное место на диаграмме, будет создан элемент с именем `Unnamed`, соответствующий родительскому классу, к которому идет связь генерализации в виде сплошной линии с треугольной стрелкой на конце. Если же перетащить пиктограмму в область другого элемента-класса, классы окажутся соединены связью, показывающей отношение генерализации, т. е. характеризующее их как наследник и прародитель. Как мы уже знаем, на UML-диаграммах принято изображать стрелки в направлении, противоположном наследованию — в направлении общно-

сти, генерализации, поэтому данный элемент называется **Generalization**. Проводить такую связь также требуется в направлении от потомка к прародителю.

- **Implementation** (реализация) — связь между классом и интерфейсом, реализующая абстракцию реализации интерфейса классом *(см. главу 9)* — наследования класса от интерфейса с реализацией в классе всех методов, заданных в интерфейсе. Изображается в виде пунктирной линии с треугольной стрелкой, указывающей направление от класса к интерфейсу. Если мы перетащим пиктограмму на свободное место на диаграмме, будет создан родительский элемент-интерфейс.
- **Association** (ассоциация) — связь между классами, реализующая абстракцию ассоциации *(см. главу 9)*. Изображается в виде сплошной линии, связывающей два класса.
- **Containment** (вложение) — связь между классами, обозначающая вложение одного класса в другой *(см. главу 12)*. Выражает отношение вложенного и родительского классов. Изображается в виде линии, на одном конце которой кружок, содержащий внутри себя крестик. Кружок с крестиком ставится у родительского класса. Если мы перетащим пиктограмму на свободное место на диаграмме, будет создан элемент, соответствующий вложенному классу.
- **Dependency** (зависимость) — связь между классами, показывающая, что один из них зависит от другого. То есть что в поле данных, списке параметров метода, типе возвращаемого значения или в списке исключительных ситуаций одного класса имеется зависимость от другого класса. Например, если в методе одного класса в качестве параметра используется экземпляр другого класса. Связь просто отмечает данную зависимость на уровне логики, не приводя к каким-либо другим последствиям. Изображается в виде пунктирной линии со стрелкой на конце.
- **Link Comment** (связать с комментарием) — позволяет создавать документационный комментарий к классу. Если мы перетащим пиктограмму на свободное место на диаграмме, будет создан элемент-комментарий *(см. далее)*.

Выделим элемент **Dot**, выберем около него пиктограмму **Generalization**, потянем ее мышью к элементу **Figure** и отпустим кнопку мыши после того, как курсор мыши зайдет в область элемента **Figure**. В результате классы окажутся соединены связью, которая является на диаграмме самостоятельным элементом (рис. 7.6).

В окне **Projects** в дереве проекта для обоих классов автоматически возникают узлы **Relationships** (отношения), а в них подузлы **Generalization**. При этом для класса `Dot` в узле **Generalization** указан класс `Figure`, что вполне естественно. А вот для класса `Figure` появление в узле **Generalization** имени класса `Dot` представляется крайне неудачным. Гораздо лучше, если бы были определены два узла: **Generalization** (генерализация, общность) и **Inheritance** (наследование), и в первом указывался бы класс-прародитель, а во втором — наследники.

Отметим несколько особенностей работы с диаграммами.

Во-первых, после установления связей изображения классов можно перемещать по экрану, и связь будет автоматически следовать за перетаскиваемым классом.

Во-вторых, можно перейти к изображению нужного класса на диаграмме путем двойного щелчка по имени класса в любом месте дерева проекта. При этом изображение класса автоматически показывается в видимой части окна и становится выделенным.

В нашем примере это не очень важно, но в реальных проектах на диаграмме бывает изображено много классов, и такой метод поиска оказывается очень удобным.

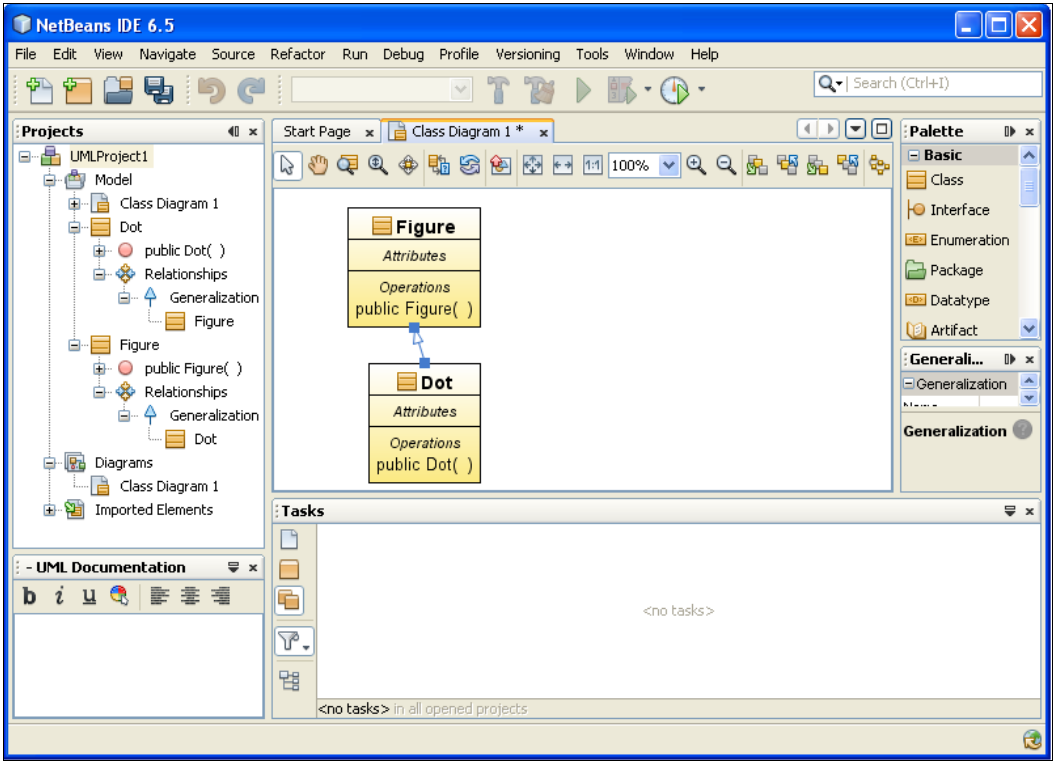


Рис. 7.6. Диаграмма после задания наследования

Среди новшеств UML-проектирования в NetBeans 6.5 хотелось бы отметить появление очень удобного средства документирования элементов диаграммы — окно **UML Documentation** (левый нижний угол на рис. 7.6). В этом окне отображается и редактируется текст-комментарий с информацией о выделенном элементе диаграммы. Например, если выделить элемент **Figure**, для него можно ввести текст "Абстрактный класс, являющийся базовым для иерархии фигур, отрисовываемых на экране" (рис. 7.7). Если на диаграмме у элемента выделить атрибут или операцию, к ним можно делать документационные комментарии. Когда не выделен ни один элемент, в окне документирования можно редактировать текст, относящийся к самой диаграмме — но этот текст в файле не сохраняется, и при повторной загрузке проекта вы его не обнаружите.

В панели инструментов окна **UML Documentation** (рис. 7.7) имеются средства форматирования текста — установка параметров шрифта (полужирный, наклонный, подчеркнутый, цветной), а также способа выравнивания текста (по левому краю, по центру или по правому краю). Если по каким-то причинам данное окно было закрыто, вновь показать его можно с помощью выбора пункта главного меню NetBeans **Window/ Other/UML Documentation**. Также можно использовать комбинацию клавиш <Ctrl>+<Shift>+<D>, но до этого следует перейти из окна редактирования диаграммы в любое другое (в окно **Project**, либо **Navigator**, либо **Output**, и т. д.).

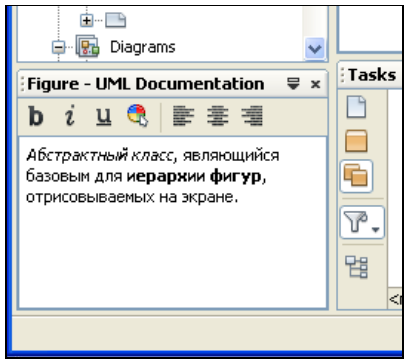


Рис. 7.7. Окно документирования UML-элемента диаграммы

При вводе длинного текста пользоваться окном документирования неудобно. В этом случае необходимо в окне **Properties** щелкнуть по полю **Documentation** (см. рис. 7.4). В результате откроется отдельное окно редактирования **Documentation** с аналогичной панелью инструментов, которое может иметь гораздо больший размер (рис. 7.8).

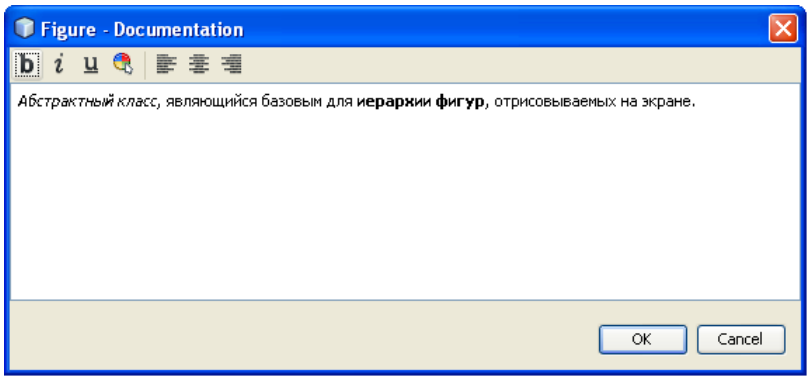


Рис. 7.8. Окно **Documentation** документирования UML-элемента диаграммы

Перечисленные возможности делают составление документационных комментариев к элементам удобным и приятным занятием, т. к. нет необходимости вручную вставлять в комментарии HTML-теги. Причем, в отличие от обычных систем UML-проектирования, сами комментарии не видны в окне диаграммы, которое в реальных условиях и так бывает перегружено большим числом элементов.

Раньше для создания документационных комментариев служило стандартное средство UML-проектирования — элемент **Comment**. Его можно выбрать в окне палитры (**Palette**), в Netbeans 6.5 элемент **Comment** последний в категории **Basic** (базовые элементы). Если данный элемент перетащить мышью из палитры в окно диаграммы, в нем появится изображение страницы с загнутым углом, а рядом — панель с пиктограммой **Link Comment** (связать комментарий) (рис. 7.9).

После чего можно связать комментарий с каким-либо классом. Для этого необходимо схватить мышью пиктограмму **Link Comment** и перетащить ее в область элемента данного класса. На диаграмме появится пунктирная линия, соединяющая элемент-комментарий с элементом-классом (рис. 7.10).

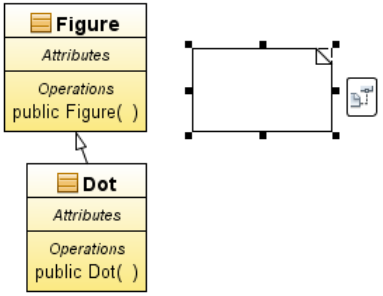


Рис. 7.9. Окно диаграммы с добавленным элементом **Comment**

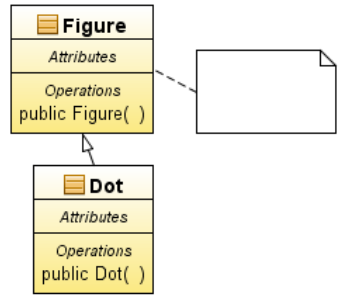


Рис. 7.10. Окно диаграммы с элементом **Comment**, связанным с элементом **Figure**

Текст в элемент-комментарий вводится в окне документирования и на самом элементе не отображается.

В отличие от окна документирования элемент **Comment** позволяет создавать документационные комментарии только для самого класса, но не для заданных в этом классе полей и методов.

Создать комментарий к классу с помощью элемента **Comment** можно и проще, не задействуя палитру компонентов. Для этого достаточно перетащить пиктограмму **Link Comment** с панели около выделенного элемента-класса (см. рис. 7.4, нижняя пиктограмма) на пустое место в области диаграммы.

7.2. Структура окон проектов с UML-диаграммами

Остановимся подробнее на структуре окон проекта с UML-диаграммами.

В первом окне **Projects** показывается дерево элементов проекта.

Во втором окне редактируются диаграммы. В верхней части окна находится палитра инструментов редактирования. О ней будет рассказано в следующем разделе.

В третьем окне **Palette** (Палитра) расположены элементы, которые можно размещать в окне проектирования. Они сгруппированы в четыре категории (рис. 7.11).

- **Basic** (базовые) — содержит наиболее часто используемые элементы, представляющие абстракции проектирования.
  - **Class** (класс) — представляет класс на диаграмме.
  - **Interface** (интерфейс) — представляет интерфейс. Интерфейсы являются аналогами полностью абстрактных классов и допускают множественное наследование. Подробное описание того, что такое интерфейсы, будет дано в *главе 9*.
  - **Enumeration** (перечисление) — представляет перечисление *(см. главу 8)*.
  - **Package** (пакет) — представляет пакеты.
  - **Datatype** (тип данных) — представляет типы данных таких величин, которые сравниваются по значениям, а не по ссылкам. В Java это примитивные типы.

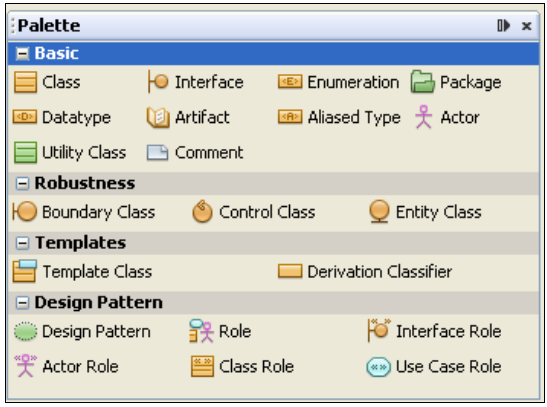


Рис. 7.11. Палитра компонентов

- **Artifact** (артефакт) — абстракция для представления информации, хранимой на каком-либо физическом устройстве (в оперативной памяти, на жестком диске и т. п.), которая используется или создается в процессе разработки приложения (например, UML-модели, исходные коды классов, двоичные коды, файлы данных).
- **Aliased Type** (тип, являющийся псевдонимом) — в C++ необходим при создании псевдонимов имен типов.
- **Actor** (актор) — роль пользователя приложения. Напомним, что в системах управления элементы, физически воздействующие на что-либо, называются *акторами* (те, кто воздействует), а элементы, регистрирующие внешние воздействия, называются *сенсорами* (те, кто воспринимает). Заметим, что привычное для зрителей слово "актер" первоначально имело тот же смысл, что и *актор* в кибернетике (науке об управлении систем с обратными связями).
- **Utility Class** (класс-утилита) — набор глобальных переменных и подпрограмм, объединенных в класс. При этом класс абстрактный, т. е. не предполагает создания объектов (экземпляров класса), а все поля и методы являются переменными и методами класса.
- **Comment** (комментарий) — элемент, задающий комментарий.

♦ **Robustness** (надежность) — содержит элементы, представляющие абстракции методики управляющих объектов.

- **Boundary Class** (пограничный класс) — класс, экземплярами которого являются пограничные объекты, т. е. объекты, непосредственно используемые акторами для взаимодействия с системой.
- **Control Class** (управляющий класс) — класс, экземплярами которого являются управляющие объекты, взаимодействующие как с пограничными объектами, так и с объектами бизнес-логики. Наличие такого промежуточного слоя отделяет логику представления данных (ввод/вывод данных в виде, необходимом акторам) от бизнес-логики работы основной части приложения. Если потребуется другой способ взаимодействия с акторами, следует заменить пограничные объекты. В простейших случаях этого обычно бывает достаточно. В более сложных системах (там, где протоколы взаимодействия с пограничными объектами при такой

замене тоже изменятся) потребуется заменить и управляющие объекты. Но протоколы взаимодействия управляющих объектов с объектами бизнес-логики и сами объекты бизнес-логики не изменятся. Такая схема обеспечивает гибкость в настройках сложных систем и гарантирует неприкосновенность объектов бизнес-логики при изменениях, относящихся к представлению информации и ее вводу/выводу и не затрагивающих бизнес-логику.

- **Entity Class** (сущностный класс, класс бизнес-логики) — класс, экземплярами которого являются объекты бизнес-логики (основной логики приложения). Они могут взаимодействовать только с управляющими объектами.

♦ **Templates** (шаблоны, заготовки).

- **Template Class** (класс, параметризованный шаблоном) — абстрактный класс, в декларации которого задана метка шаблона, обеспечивающая имена и допустимые типы параметров шаблона. Шаблоны (templates, generics) в данной книге не рассматриваются.
- **Derivation Classifier** (производный элемент) — класс, который может быть вычислен по параметризованному шаблоном классу, но указан на диаграмме для большей ясности.

♦ **Design Pattern** (паттерн проектирования).

Примечание

Паттерном в программировании принято называть часто встречающуюся типовую схему действий, применяемую для достижения заданной цели.

Данные элементы служат для организации процесса коллективной разработки проектов. Современное профессиональное программирование в значительной степени основано на использовании паттернов проектирования (см. страницу **http://barsic.spbu.ru/www/comlan/patterns.html**, на которой автор собрал ссылки на полезные ресурсы по этой тематике).

- **Design Pattern** (паттерн проектирования) — элемент, представляющий паттерн проектирования.
- **Role** (роль) — элемент, задающий роль типа данных при его задании в паттерне проектирования.
- **Interface Role** (роль интерфейса) — элемент, задающий роль интерфейса в паттерне проектирования.
- **Actor Role** (роль актора) — элемент, задающий роль актора в паттерне проектирования.
- **Class Role** (роль класса) — элемент, задающий роль класса в паттерне проектирования.
- **Use Case Role** (роль "использовать в случае, если...") — фактически, это опции условной компиляции.
- **Role Binding** (задание роли) — задание роли элементу в паттерне проектирования.

В четвертом окне **Properties** (Свойства), расположенном снизу под окном палитры элементов, показываются свойства диаграммы (если ни один из элементов не выбран)

или свойства выбранного элемента. Если же выделено сразу несколько элементов, то отображаются свойства только первого из них. Окно редактирования свойств обычно не нужно при проектировании, но иногда оно играет очень важную роль. Например, с его помощью класс задается как абстрактный — делают щелчок мышью в области имени элемента и в окне свойств ставят флажок против пункта **Abstract**. Аналогично класс помечают как окончательный (ставят флажок против пункта **Final** либо изменяют с помощью выпадающего списка спецификатор видимости с **public** на **protected**, **private** или **package**. Если в элементе выделить строку какого-либо поля или метода (сделать щелчок по этой строке), то им также можно назначить различные спецификаторы. Конечно, спецификаторы можно задавать и вручную, путем редактирования соответствующей строки непосредственно в элементе, но так удобнее — проставлять флажки и выбирать пункт из выпадающего списка быстрее и надежнее.

Еще одно важное применение окна редактирования свойств — задание документации для класса, поля или метода. Для этого служит пункт **Documentation**, о котором уже говорилось в *разд. 7.1*. Нажатие на кнопку с тремя точками, находящуюся напротив этого пункта, открывает редактор документации.

7.3. Панель инструментов проектов с UML-диаграммами

В верхней части окна проектирования (редактирования диаграммы) находится палитра инструментов (действий), которые доступны при проектировании (рис. 7.12).

Рис. 7.12. Палитра инструментов редактирования диаграмм

К левой части палитры инструментов будем относить все элементы левее выпадающего списка с текущим масштабом диаграммы (в процентах). Сам этот пункт и все расположенные правее будем считать относящимися к правой части палитры.

Рассмотрим элементы левой части палитры инструментов поочередно (слева направо).

- Первый инструмент с изображением стрелки (**Select** — выделение) служит для отмены выбранного режима (продолжения действия, использования инструмента). К аналогичному эффекту приводит нажатие клавиши <Esc>.
- Второй инструмент с изображением открытой руки (**Pan** — всеобщий) служит для сдвига мышью или стрелками клавиатуры всех элементов диаграммы на новое место. В обычном режиме перемещаются только выделенные элементы, а для перемещения их с помощью клавиатуры требуется нажатие комбинации клавиш <Ctrl>+<↑>. Отменить режим **Pan**, как уже говорилось, можно с помощью первого инструмента "стрелка" или клавиши <Esc>.

Замечание

Перемещать элементы можно и по-другому: выделить все элементы нажатием комбинации клавиш <Ctrl>+<A>, а затем двигать их мышью или стрелками клавиатуры. А выделить не-

сколько элементов мышью можно либо последовательным щелчком по ним при нажатых клавишах <Ctrl> или <Shift>, либо просто обведя нужную область мышью при нажатой кнопке мыши.

- Третий инструмент с изображением лупы с рамкой выделения (**Marquee Zoom** — увеличитель с рамкой выделения) позволяет выделить мышью нужную часть диаграммы для растягивания содержимого выделенной области на все окно.
- Четвертый инструмент (**Interactive Zoom** — интерактивное увеличение) дает возможность выбирать масштаб увеличения/сжатия изображения диаграммы путем движения мыши при нажатой кнопке. Движение вниз приводит к уменьшению масштаба (сжатию изображения), движение вверх — к увеличению изображения.
- Пятый инструмент (**Navigate Link** — перейти по связи) — небольшой оранжевый квадрат в центре с выходящими из него в разные стороны четырьмя стрелками, позволяет перейти к тому элементу, на который указывает связь. Для этого следует выбрать данный инструмент и щелкнуть мышью по связи. После этого тот элемент (и все его окружение), который находится на противоположном конце связи, будет показан в центре окна.

  Например, если мы переместим элемент **Dot** далеко вправо за пределы окна, то связи перетащатся вместе с ним. Но куда они ведут, определить будет трудно. Выберем инструмент **Navigate Link**, щелкнем по связи наследования — и мы увидим в центре окна класс `Figure`. Снова выберем инструмент **Navigate Link**, щелкнем по той же связи наследования — и мы вернемся обратно к классу `Dot`. В прежней версии NetBeans точно так же можно было переходить по связям комментариев, связям интерфейсов и т. п. Но в NetBeans 6.5 навигация по связям пока работает только для классов.
- Шестой инструмент (**Discover Relationship** — обнаружить отношение) позволяет обнаружить и показать на диаграмме отношения между элементами, которые до того не были показаны. Например, если мы по какой-то причине удалим с диаграммы стрелку наследования, но без полного удаления соответствующей связи из проекта, то нажатие на кнопку **Discover Relationship** приведет к восстановлению этой связи. Таким способом можно скрыть не очень важные связи, чтобы не загромождать проект, и восстановить их, когда они понадобятся.
- Седьмой инструмент (**Sync Diagram Synchronization** — сокращение от Synchronize Diagram, синхронизовать диаграмму) обеспечивает синхронизацию показываемой диаграммы с ее данными, хранящимися в файле. Это действие важно, когда данные в файле могут изменяться (например, при создании диаграмм по исходному коду классов). В этом случае после редактирования исходного кода следует синхронизовать диаграммы.
- Восьмой инструмент (**Export Image** — экспортировать как изображение) позволяет сохранить изображение диаграммы в формате JPEG, PNG или SVG.

  Формат JPEG наиболее известен, но мало пригоден для хранения изображений с четкими линиями и мелким текстом, т. к. размывает границы линий и букв. Поэтому рекомендуется сохранять рисунок в форматах PNG или SVG.

  Формат PNG — современная альтернатива хорошо известному формату GIF, позволяющему сохранять изображения без потери качества, он имеет перед ним ряд за-

метных преимуществ и единственный, но очень существенный недостаток — не на всех компьютерах установлены программы, позволяющие просматривать и редактировать такие файлы. Хотя в последнее время ситуация, наконец-то, меняется в лучшую сторону.

Формат SVG (Scalable Vector Graphics — масштабируемая векторная графика), по идее, лучше всего подходит для сохранения диаграмм, т. к. не только сохраняет изображение диаграммы без потери качества, но и поддерживает возможность растяжения или сжатия рисунка без потери качества. Но этот формат пока поддерживается гораздо хуже, чем даже PNG, поэтому при сохранении рисунка в этом формате нужно быть уверенным в том, что у вас и ваших коллег не возникнет проблем с дальнейшей работой.

- Девятый инструмент (**Fit Diagram** — подогнать диаграмму по размеру окна) позволяет отмасштабировать диаграмму так, чтобы она полностью помещалась в пределы окна редактирования. Пропорции диаграммы (отношение высоты к ширине) при этом не меняются.
- Десятый (предпоследний) инструмент (**Fit Width** — подогнать ширину диаграммы по размеру окна) позволяет отмасштабировать диаграмму так, чтобы она полностью помещалась по ширине в пределы окна редактирования. При этом по высоте диаграмма может выходить за пределы окна. Пропорции диаграммы (отношение высоты к ширине) при этом не меняются.
- Одиннадцатый (последний) инструмент (**100%**) позволяет одним щелчком мыши установить масштаб диаграммы 1:1, т. е. установить масштаб просмотра равным 100%.

Правая часть палитры инструментов представлена на рис. 7.13.

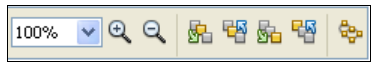


Рис. 7.13. Правая часть палитры инструментов

Рассмотрим элементы правой части палитры инструментов:

- Первый инструмент — текущий масштаб диаграммы (в процентах), а в пунктах его выпадающего списка можно выбрать другой масштаб. Также можно вручную отредактировать число в данном пункте ввода.
- Второй и третий инструменты — увеличительные стекла:
  - со знаком плюс (**Zoom In** — увеличить масштаб);
  - со знаком минус (**Zoom Out** — уменьшить масштаб).

  Эти элементы служат для увеличения/уменьшения масштаба на 5% от текущего значения при каждом нажатии.
- Следующая группа из четырех инструментов предназначена для изменения порядка перекрывания элементов при отрисовке:
  - **Move Forward** — передвинуть на один уровень вперед;
  - **Move To Front** — передвинуть на самый передний план;

- **Move Backward** — передвинуть на один уровень вглубь;
- **Move To Back** — передвинуть на задний план.

Эти инструменты полезны в тех случаях, когда изображения на диаграмме перекрываются и хочется задать порядок этого перекрытия (такие элементы располагаются выше других, перекрывая их). Обычно такая проблема возникает со связями при перетаскивании элементов, к которым они ведут.

♦ Последний инструмент (**Hierarchical Layout** — иерархическое расположение) предназначен для автоматической расстановки элементов диаграммы на экране. Рассмотрим действие менеджера иерархического расположения на примере иерархии фигур (рис. 7.14).

Замечание

Элементы-классы показаны на диаграмме в свернутом виде, когда их атрибуты и операции не видны на диаграмме. В NetBeans 6.5 появилось удобное средство для осуществления таких действий — операции сворачивания/разворачивания атрибутов и операций на диаграмме с помощью пунктов контекстного меню **Show/Hide List Compartment/Collapse All** (свернуть все) и **Show/Hide List Compartment/Expans All** (развернуть все). Для сворачивания/разворачивания атрибутов служит пункт **Show/Hide List Compartment/Attributes Compartment**, а для сворачивания/разворачивания операций — пункт **Show/Hide List Compartment/Operations Compartment**.

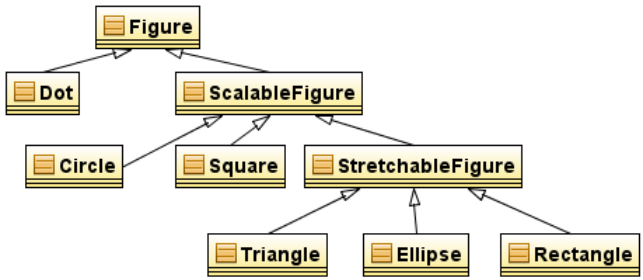


Рис. 7.14. Первоначальное расположение элементов на диаграмме, которое сделал автор

Инструмент **Hierarchical Layout** выстраивает классы на данной диаграмме согласно их положению в иерархии. При нажатии соответствующей кнопки на палитре инструментов сначала появляется окно с предупреждением (рис. 7.15).

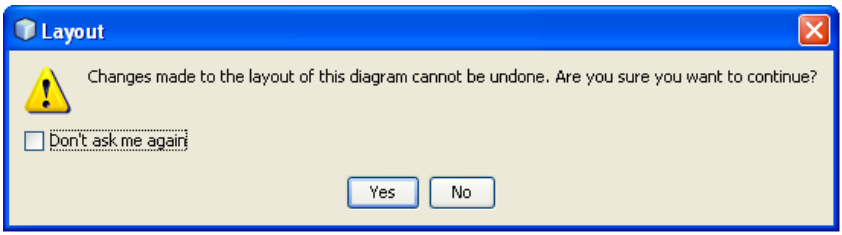


Рис. 7.15. Окно с предупреждением о том, что откат (операция undo) при изменении размещения с помощью инструмента **Hierarchical Layout** невозможен

Если пользователь подтверждает необходимость применения менеджера размещения, расположение элементов становится таким, как показано на рис. 7.16.

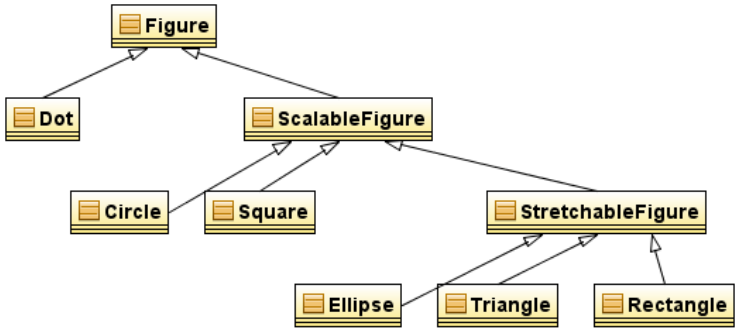


Рис. 7.16. Расположение элементов на диаграмме после применения менеджера иерархического расположения

7.4. Прямое проектирование (Forward engineering) — построение кода классов по UML-диаграммам

В NetBeans 6.5 имеются развитые средства прямого проектирования.

В проекте 07_4_1_UML_Figures приведена диаграмма нескольких классов нашей иерархии фигур, но теперь атрибуты и операции элементов на диаграмме развернуты, поэтому показываются не только имена классов, но также и поля данных (attributes — атрибуты) и заголовки методов (operations — операции).

Для добавления в наш базовый класс `Figure` нового поля `x`, задающего координату фигуры по оси *x*, можно в дереве проекта вызвать правой кнопкой мыши контекстное меню, а в нем выбрать пункты **New/Attribute** (рис. 7.17).

Также можно в области редактирования диаграммы щелкнуть правой кнопкой мыши в области надписи ***Attributes*** для элемента **Figure** и выбрать пункт **Create Attribute** (создать атрибут) (рис. 7.18).

На диаграмме у элемента **Figure** в разделе атрибутов появится декларация:

```
private int Unnamed
```

Переименуем поле из `Unnamed` в `x` и зададим ему нулевое начальное значение. Это можно сделать непосредственно в области элемента — двойной щелчок по имени атрибута или операции приводит к открытию соответствующего пункта для редактирования. Редактирование заканчивается нажатием клавиши <Enter> или щелчком мышью вне области элемента. Напрямую давать доступ к полю данных — дурной тон программирования. Поэтому рекомендуется давать полям уровень видимости `private`, а доступ к ним по чтению и записи осуществлять с помощью методов `getИмяПоля` и `setИмяПоля` (получить и установить значение этого поля).

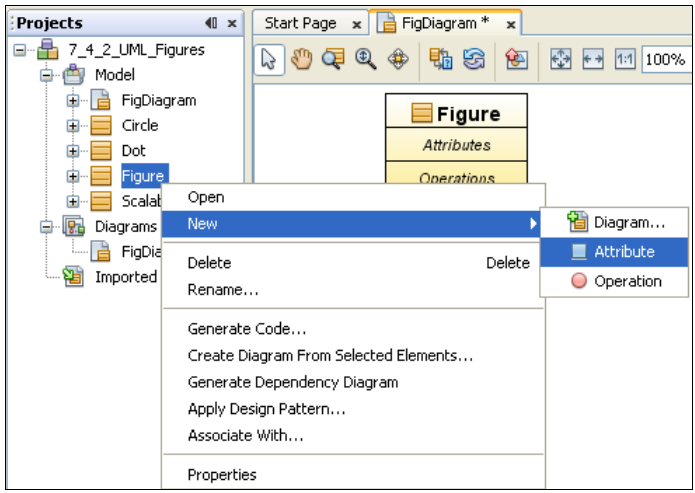


Рис. 7.17. Добавление в класс нового поля через дерево проекта

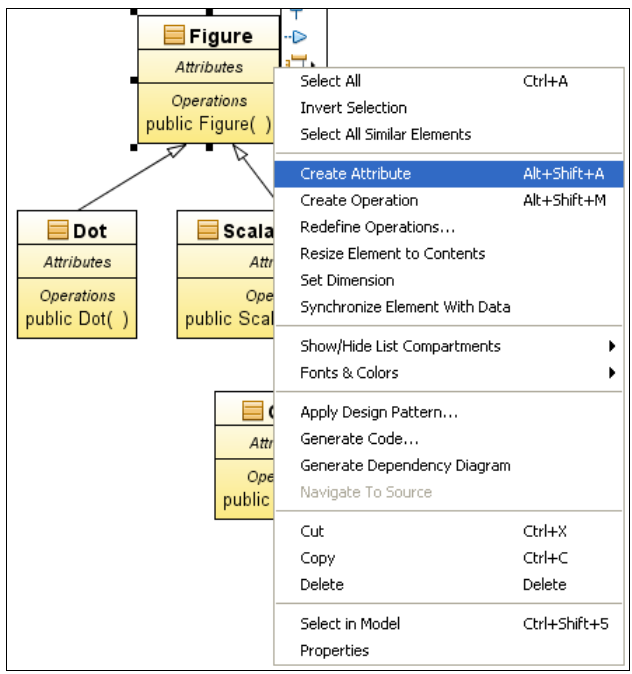


Рис. 7.18. Добавление в класс нового поля непосредственно в диаграмме

Такие методы в Java называют геттерами (getters) и сеттерами (setters). При UML-проектировании в среде NetBeans при задании поля для него автоматически генерируются геттер и сеттер. Поэтому в результате задания полю имени `x` автоматически появятся два метода — геттер — `public int getX()` и сеттер — `public void setX(int val)` (рис. 7.19). Инициализировать поле не обязательно, но в большинстве случаев для полей примитивного типа это лучше сделать сразу.

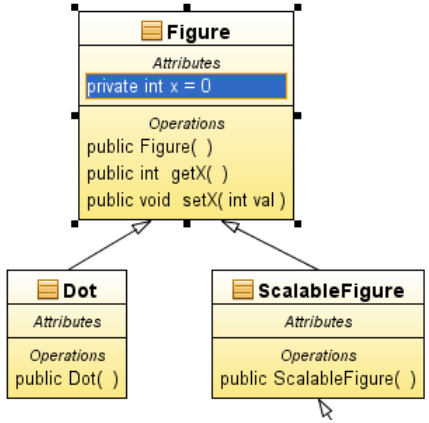


Рис. 7.19. Результат переименования и инициализации нового поля

По умолчанию новому полю всегда назначается тип `int`, но его, как и начальное значение поля, можно изменить редактированием строки.

Если мы добавим еще и поле **y**, то элемент **Figure** станет занимать слишком большую часть экрана из-за того, что кроме самого этого поля появятся еще и геттер и сеттер к нему. Как уже говорилось, в NetBeans можно сворачивать на диаграмме разделы, относящиеся к полям и методам, если они не нужны для просмотра. Для этого нажатием правой кнопки мыши нужно вызвать для данного элемента контекстное меню и выбрать в нем пункт **Show/Hide List Compartments** (Показать/Спрятать часть элемента).

Теперь с помощью редактора свойств (окно **Properties**) пометим класс `Figure` как абстрактный. После установки флажка **Abstract** начертание имени класса `Figure` изменится — оно станет наклонным (рис. 7.20). Абстрактные методы на диаграммах также отмечаются наклонным шрифтом.

Пометка класса абстрактным не приводит к исчезновению конструктора `Figure()`. На первый взгляд это кажется странным, поскольку экземпляров у абстрактного класса не может быть, и воспользоваться конструктором `Figure()` для создания экземпляра класса `Figure` невозможно. Однако конструктор может служить не только для создания экземпляра, но и для инициализации полей создаваемого объекта. Конструктор без параметров для класса `Figure` смысла не имеет, а вот конструктор `Figure(int x, int y)` полезен для использования в классах-наследниках. Но он не должен быть виден из других классов, поэтому установим для него модификатор видимости `protected`. Для этого необходимо выделить конструктор на диаграмме и в окне **Properties** выбрать в выпадающем списке в поле **Visibility** (видимость) значение **protected** (рис. 7.21).

После этого изменим список параметров конструктора — добавим в него параметры `int x` и `int y`. Для этого необходимо щелкнуть по кнопке ..., расположенной в окне **Properties** справа от пункта **Parameters**. Появится окно задания параметров (рис. 7.22).

Нажмем кнопку **New Parameter...**, в появившемся окне (рис. 7.23) введем имя параметра `x` и в выпадающем списке выберем его тип — нам подойдет стоящий по умолчанию тип `int`.

После нажатия на кнопку **OK** форма задания нового параметра закроется, и мы вернемся в форму задания параметров.

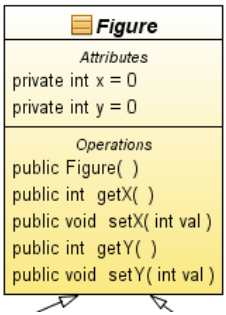


Рис. 7.20. Результат установки флажка **Abstract** для элемента **Figure**

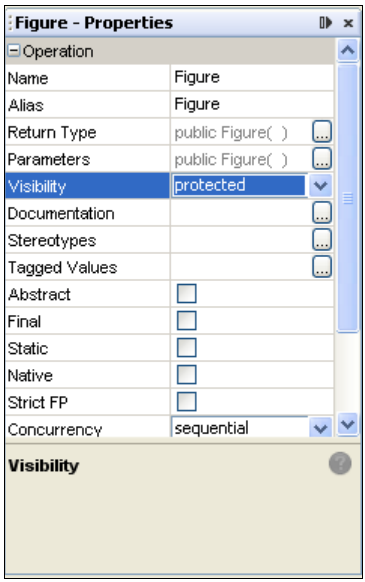


Рис. 7.21. Вид окна свойств для конструктора `Figure()`

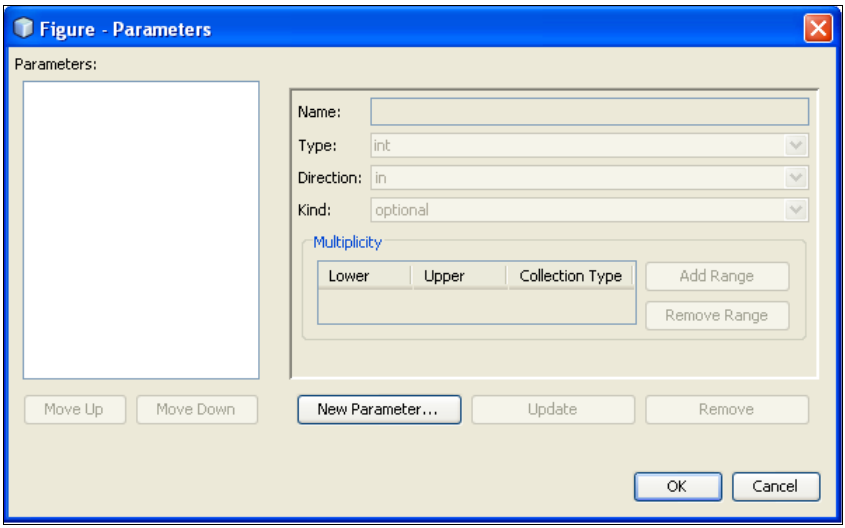


Рис. 7.22. Окно задания параметров метода или конструктора

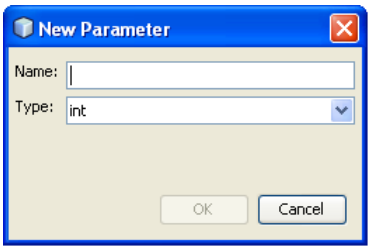


Рис. 7.23. Окно задания нового параметра метода или конструктора

Аналогично зададим параметр `int y`. Форма задания параметров приобретет вид, показанный на рис. 7.24.

Остановимся подробнее на элементах данной формы. Кнопки **Move Up** (Передвинуть вверх) и **Move Down** (Передвинуть вниз) позволяют менять порядок параметров в списке параметров. В правой части формы расположены следующие пункты: **Name** (имя), позволяющий переименовать параметр, выделенный в списке параметров в левой части формы; **Type** (Тип), позволяющий изменить тип параметра; **Direction** (Направление) и **Kind** (Вид) — задают способ передачи параметров в подпрограмму. В Java передача параметра всегда осуществляется по значению, поэтому **Direction** всегда должно быть **in** (передача внутрь подпрограммы), а **Kind** должен быть **byValue** (по значению) для примитивных типов и **addressOf** (по адресу, т. е. по ссылке) для объектных.

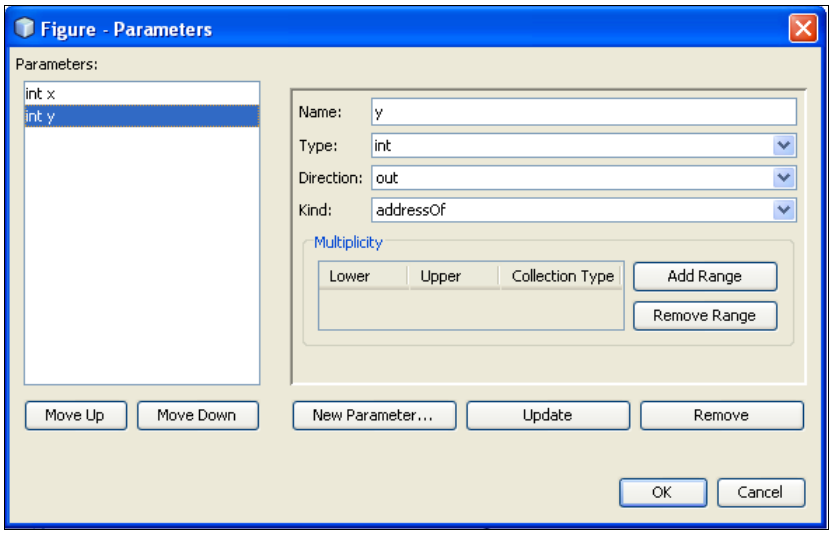


Рис. 7.24. Окно задания параметров конструктора после добавления параметра `x`

Чтобы выбранные атрибуты параметра были установлены, следует нажать кнопку **Update** (Применить) — она действует только на выделенный параметр, не затрагивая атрибуты других параметров. Если мы хотим убрать параметр из списка, следует воспользоваться кнопкой **Remove** (Удалить).

Вернемся к редактированию диаграммы. Если мы хотим удалить конструктор (или другую операцию или атрибут), необходимо щелкнуть в соответствующей области правой кнопкой мыши и в появившемся контекстном меню выбрать пункт **Delete** (Удалить).

Добавим в класс `Figure` абстрактные методы `show()` и `hide()`, а также метод `moveTo(int newX, int newY)`. Пометим класс `ScalableFigure` как абстрактный, удалим из него конструктор и добавим в него поле `int size`. В класс `Circle` добавим методы `getR` и `setR`, а в конструктор `Circle()` добавим список параметров `int x, int y, int r`.

Добавим в классы `Dot` и `Circle` методы `show()` и `hide()` — они должны переопределять одноименные абстрактные методы из класса `Figure`.

При переопределении метода в классе-наследнике возникает диалоговая форма с предупреждением о том, что сигнатура задаваемого метода совпадает с сигнатурой уже

имеющегося в базовом классе метода (рис. 7.25). Следует подтвердить, что мы действительно хотим переопределить метод.

Проект 7_4_3_UML_Figures показывает, как будет выглядеть наша иерархия после внесения всех указанных изменений.

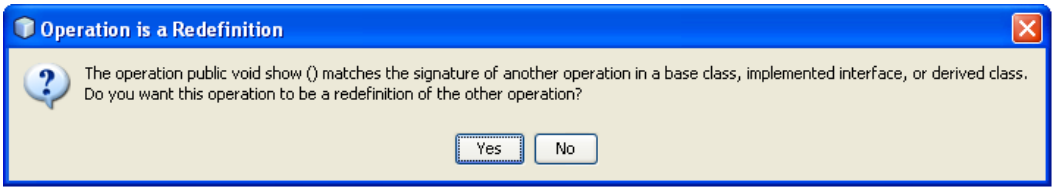


Рис. 7.25. Окно с предупреждением о переопределении метода

Будем теперь рассматривать генерацию кода Java по диаграммам данного проекта.

Для этого сначала создадим проект 7_4_3_JavaLibrary1 по схеме: **File/New Project /Java/Java Class Library/Next>**, введем имя `7_4_3_JavaLibrary1`, нажмем кнопку **Finish**.

Выделим в UML-проекте на диаграмме или в каком-либо из деревьев нужный элемент (**Figure**, **Dot** и т. д.). Если выделено несколько элементов, генерация кода идет для всех них. Так что нет необходимости заниматься генерацией по очереди для каждого класса, как это было в предыдущих версиях NetBeans.

Вызовем правой кнопкой мыши контекстное меню для выделенных классов и выберем там пункт **Generate Code...** (Генерировать код).

Появится диалоговое окно (рис. 7.26), в котором предлагается выбрать **Target Project** (Проект-мишень) для размещения сгенерированных исходных кодов.

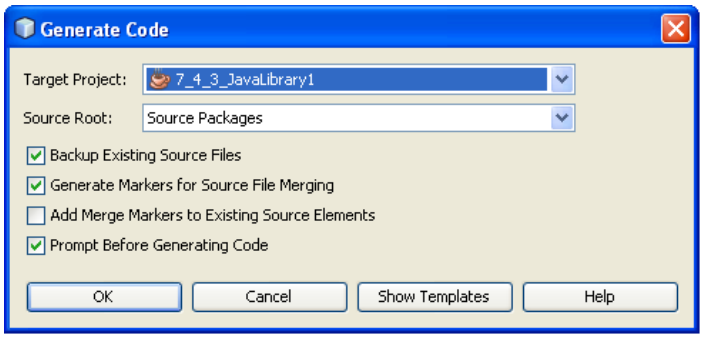


Рис. 7.26. Диалоговое окно, появляющееся после выбора пункта **Generate Code**

Выберем наш Java-проект (**7_4_3_JavaLibrary1**) в качестве "мишени" для генерации кода.

В диалоге имеются следующие установки:

- **Backup Existing Source Files** — (Создавать резервные копии существующим файлам с исходными кодами). При этом сохраняются все резервные версии всех модификаций файлов, а не только одна, как это обычно бывает в текстовых редакторах. Кроме файлов java добавляются файлы с расширениями java1~, java2~ и т. д. (с но-

мером модификации файла java). Рекомендуется оставить данный пункт выбора отмеченным флажком;

- **Generate Markers for Source File Merging** — (Генерировать маркеры для слияния с файлами исходных кодов). Помеченные маркерами коды объединяются с кодами, создаваемыми по измененным UML-диаграммам. Рекомендуется оставить данный пункт выбора отмеченным, чтобы классы, их поля данных (атрибуты) и методы в проекте изменялись при изменении UML-диаграмм;
- **Add Merge Markers to Existing Source Elements** — (Добавить маркеры слияния к существующим элементам исходных кодов). Этот пункт выбора по умолчанию не отмечен. Его следует отметить в случае, если в выбранном проекте уже имеются исходные коды, не связанные с текущим UML-проектом;
- **Prompt Before Generating Code** — (Запрос перед генерацией кода).

Кнопка **Show Templates** (Показать шаблоны) позволяет осуществлять дополнительные настройки — отключать шаблоны для генерации класса, интерфейса или перечисления (по умолчанию все три шаблона подключены).

Аналогично сгенерируем коды для классов `Dot`, `ScalableFigure`, `Circle`, в результате чего в проекте 07_4_3_JavaLibrary1 появятся соответствующие классы. Исходный код класса `Figure` с маркерами UML-проектирования показан на рис. 7.27.

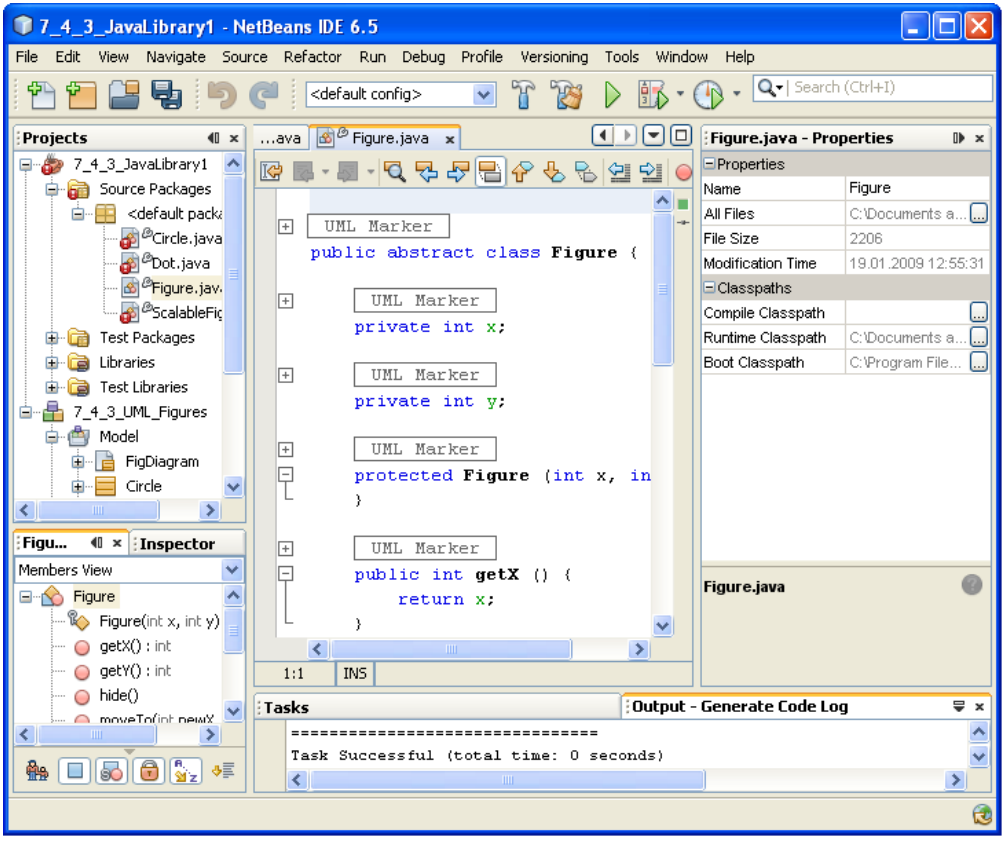


Рис. 7.27. Сгенерированный исходный код для класса `Figure`

В сгенерированном коде классы `Dot`, `Circle` и `ScalableFigure` отмечены как имеющие ошибки в реализации (восклицательные знаки около имен классов в дереве проектов). Для класса `Dot` это связано с отсутствием вызова прародительского конструктора в конструкторе `Dot(int x, int y)`. Добавление в сгенерированный код

```
public Dot (int x, int y) {
}
```

вызова `super(x,y)` решает проблему:

```
public Dot (int x, int y) {
    super(x,y);
}
```

Внесение исправлений в классы `Circle` и `ScalableFigure` предоставляется читателю в качестве упражнения.

В проекте 07_4_3_JavaLibrary1 наши классы будут помещены в пакет по умолчанию (default package). Использование пакета по умолчанию — плохой стиль программирования. Но переименовывать пакет по умолчанию нельзя. Нужно создать новый пакет и переместить в него наши классы. Как это сделать, будет рассказано в *разд. 7.6*.

Если мы внесем изменения в исходный код классов фигур в нашем получившемся Java-проекте, то они никак не отразятся на UML-диаграмме. Точно так же изменения на UML-диаграмме не повлияют на уже существующий Java-проект. Для того чтобы иметь возможность одновременно работать с исходными кодами и диаграммами и в то же время быть уверенными, что изменения в исходных кодах отразятся на диаграммах, и наоборот, необходимо использовать технологию *обратного проектирования*.

7.5. Обратное проектирование (Reverse engineering) — построение UML-диаграмм по разработанным классам

Среда NetBeans 6.5 позволяет по имеющемуся исходному коду построить UML-диаграммы. Для этого следует открыть проект (в нашем случае 7_5_JavaLibrary_Figure), щелкнуть правой кнопкой мыши по имени проекта, тем самым вызвав контекстное меню, и выбрать пункт **Reverse Engineer...** (рис. 7.28).

Появится диалоговая форма задания параметров создаваемого проекта (рис. 7.29). В ней название UML-проекта по умолчанию строится из названия проекта Java, к которому в конце добавлено **-Model**. Поэтому легко определить, к какому проекту Java относится UML-проект, и наоборот.

После нажатия на кнопку **OK** будет создан UML-проект (рис. 7.30), в котором можно просмотреть параметры, относящиеся к каждому классу. В нашей иерархии наиболее сложной является структура класса `Figure` (рис. 7.31).

В получившемся UML-проекте отсутствуют UML-диаграммы. Но их можно сгенерировать, щелкнув правой кнопкой мыши по имени соответствующего класса (рис. 7.32).

Если для элемента **Circle** выбрать пункт **Create Diagram From Selected Elements** (Создать диаграмму из выбранных элементов), далее выбрать тип диаграммы **Class**

Diagram и нажать кнопку **Finish**, то можно получить диаграмму, представленную на рис. 7.33.

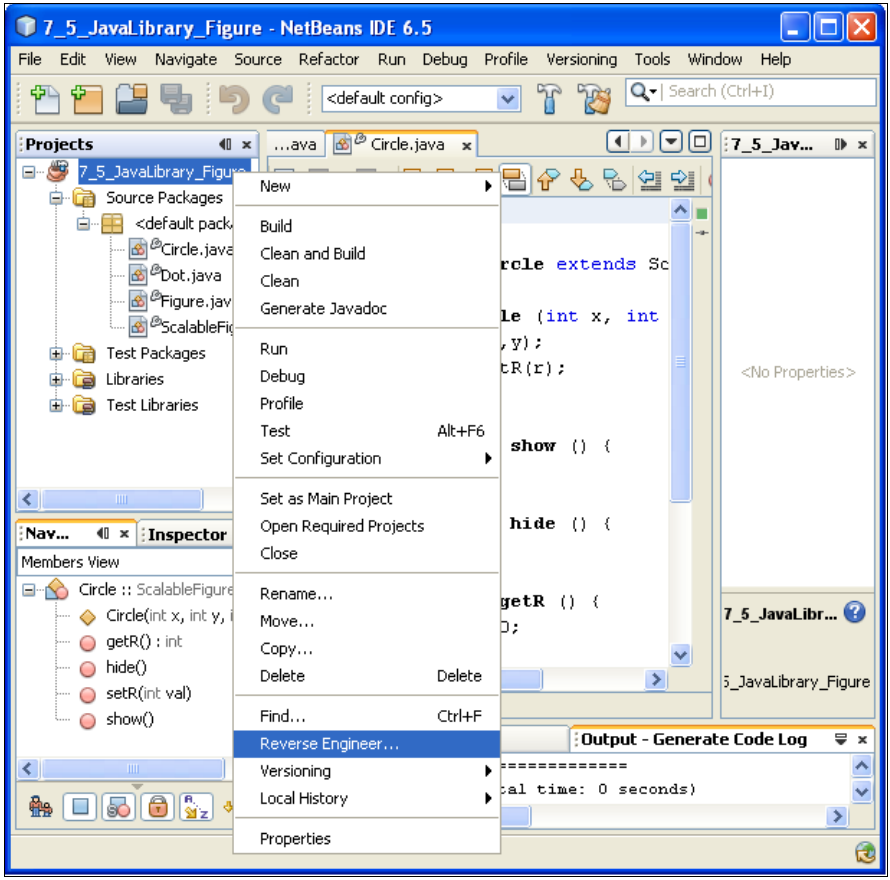


Рис. 7.28. Начало обратного проектирования — создания UML-диаграммы по исходному коду

При этом лучше заменить имя создаваемой диаграммы с **Class Diagram 1** на **Circle Class Diagram**. Переименование можно сделать и позже, щелкнув правой кнопкой мыши по имени диаграммы и выбрав в появившемся контекстном меню пункт **Rename** (Переименовать), либо щелкнув второй раз (не двойной щелчок!) и просто отредактировав имя.

Если же одновременно выделить в дереве **Projects** элементы **Circle**, **Dot**, **Figure**, **ScalableFigure** и проделать такую же операцию, то получим диаграмму наследования (рис. 7.34), которой можно дать имя **Inheritance Class Diagram**.

Если для какого-либо класса в контекстном меню выбрать пункт **Generate Dependency Diagram** (Сгенерировать диаграмму зависимостей), то получим диаграмму, на которой изображен выбранный класс и его родитель. В этом случае по умолчанию имя диаграммы образуется из имени класса, к которому в конце дописано слово Dependencies (зависимости). Например, для элемента **Circle** имя диаграммы зависимостей будет **CircleDependencies**.

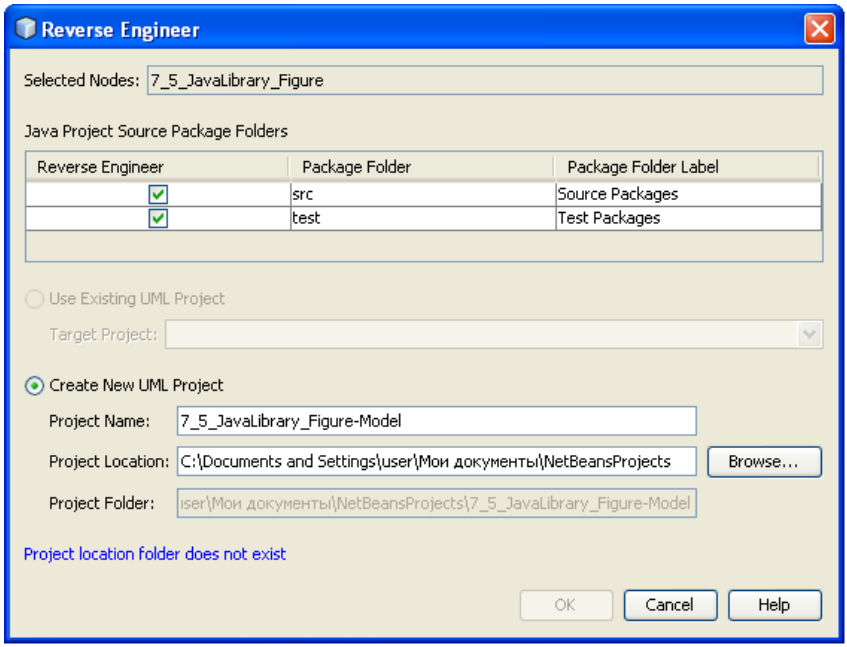


Рис. 7.29. Диалоговая форма задания параметров создаваемого UML-проекта

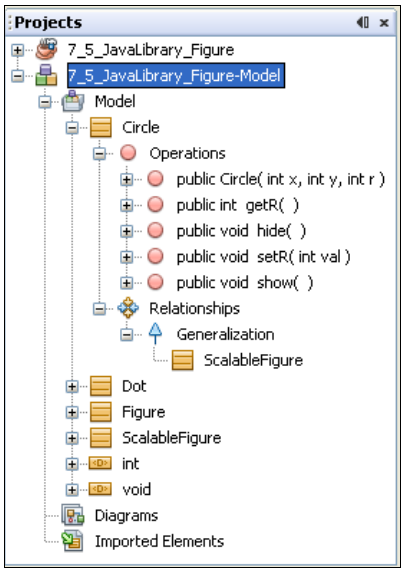


Рис. 7.30. Параметры UML-проекта, относящиеся к классу `Circle`

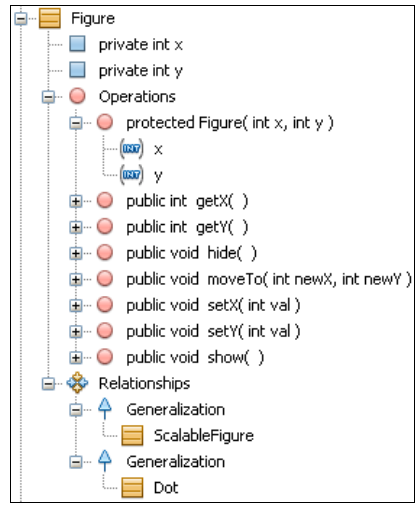


Рис. 7.31. Параметры UML-проекта, относящиеся к классу `Figure`

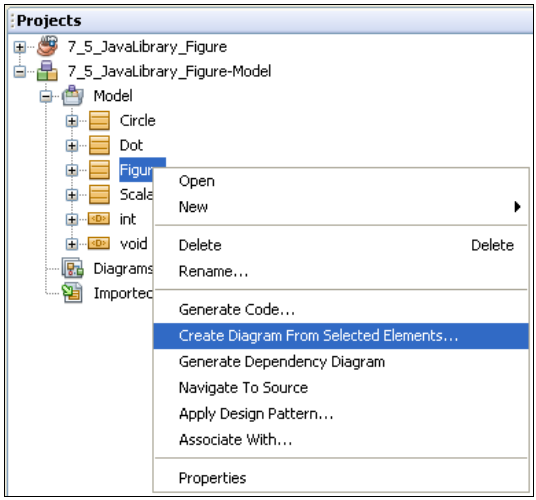


Рис. 7.32. Контекстное меню действий с классом в UML-проекте

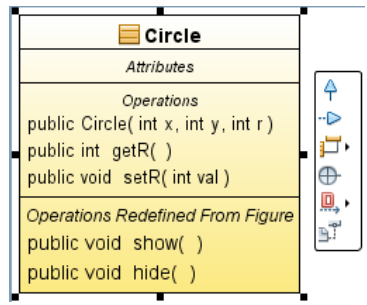


Рис. 7.33. Диаграмма для класса `Circle`

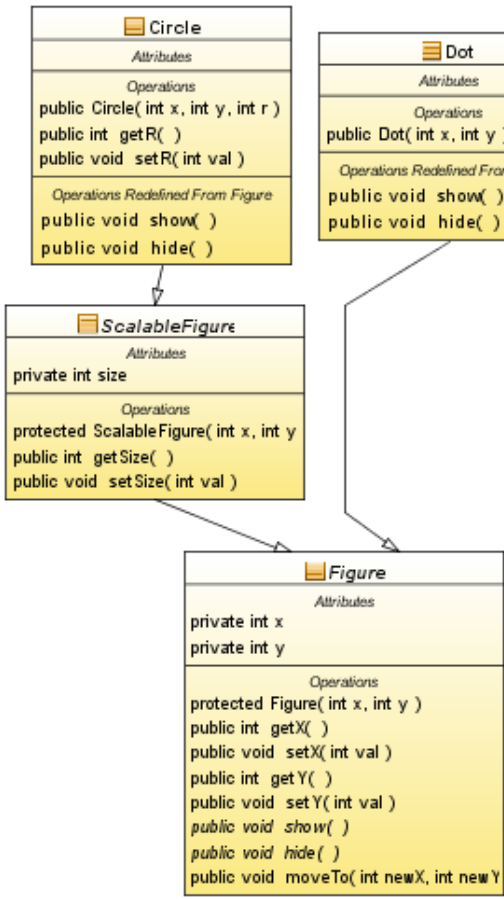


Рис. 7.34. Диаграмма для нескольких выделенных классов

Мы можем изменить исходный код проекта и заново сгенерировать диаграммы. Например, если мы добавим в класс `Circle` конструктор

```
public Circle(java.awt.Graphics g, int x, int y, int r)
```

и вызовем в проекте 7_5_JavaLibrary_Figure в контекстном меню для класса `Circle` пункт **Reverse Engineer...**, сначала появится диалоговая форма с предложением переписать в модели элемент **Circle** (рис. 7.35).

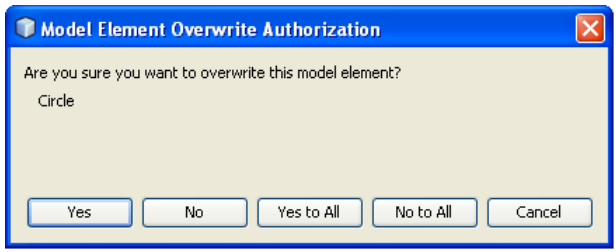


Рис. 7.35. Диалог с предложением переписать в модели элемент **Circle**

Если после этого заново сгенерировать диаграмму зависимостей для класса `Circle`, она приобретет вид, показанный на рис. 7.36.

Пункт меню **Navigate to Source** всплывающего окна позволяет показывать редактор исходного кода, соответствующего элементу диаграммы, из которого осуществляется вызов всплывающего окна.

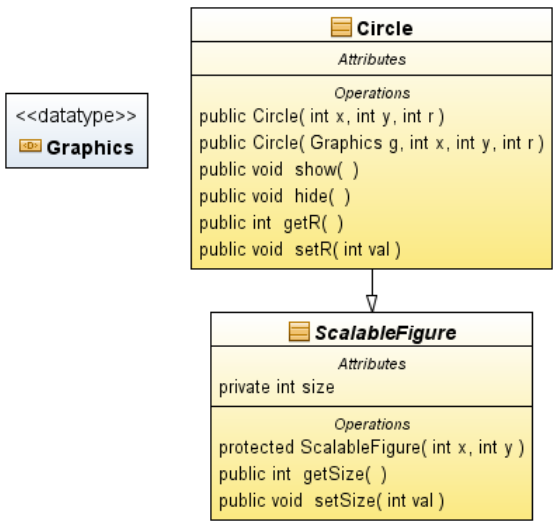


Рис. 7.36. Диаграмма зависимостей для класса `Circle`

На диаграммах можно добавлять в классы или удалять из них поля и методы, проводить переименования, менять модификаторы. Причем изменения, сделанные на любой из диаграмм, автоматически отражаются на других диаграммах UML-проекта. Но для того чтобы они отобразились в исходном коде проекта Java, требуется воспользоваться

средствами прямого проектирования (Forward Engineering) — вызвать пункт меню **Generate Code...** контекстного меню.

Точно так же изменения, сделанные в исходном коде Java, не отражаются автоматически на диаграммах UML. Для их реализации требуется использовать средства обратного проектирования — вызывать пункт **Reverse Engineer...** для соответствующего класса либо для всего проекта с исходным кодом. При этом иногда для правильного отображения диаграмм приходится применять кнопку панели инструментов **Sync Diagram**.

При последующей работе с проектами следует открывать проект с UML-диаграммой и при этом в диалоге выбора папки с проектом проставлять флажок против пункта **Open Required Projects** (Открыть необходимые проекты) (рис. 7.37).

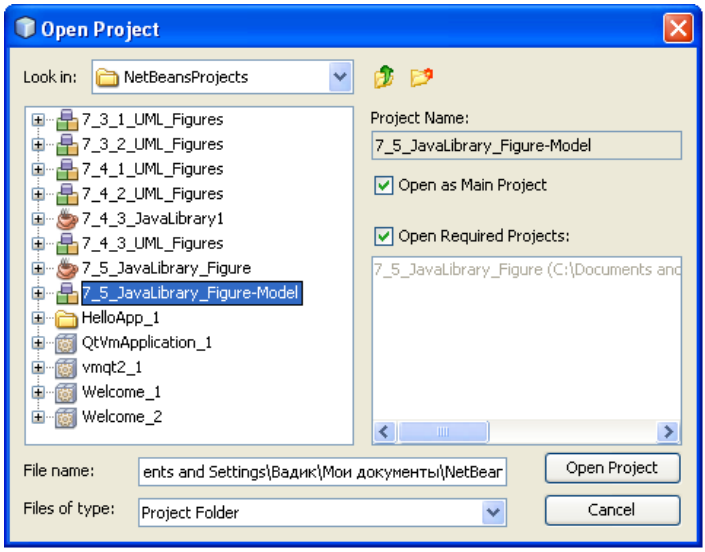


Рис. 7.37. Открытие связанных друг с другом проектов

В дальнейшем лучше открывать проект с UML-диаграммами, при этом автоматически будет открываться и проект с исходными кодами. А закрывать эти проекты лучше сразу оба — выделить их имена в окне **Projects** и в контекстном меню выбрать пункт **Close 2 Projects**.

При переносе папок с проектами на другое место связь между проектом с UML-диаграммами и проектом с исходным кодом иногда теряется, и при открытии проекта с UML-диаграммами в NetBeans 6.5 имя UML-проекта высвечивается в виде трех вопросов. Эта проблема связана с тем, что имена проектов в данной книге начинаются с цифры. Для проектов, названия которых начинаются с буквы, данная проблема отсутствует.

7.6. Рефакторинг

Один из важнейших элементов современного программирования — *рефакторинг* — изменение структуры существующего проекта без изменения его функциональности. Зачем он нужен?

Развитие любого проекта проходит несколько этапов. Сначала создают черновую структуру проекта — разрабатывают классы с набором полей и методов, пишут реализацию этих методов. Но не бывает программистов, настолько гениальных, чтобы сразу предусмотреть все тонкости взаимосвязей внутри классов, объектов, а также между классами и объектами. Поэтому по мере разработки классов приходится вносить уточнения, дополнения, изменения. А в ряде случаев — менять структуру отдельных классов, пакетов, имеющиеся взаимосвязи между классами и объектами. Причем эти изменения делают для того, чтобы проект оставался управляемым и мог дальше развиваться, а не для добавления новой функциональности непосредственно в результате сделанных изменений.

Если открыть проект 07_6_refactoring1, перейти в исходный код класса `Integrator`, установить курсор на имя **Integrator** в заголовке класса и вызвать меню рефакторинга через пункт главного меню **Refactor**, то увидим меню с возможными вариантами рефакторинга (рис. 7.38). Аналогичное меню мы увидим при вызове пункта **Refactor** контекстного меню, появляющегося при щелчке правой кнопки мыши по данному имени в исходном коде. Активные пункты меню (выделены нормальным цветом, а не бледно-серым) показывают, какие виды рефакторинга применимы к текущему элементу или выделенному фрагменту исходного кода.

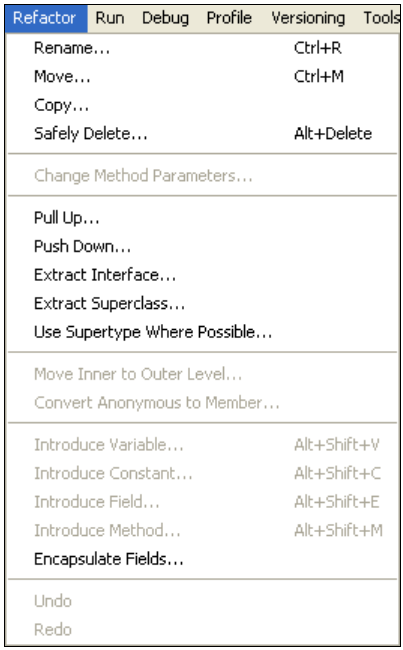


Рис. 7.38. Меню с возможными вариантами рефакторинга

Приведем шесть наиболее часто встречающихся примеров рефакторинга:

- **Rename** — переименование элементов программы (пакетов, классов, полей данных, методов, переменных);
- **Move** — перемещение классов с одного места на другое;
- **Copy** — копирование класса для его дальнейшей модификации;

- **Safely Delete** — безопасное удаление элемента программы;
- **Change Method Parameters** — изменение списка параметров метода;
- **Encapsulate Fields** — инкапсуляция полей данных (генерация геттеров и сеттеров для поля и замена непосредственного доступа к полям на вызовы геттеров и сеттеров).

В сложных проектах, конечно, возникают и другие варианты рефакторинга:

- **Pull Up** — передвижение класса вверх по иерархии;
- **Push Down** — передвижение класса вниз по иерархии;
- **Extract Interface** — извлечение из класса интерфейса (выделение тех методов, которые должны быть продекларированы в интерфейсе). Обсуждению концепции интерфейсов посвящена *глава 9*;
- **Extract Superclass** — извлечение из класса прародительского класса (выделение тех полей и методов, которые должны быть продекларированы в прародительском классе);
- **Use Supertype Where Possible** — использование везде, где можно, прародительского типа наиболее высокого уровня в иерархии (для обеспечения максимально возможной полиморфности кода);
- **Move Inner to Outer Level** — перемещение внутреннего класса на уровень его внешнего класса (про внутренние классы рассказывается в *главе 12*);
- **Convert Anonymous to Member** — преобразовать анонимный класс во внутренний (про эти классы рассказывается в *главе 12*).

Существует еще одна группа команд рефакторинга, которая относится к добавлению в класс новых элементов:

- **Introduce Variable** — ввести новую переменную (в метод);
- **Introduce Constant** — ввести новую константу;
- **Introduce Field** — ввести новое поле данных (в класс);
- **Introduce Method** — ввести новый метод.

И, наконец, в меню рефакторинга имеются две очень полезные команды:

- **Undo** [*название операции*] — отменить изменения, совершенные при операции рефакторинга (сделать откат на один шаг). Возможен постадийный откат на несколько шагов;
- **Redo** [*название операции*] — вернуть изменения, совершенные при операции рефакторинга, для которых был совершен откат назад операцией **Undo**. Возможно постадийное восстановление после откатов на несколько шагов.

Существуют и другие виды рефакторинга, которые среда NetBeans пока не поддерживает. Про некоторые из них можно прочитать в книге М. Фаулера [10].

Но с первыми шестью вариантами рефакторинга, а также с командами, относящимися к добавлению в класс новых элементов (кроме последней), приходится встречаться очень часто. Поэтому рассмотрим эти случаи подробнее.

Первый случай — переименование элементов программы

В качестве примера используем код уже известного нам приложения 05_8_effectiveCalc. Но его мы трогать не будем, а станем проводить эксперименты с его копией 07_6_refactoring1.

Для того чтобы в среде NetBeans переименовать элемент (класс, интерфейс, метод, поле данных, переменную), следует щелкнуть по его имени правой кнопкой мыши. Например, мы хотим заменить имя **ApplicationCalc** на **ExampleCalc**. Это можно сделать в исходном коде программы, а можно и в окне **Projects** или **Navigator**. В появившемся контекстном меню следует выбрать **Refactor/Rename...**, после чего ввести новое имя и нажать кнопку **Preview** или **Refactor** (рис. 7.39).

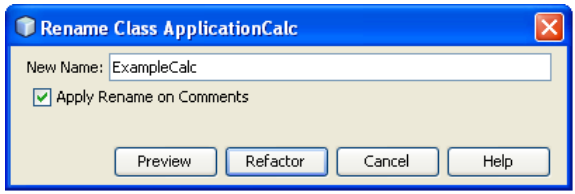


Рис. 7.39. Переименование класса. Шаг 2

Если после изменения в строке ввода New Name имени ApplicationCalc на ExampleCalc нажать кнопку **Preview** (Предварительный просмотр) в самом нижнем окне под редактором исходного кода, где мы обычно просматривали результаты вывода в окне **Output** (Вывод), то появится набор окон **Refactoring**.

Слева показывается дерево со списком мест, где будут проведены исправления (рис. 7.40), затем идет окно с первоначальным вариантом исходного кода и окно с вариантом, который будет после рефакторинга (рис. 7.40, в увеличенном масштабе рис. 7.41). В случае необходимости флажки в любых местах можно снять, и в этих местах рефакторинг проводиться не будет.

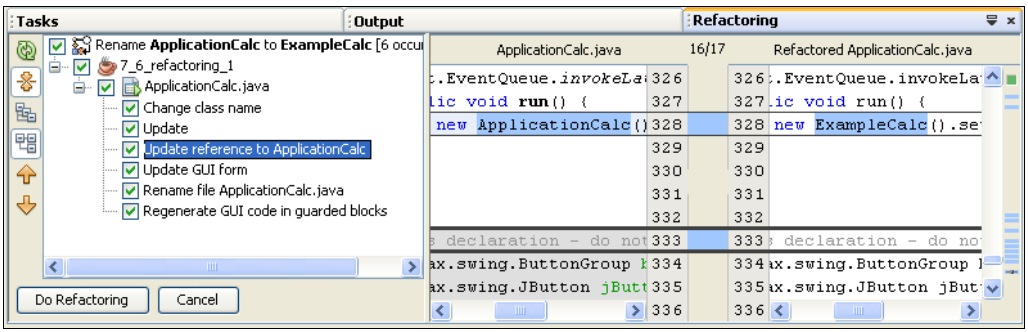


Рис. 7.40. Переименование класса. Шаг 3. Предварительный просмотр дерева изменений

Необходимо быть внимательными: довольно часто начинающие программисты не замечают появления окон **Refactoring** и кнопки **Do Refactoring**, особенно если высота окна **Output** сделана очень маленькой.

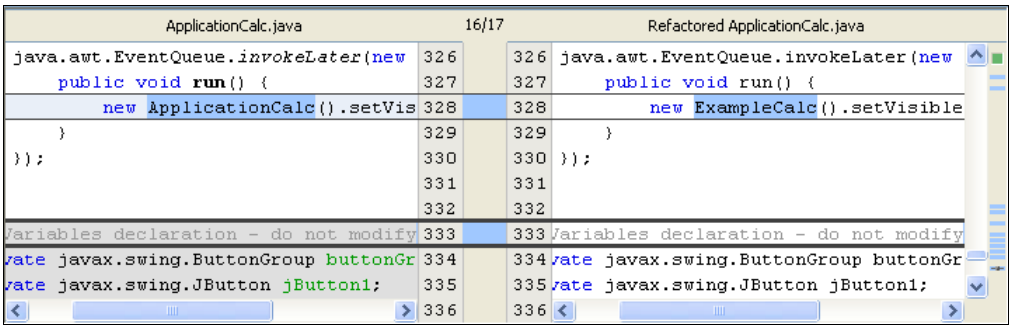


Рис. 7.41. Переименование класса. Предварительный просмотр изменений

Слева от дерева находится вертикальная панель инструментов.

Первая кнопка этой панели предназначена для обновления информации о местах рефакторинга в случае, если процесс рефакторинга не был доведен до конца, и происходило редактирование исходного кода.

Вторая кнопка предназначена для сворачивания всех узлов дерева и раскрытия их обратно.

Третья кнопка переключает вид дерева к логическому, четвертая — к физическому (он установлен по умолчанию).

Две нижних кнопки позволяют перейти к предыдущему месту рефакторинга или к следующему месту (вхождению — occurrence).

При нажатии кнопки **Do Refactoring** (Провести рефакторинг) проводится операция переименования в выбранных местах программы. Точно такой же эффект будет при нажатии кнопки **Refactor** в диалоговом окне шага 2 рефакторинга, но без предварительного просмотра возможных изменений и, что особенно важно, возможных проблем.

Когда в результате рефакторинга возникают ошибки, в дереве предварительного просмотра сообщается о проблемных местах и причинах ошибки. Поэтому при рефакторинге рекомендуется всегда пользоваться режимом предварительного просмотра.

В отличие от обычных текстовых процессоров переименование происходит с учетом синтаксиса программы, так что элементы, не имеющие отношения к переименовываемому, но имеющие такие же имена, не затрагиваются (что в выгодную сторону отличает NetBeans от многих других сред разработки, не говоря уже об обычных текстовых редакторах).

Следует отметить, что по окончании рефакторинга возможен возврат к первоначальному состоянию (откат, операция **Undo**). Обычно такая операция осуществляется с помощью главного меню проекта (кнопка **Undo** или пункт меню **Edit/Undo**), но в случае рефакторинга требуется правой кнопкой мыши вызвать контекстное меню и выбрать пункт **Refactor/Undo**.

Откат может быть на несколько шагов назад путем повторения данного действия. При необходимости отказа от отката в меню рефакторинга следует выбрать пункт **Redo** — с его помощью можно делать возврат вперед на один или более шагов.

В NetBeans 6.5 имеется еще один способ вызова рефакторинга-переименования — путем изменения имени непосредственно в дереве проекта. При щелчке мышью по имени

какого-либо класса или пакета появляется возможность непосредственно в этом месте изменить это имя, но после изменения имени и нажатия клавиши <Enter> появляется диалог, предлагающий провести рефакторинг.

Второй случай — перемещение класса

Например, мы хотим переместить класс из одного пакета в другой. Для этого достаточно перетащить мышью в окне **Projects** узел, связанный с данным классом, в соответствующий пакет. При таком перемещении там, где это необходимо, автоматически добавляются операторы импорта.

Если при перемещении возникают проблемы, о них выдается сообщение. Как правило, проблемы бывают связаны с неправильными уровнями видимости. Например, если указан пакетный уровень видимости метода, то он доступен другим классам этого пакета.

А при переносе класса в другой пакет в месте исходного кода, где осуществляется такой доступ, в новом варианте кода возникает ошибка доступа. Перенос класса в отдельный пакет, отличающийся от пакета приложения, — это хороший способ проверить правильность выбранных уровней доступа для членов класса.

Аналогично перемещают пакеты. При этом все пакеты в дереве элементов показываются на одном уровне вложенности, но у вложенных пакетов имена квалифицируются именем родительского пакета.

Для перемещения элементов проекта, кроме перетаскивания мышью в дереве проектов, можно также воспользоваться пунктом меню **Refactor/Move** (либо в главном меню NetBeans, либо в контекстном меню, вызываемом правой кнопкой мыши).

Третий случай — копирование класса для его дальнейшей модификации

Пункт **Copy** предназначен для создания копии класса с целью его дальнейшей модификации. Например, в нашем приложении **ApplicationCalc** (проект 07.6_refactoring) многие классы (`Fnc`, `Fnc1`, `Fnc1_x`, `FncSin` и т. д.), устроены однотипно. Новый класс для очередной функции, интеграл от которой мы хотим вычислить, проще всего создать с помощью пункта меню **Refactor/Copy**. Появится диалог проведения рефакторинга (рис. 7.42).

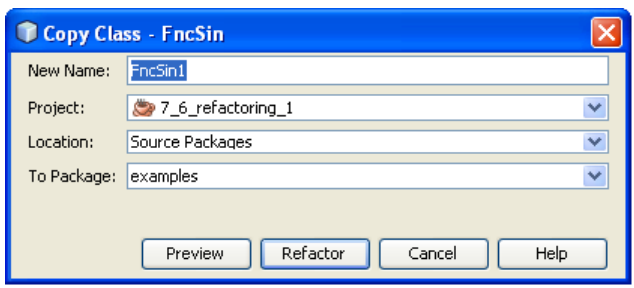


Рис. 7.42. Диалог рефакторинга для копирования класса

Имя для создаваемой копии класса по умолчанию образуется путем добавления символа 1 к имени оригинала. Причем копия помещается в тот же пакет, что и оригинал. Однако можно не только ввести требуемое имя, но и указать местоположение копии.

Четвертый случай — безопасное удаление элемента программы

Если в нашем приложении попытаемся удалить класс `Fnc1`, то приложение перестанет компилироваться. Пункт **Refactor/Safely Delete** (Провести рефакторинг/Безопасно удалить) позволяет до удаления убедиться, что оно не приведет к проблемам, ну а если они все же возникнут, то получить список соответствующих мест приложения.

Аналогично удаляют интерфейсы.

Пятый случай — изменение списка параметров метода

Установим в исходном коде в созданном в результате рефакторинга-копирования класса `FncSin1` курсор на имя функции `f1` (зададим ее путем обычного копирования тела и списка параметров функции `f`) и вызовем диалог рефакторинга-изменения списка параметров метода (рис. 7.43). Функцию `f` лучше не трогать, т. к. иначе произойдет изменение списка параметров функции `f` для всей иерархии.

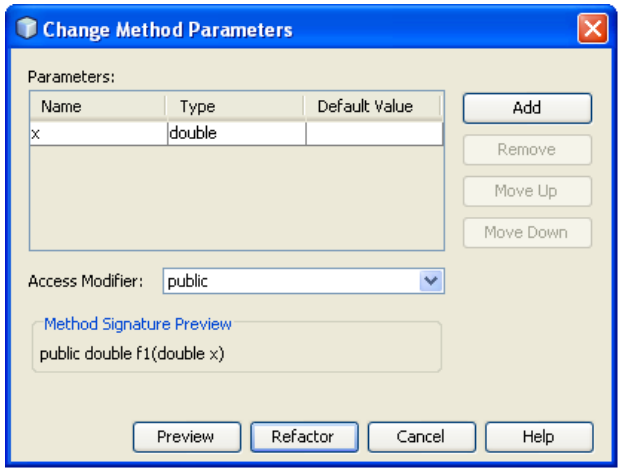


Рис. 7.43. Диалог изменения параметров метода

В диалоге в ячейках таблицы имеется возможность изменить имя параметра, его тип и значение параметра по умолчанию (по-видимому, в Java планируется добавление в методы параметров по умолчанию. Но пока этого нет, данное значение лучше оставлять пустым). В выпадающем списке можно выбрать необходимый модификатор доступа к методу.

Кроме того, с помощью кнопки **Add** можно добавлять новые параметры метода, а кнопка **Remove** позволяет удалять ненужный параметр, соответствующий выделенной строке в таблице. Например, если мы нажмем кнопку **Remove**, то в нашем методе не

останется параметров, а если нажмем кнопку **Add**, то в таблице появится вторая строка с параметрами, соответствующими второму параметру. По умолчанию добавляется параметр типа `Object` и значение по умолчанию `null`, но значения соответствующих клеток в таблице можно отредактировать, изменив на нужные. Кнопки **MoveUp** и **MoveDown** позволяют изменять порядок параметров.

Шестой случай — инкапсуляция полей данных

Как уже упоминалось, напрямую давать доступ к полю данных — дурной тон программирования. Но задание вручную геттеров и сеттеров — скучная утомительная работа. Поэтому при введении в класс новых полей на первом этапе удобнее задать поля с модификатором `public` и обеспечивать чтение значения полей напрямую, а его изменение — присвоением полям новых значений. А затем можно исправить данный недостаток программы с помощью рефакторинга — инкапсуляцией полей данных.

Например, если мы перейдем в класс `Integrator` и вызовем данный тип рефакторинга, то появится диалог, позволяющий выбрать поля, которые необходимо инкапсулировать (рис. 7.44).

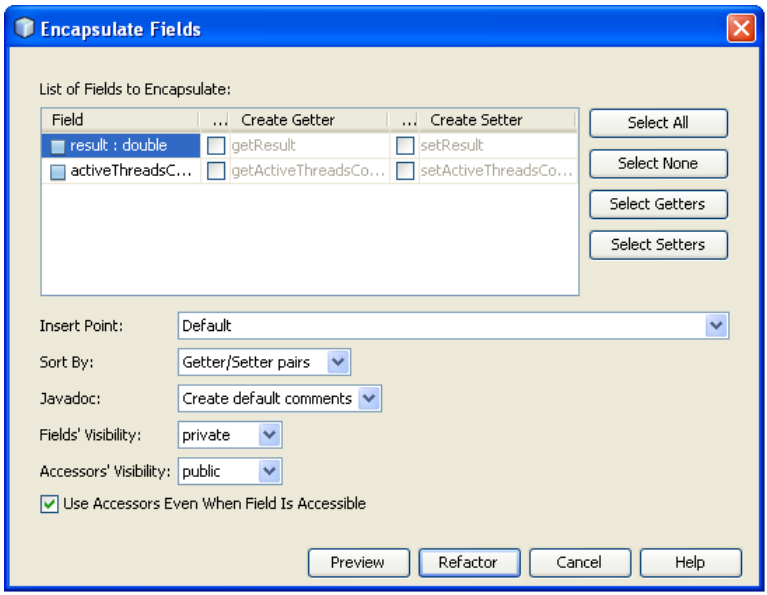


Рис. 7.44. Диалог, позволяющий выбрать поля, которые необходимо инкапсулировать

При этом видимость инкапсулируемых полей по умолчанию меняется на `private`, а видимость геттера и сеттера (вместе их часто называют *аксессорами*, accessors, от англ. *access* — доступ) задается как `public`. Но эти значения можно изменить.

Опция **Use Accessors Even When Field Is Accessible** (Использовать аксессоры даже если поле доступно), стоящая по умолчанию, заставляет проводить замену доступа к полю на соответствующий метод (геттер или сеттер) даже в тех случаях, когда по правилам видимости поле доступно напрямую. Вообще говоря, для private-полей это не лучшее решение, т. к. из-за такого опосредованного доступа скорость работы програм-

мы может заметно снизиться. А вот если модификатор видимости поля установлен другим, такое действие становится крайне желательным для обеспечения надежной работы класса.

Более подробную информацию по идеологии и методам рефакторинга проектов, написанных на языке Java, можно найти в монографии [10]. Правда, эта книга уже несколько устарела — среда NetBeans позволяет делать в автоматическом режиме многие из описанных в [10] действий.

Краткие итоги по главе

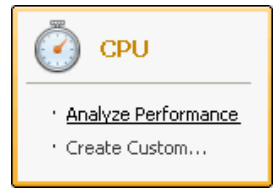
- *Проектирование классов* осуществляется с помощью UML-диаграмм.
- С помощью средств прямого проектирования (Forward Engineering) можно создавать и изменять исходный код классов путем создания и редактирования UML-диаграмм.
- С помощью средств обратного проектирования (Reverse Engineering) можно создавать UML-диаграммы классов и зависимостей классов. Причем после создания UML-проекта, сопровождающего Java-проект, изменения, сделанные в исходном коде Java, автоматически применяются к диаграммам UML, и наоборот.
- *Рефакторинг* — изменение структуры существующего проекта с сохранением его функциональности. Шесть наиболее часто встречающихся примеров рефакторинга:
  - **Rename** — переименование элементов программы (пакетов, классов, полей данных, методов, переменных);
  - **Move** — перемещение классов с одного места на другое;
  - **Copy** — копирование класса для его дальнейшей модификации;
  - **Safely Delete** — безопасное удаление элемента программы;
  - **Change Method Parameters** — изменение списка параметров метода;
  - **Encapsulate Fields** — инкапсуляция полей данных.
- *Аксессоры* — это методы, обеспечивающие доступ к полю данных по чтению *(геттер)* и по записи *(сеттер)*.

Типичные ошибки

- Пытаются переименовать "пакет по умолчанию", помеченный в дереве проектов как <default package>. Вместо этого следует создать новый пакет и переместить в него содержимое пакета по умолчанию.
- Забывают, что в первый раз после выполнения обратного проектирования в получившемся UML-проекте отсутствуют UML-диаграммы. Их требуется сгенерировать, щелкнув правой кнопкой мыши по имени соответствующего класса.
- При выборе режима **Preview** во время проведения рефакторинга не замечают появления окон **Refactoring** и кнопки **Do Refactoring**, особенно если высота окна **Output** установлена очень маленькой. И недоумевают, что же делать.
- Забывают, что при рефакторинге операции **Undo** и **Redo** выполняются через пункты меню **Refactor**, а не с помощью обычных кнопок и пунктов меню **Undo** и **Redo**.

Задания

- Завершить проектирование иерархии фигур из примера 07_4_3_UML_Figures. По UML-диаграмме сгенерировать Java-код для созданной иерархии.
- На основе примера 07_5_JavaLibrary_Figure/07_5_JavaLibrary_Figure-Model создать приложение **MyFigures**, а в нем — пакет **figures**. С помощью средств рефакторинга переместить классы фигур в пакет **figures**.
- С помощью средств **Reverse Engineering** создать UML-диаграммы классов и зависимости классов для разработанного пакета **figures**.



ГЛАВА 8

Важнейшие объектные типы

8.1. Массивы

Массив (array) — это упорядоченный набор одинаково устроенных ячеек, доступ к которым осуществляется по индексу. Например, если у массива имя `a1`, то `a1[i]` — имя ячейки этого массива, имеющей индекс `i`.

В Java массивы являются объектами, но особого рода — их объявление отличается от объявления других видов объектов. Переменная типа `массив` ссылочная — в ней содержится адрес объекта, а не сам объект, как и для всех других объектных переменных в Java. В качестве элементов (ячеек) массива могут выступать значения как примитивных, так и ссылочных типов, в том числе переменных типа `массив`.

Тип ячейки называется *базовым типом* для массива.

Для задания массива, в отличие от объектов других типов, не требуется предварительно определять класс и иметь специальное имя для данного объектного типа. Вместо имени класса при объявлении переменной указывают имя базового типа, после которого идут пустые квадратные скобки.

Например, объявление:

```
int[] a1;
```

задает переменную `a1` типа `массив`.

При этом размер массива (число ячеек в нем) заранее не задается и не является частью типа.

Для того чтобы создать объект типа `массив`, следует воспользоваться зарезервированным словом `new`, после чего указать имя базового типа, а за ним — в квадратных скобках число ячеек в создаваемом массиве:

```
a1=new int[10];
```

Можно совместить объявление типа переменной и создание массива:

```
int[] a1=new int[10];
```

После создания массивы Java всегда инициализированы — в ячейках содержатся нули. Поэтому если базовый тип массива примитивный, то элементы массива будут нулями соответствующего типа. А если базовый тип ссылочный, то в ячейках будут значения `null`.

Ячейки в массиве имеют индексы, всегда начинающиеся с нуля (т. е. первая ячейка имеет номер 0, вторая — номер 1 и т. д.). Если число элементов в массиве равно `n`, то последняя ячейка имеет индекс `n-1`. Такая своеобразная нумерация принята в языках C и C++, а язык Java унаследовал эту особенность, часто приводящую к ошибкам при организации циклов.

Длина массива хранится в поле `length`, которое доступно только по чтению — изменять его путем присваивания нового значения нельзя, т. к. длина уже созданного объекта-массива фиксирована.

Пример работы с массивом:

```
int[] a=new int[100];
for(int i=0;i<a.length;i++){
  a[i]=i+1;
}
```

Если у нас имеется переменная типа `массив`, и ей сопоставлен массив заданной длины, то в любой момент этой переменной можно сопоставить новый массив, например

```
a1=new int[20];
```

При этом прежний объект-массив, находящийся в динамической области памяти, будет утерян и превратится в мусор.

Переменные типа `массив` можно присваивать друг другу.

Например, если мы задали переменную:

```
int[] a2;
```

то изначально в ней хранится значение `null` (ссылка направлена "в никуда") (рис. 8.1).

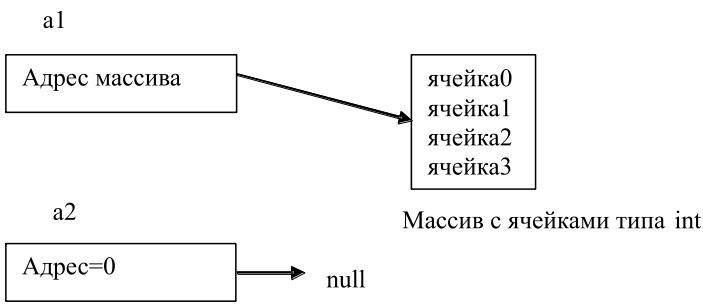


Рис. 8.1. Работа со ссылочными переменными типа `массив` (состояние до присваивания)

Присваивание

```
a2=a1;
```

приведет к тому, что ссылочные переменные `a1` и `a2` будут ссылаться на один и тот же массив, расположенный в динамической области памяти (рис. 8.2), т. е. присваивание переменных типа `массив` приводит к тому, что имена переменных становятся синонимами одного и того же массива — копируется адрес массива, а вовсе не элементы из одного массива в другой, как это бывает в некоторых других языках программирования.

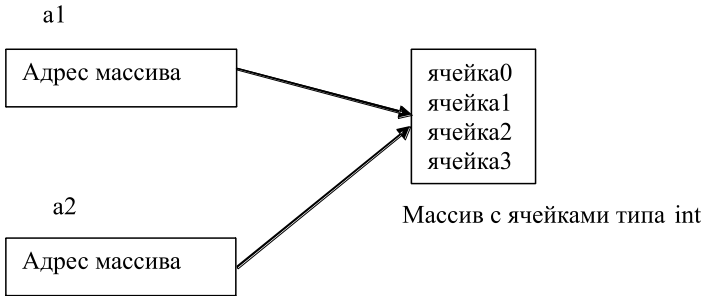


Рис. 8.2. Работа со ссылочными переменными типа `массив` (состояние после присваивания)

В качестве элементов массивов могут выступать объекты. В этом случае доступ к полям и методам этих объектов производится через имя ячейки массива, после которого через точку указывается имя поля или метода. Например, если у нас есть класс `Circle` (окружность) с полями `x`, `y` и `r`, а также методами `show()` и `hide()`, то массив `circles` из 10 объектов такого типа может быть задан и инициализирован, например, так, как в листинге 8.1.

Листинг 8.1. Задание и инициализация массива из 10 объектов

```
int n=10;
Circle[] circles=new Circle[n];
for(int i=0;i<n;i++){
 circles[i]=new Circle();
 circles[i].x=40*i;
 circles[i].y= circles[i].x/2;
 circles[i].r=50;
 circles[i].show();
}
```

В такого рода программах для повышения читаемости часто используется вспомогательная ссылка, позволяющая предотвратить многократное обращение по индексу. В нашем случае мы применяем в этих целях переменную `circle`. На скорость работы программы это почти не влияет (хотя и может немного повысить быстродействие), но код делается более читаемым (листинг 8.2).

Листинг 8.2. Применение вспомогательной ссылки

```
int n=10;
Circle[] circles=new Circle[n];
Circle circle;
for(int i=0;i<n;i++){
 circle=new Circle();
 circle.x=40*i;
 circle.y= circle.x/2;
 circle.r=50;
```

```
 circle.show();
 circles[i]= circle;
}
```

С помощью переменной `circle` мы инициализируем создаваемые объекты и показываем их на экране, после чего присваиваем ссылку на них ячейкам массива.

Двумерный массив представляет собой массив ячеек, каждая из которых имеет тип `одномерный массив`. Соответствующим образом он и задается. Например, задание двумерного массива целых чисел будет выглядеть так:

```
int[][] a=new int[10][20];
```

Будет задана ячейка типа `двумерный массив`, а также создан и назначен этой ссылочной переменной массив, имеющий по первому индексу 10 элементов, а по второму 20 (т. е. мы имеем 10 ячеек типа `одномерный массив`, каждая из которых ссылается на массив из 20 целых чисел). При этом базовым типом для ячеек по первому индексу является `int[]`, а для ячеек по второму индексу `int`.

Рассмотрим работу с двумерными массивами на примере заполнения двумерного массива случайными числами (листинг 8.3).

Листинг 8.3. Заполнение двумерного массива случайными числами

```
int m=10;//10 строк
int n=20;//20 столбцов
int[][] a=new int[m][n];
for(int i=0;i<m;i++){     //цикл по строкам
    for(int j=0;j<n;j++){    //цикл по столбцам
        a[i][j]=(int)(100*Math.random());
        System.out.print(a[i][j]+"  ");
    }
    System.out.println();//перевод на новую строку после вывода строки
                         //матрицы
}
```

Многомерные массивы задают аналогично двумерным, только указывают необходимое количество прямоугольных скобок. Следует отметить, что массивы, имеющие размерность более 3, встречаются крайне редко.

Обычно в двумерных и многомерных массивах задают одинаковый размер всех массивов, связанных с ячейками по какому-либо индексу. Такие массивы называют регулярными. В Java, в отличие от большинства других языков программирования, возможны двумерные и многомерные массивы с разным размером составляющих их массивов, связанных с ячейками по какому-либо индексу. Такие "непрямоугольные" массивы называют *иррегулярными*. Обычно они позволяют сэкономить память. Работать с иррегулярными массивами следует особенно аккуратно, т. к. разный размер вложенных массивов часто приводит к ошибкам при реализации алгоритмов.

Пример задания иррегулярного двумерного массива треугольной формы иллюстрирует листинг 8.4.

Листинг 8.4. Задание иррегулярного двумерного массива треугольной формы

```
int n=9;
int[][] a=new int[n][];
for(int i=0;i<a.length;i++){     //цикл по строкам
    a[i]=new int[i+1];           //число элементов в строке равно i+1
    for(int j=0;j<a[i].length;j++){ //цикл по столбцам
        a[i][j]=100*i+j;
        System.out.print(a[i][j]+"  ");
    }
    System.out.println();//перевод на новую строку после вывода
                         //строки матрицы
 }
```

После создания массива необходимо его инициализировать — записать нужные значения в ячейки. До сих пор мы делали это путем задания значений в цикле по некоторой формуле, однако часто требуется задать конкретные значения. Конечно, можно это сделать в виде:

```
int[] a=new int[4];
a[0]=2;
a[1]=0;
a[2]=0;
a[3]=6;
```

Но гораздо удобнее другой вариант синтаксиса:

```
int[] a=new int[] {2,0,0,6};
```

При этом приходится задавать массив без указания его размера непосредственно с помощью указания значений в фигурных скобках.

В правой части оператора присваивания стоит так называемый *анонимный массив* (или, что то же, *безымянный массив* — у него нет имени). Такие массивы обычно используют для инициализации, а также при написании кода для различного рода проверок.

Если мы хотим присвоить новые значения, то придется либо присваивать поэлементно, либо создавать новый объект:

```
a=new int[] {2,0,0,6};
```

Двумерные и многомерные массивы инициализируют посредством вложенных массивов, задаваемых с помощью фигурных скобок.

Например, фрагмент кода из листинга 8.5 приведет к заданию целочисленного двумерного массива `b`, состоящего из шести строк и четырех столбцов, т. е. `int[6][4]`.

Листинг 8.5. Инициализация двумерного массива с помощью анонимного массива

```
int[][] b= new int[][]
{
  {2,0,0,0}, //это b[0]
  {2,0,0,1}, //это b[1]
  {2,0,0,2}, //это b[2]
```

```
  {1,0,0,0}, //это b[3]
  {2,0,0,0}, //это b[4]
  {3,0,0,0}, //это b[5]
};
```

Подобным образом можно задавать как регулярные, так и иррегулярные массивы. Но следует помнить, что в таком варианте синтаксиса проверка правильности размера массива по индексам не делается, что может привести к ошибкам.

Например, при компиляции кода листинга 8.6 ошибка не возникнет, а будет создан иррегулярный массив.

Листинг 8.6. Не диагностируемая ошибка при задании массива

```
int[][] b= new int[][]
{
  {2,0,0,0}, //это b[0]
  {2,0,0,1}, //это b[1]
  {2,0,0,2}, //это b[2]
  {1,0,0,0}, //это b[3]
  {2,0,0,0}, //это b[4]
  {3,0,0}, //это b[5] — массив из трех элементов
};
```

Из объектов-массивов можно вызывать метод `clone()`, позволяющий создавать копию (клон) массива:

```
a=new int[] {2,0,0,6};
int[] a1=a.clone();
```

Напомним, что присваивание

```
int[] b=a;
```

не приведет к копированию массива — просто переменная `b` станет ссылаться на тот же объект-массив.

Копировать массивы можно в цикле, но лучше использовать метод `System.arraycopy`, который работает гораздо быстрее:

```
int[] b=new int[a.length+10];
System.arraycopy(a,index1a,b, index1b,count);
```

Из `a` в `b` копируется `count` элементов, начиная с индекса `index1a` в массиве `a`. Они размещаются в массиве `b`, начиная с индекса `index1b`. Содержимое остальных элементов `b` не меняется. Для использования метода требуется, чтобы массив `b` существовал и имел необходимую длину (при выходе за границы массивов возбуждается исключительная ситуация — *см. разд. 8.8*).

Быстро заполнить массив одинаковыми значениями позволяет метод `Arrays.fill(массив, значение)`. Класс `Arrays` расположен в пакете `java.util`.

Поэлементное сравнение массива следует выполнять с помощью метода `Arrays.equals(a,a1)`.

Заметим, что у любого массива имеется метод `equals`, унаследованный от класса `Object` и позволяющий сравнивать массивы. Но, к сожалению, метод не переопределен и сравнение идет по адресам объектов, а не по содержимому. Поэтому `a.equals(a1)` это то же самое, что `a==a1`. Оба сравнения вернут `false`, т. к. адреса объектов, на которые ссылаются переменные `a` и `a1`, различаются, хотя содержимое массива `a1` скопировано с `a`. Напротив, `Arrays.equals(a,a1)` в нашем случае возвратит `true`.

Сортировка (упорядочение по значениям) массива `a` производится методами `Arrays.sort(a)` и `Arrays.sort(a,index1,index2)`. Первый из них упорядочивает в порядке возрастания весь массив, второй — часть элементов (от индекса `index1` до индекса `index2`). Имеются и более сложные методы сортировки. Элементы массива должны быть сравниваемы (поддерживать операцию сравнения).

`Arrays.deepEquals(a1,a2)` — сравнение на равенство содержимого массивов объектов `a1` и `a2` путем глубокого сравнения (на равенство содержимого, а не ссылок, на произвольном уровне вложенности).

В JDK 6 появились две новые функции копирования массивов:

- `Arrays.copyOf(оригинальный_массив,новая_длина)` — возвращает массив-копию новой длины. Если новая длина меньше оригинальной, то массив усекается до этой длины, а если больше, то дополняется нулями (что рассматривается как нулевые значения соответствующего типа);
- `copyOfRange(оригинальный_массив,начальный_индекс,конечный_индекс)` — также возвращает массив-копию новой длины, при этом копируется часть оригинального массива от начального индекса до конечного –1.

Функция `Arrays.toString(массив)` возвращает строковое представление массива со строковым представлением элементов, заключенных в квадратные скобки. Заметим, что вызов `массив.toString()` возвратит результат в виде, очень далеком от такого представления *(см. разд. 8.4)*.

Также в классе `Arrays` содержится функция `asList` преобразования массива в список, функция `binarySearch` поиска индекса элемента с заданным значением и ряд других полезных методов.

8.2. Коллекции, списки, итераторы

В Java получили широкое использование *коллекции* (Collections) — "умные" массивы с динамически изменяемой длиной, поддерживающие ряд важных дополнительных операций по сравнению с обычными массивами. Базовым для иерархии коллекций является класс `java.util.AbstractCollection` (в общем случае класс коллекции не обязан быть потомком `AbstractCollection` — он может являться любым классом, реализующим интерфейс `Collection`).

Основные классы и интерфейсы (особый вид полностью абстрактных классов, *см. главу 9*) коллекций:

- интерфейсы `Set`, `SortedSet`; классы `HashSet`, `TreeSet`, `EnumSet`, `LinkedHashSet` — множества (неупорядоченные или некоторым образом упорядоченные наборы неповторяющихся элементов);

- интерфейс `java.util.List`; классы `java.awt.List`, `ArrayList`, `LinkedList`, `Vector` — списки (упорядоченные наборы элементов, причем элементы могут повторяться в разных местах списка);
- интерфейсы `Map`, `SortedMap`; классы `HashMap`, `HashTable`, `ConcurrentHashMap`, `TreeMap`, `EnumMap`, `Properties` — таблицы (списки пар "имя-значение").

Доступ к элементам коллекции в общем случае не может осуществляться по индексу, т. к. не все коллекции поддерживают индексацию элементов. Эту функцию осуществляют с помощью специального объекта — итератора (iterator). У каждой коллекции (collection) есть свой итератор, который умеет с ней работать, поэтому итератор вводят следующим образом:

```
Iterator iter = collection.iterator()
```

У итераторов имеются следующие три метода:

- `boolean hasNext()` — дает информацию, имеется ли в коллекции следующий объект;
- `Object next()` — возвращает ссылку на следующий объект коллекции;
- `void remove()` — удаляет из коллекции текущий объект, т. е. тот, ссылка на который была получена последним вызовом `next()`.

Листинг 8.7 содержит пример преобразования массива в коллекцию и цикл с доступом к элементам этой коллекции, осуществляемый с помощью итератора.

Листинг 8.7. Преобразование коллекции в массив и использование итератора

```
java.util.List components=
  java.util.Arrays.asList(this.getComponents());

for (Iterator iter = components.iterator();iter.hasNext();) {
      Object elem = (Object) iter.next();
      javax.swing.JOptionPane.showMessageDialog(null,"Компонент: "+
                                                elem.toString());
}
```

В табл. 8.1 приведены основные методы коллекций.

***Таблица 8.1.** Основные методы коллекций*

Имя метода	Действие
`boolean add(Object obj)`	Добавление объекта в коллекцию (в конец списка). Возвращает `true` в случае успешного добавления (изменения коллекции). Коллекция может не позволить добавление элементов несовместимого типа или не подходящих по какому-либо другому признаку
`boolean addAll(Collection c)`	Добавление в коллекцию всех объектов из другой коллекции. Возвращает `true` в случае успешного добавления (т. е. если добавлен хотя бы один элемент)

Таблица 8.1 *(продолжение)*

Имя метода	Действие
`void clear()`	Очистка коллекции — удаление из нее ссылок на все входящие в коллекцию объекты. При этом те объекты, на которые имеются ссылки у других элементов программы, не удаляются из памяти
`boolean contains(Object obj)`	Возвращает `true` в случае, если коллекция содержит объект `obj`. Проверка осуществляется с помощью поочередного вызова метода `obj.equals(e)` для элементов `e`, входящих в коллекцию
`boolean containsAll(Collection c)`	Возвращает `true` в случае, если коллекция содержит все элементы коллекции `c`
`boolean isEmpty()`	Возвращает `true` в случае, если коллекция пуста (т. е. не содержит ни одного элемента)
`Iterator iterator()`	Возвращает ссылку на итератор — объект, позволяющий получать поочередный доступ к элементам коллекции. Для одной коллекции разрешается иметь произвольное число объектов-итераторов (в том числе разных типов). В процессе работы они могут указывать на разные элементы коллекции. После создания итератор всегда указывает на начало коллекции — вызов его метода `next()` дает ссылку на начальный элемент коллекции
`boolean remove(Object obj)`	Удаляет из коллекции первое встретившееся вхождение объекта `obj`. Поиск и удаление осуществляется с помощью итератора. Возвращает `true` в случае, если удаление удалось (т. е. если коллекция изменилась)
`boolean remove All(Collection c)`	Удаляет из коллекции все элементы коллекции `c`. Возвращает `true` в случае, если удаление удалось (т. е. если коллекция изменилась)
`boolean retainAll(Collection c)`	Оставляет в коллекции только те из входящих в нее элементов, которые входят в коллекцию `c`
`int size()`	Возвращает число элементов в коллекции
`Object[] toArray()`	Возвращает массив ссылок на объекты, содержащиеся в коллекции (т. е. преобразует коллекцию в массив)
`T[] toArray(T[n] a)`	Возвращает массив элементов типа `T`, полученных в результате преобразования элементов, содержащихся в коллекции (т. е. преобразует коллекцию в массив). Если число элементов коллекции не превышает размер `n` массива `a`, то размещение данных производится в существующих ячейках памяти, отведенных под массив. Если число элементов превышает `n`, то в памяти динамически создается и заполняется новый набор ячеек, а их число делается равным числу элементов коллекции, после чего переменная `a` начинает ссылаться на новый набор ячеек

Таблица 8.1 *(окончание)*

Имя метода	Действие
`String toString()`	Метод переопределен — он возвращает строку со списком элементов коллекции. В списке выводятся заключенные в квадратные скобки строковые представления элементов, разделяемые комбинацией ", " (запятая с пробелом после нее)

Самые распространенные варианты коллекций — списки (`Lists`) — во многом похожи на массивы, но отличаются от них тем, что в списках основными операциями являются добавление и удаление элементов, а не доступ к элементам по индексу, как в массивах.

Интерфейс `java.util.List` наследует от интерфейса `Collection` методы коллекций (про интерфейсы и их наследование *см. главу 9*) и добавляет к ним ряд дополнительных методов:

- `list.get(i)` — получение ссылки на элемент списка `list` по индексу `i`;
- `list.indexOf(obj)` — получение индекса элемента `obj` в списке `list`. Возвращает –1, если объект не найден;
- `list.listIterator()` — получение ссылки на итератор типа `ListIterator`, обладающий дополнительными методами по сравнению с итераторами типа `Iterator`;
- `list.listIterator(i)` — то же с позиционированием итератора на элемент с индексом `i`;
- `list.remove(i)` — удаление из списка элемента с индексом `i`;
- `list.set(i,obj)` — замена в списке элемента с индексом `i` на объект `obj`;
- `list.subList(i1,i2)` — возвращает ссылку на подсписок, состоящий из элементов списка с индексами от `i1` до `i2`.

Кроме них, в интерфейсе `List` продекларированы и многие другие полезные методы.

Ряд полезных методов для работы с коллекциями содержит класс `Collections`:

- `Collections.addAll(c,e1,e2,...,eN)` — добавление в коллекцию `c` произвольного числа элементов `e1,e2,...,eN`;
- `Collections.frequency(c,obj)` — возвращает число вхождений элемента `obj` в коллекцию `c`;
- `Collections.reverse(list)` — обращает порядок следования элементов в списке `list` (первые становятся последними и наоборот);
- `Collections.sort(list)` — сортирует список в порядке возрастания элементов. Сравнение идет вызовом метода `e1.compareTo(e2)` для очередных элементов списка `e1` и `e2`.

Кроме них, в классе `Collections` имеются и многие другие полезные методы.

В классе `Arrays` имеется метод:

`Arrays.asList(a)` — возвращает ссылку на список элементов типа `T`, являющийся оболочкой над массивом `T[] a`. При этом и массив, и список содержат одни и те же элементы, а изменение элемента списка приводит к изменению элемента массива, и наоборот.

При работе с коллекциями многие методы используют сравнение объектов с помощью метода `equals`. Например, проверка того, содержится ли в коллекции `collection` объект `obj`, осуществляемая методом `collection.contains(obj)`, неявно использует метод `equals`.

В Java очень распространены коллекции, в имени которых как составная часть имеется слово Hash (в переводе с английского — "мелко нарезать"). Такие коллекции для ускорения доступа к элементам и их сравнения используют специальную форму хранения элементов — коллекцию "нарезают" на набор небольших подтаблиц (откуда, собственно, и пошло название). Причем каждому элементу сопоставляется хэш-код, "почти однозначно" характеризующий этот элемент. Элемент в хэш-коллекции ищется не путем перебора элементов, а по его хэш-коду, получаемому методом `hashCode()`. О нем было написано в *разд. 6.8*. Основное преимущество хэш-коллекций по сравнению с массивами заключается в гораздо большей скорости поиска элементов. Например, если необходимо проверить, содержится ли объект в массиве, содержащем `n` объектов, в среднем потребуется `n/2` проверок. То есть время проверки будет пропорционально `n`. А для хэш-коллекций гарантируется, что время проверки будет пропорционально логарифму от `n`. Для больших `n` это дает выигрыш в скорости до нескольких порядков.

Если в имени класса коллекции как составная часть содержится слово Linked (в переводе с английского — "связный"), таблицы, входящие в состав коллекции, основаны на связных списках. Если же в имени есть слово Array — таблицы с элементами коллекции основаны на использовании массивов. Если коллекция является списком (List), она содержит только одну таблицу с элементами, и размер этой таблицы может быть произвольно большим (он задается длиной списка). А вот хэш-коллекции всегда содержат таблицы с относительно небольшим числом элементов, обычно от нескольких сотен до нескольких тысяч, но таких таблиц может быть произвольное количество (определяется числом элементов в коллекции).

8.3. Перебор в цикле элементов коллекций. Оператор цикла *for-each*

В Java старых версий (до JDK 5) отсутствовала специальная форма оператора `for` для перебора в цикле элементов коллекции (или, что то же самое, наборов). Тем не менее, оператор `for` позволял и в этих случаях последовательно обработать все элементы массива или набора.

Рассмотрим пример вывода в текстовую область `jTextArea1` значений свойств компонентов, являющихся элементами массива компонентов панели главной формы приложения (пример 08_3_for_each, листинг 8.8).

Листинг 8.8. Перебор элементов коллекции в цикле `for`

```
java.util.List components=
    java.util.Arrays.asList(this.rootPane.getComponents());
for (java.util.Iterator iter = components.iterator();iter.hasNext();) {
    Object elem = (Object) iter.next();
    jTextArea1.setText(jTextArea1.getText()+"Component: "+
                       elem.toString()+"\n");
}
```

В данном примере переменной components назначается ссылка на список компонентов главной панели формы GUI_application, являющейся наследником класса javax.swing.JFrame. В секции инициализации цикла for переменной iter назначается ссылка на объект-итератор для данного списка. В секции проверки условия продолжения цикла стоит оператор iter.hasNext(), т. е. цикл продолжается, пока итератор не дошел до конца списка (еще имеются не перебранные в цикле элементы). Секция изменения счетчиков цикла for пуста, поскольку переход итератора к следующему элементу списка осуществляется в теле цикла оператором iter.next().

В версии языка Java, соответствующей JDK 5, появился вариант цикла for, который по традиции, воспринятой из JavaScript и C#, был назван for-each (для каждого). Достаточно указать в качестве параметров цикла for имя переменной, которая будет по очереди перещелкиваться на очередной элемент массива (или коллекции), а затем через двоеточие указать тот массив (или коллекцию), элементы которого должны перебираться. В нашем случае это elem:components. Читается: элемент elem в коллекции components.

Цикл из листинга 8.8 теперь можно записать в виде листинга 8.9.

Листинг 8.9. Перебор элементов коллекции в цикле for-each

```
java.util.List components=
    java.util.Arrays.asList(this.rootPane.getComponents());
for (Object elem:components) {
    jTextArea1.setText(jTextArea1.getText()+"Component: "+
                       elem.toString()+"\n");
}
```

8.4. Работа со строками в Java. Строки как объекты. Классы *String*, *StringBuffer* и *StringBuilder*

Класс String инкапсулирует действия со строками. Объект типа String — это строка, состоящая из произвольного числа символов (от 0 до $2 \cdot 10^9$). Литерные константы типа String представляют собой последовательности символов, заключенные в двойные кавычки:

"A", "abcd", "abcd", "Мама моет раму", " ".

Это так называемые длинные строки. Внутри литерной строковой константы не разрешается использовать ряд символов — вместо них применяются управляющие последовательности.

Внутри строки допустимы переносы на новую строку. Но литерные константы с такими переносами запрещены, и нужно ставить управляющую последовательность "\n". К сожалению, такой перенос строки не срабатывает при показе текста в большинстве компонентов (JLabel,JButton и др.). Но для компонентов JTextArea такой перенос строки действует.

Разрешены пустые строки, не содержащие ни одного символа. Например, если задана строковая переменная

```
String s;
```

то после присваивания

```
s="";
```

эта переменная будет ссылаться на пустую строку.

Отметим, что существует принципиальная разница между ситуацией, когда строковая переменная не инициализирована (как у нас `s` сразу после объявления), и ситуацией, когда она ссылается на пустую строку. В первом случае в переменной хранится значение `null`, во втором — она ссылается на полноценный объект-строку, просто объект хранит пустую последовательность символов. Но во втором случае из этого объекта можно вызывать методы. Например, `s.length()` возвратит длину строки, равную нулю. А попытка вызвать любой метод (хотя бы ту же функцию `s.length()`) в случае, когда `s` содержит значение `null`, приведет к возбуждению исключения `NullPointerException` — *см. разд. 8.8.*

В языке Java строковый и символьный типы несовместимы:

- "A" — строка из одного символа;
- 'A' — число с ASCII-кодом символа "A".

Это несколько усложняет совместную работу со строками и символами.

Строки можно складывать.

Если `s1` и `s2` — строковые литерные константы или переменные, то результатом операции `s1+s2` будет строка, являющаяся сцеплением (конкатенацией) строк, хранящихся в `s1` и `s2`.

Например, в результате операции:

```
String s="Это "+"моя строка";
```

в переменной `s` будет храниться строковое значение "`Это моя строка`".

Для строк разрешен оператор "`+=`". Для строковых операндов `s1` и `s2` выражение `s1+=s2` эквивалентно выражению `s1=s1+s2`.

Любая строка (набор символов) является объектом — экземпляром класса `String`. Переменные типа `String` — это ссылки на объекты, что следует учитывать при передаче параметров строкового типа в подпрограммы, а также при многократных изменениях строк. При каждом изменении строки в динамической области памяти создается новый объект, а прежний превращается в мусор. Поэтому при многократных изменениях строк в цикле возникает много мусора, что нежелательно.

Очень частая ошибка — попытка сравнения строк с помощью оператора `==`.

Например, результатом выполнения листинга 8.10 будет вывод в консольное окно строки:

```
s1 не равно s2
```

т. к. объекты-строки имеют в памяти разные адреса.

Листинг 8.10. Проблема сравнения строк на равенство

```
String s1="Строка типа String";
String s2="Строка";
s2+=" типа String";
if(s1==s2)
    System.out.println("s1 равно s2");
else
    System.out.println("s1 не равно s2");
```

Оператор `equals` сравнивает строки по содержанию.

Поэтому если бы вместо `s1==s2` мы написали `s1.equals(s2)`, то получили бы ответ:

```
s1 равно s2.
```

Удивительным может показаться факт, что результатом выполнения листинга 8.11 будет вывод в консольное окно строки:

```
s1 равно s2.
```

Листинг 8.11. Проблема сравнения на равенство литерных констант

```
String s1="Строка";
String s2="Строка";
if(s1==s2)
    System.out.println("s1 равно s2");
else
    System.out.println("s1 не равно s2");
```

Дело в том, что компилятор Java анализирует имеющиеся в коде программы литерные константы и на этапе компиляции для одинаковых по содержанию констант использует одни и те же объекты-строки. И даже если задать переменную `s2` как

```
String s2 = "Стр"+"ока";
```

мы получим тот же результат. Но вот если динамически создать новый независимый объект с помощью конструктора

```
String s2 =new String("Строка");
```

или, например, прочитать значение `s2` из пункта редактирования текста

```
String s2 = jTextArea1.getText();
```

мы получим сообщение, что `s1` не равно `s2`.

В остальных случаях то, что строковые переменные ссылочные, обычно никак не влияет на работу со строковыми переменными, и с ними можно действовать так, как если бы они содержали сами строки.

В классе `String` есть много конструкторов, предназначенных для преобразования текстовой информации из первоначального формата в объект-строку. Перечислим некоторые из них.

Пусть:

- `s1`, `s2`, `charsetName` и `subS` имеют тип `String`;
- `charArray` — массив символов `char[]`;

- `byteArray` — массив символов `byte[]`;
- `i`, `index` и `count` (счет, количество) — целочисленные переменные или значения.

Тогда:

- `String()` — создание объекта-строки с пустым содержимым строки. Присваивание `s1=new String()` эквивалентно присваиванию `s=""` (за исключением того, что объект с пустой строкой для литерной константы "" хранится в единственном экземпляре, как разъяснялось ранее, а конструктор `String()` всегда создает новый, отличный от прежних объект с пустым содержимым строки).
- `String(s1)` — создание копии объекта-строки, на которую ссылается строковая переменная `s1`.
- `String(charArray)` — массив символов копируется в объект-строку.
- `String(charArray,index,count)` — то же, что в предыдущем случае, но копируется `count` символов, начиная с символа, имеющего позицию `index`. При выходе за границы массива возбуждается исключение `IndexOutOfBoundsException` — *см. разд. 8.8*.
- `String(stringBuffer)` — копирование последовательности символов объекта-строки `stringBuffer`, имеющей тип `StringBuffer`, в объект типа `String`. Про класс `StringBuffer` рассказывается далее.
- `String(stringBuilder)` — копирование последовательности символов объекта-строки `stringBuilder`, имеющей тип `StringBuilder`, в объект типа `String`. Про класс `StringBuilder` рассказывается далее.
- `String(byteArray)` — массив байтов копируется и преобразуется (декодируется) в объект-строку платформо-зависимым образом. При этом считается, что в байтовом массиве находится последовательность символов из набора (charset), установленного по умолчанию (default) для данного компьютера. В связи с этим результат декодирования будет гарантированно неправильным на компьютерах с другим национальным набором символов по умолчанию. Поэтому данный вариант конструктора имеет смысл использовать только для дешифровки последовательностей англоязычных символов.
- `String(byteArray,index,count)` — то же, что в предыдущем случае, но декодируется `count` байт начиная с байта, имеющего позицию `index`. При выходе за границы массива возбуждается исключение `IndexOutOfBoundsException` — *см. разд. 8.8*.
- `String(byteArray,charsetName)` — массив байтов копируется и преобразуется (декодируется) в объект-строку платформо-независимым образом. При этом считается, что в байтовом массиве находится последовательность символов из набора символов, заданного с помощью строки `charsetName`. Например, могут быть использованы имена "`ISO-8859-1`" (стандартный латинский алфавит в 8-битовой кодировке), "`UTF-8`", "`UTF-16`" (символы Unicode) и др. Если указанное имя набора символов не поддерживается, возбуждается исключение `UnsupportedEncodingException` — *см. разд. 8.8*.
- `String(byteArray,index,count,charsetName)` — то же, что в предыдущем варианте конструктора, но декодируется не весь байтовый массив, а `count` байтов начиная с байта, имеющего позицию `index`. При выходе за границы массива возбуждается исключение `IndexOutOfBoundsException` — *см. разд. 8.8*.

Кроме конструкторов в классе `String` имеется ряд других методов класса. Важнейшие из них:

- `String.valueOf(параметр)` — возвращает строку типа `String`, являющуюся результатом преобразования параметра в строку. Параметр может быть любого примитивного или объектного типа.
- `String.valueOf(charArray, index, count)` — функция, аналогичная предыдущей в случае, когда параметр является массивом символов, но здесь преобразуется `count` символов, начиная с символа, имеющего индекс `index`.

В классе `String` предусмотрен ряд методов объектов, т. е. таких методов, которые могут быть вызваны из объектов типа `String`. Перечислим важнейшие из них, разбив на несколько категорий.

Общие характеристики строки:

- `s1.length()` — возвращает длину строки (число 16-битовых символов Unicode, содержащихся в строке). Длина пустой строки равна нулю;
- `s1.isEmpty()` — возвращает `true` в случае, если строка, на которую ссылается переменная `s1`, пустая. Если строка не пустая, возвращается `false`;
- `s1.intern()` — возвращает ссылку на объект-строку, содержащуюся в хранилище строковых констант и имеющую такую же последовательность символов. Если же такой строки в хранилище не найдено, объект-строка, на который ссылается `s1`, добавляется в хранилище, и возвращается ссылка на нее. Данный метод имеет смысл использовать для экономии памяти только при хранении очень большого количества объектов-строк.

Преобразование строки в массив:

- `s1.getBytes()` — возвращает массив типа `byte[]`, полученный в результате платформо-зависимого преобразования символов строки в последовательность байтов;
- `s1.getBytes(charsetName)` — то же, но с указанием кодировки (`charset`). Например, могут быть использованы имена `"ISO-8859-1"`, `"UTF-8"`, `"UTF-16"` (символы Unicode) и др.;
- `s1.toCharArray()` — возвращает массив символов, скопированных из строки `s1`.

Преобразование регистра строки:

- `s1.toLowerCase()` — возвращает строку с символами, скопированными из строки `s1` и преобразованными к нижнему регистру (строчным буквам). Имеется вариант метода, делающего такое преобразование с учетом конкретной кодировки (`locale`);
- `s1.toUpperCase()` — возвращает строку с символами, скопированными из строки `s1` и преобразованными к верхнему регистру (заглавным буквам). Имеется вариант метода, делающего такое преобразование с учетом конкретной кодировки (`locale`).

Сравнение строк и их частей:

- `s1.equals(s2)` — возвращает `true` в случае, когда строка `s2` имеет тип `String` и ее содержимое (последовательность символов) совпадает с содержимым строки `s1`. В случае несовпадения последовательностей символов `s1` и `s2` или если `s2` имеет тип, отличный от `String`, возвращается `false`;

Внимание!

При сравнении с буферизованными строками `StringBuffer` и `StringBuilder` *(см. далее)* функция всегда возвращает `false`, даже если содержимое строки `s2` совпадает с содержимым строки `s1`. Для сравнения с содержимым строк `StringBuffer` и `StringBuilder` необходимо использовать функцию `contentEquals`.

- `s1.equalsIgnoreCase(s2)` — то же, но при сравнении строк игнорируются различия в регистре символов (строчные и заглавные буквы не различаются);
- `s1.contentEquals(sb1)` — булева функция, проверяющая на совпадение содержимого строки `s1` типа `String` и объекта `sb1`. В случае если `sb1` — строка, имеющая тип `String`, `StringBuffer` или `StringBuilder` *(см. далее)*, и ее содержимое (последовательность символов) совпадает с содержимым `s1`, возвращается `true`. Иначе возвращается `false`;
- `s1.startsWith(subS)` — возвращает `true` в случае, когда строка `s1` начинается с символов строки `subs`;
- `s1.startsWith(subs,index1)` — возвращает `true` в случае, когда символы строки `s1` с позиции `index1` начинаются с символов строки `subs`;
- `s1.endsWith(subS)` — возвращает `true` в случае, когда строка `s1` заканчивается последовательностью символов, содержащихся в строке `subS`;
- `s1.regionMatches(index1,s2,index2,count)` — булева функция, сравнивающая подстроки двух строк, `s1` и `s2`. Сравнение начинается с позиции `index1` для первой строки и с позиции `index2` для второй, сравнивается `count` символов. В случае совпадения последовательностей символов возвращается `true`, иначе `false`. При недопустимых значениях `index1`, `index2` или `count` возвращается `false`;
- `s1.regionMatches(ignoreCase,index1,s2,index2,count)` — то же, но с дополнительным булевым параметром `ignoreCase`. Если он имеет значение `false`, то сравнение происходит так же, как в предыдущем методе (регистрочувствительно). Если `true` — сравнение идет регистронечувствительно, т. е. заглавные и строчные буквы считаются эквивалентными, и регистр букв не влияет на результат сравнения.

Копирование части строки:

- `s1.substring(index1)` — возвращает строку типа `String` с символами, скопированными из строки `s1`, начиная с позиции `index1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` — *см. разд. 8.8*;
- `s1.substring(index1,index2)` — возвращает строку типа `String` с символами, скопированными из строки `s1`, начиная с позиции `index1` и кончая позицией `index2-1`. Длина получившеейся строки равна `index2-index1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` — *см. разд. 8.8*;
- `s1.subSequence(index1,index2)` — то же, что предыдущий метод, но возвращаемое значение имеет более общий тип `CharSequence`. Поэтому ссылку на возвращаемую строку можно присваивать как переменным типа `String`, так и переменным типа `StringBuffer` и `StringBuilder` *(см. далее)*, а также подставлять в методы в качестве параметров соответствующих типов.

Поиск подстрок в строке:

- `s1.indexOf(subS)` — индекс позиции, где в строке `s1` первый раз встретилась последовательность символов `subS`. Если подстрока не найдена, возвращается значение `-1`;
- `s1.indexOf(subS,i)` — то же, но поиск начинается с символа, имеющего индекс `i`;
- `s1.lastIndexOf(subS)` — индекс позиции, где в строке `s1` последний раз встретилась последовательность символов `subS`. Если подстрока не найдена, возвращается значение `-1`;
- `s1.lastIndexOf(subS,i)` — то же, но поиск вхождения подстроки идет к началу строки с позиции `i`, а не с конца строки.

Замена подстрок в строке:

- `s1.replaceFirst(oldSubS,newSubS)` — возвращает строку на основе строки `s1`, в которой произведена замена первого вхождения символов строки `oldSubS` на символы строки `newSubS`;
- `s1.replaceAll(oldSubS,newSubS)` — возвращает строку на основе строки `s1`, в которой произведена замена всех вхождений символов строки `oldSubS` на символы строки `newSubS`;
- `s1.trim()` — возвращает копию строки `s1`, из которой убраны ведущие и завершающие пробелы.

Расщепление строки на массив строк:

- `s1.split(separator)` — возвращает массив строк `String[]`, полученный разделением строки `s1` на независимые строки по местам вхождения сепаратора, задаваемого строкой `separator`. При этом символы, содержащиеся в строке `separator`, в получившиеся строки не входят. Пустые строки из конца получившегося массива удаляются;
- `s1.split(separator, i)` — то же, но положительное `i` задает максимально допустимое число элементов массива. В этом случае последним элементом массива становится окончание строки `s1`, которое не было расщеплено на строки вместе с входящими в это окончание символами сепараторов.

  При `i = 0` ограничений нет, но пустые строки из конца получившегося массива удаляются.

  При `i < 0` ограничений нет, а пустые строки из конца получившегося массива не удаляются.

Работа со строками поддерживается не только классом `String`, но и многими другими классами. В классе `Object` имеется метод `toString()`, обеспечивающий строковое представление объекта. Конечно, выводимая им строка не может отражать все особенности объекта, а является представлением содержимого объекта по мере возможности. В самом классе `Object` это представление обеспечивает возврат методом полного имени класса (квалифицированное именем пакета), затем идет символ `@`, после которого следует число — хэш-код объекта (число, однозначно характеризующее данный объект во время сеанса работы) в шестнадцатеричном представлении. Поэтому во всех классах-наследниках, где этот метод не переопределен, он возвращает такую же конструкцию.

Во многих стандартных классах этот метод переопределен. Например, для числовых классов метод `toString()` обеспечивает вывод строкового представления соответствующего числового значения. Для строковых объектов возвращает саму строку, а для символьных (тип `char`) — символ.

При операциях `+` и `+=` с операндами, один из которых строковый, а другой нет, метод `toString()` вызывается автоматически для нестрокового операнда. В результате получается сложение (конкатенация) двух строк.

Для преобразования значения типа `char` в строку нельзя воспользоваться вызовом метода `toString()`, т. к. соответствующая переменная или литерная константа не является объектом. Но с помощью сложения с пустой строкой можно преобразовать значение типа `char` в строковое. Пусть `ch1` — переменная или значение типа `char`. Тогда `ch1+""` будет объектом типа `String`.

При сложении числовых величин со строками следует быть очень внимательными, т. к. результат сложения более чем двух слагаемых может оказаться сильно отличающимся от ожидаемого.

Например

```
String s=1+2+3;
```

даст вполне ожидаемое значение `s=="6"`.

А вот присваивание

```
String s="Сумма ="+1+2+3;
```

даст не очень понятное начинающим программистам значение:

`"Сумма =123"`.

Дело в том, что в первом случае сначала выполняются арифметические сложения, а затем результат преобразуется в строку и присваивается левой части.

А во втором случае сначала производится сложение:

`"Сумма ="+1`.

Первый операнд строковый, а второй — числовой. Поэтому для второго операнда вызывается метод `toString()` и складываются две строки.

Результатом будет строка:

`"Сумма =1"`.

Затем складывается строка `"Сумма =1"` и число `2`. Опять для второго операнда вызывается метод `toString()` и складываются две строки.

Результатом будет строка:

`"Сумма =12"`.

Совершенно так же выполняется сложение строки `"Сумма =12"` и числа `3`.

Еще более странный для начинающих программистов результат получится при присваивании

```
String s=1+2+" не равно "+1+2;
```

Следуя изложенной ранее логике, мы получаем, что результатом будет строка:

```
"3 не равно 12"
```

Совместная работа с типами `String` и `char` требует особой аккуратности. Например, несмотря на то, что тип `char` является числовым, и `'S'==83`, в результате присваивания

```
String s='S'+"";
```

в переменной `s` окажется значение "`S`", а не "`83`". Это происходит из-за того, что символ `'S'` сначала автоматически преобразуется в объект оболочечного типа `Character`, который методом `toString()` возвращает строковое значение "`S`".

А в результате присваивания

```
String s=0+'S'+"";
```

в переменной `s` окажется значение "`83`", поскольку `0+'S'` является числом типа `int` и имеет значение `83`. А затем сумма `83+""` преобразуется в строку "`83`".

Аналогично, в результате присваивания

```
String s='S'+"t"+"ring";
```

в переменной `s` окажется значение "`String`". А в результате присваивания

```
String s='S'+'t'+"ring";
```

в переменной `s` окажется значение "`199ring`", т. к. сначала складываются два числовых значения — `'S'`, равное `83`, и `'t'`, равное `116`. Затем получившееся число `199` складывается со строкой "`ring`" и получается "`199ring`".

До сих пор были приведены простейшие примеры, для которых все достаточно очевидно. Если же в такого рода выражениях присутствуют числовые и строковые функции, да еще с оператором `? :`, то результат может оказаться совершенно непредсказуемым.

Кроме указанных, имеется ряд строковых операторов, заданных в оболочечных числовых классах. Например, мы уже хорошо знаем методы преобразования строковых представлений чисел в числовые значения:

```
Byte.parseByte(строка)
Short.parseShort(строка)
Integer.parseInt(строка)
Long.parseLong(строка)
Float.parseFloat(строка)
Double.parseDouble(строка)
```

и метод, преобразующий строковые представления чисел в числовые объекты, — экземпляры оболочечных классов `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`:

```
valueOf(строка)
```

Например:

```
Byte.valueOf(строка)
```

и т. п.

Кроме того, имеются методы классов `Integer` и `Long` для преобразования чисел в двоичное и шестнадцатеричное строковое представление:

```
Integer.toBinaryString(число)
Integer.toHexString(число)
Long.toBinaryString(число)
Long.toHexString(число)
```

Существует возможность обратного преобразования — из строки в объект соответствующего класса (`Byte`, `Short`, `Integer`, `Long`) с помощью метода `decode`:

```
Byte.decode(строка)
```

и т. п.

Также полезны методы для анализа отдельных символов, имеющиеся в классах `String` и `Character`:

- `s1.charAt(i)` — символ в строке `s1`, имеющий индекс `i` (индексация начинается с нуля). Метод возбуждает исключительную ситуацию `IndexOutOfBoundsException` при выходе за пределы допустимых индексов — *см. разд. 8.8*;
- `s1.indexOf(символ)` — индекс позиции, где в строке `s1` первый раз встретился символ;
- `s1.indexOf(символ,i)` — индекс позиции, начиная с `i`, где в строке `s1` первый раз встретился символ. Если символ не найден или при выходе за пределы допустимых индексов возвращается значение `-1`;
- `s1.lastIndexOf(символ)` — индекс позиции, где в строке `s1` последний раз встретился символ;
- `s1.lastIndexOf(символ,i)` — то же, но поиск вхождения символа идет к началу строки с позиции `i`, а не с конца строки;
- `Character.isDigit(символ)` — булева функция, проверяющая, является ли символ цифрой;
- `Character.isLetter(символ)` — булева функция, проверяющая, является ли символ буквой;
- `Character.isLetterOrDigit(символ)` — булева функция, проверяющая, является ли символ буквой или цифрой;
- `Character.isLowerCase(символ)` — булева функция, проверяющая, набран ли символ в нижнем регистре;
- `Character.isUpperCase(символ)` — булева функция, проверяющая, набран ли символ в верхнем регистре;
- `Character.isWhitespace(символ)` — булева функция, проверяющая, является ли символ "пробелом в широком смысле" (пробелом, символом табуляции, символом перехода на новую строку и т. д.).

Для того чтобы сделать работу с многочисленными присваиваниями более эффективной, используются классы `StringBuffer` и `StringBuilder`. Они особенно удобны в тех случаях, когда требуется проводить изменения внутри одной и той же строки (убирать или вставлять символы, менять их местами, заменять одни на другие). Изменение значений переменных этого класса не приводит к созданию мусора, и обычно происходит значительно быстрее, чем при работе с переменными типа `String`.

В примере работы со строками, который будет рассмотрен в *главе 17*, благодаря использованию переменных типа `StringBuffer` вместо переменных типа `String` удалось по-

высить быстродействие приложения примерно в 30 раз и уменьшить потребление памяти в 4000 раз. Эта ситуация типична для задач, требующих многократного изменения содержимого строк.

Класс `StringBuffer` рекомендуется для тех случаев, когда используются потоки выполнения (`threads`), этот класс, в отличие от `String` и `StringBuilder`, обеспечивает синхронизацию строк *(см. главу 10)*. Класс `StringBuilder`, введенный в JDK 5, полностью ему подобен, но синхронизации не поддерживает. Зато обеспечивает бóльшую скорость работы со строками.

К сожалению, совместимости по присваиванию между переменными типа `String`, `StringBuffer` и `StringBuilder` нет. Но в классах `StringBuffer` и `StringBuilder` имеется метод `sb.append(s)`, позволяющий добавлять в конец буферизуемой строки `sb` (имеющей тип `StringBuffer` или `StringBuilder`) обычную строку `s`. Также имеется метод `sb.insert(index,s)`, позволяющий вставлять, начиная с места символа, имеющего индекс `index`, строку `s`. Значения типа `StringBuffer` или `StringBuilder` преобразуются в тип `String` с помощью метода `toString()` (листинг 8.12).

Листинг 8.12. Работа со строками типа `StringBuffer`

```
StringBuffer sb=new StringBuffer();
sb.append("типа StringBuffer");
sb.insert(0, "Строка ");
System.out.println(sb);
String s=sb.toString();
```

Кроме строк, в методы `append` и `insert` можно подставлять в качестве параметров значения различных примитивных и объектных типов. Значение параметра сначала методом `String.valueOf(значение)` автоматически преобразуется в значение типа `String`, после чего выполняется метод `append` или `insert`, соответственно, для параметра типа `String`.

Самый простой способ преобразования объекта-строки в объект типа `StringBuffer` — это использовать конструктор `StringBuffer(String str)`. Например, в предыдущем фрагменте кода вместо создания объекта с пустым содержимым и последующего добавления строки методом можно было бы выполнить вызов следующего конструктора:

```
StringBuffer sb=new StringBuffer("типа StringBuffer");
```

Обычные строки можно сравнивать с буферизуемыми на совпадение содержания. Для этого предусмотрена булева функция `contentEquals`.

Вызов `s1.contentEquals(sb)` возвращает `true` в случае, когда строка `s1` содержит такую же последовательность символов, как и строка `sb`.

У буферизуемых строк (объектов типа `StringBuffer` или `StringBuilder`) помимо длины имеется дополнительное поле `capacity` ("емкость") — размер буфера под символы строки. Естественно, значение `capacity` всегда больше или равно длине строки `length`. При присваиваниях размер буфера при необходимости увеличивается автоматически с некоторым запасом.

Приведем описание важнейших методов для класса `StringBuffer` (для класса `StringBuilder` все совершенно аналогично, но, как уже упоминалось, методы не являются синхронизованными).

Конструкторы:

- `StringBuffer()` — создание объекта-строки `StringBuffer` с пустым содержимым строки и размером буфера в 16 символов.
- `StringBuffer(capacity)` — то же, что в предыдущем случае, но размер буфера задается равным значению параметра `capacity`, имеющего тип `int`.
- `StringBuffer(s1)` — создание объекта-строки `StringBuffer` с копией содержимого объекта-строки типа `String`, на которую ссылается строковая переменная `s1`.
- `StringBuffer(sb)` — создание объекта-строки `StringBuffer` с копией содержимого буферизуемой строки `sb`, имеющей тип `StringBuffer` или `StringBuilder`.

Конструкторы, позволяющие преобразовывать массивы символов и массивы байтов, а также их подмассивы, в объекты-строки типа `StringBuffer`, в отличие от класса `String`, отсутствуют.

Также, в отличие от класса `String`, в классах `StringBuffer` и `StringBuilder` кроме конструкторов нет других методов класса.

Функции, аналогичные `String.valueOf(параметр)` и `String.valueOf(charArray,index, count)`, отсутствуют.

В классах `StringBuffer` и `StringBuilder` есть множество методов объектов, хотя пока и не обеспечивающее всех возможностей, имеющихся для класса `String`. Перечислим важнейшие из них, разбив на те же категории, что и для класса `String`.

Общие характеристики строки:

- `sb.length()` — возвращает длину строки (число 16-битовых символов Unicode, содержащихся в строке). Длина пустой строки равна нулю;
- аналог метода `isEmpty()`, поддерживаемого классом `String`, отсутствует, но может быть легко заменен проверкой `sb.length()==0`;
- аналог метода `intern()`, поддерживаемого классом `String`, отсутствует.

Преобразование строки в массив или строку другого типа:

- `sb.getChars(index1,index2,charArray,index)` — возвращает массив типа `char[]`, полученный в результате копирования символов строки `sb` в массив `charArray`. Копирование начинается с символа, имеющего индекс `index1`, и заканчивается символом с индексом `index2-1`. В массиве символы копируются на место, начинающееся с позиции `index`. Количество скопированных символов равно `index2-index1`, индекс последнего скопированного символа в массиве `charArray` равен `index+index2-index1-1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` — *см. разд. 8.8*;
- `sb.toString()` — копирование символов строки типа `StringBuffer` в символы строки типа `String`;
- аналог метода `getBytes(charsetName)`, поддерживаемого классом `String`, отсутствует.

Преобразование регистра строки:

- аналоги методов `toLowerCase()` и `toUpperCase()`, поддерживаемых классом `String`, отсутствуют.

Сравнение строк и их частей:

- `sb.equals(subS)` — метод не переопределен, а просто унаследован от класса `Object`. Поэтому он не только бесполезен, но даже опасен, т. к. вводит программистов в заблуждение — его лучше заменять эквивалентным оператором сравнения `==`. А для сравнения содержимого обычной строки с буферизованной необходимо использовать булеву функцию `s1.contentEquals(sb1)`. В классах `StringBuffer` и `StringBuilder` метод `contentEquals`, к сожалению, отсутствует. Поэтому эффективно сравнивать содержимое этих строк друг с другом пока нельзя;
- аналог метода `s1.equalsIgnoreCase(s2)` отсутствует;
- аналоги методов `startsWith` и `endsWith` отсутствуют;
- аналог метода `regionMatches` отсутствует.

Копирование части строки:

- `sb.charAt(index)` — возвращает символ с индексом `index`;
- `sb.subSequence(index1,index2)` — возвращает строку с символами, скопированными из строки `sb`, начиная с позиции `index1` и кончая позицией `index2-1`. Длина получившейся строки равна `index2-index1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` *(см. разд. 8.8)*. Возвращаемое значение имеет тип `CharSequence`, прародительский для классов `String`, `StringBuffer` и `StringBuilder`. Поэтому ссылку на возвращаемую строку можно присваивать как переменным типа `String`, так и переменным типа `StringBuffer` или `StringBuilder`, а также подставлять в методы в качестве параметров соответствующих типов. Метод может быть заменой методу `substring` *(см. далее)* — для работы без использования строк типа `String`;
- `sb.substring(index1)` — возвращает строку типа `String` с символами, скопированными из строки `sb`, начиная с позиции `index1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `StringIndexOutOfBoundsException` *(см. разд. 8.8)*;
- `sb.substring(index1,index2)` — возвращает строку типа `String` с символами, скопированными из строки `sb`, начиная с позиции `index1` и кончая позицией `index2-1`. Длина получившейся строки равна `index2-index1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `StringIndexOutOfBoundsException` *(см. разд. 8.8)*.

Поиск подстрок в строке:

- `sb.indexOf(subS)` — индекс позиции, где в строке `sb` первый раз встретилась последовательность символов подстроки `subS`, имеющей тип `String`. Если подстрока не найдена, возвращается значение `-1`;
- `sb.indexOf(subS,i)` — то же, но поиск начинается с индекса `i`;
- `sb.lastIndexOf(subS)` — индекс позиции, где в строке `sb` последний раз встретилась последовательность символов подстроки `subS`, имеющей тип `String`. Если подстрока не найдена, возвращается значение `-1`;
- `sb.lastIndexOf(subS,i)` — то же, но поиск вхождения подстроки идет к началу строки с позиции `i`, а не с конца строки.

Варианты этих методов, использующие поиск по буферизуемой подстроке, отсутствуют.

Замена подстрок в строке:

- `sb.replace(index1,index2,newSubS)` — заменяет в строке `sb` символы в диапазоне от `index1` до `index2-1` на символы строки `newSubS`, имеющей тип `String`. Возвращает ссылку на строку `sb` (но ее можно не использовать, т. к. она уже хранится в переменной `sb`). Варианты метода, осуществляющие поиск по буферизуемой подстроке, отсутствуют. Данный метод обеспечивает дополнительные возможности по сравнению с методом `replaceAll`, поддерживаемым классом `String`;
- аналог метода `replaceFirst` отсутствует;
- аналог метода `trim()` отсутствует.

Расщепление строки на массив строк:

- аналог метода `split` отсутствует.

Несмотря на отсутствие в классах `StringBuffer` и `StringBuilder` многих методов, поддерживаемых классом `String`, нельзя считать, что эти классы обеспечивают меньшую функциональность, чем `String`, поскольку в них имеется ряд методов, основанных на изменяемости содержимого строки. С помощью таких изменяемых (mutable) строк можно эффективно выполнять действия, которые невозможно выполнить с помощью неизменяемых (immutable) строк типа `String`. Перечислим эти методы:

- `sb.append(v)` — добавляет в конец строки строковое представление параметра `v`, который может быть любого объектного или примитивного типа. Параметр сначала преобразуется в строку методом `String.valueOf(v)`, после чего результат добавляется в конец строки;
- `sb.insert(index,v)` — вставляет в строку на место с позицией `index` строковое представление параметра `v`, который может быть любого объектного или примитивного типа. Параметр сначала преобразуется в строку методом `String.valueOf(v)`, после чего результат вставляется в соответствующее место строки. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` *(см. разд. 8.8)*;
- `sb.insert(index,sb2,index1,index2)` — вставляет в строку `sb` на место с позицией `index` часть строки `sb2`, задаваемую диапазоном индексов от `index1` до `index2-1`. Строка `sb2` может иметь тип `String`, `StringBuffer`, `StringBuilder` или любой другой, являющийся наследником `CharSequence`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` *(см. разд. 8.8)*;
- `sb.deleteCharAt(index)` — удаляет из строки `sb` символ с индексом `index`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `StringIndexOutOfBoundsException` *(см. разд. 8.8)*;
- `sb.setCharAt(index,символ)` — заменяет в строке `sb` символ с индексом `index` на новое значение `символ`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `IndexOutOfBoundsException` — *см. разд. 8.8*;
- `sb.delete(index1,index2)` — удаляет из строки `sb` символы с индексами от `index1` до `index2-1`. При выходе за пределы допустимых индексов возбуждается исключительная ситуация `StringIndexOutOfBoundsException` — *см. разд. 8.8*;

- `sb.setLength()` — установка новой длины строки;
- `sb.capacity()` — возвращает текущий размер буфера под символы строки;
- `sb.ensureCapacity(n)` — установка новой емкости строки (размера буфера), задаваемой переменной `n`, имеющей тип `int`. Если текущий размер буфера равен `old_n` символов, то новый размер буфера становится равным наибольшему из двух значений: `n` и `2*old_n+2`. Если `n` отрицательно или равно нулю, никаких действий не производится;
- `sb.trimToSize()` — пытается уменьшить размер буфера строки `sb` до минимально допустимого размера;
- `sb.reverse()` — инвертирует последовательность символов в строке `sb`, т. е. первый символ становится последним, второй — предпоследним и т. д. Возвращает ссылку на строку `sb` (но ее можно не использовать, т. к. она уже хранится в переменной `sb`).

Подведем краткие итоги.

Класс `String` дает два преимущества по сравнению с `StringBuffer` и `StringBuilder`. Во-первых, этот класс обеспечивает наибольшее количество методов для работы со строками, поэтому алгоритмы с использованием строк типа `String` оказываются более простыми и прозрачными. Во-вторых, метод `intern()` позволяет заметно экономить память при хранении строковых данных с большим числом повторяющихся строк (например, имен людей или почтовых адресов). В то же время крайне неэффективная работа со строками типа `String` при изменении их содержимого в циклах заметно ограничивает возможности этого класса.

Но неизменяемость объектов-строк типа `String` приводит к крайне неэффективной работе с памятью и весьма медленной обработке таких строк *(см. главу 17)*. Лучшие результаты во многих случаях дают экземпляры классов `StringBuffer` и `StringBuilder`, но они пока не обеспечивают всей необходимой функциональности, и приходится использовать строки типа `String`.

8.5. Типы-перечисления (enum)

Иногда требуются элементы, которые не являются ни числами, ни строками, но ведут себя как имена элементов и одновременно обладают порядковыми номерами. Например, названия месяцев или дней недели. В этих случаях используют *перечисления*. Для задания типа какого-либо перечисления следует написать зарезервированное слово `enum` (сокращение от enumeration — перечисление), после которого — имя задаваемого типа, а затем — в фигурных скобках через запятую элементы перечисления. В качестве элементов можно использовать любые простые идентификаторы (не содержащие квалификаторов вида `имя1.имя2`).

Тип-перечисление обязан быть глобальным — он может задаваться либо на уровне пакета, либо в каком-либо классе. Но его нельзя задавать внутри какого-либо метода. Элементы перечисления могут иметь любые имена, в том числе совпадающие в разных перечислениях или совпадающие с именами классов или их членов (каждое перечисление имеет свое собственное *пространство имен*). Доступ к элементу перечисления осуществляется с квалификацией именем типа-перечисления:

```
ИмяТипа.имяЭлемента
```

У каждого элемента перечисления имеется порядковый номер, соответствующий его положению в наборе (нумерация начинается с нуля). Поэтому первый элемент имеет номер 0, второй элемент — номер 1 и т. д. Имеется функция `ordinal()`, возвращающая порядковый номер элемента в перечислении. Также имеется функция `compareTo()`, позволяющая сравнивать два элемента перечисления (она возвращает разницу в их порядковых номерах).

Строковое представление значения можно получить с помощью функции `name()`. Преобразование из строки в значение типа "перечисление" осуществляется с помощью функции класса `valueOf`, в которую передается строковое представление значения.

Если требуется рассматривать элементы перечисления как массив, можно воспользоваться функцией `values()` — она возвращает массив элементов, к которым можно обращаться по индексу.

Формат вызова функции такой:

```
ИмяТипа.values()
```

Для примера зададим типы-перечисления `Months` (месяцы) и `Spring` (весна), соответствующие различным наборам месяцев:

```
enum Months {jan, feb, mar, apr, may, jun, jul, aug, sept, oct, nov, dec}
enum Spring {march, apr, may }
```

Названия месяцев мы намеренно пишем со строчной буквы, чтобы было понятно, что это идентификаторы переменных, а не типы. А имя месяца марта написано по-разному в типах `Months` и `Spring`, чтобы показать независимость их пространств имен.

Переменные типа "перечисление" объявляются так же, как и все остальные типы, при этом переменные могут быть как неинициализированы, так и инициализированы:

```
public Months m1 ,m2=Months.mar, m3;
```

(при задании в классе общедоступных полей `m1`, `m2` и `m3`),

```
Spring spr1=Spring.apr, spr2;
```

(при задании в методе локальной переменной или задании в классе поля `spr1` с пакетным уровнем доступа).

После этого возможны следующие операторы:

```
spr2=spr1;
spr1=Spring.may;
System.out.println("Результат сравнения="+spr2.compareTo(Spring.march));
```

В переменной `spr2` оказывается значение `Spring.apr`, порядковый номер которого на единицу больше, чем у значения `Spring.march`, с которым идет сравнение. Поэтому после выполнения этих операторов в консольное окно будет выведен текст:

```
Результат сравнения=1
```

Пусть в переменной `spr2` хранится значение `Spring.may`. Порядковый номер значения, хранящегося в переменной, можно получить с помощью вызова `spr2.ordinal()`. Он возвратит число 2, т. к. `may` — третий элемент перечисления (сдвиг на единицу получается из-за того, что нумерация начинается с нуля).

Строковое представление значения, хранящегося в переменной spr2, можно получить с помощью вызова spr2.name(). Он возвратит строку may — имя типа в возвращаемое значение не входит.

Если переменная типа "перечисление" не инициализирована, в ней хранится значение null.

Поэтому вызов

```
System.out.println("spr2="+spr2);
```

осуществленный до присваивания переменной spr2 значения, возвратит строку:

```
spr2=null
```

А вот попытки вызовов spr2.ordinal() или spr2.name() приведут к возникновению ошибки (исключительной ситуации) с диагностикой примерно такого вида:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
```

Получение значения типа Spring по номеру, хранящемуся в переменной i, осуществляется так:

```
spr1=Spring.values()[i];
```

Преобразование из строки в значение типа Spring будет выглядеть так:

```
spr1=Spring.valueOf("march");
```

Необходимо отметить, что перечисления — объектные типы — наследники класса Enum, несмотря на отличный от других классов синтаксис задания. Элементы перечисления являются переменными класса.

8.6. Работа с датами и временем. Класс *GregorianCalendar*

Мы уже знаем функцию System.currentTimeMillis(), позволяющую узнать текущее время в миллисекундах, прошедшее с момента 0 часов 0 минут 1 января 1970 года.

В старых версиях Java при работе с временем и датами обычно использовались классы java.util.Time и java.util.Date. В настоящее время почти все методы этих классов обозначены как устаревшие и нерекомендуемые (deprecated), хотя сами классы пока использовать можно. Тем не менее, без устаревших методов классы Time и Date нефункциональны, и их следует заменять классом java.util.GregorianCalendar.

Конструктор GregorianCalendar() создает объект с датой и временем, соответствующими дате и времени для компьютера, на котором создан объект, на момент создания объекта. Также имеются варианты конструкторов, позволяющие создавать объекты с заданной датой и временем. После вызова метода clear(), очищающего все поля, значения констант сбрасываются в значения по умолчанию.

Из объекта типа GregorianCalendar можно осуществлять доступ к методу get(*имяКонстанты*), возвращающему значение, соответствующее дате и времени этого объекта и тому, какая константа задана в качестве параметра. Имеются следующие константы, с помощью которых можно получить информацию о дате и времени:

- ERA — эра. При задании константы в качестве параметра метода get возвращаемое значение AD==1 (от латинского Anno Domini — наша эра) или BC==0 (от Before Christ — до Рождества Христова). По умолчанию объект имеет значение эры AC.
- YEAR — год. По умолчанию объект имеет значение года 1970.
- MONTH — месяц. Возвращаемые значения от JANUARY до DECEMBER (т. е. от 0 до 11). По умолчанию объект имеет значение месяца JANUARY==0.
- DAY_OF_MONTH — день месяца. Возвращаемые значения от 1 до 31. По умолчанию объект имеет значение дня месяца 1.
- DAY_OF_WEEK — день недели. Возвращаемые значения от SUNDAY до SATURDAY. По умолчанию объект имеет значение дня недели SUNDAY==1.
- WEEK_OF_MONTH — неделя месяца. Возвращаемые значения от 0 до 6. По умолчанию объект имеет значение недели месяца 0. Номер недели учитывает возможность существования неполных недель, состоящих менее чем из 7 дней.
- DAY_OF_WEEK_IN_MONTH — номер соответствующего дня недели (воскресенья, понедельника, вторника и т. п.) в месяце. Например, второе воскресенье месяца. Может принимать не только положительные, но и нулевые или отрицательные значения — в этом случае идет отсчет для предыдущих недель. По умолчанию объект имеет значение номера дня недели 1.
- AM_PM — до полудня или после полудня. Возвращаемые значения AM (от латинского Ante Meridiem — до полудня) и PM (от латинского Post Meridiem — после полудня) типа int. По умолчанию имеет значение AM.
- HOUR — номер часа для объекта календаря в 12-часовом варианте исчисления времени. Возвращаемые значения от 0 до 11.
- HOUR_OF_DAY — номер часа для объекта календаря в 24-часовом варианте исчисления времени. Возвращаемые значения от 0 до 23.
- MINUTE — номер минуты в соответствующем часе. Возвращаемые значения от 0 до 59.
- SECOND — номер секунды в соответствующей минуте. Возвращаемые значения от 0 до 59.
- MILLISECOND — номер миллисекунды в соответствующей секунде. Возвращаемые значения от 0 до 999.

Дату и время объекта календаря можно изменить с помощью соответствующих сеттеров:

```
set(int year, int dayOfMonth, int date)
set(int year, int month, int dayOfMonth, int hourOfDay, int minute)
set(int year, int month, int dayOfMonth, int hourOfDay, int minute, int second)
setTimeInMillis(long millis).
```

Метод setFirstDayOfWeek(int value) позволяет задавать, какой день недели считается первым. По умолчанию это воскресенье (GregorianCalendar.SUNDAY), как принято в США. В России это понедельник (GregorianCalendar.MONDAY), поэтому необходимо задать

```
calendar.setFirstDayOfWeek(GregorianCalendar.MONDAY);
```

Следует отметить, что эта установка влияет на значение, возвращаемое не методом `calendar.get(DAY_OF_WEEK)`, а методом `calendar.getFirstDayOfWeek()`.

Также имеется возможность установки временной зоны и ряда других параметров.

В качестве примера работы с датой и временем, а также перечислениями приведем код класса `GUI_application` (листинг 8.13) из проекта 08_6_DateTime с прилагаемого к книге DVD.

Листинг 8.13. Иллюстрация работы с датой, временем и перечислениями

```
package javaapplication;
import java.util.GregorianCalendar;
import static java.util.GregorianCalendar.*;

public class GUI_application extends javax.swing.JFrame {

    enum WeekDays {
        Воскресенье,
        Понедельник,
        Вторник,
        Среда,
        Четверг,
        Пятница,
        Суббота;
    }

    public static void main(String[] args) {
         GregorianCalendar calendar = new GregorianCalendar();
         //calendar.setTime(new Date());
         System.out.println("Сегодня эра "+calendar.get(ERA));
         System.out.println("Сегодня год номер "+calendar.get(YEAR));
         System.out.println("Сегодня месяц номер "+calendar.get(MONTH));
         System.out.println("День недели номер "+
                            calendar.get(DAY_OF_WEEK));
         calendar.setFirstDayOfWeek(MONDAY);
         System.out.println("Номер дня недели не изменился: " +
                            calendar.get(DAY_OF_WEEK));
         System.out.println("Сегодня день месяца номер "+
                             calendar.get(DAY_OF_MONTH));
         System.out.println("Сегодня день года номер "+
                             calendar.get(DAY_OF_YEAR));
         System.out.println("Час "+calendar.get(HOUR));
         System.out.println("Минута "+calendar.get(MINUTE));

         System.out.println("В РФ день недели под названием " +
            calendar.getDisplayName(DAY_OF_WEEK,LONG,
                                    new java.util.Locale("RU")));
         System.out.println("В США день недели под названием " +
          calendar.getDisplayName(DAY_OF_WEEK,LONG,java.util.Locale.US));
```

```
        WeekDays w=WeekDays.values()[calendar.get(DAY_OF_WEEK)-1];
        System.out.println("Сегодня день недели под названием "+w);
        String s=w.name();
        System.out.println("Сегодня день недели под названием "+s);
    }
}
```

Проанализируем данный фрагмент.

Во-первых, обратим внимание, что вместо того, чтобы использовать вызовы вида `calendar.get(GregorianCalendar.ERA)`, `calendar.get(GregorianCalendar.YEAR)` и т. п., мы импортировали все содержащиеся в классе `GregorianCalendar` константы (статические поля).

Во-вторых, мы применили два различных способа для вывода в консоль названия дня недели:

- Метод `getDisplayName` возвращает строку с названием дня недели в полном (константа `LONG` в качестве параметра) или сокращенном виде (константа `SHORT`) с учетом заданного языкового формата (locale). Для России языковый формат приходится создавать с помощью конструктора, а для ряда других стран можно задавать его в более простом виде: `Locale.US` для США, `Locale.FR` для Франции, `Locale.CHINA` для Китая и т. д.
- Мы задали перечисление с русскими названиями элементов и использовали эти названия для вывода. При этом для нахождения нужного нам дня недели пришлось учесть, что дни недели нумеруются от 1 до 7, а элементы нашего перечисления — от 0 до 6. В последних трех строчках листинга 8.11 иллюстрируется как автоматическое, так и явное (с помощью метода `name()`) преобразование имени перечисления в строку.

Наконец, помимо перечисленных классов для отсчета времени часто используют класс `java.util.Timer`. Он работает только в многопоточном режиме *(см. главу 10)*.

8.7. Работа с графикой и графическим пользовательским интерфейсом

Код визуализации и код бизнес-логики приложения

Код любой программы с графическим пользовательским интерфейсом состоит из двух частей — кода, относящегося к взаимодействию с элементами этого интерфейса, и кода, связанного с выполнением основных действий приложения. Например, проведение вычислений; управление устройствами, объектами или базами данных; запись в файлы или чтение из файлов и т. д.

Код, относящийся к работе с пользовательским интерфейсом, называется кодом *визуализации* или кодом *представления данных*. А код, отвечающий за выполнение основных действий приложения, называется кодом *бизнес-логики*. Он отвечает за получение и обработку данных.

В правильно написанной программе их следует разделять. Это связано с тем, что задачи, связанные с получением и обработкой данных, не имеют никакого отношения к

представлению этих данных в том или ином виде. Вычисление интеграла не имеет никакого отношения к тому, как мы захотим увидеть результаты этого вычисления. Причем очевидно, что код бизнес-логики меньше подвержен изменениям, чем код визуализации.

Как будет выглядеть пользовательский интерфейс, зависит от того, на какого пользователя он рассчитан, какие устройства ввода/вывода используются, и даже от художественных вкусов дизайнера.

Код бизнес-логики, напротив, должен быть одним и тем же для разных вариантов пользовательского интерфейса, поэтому он должен быть отделен от кода визуализации. Код визуализации всегда использует код бизнес-логики, но не наоборот. Обращение к коду визуализации из кода бизнес-логики категорически запрещено.

Несмотря на простоту и очевидность изложенного правила, на практике часто бывает очень трудно ему следовать. Основная рекомендация — для реализации любых сколько-нибудь значительных вычислительных или управляющих алгоритмов следует писать отдельные классы, а не реализовывать их в обработчиках событий или методах приложения. В простейших учебных примерах мы не будем следовать этому правилу, но в более сложных всегда будем им руководствоваться.

Графические примитивы

Вывод графики осуществляется с помощью объектов типа `java.awt.Graphics`. Для них определен ряд методов, описанных табл. 8.2.

Подразумевается, что:

- `w` — ширина области или фигуры;
- `h` — высота;
- `x`, `y` — координаты левого верхнего угла области (для фигуры это координаты левого верхнего угла прямоугольника, в который вписана фигура).

Таблица 8.2. *Методы, с помощью которых осуществляется вывод графики*

Метод	Что делает
Параметры вывода графики	
`Color getColor()`	Возвращает текущий цвет рисования
`setColor(Color c)`	Устанавливает текущий цвет рисования
`Font getFont()`	Возвращает текущий шрифт для вывода текстовой информации
`setFont(Font f)`	Устанавливает текущий шрифт для вывода текстовой информации. Экземпляр шрифта создается с помощью конструктора `Font(имяШрифта,стильШрифта,размерШрифта)`
`FontMetrics getFontMetrics()`	Возвращает параметры текущего шрифта
`FontMetrics getFontMetrics(Font f)`	Возвращает параметры для произвольного шрифта `f`

Таблица 8.2 *(продолжение)*

Метод	Что делает
`setXORMode(Color c1)`	Устанавливает режим рисования `XOR` (исключающее `ИЛИ`) для цвета `c1`. При этом вывод точки цвета `color` дает цвет, равный побитовому значению `color ^ c1` (т. е. `color XOR c1`) для числовой RGB-кодировки цвета. Повторный вывод графического изображения на то же место приводит к восстановлению первоначального изображения в области вывода
`setPaintMode()`	Обеспечивает переключение в обычный режим из режима рисования `XOR`
`translate(x0,y0)`	Обеспечивает сдвиг начала координат графического контекста в точку `x0,y0`. Все координаты, указанные при выводе графических примитивов, отсчитываются относительно этого начала координат
Рисование контурных фигур	
`drawLine(x1,y1,x2,y2)`	Рисует линию из точки с координатами `x1,y1` в точку `x2,y2`
`drawRect(x,y,w,h)`	Рисует прямоугольник
`drawRoundRect(x,y,w,h,arcWidth, arcHeight)`	Рисует скругленный прямоугольник
`draw3DRect(x,y,w,h,isRaised)`	Рисует объемный прямоугольник. Если переменная `isRaised ==true`, то он выпуклый (raised), иначе — вдавленный
`drawPolygon(Polygon p);` `drawPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Рисует многоугольник по массиву точек (`nPoints` — число точек)
`drawPolyline(int[] xPoints, int[] yPoints, int nPoints)`	Рисует незамкнутую ломаную линию по массиву точек (`nPoints` — число точек)
`drawOval(x,y,w,h)`	Рисует эллипс
`drawArc(x,y,w,h,startAngle, arcAngle)`	Рисует дугу эллипса. Начальный угол `startAngle` и угол, задающий угловой размер дуги `arcAngle`, задаются в градусах
`drawImage(Image img,int x, int y, ImageObserver observer)` и другие перегруженные варианты метода	Рисует изображение
Рисование заполненных фигур	
`clearRect(x,y,w,h)`	Очищает прямоугольник (заполняет текущим цветом)
`fillRect(x,y,w,h)`	Рисует прямоугольник, заполненный текущим цветом
`fillRoundRect(x,y,w,h,arcWidth, arcHeight)`	Рисует скругленный прямоугольник, заполненный текущим цветом

***Таблица 8.2** (окончание)*

Метод	Что делает
`fill3DRect(x,y,w,h, isRaised)`	Рисует объемный прямоугольник, заполненный текущим цветом. Если переменная `isRaised` имеет значение `true`, то он выпуклый (raised), иначе — вдавленный
`fillPolygon(Polygon p)` `fillPolygon(int[] xPoints, int[] yPoints, int nPoints)`	Рисует многоугольник, заполненный текущим цветом
`fillOval(x,y,w,h)`	Рисует эллипс, заполненный текущим цветом
`fillArc(x,y,w,h,startAngle, arcAngle)`	Рисует сектор эллипса, заполненный текущим цветом. Заполняется сектор, ограниченный отрезками из центра эллипса в концы дуги и самой дугой
`copyArea(x,y,w,h,dx,dy)`	Копирует содержимое области на новое место, сдвинутое от старого на `dx,dy`
Вывод текстовой информации	
`drawString(s,x,y)`	Выводит содержимое строки `s`
`drawChars(char[] data,int offset,int length,int x,int y)`	Выводит массив символов
`drawBytes(byte[] data,int offset,int length,int x,int y)`	Выводит символы, представленные как последовательность байтов
Управление областью вывода	
`setClip(x,y,w,h)` `setClip(Shape clip)`	Устанавливает новые границы области вывода. Вне этой области при выводе графических примитивов они усекаются (не выводятся)
`clipRect(x,y,w,h)`	Сужает область вывода
`Rectangle getClipBounds()` `Rectangle getClipBounds(Rectangle r)`	Возвращает параметры прямоугольника, в который вписана область вывода
`Graphics create()`	`g1=g.create()` — создает копию графического объекта `g`
`dispose()`	Деструктор — уничтожает графический объект с одновременным высвобождением ресурсов (без ожидания, когда это сделает сборщик мусора)

Пример метода, работающего с графикой

Листинг 8.14 (пример 08_7_Graphics1 на прилагаемом к книге DVD) иллюстрирует работу с графикой в обработчике события нажатия на кнопку. На первый взгляд кажется, что в нем все сделано правильно.

Листинг 8.14. Рисование по панели типа `JPanel`

```
java.awt.Graphics g,g1;
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
     java.awt.Graphics g,g1;
     g=jPanel1.getGraphics();
     int x1=20,x2=120,y1=20,y2=120;
     int x3=20,y3=20,w3=60,h3=80;
     int x4=30,y4=60,w4=30,h4=40;
     int x0=10,y0=10,w0=10,h0=10;
     int w1=80,h1=120;
     g.setClip(0,0,60,80);//границы области вывода
     g.drawLine(x1,y1,x2,y2);//линия
     g.drawOval(x3,y3,w3,h3);//эллипс
     g.clipRect(x4,y4,20,20);//сужение области вывода
     g.clearRect(x4,y4,w4,h4);//очистка прямоугольника
     g.setClip(0,0,200,280); //новые границы области вывода
     g.copyArea(x1,y1,w1,h1,60,0);
     g.draw3DRect(10,20,w1,h1,false);
     g.drawPolygon(new java.awt.Polygon(new int[]{10,10,20,40},
                                        new int[]{10,20,30,60},4) );
}
```

Но при перерисовке графического контекста возникает проблема: все выведенное изображение исчезает. Такая перерисовка вызывается автоматически при изменении размера окна приложения, а также при его восстановлении после минимизации или перекрытия другим окном.

Для того чтобы результаты вывода не пропадали, в классе приложения требуется переопределить метод `paint`, вызываемый при отрисовке. Код этого метода может выглядеть так, как в листинге 8.15.

Листинг 8.15. Переопределение метода `paint`

```
 public void paint(java.awt.Graphics g){
    super.paint(g);
    g=jPanel1.getGraphics();
    ... — команды графического вывода
 }
```

Правда, при изменении размера окна приложения этот код не сработает, и для панели нужно будет назначить обработчик:

```
private void jPanel1ComponentResized(java.awt.event.ComponentEvent evt) {
    ... — команды графического вывода
}
```

То, что для изменения размера компонента следует писать отдельный обработчик, вполне разумно — ведь при восстановлении окна требуется только воссоздать изображение прежнего размера. А при изменении размера может потребоваться масштабиро-

вание выводимых элементов. Поэтому алгоритмы вывода графики в этих случаях заметно отличаются.

Если отрисовка происходит из обработчика, то какого-либо события изменения графического контекста не возникает до окончания обработчика. Это принципиальная особенность работы по идеологии обработчиков событий — пока не кончится один обработчик, следующий не начинается. Для досрочной отрисовки непосредственно во время выполнения обработчика события служит вызов метода:

```
update(Graphics g).
```

Пример:

```
for(int i=0;i<=100;i++){
   FiguresUtil.moveFigureBy(figure,dx,dy);
   update(g);
}
```

При работе со статическими изображениями изложенных алгоритмов вполне достаточно. Однако при использовании движущихся элементов во многих графических системах возникает мельтешение, связанное с постоянными перерисовками. В этих случаях обычно применяют идеологию двойной буферизации — отрисовку элементов по невидимому буферному изображению, а затем показ этого изображения в качестве видимого. А то изображение, которое было видимо, при этом становится невидимым буфером. Многие компоненты `Swing` (например, `JPanel`) обладают поддержкой двойной буферизации, которая начинает работать при установке свойства `doubleBuffered` в `true`.

При рисовании по поверхности какого-либо компонента вместо того, чтобы мучаться с заданием обработчиков событий, бывает гораздо проще задать класс-наследник и переопределить в нем метод `paint()` соответствующего компонента (пример 08_7_Graphics2 на прилагаемом к книге DVD).

В листинге 8.16 приведен код такого класса-наследника.

Листинг 8.16. Задание потомка `JPanel`

```
package examples;

import java.awt.Graphics;
public class JPanelVM extends javax.swing.JPanel {

    public JPanelVM(java.awt.LayoutManager layout, boolean isDoubleBuffered) {
        super(layout, isDoubleBuffered);
    }

    public JPanelVM(boolean isDoubleBuffered) {
        this(null, isDoubleBuffered);
    }

    public JPanelVM() {
        this(true);
    }
```

```
    @Override
    public void paint(Graphics g) {
        super.paint(g);
        int x1 = 20, x2 = 120, y1 = 20, y2 = 120;
        int x3 = 20, y3 = 20, w3 = 60, h3 = 80;
        int x4 = 30, y4 = 60, w4 = 30, h4 = 40;
        int x0 = 10, y0 = 10, w0 = 10, h0 = 10;
        int w1 = 80, h1 = 120;
        g.setClip(0, 0, 60, 80);//границы области вывода
        g.drawLine(x1, y1, x2, y2);//линия
        g.drawOval(x3, y3, w3, h3);//эллипс
        g.clipRect(x4, y4, 20, 20);//сужение области вывода
        g.clearRect(x4, y4, w4, h4);//очистка прямоугольника
        g.setClip(0, 0, 200, 280); //новые границы области вывода
        g.copyArea(x1, y1, w1, h1, 60, 0);
        g.draw3DRect(10, 20, w1, h1, false);
        g.drawPolygon(new java.awt.Polygon(new int[]{10, 10, 20, 40},
                new int[]{10, 20, 30, 60}, 4));
    }
}
```

Видно, что мы практически один к одному перенесли алгоритм отрисовки из обработчика события в метод `paint()`. Но при создании нового компонента возникает вопрос его использования в приложении. Имеется два пути: либо создавать необходимый объект-компонент динамически во время работы программы, либо добавить новый компонент в палитру компонентов.

Рассмотрим первый вариант. Сначала добавим на нашу форму кнопку **JButton** по нажатию кнопки `jButton1` (листинг 8.17).

Листинг 8.17. Динамическое добавление кнопки на панель

```
    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
        btn1 = new javax.swing.JButton();
        btn1.setBounds(200, 20, 300, 30);
        getContentPane().add(btn1);
        btn1.setText("Динамически созданная кнопка");
    }
```

В данном обработчике сначала создается объект-кнопка, затем методом `setBounds(200, 20, 300, 30)` устанавливаются координаты ее левого верхнего угла (`x=200`, `y=20`) и размеры (ширина `width=300`, высота `height=30`). Затем кнопка добавляется методом `add` (добавить) на главную панель нашей формы, ссылка на которую возвращается методом `getContentPane()`.

И наконец, для того чтобы наша добавленная кнопка отрисовалась, мы в последнем операторе назначаем кнопке надпись. Если бы мы поменяли местами два последних оператора, отрисовка произошла бы только при изменении размера формы или перемещении курсора мыши над созданной пока еще невидимой кнопкой. Поэтому обработчик должен был бы выглядеть так, как в листинге 8.18.

Листинг 8.18. Форсирование отрисовки методом `update`

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    btn1 = new javax.swing.JButton();
    btn1.setBounds(200, 50, 300, 30);
    btn1.setText("Динамически созданная кнопка");
    getContentPane().add(btn1);
    btn1.update(btn1.getGraphics());
}
```

Метод `update` вызывает форсированную перерисовку кнопки. Мы привели пример динамического добавления на форму обычной кнопки `JButton`, но `JPanel` или наш собственный компонент `JPanelVM`, являющийся наследником от `JPanel`, добавляются абсолютно так же.

Второй вариант использования нового компонента заключается в добавлении его в палитру компонентов. Это можно сделать из дерева проектов, вызвав для узла **JPanelVM** контекстное меню, выбрав в нем пункт **Tools/Add to Palette...**, в появившейся форме выбрав категорию **Beans** и нажав кнопку **OK**. После этого компонент **JPanelVM** будет доступен в палитре компонентов в категории **Beans**, и его можно добавлять в проект простым перетаскиванием на форму. При таком перетаскивании компонент будет вести себя достаточно необычно. А еще необычнее наша панель будет себя вести при перетаскивании на нее кнопки. Дело в том, что наш переопределенный метод `paint()` вызывается не только во время работы приложения, но и в режиме дизайна. Поэтому в наш компонент полезно добавить булево поле, разрешающее дополнительную отрисовку. По умолчанию это поле должно иметь значение `false`, и панель будет себя вести как обычная панель. Но в приложении можно установить данное поле в `true`, и тогда в методе `paint()` будет происходить дополнительная отрисовка.

8.8. Исключительные ситуации

Обработка исключительных ситуаций

При работе программы операторы обычно выполняются в рамках основного алгоритма (когда все идет нормально). Но время от времени возникают *исключительные ситуации* (exceptions), приводящие к ответвлению от основного алгоритма: деление на 0, отсутствие места на диске или попытка писать на защищенную для записи дискету, ввод с клавиатуры ошибочного символа (например, буквы вместо цифры) и др. В отличие от катастрофических проявлений (ошибок), такие ситуации в большинстве случаев могут быть учтены в программе (и в частности, они не должны вызывать аварийное завершение программы).

В языках программирования предыдущих поколений для решения указанных проблем приходилось выполнять огромное число проверок на допустимость присваиваний и математических операций. Мало того, что эти проверки резко замедляли работу программы, не было гарантии, что они достаточны, и что во время работы программы не возникнет аварийного завершения из-за непредусмотренной ситуации.

В Java, как и в других современных языках программирования, для таких целей существует специальное средство — обработка исключительных ситуаций. При этом ис-

пользуется так называемый защищенный блок программного кода try (попытаться), после которого следуют необязательные блоки перехвата исключений catch, а затем необязательный блок очистки ресурсов finally.

Про наступившую исключительную ситуацию говорят, что она возникает либо возбуждается. В английском это обозначается словом throw (бросить). Поэтому иногда в переводной литературе встречается дословный перевод "бросается исключительная ситуация".

Листинг 8.19 иллюстрирует общий случай использования защищенного блока программного кода и перехвата исключительных ситуаций.

Листинг 8.19. Обработка исключительных ситуаций

```
try{
 операторы0;
}
catch (ТипИсключения1 переменная1){
 операторы1;
}
catch (ТипИсключения2 переменная2){
 операторы2;
}
catch (ТипИсключенияN переменнаяN){
 операторыN;
}
finally{
 операторы;
}
```

Отметим, что при задании блоков try-catch-finally после фигурных скобок точку с запятой лучше опустить (как и обычно, в случае использования фигурных скобок), но можно и поставить — это будет рассматриваться как пустой оператор.

Если исключительных ситуаций не было, то операторы0 в блоке try следуют в обычном порядке, после чего выполняются операторы в блоке finally.

Если же возникла исключительная ситуация в блоке try, то выполнение блока прерывается, идет перехват исключений в блоках catch (перехватить). В качестве параметра оператора catch задается ссылочная переменная, имеющая тип той исключительной ситуации, которую должен перехватить данный блок. Чаще всего эту переменную называют e (по первой букве от exception). Если тип исключения совместим с типом, указанным в качестве параметра, то выполняется соответствующий оператор, после чего проверок в следующих блоках catch не делается.

После проверок и, возможно, перехвата исключения в блоках catch всегда выполняются операторы блока finally. Его обычно используют для высвобождения ресурсов и поэтому часто называют блоком очистки ресурсов. Специальных операторов или зарезервированных конструкций для обработки в блоке finally нет.

Отличие кода внутри блока finally от кода, стоящего после оператора try...finally, возникает только при наличии внутри блока try или catch операторов break, continue

или `return`, т. е. операторов, прерывающих работу данного блока программного кода. В этом случае сначала происходит выполнение операторов блока `finally` и только потом переход в другое место программы в соответствии с оператором прерывания.

Только один из операторов прерывания — `System.exit(n)` вызывает выход из блока `try` без прохождения через блок `finally`.

В листинге 8.20 приведен пример обработки исключений.

Листинг 8.20. Пример обработки исключений

```
void myETest(String s,double y){
    double x, z=0;
    int i;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(y/x);
        i=10/(int)x;
        System.out.println("x="+x+" z="+z+" i="+i);
    } catch(ArithmeticException e){
        System.out.println("Ошибка: деление на ноль");
    } catch(NumberFormatException  e){
        System.out.println("Ошибка преобразования из строки в число!");
    }
   System.out.println("В конце вызова z="+z);
}
```

В результате выполнения вызова `myETest("5",5.0)` исключительных ситуаций не возникнет и в окно консоли будет выведено:

```
x=5.0 z=1.0 i=2
В конце вызова z=1.0
```

При вызове `myETest("0",5.0)` возбуждается исключительная ситуация `ArithmeticException` — целочисленное деление на ноль, которая перехватывается в соответствующем блоке `catch`.

В окно консоли будет выведено:

```
Ошибка: деление на ноль
В конце вызова z=Infinity
```

При вычислении `y/x` в случае `x=0.0` возбуждения исключительной ситуации не происходит — вещественное деление на ноль разрешено. Отметим, что оператор `System.out.println("x="+x+" z="+z+" i="+i)` в данном случае не выполняется из-за прерывания нормального хода операций вследствие возникновения исключительной ситуации.

Если `y/x` отрицательно, то исключительная ситуация также не возникает, т. к. извлечение корня из отрицательного числа в Java разрешено и дает значение `NaN`. Поэтому, например, в результате выполнения вызова

```
myETest("-5",5.0)
```

в окно консоли будет выведено:

```
x=-5.0 z=NaN i=-2
В конце вызова z=NaN
```

Если s не содержит строкового представления числа типа Double, то возбуждается исключительная ситуация ошибки преобразования из строки в число NumberFormatException. А затем она перехватывается в соответствующем блоке catch.

В окно консоли выводится:

```
Ошибка преобразования из строки в число!
В конце вызова z=0.0
```

Заметим, что после перехвата и обработки исключений "выпадания" из вызова myETest не происходит, а выполняется оператор

```
System.out.println("В конце вызова z="+z).
```

Иерархия исключительных ситуаций

Исключительные ситуации в Java являются объектами. Их типы — это классы-потомки объектного типа Throwable (от throw able — способный возбудить исключительную ситуацию). От Throwable наследуются классы Error (ошибка) и Exception (исключение). Экземпляры класса Error — непроверяемые исключительные ситуации, которые невозможно перехватить в блоках catch. Такие исключительные ситуации представляют катастрофические ошибки, после которых невозможна нормальная работа приложения. Экземпляры класса Exception и его потомков — проверяемые исключительные ситуации, кроме одного потомка — класса RuntimeException (и его потомков). Имя этого класса переводится как "исключительные ситуации времени выполнения".

Исключения типа RuntimeException, как и исключения типа Error, не являются проверяемыми, и методы, возбуждающие их, не требуют в заголовке декларировать типы этих исключений после слова throws (о такой декларации будет рассказано далее). Но, в отличие от ошибок Error, исключения RuntimeException не препятствуют дальнейшей нормальной работе виртуальной Java-машины (JVM).

Классы исключительных ситуаций либо предопределены в стандартных пакетах (существуют исключительные ситуации ArithmeticException для арифметических операций в пакете java.lang, IOException в пакете java.io и т. д.), либо описываются пользователем как потомки класса Exception или его потомков.

В Java типы-исключения принято именовать, оканчивая имя класса на "Exception" (исключение) для проверяемых исключений или на "Error" (ошибка) для непроверяемых.

Приведем иерархию исключений из пакета java.lang (важнейшие из них, чаще всего используемые при написании программ или возникающие при их выполнении, подчеркнуты).

Прародителем для всех классов исключений является класс Throwable. Он обеспечивает трассировку стека исключения — сохранение информации о стеке вызова и выдачу ее в окно консоли при выходе исключения за уровень приложения.

От него наследуются:

- `Exception` — прародитель для всех классов, не препятствующих нормальной работе JVM (классов перехватываемых исключений и классов-потомков `RuntimeException`):
  - `ClassNotFoundException` — неудачная попытка загрузки класса по имени;
  - `CloneNotSupportedException` — класс не поддерживает клонирование объектов;
  - `IllegalAccessException` — неверный доступ при использовании механизма `reflection`;
  - `InstantiationException` — попытка создать экземпляр абстрактного класса;
  - `InterruptedException` — попытка прерывания спящего потока выполнения;
  - `NoSuchFieldException` — в классе отсутствует поле с указанным именем;
  - `NoSuchMethodException` — в классе отсутствует метод с указанным именем;
  - `RuntimeException` — прародитель всех непроверяемых исключений, не препятствующих нормальной работе JVM:
    - `ArithmeticException` — арифметические ошибки (целочисленное деление на 0);
    - `ArrayStoreException` — попытка сохранить в массиве объектов объект недопустимого типа;
    - `ClassCastException` — попытка приведения объекта к недопустимому типу;
    - `EnumConstantNotPresentException` — неверный доступ к перечислению по имени;
    - `IllegalArgumentException` — попытка вызова метода с недопустимым аргументом:
      - `IllegalThreadStateException` — попытка доступа к потоку выполнения, находящемуся в недопустимом для этого состоянии;
      - `NumberFormatException` — неудачная попытка преобразования строки в число;
    - `IllegalMonitorStateException` — недопустимая попытка обращения к потоку выполнения (методы `wait`, `notify`, `notifyAll`);
    - `IllegalStateException` — приложение или JVM не готовы к вызову;
    - `IndexOutOfBoundsException` — индекс при обращении к массиву, строке или вектору вышел за пределы допустимого диапазона:
      - `ArrayIndexOutOfBoundsException` — при обращении к массиву;
      - `StringIndexOutOfBoundsException` — при обращении к строке;
    - `NegativeArraySizeException` — попытка создать массив отрицательной длины;
    - `NullPointerException` — попытка обращения к объекту по ссылке `null`;
    - `SecurityException` — порождается менеджером безопасности;
    - `TypeNotPresentException` — неудачная попытка загрузки типа по имени;
    - `UnsupportedOperationException` — запрашиваемая операция не поддерживается;
- `Error` — прародитель для классов неперехватываемых исключений:
  - `AssertionError` — при провале теста, когда необходимое для прохождения теста утверждение (assertion) не выполнено;

- `LinkageError` — ошибки компиляции, связанные со взаимно зависимыми классами:
  - `ClassCircularityError`;
  - `ClassFormatError`;
    - `UnsupportedClassVersionError`;
  - `ExceptionInInitializerError`;
  - `IncompatibleClassChangeError`:
    - `IllegalAccessError`;
    - `InstantiationError`;
    - `NoSuchFieldError`;
    - `NoSuchMethodError`;
  - `NoClassDefFoundError`;
  - `UnsatisfiedLinkError`;
  - `VerifyError`;
- `VirtualMachineError` — JVM не может больше нормально работать:
  - `InternalError` — внутренняя ошибка в JVM;
  - `OutOfMemoryError` — JVM не может выделить память под объект;
  - `StackOverflowError` — переполнение стека из-за слишком большой глубины рекурсии вызова в приложении;
  - `UnknownError` — в JVM возникла неизвестная исключительная ситуация, препятствующая дальнейшей работе.

По правилу совместимости типов исключительная ситуация типа-потомка всегда может быть обработана как исключение прародительского типа. Поэтому порядок следования блоков `catch` имеет большое значение: обработчик исключения *более общего* типа следует писать всегда *после* обработчика для его типа-потомка, иначе обработчик потомка никогда не будет вызван.

В приведенном ранее примере (см. листинг 8.20) вместо `NumberFormatException` можно поставить `Exception`, т. к. других типов исключений, кроме `NumberFormatException`, сюда доходить не может.

Листинг 8.21 иллюстрирует, как выглядит метод в этом случае.

Листинг 8.21. Правильный порядок обработки исключений

```
void myETest(String s,double y){
    double x, z=0;
    int i;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(y/x);
        i=10/(int)x;
        System.out.println("x="+x+" z="+z+" i="+i);
    } catch(ArithmeticException e){
        System.out.println("Ошибка: деление на ноль");
```

```
    } catch(Exception  e){
        System.out.println("Ошибка преобразования из строки в число!");
    }
    System.out.println("В конце вызова z="+z);
}
```

Но если бы мы попробовали таким же образом заменить тип исключения в первом блоке `catch`, то блок для исключений типа `Exception` всегда перехватывал бы управление и обработчик для `NumberFormatException` никогда бы не срабатывал.

Листинг 8.22 содержит пример такого неправильно написанного кода.

Листинг 8.22. Неправильный порядок обработки исключений

```
void myETest(String s,double y){
    double x, z=0;
    int i;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(y/x);
        i=10/(int)x;
        System.out.println("x="+x+" z="+z+" i="+i);
    } catch(Exception  e){
        System.out.println("Ошибка преобразования из строки в число!");
    } catch(ArithmeticException e){
        System.out.println("Ошибка: деление на ноль");
    }
    System.out.println("В конце вызова z="+z);
}
```

В таких случаях среда разработки NetBeans при компиляции выдает сообщение об ошибке вида:

```
exception java.lang.NumberFormatException has already been caught
```

(исключение `java.lang.NumberFormatException` уже было перехвачено),

поэтому разработчику не удастся реализовать такой неправильный вариант.

Объявление типа исключительной ситуации и оператор *throw*

Для того чтобы создать собственный тип исключительной ситуации, требуется задать соответствующий класс, который должен быть наследником от какого-либо класса исключительной ситуации.

Например, зададим класс исключения, возникающего при неправильном вводе пользователем пароля:

```
class WrongPasswordException extends Exception {
   WrongPasswordException(){  //  конструктор
     System.out.println("Wrong password!");
   }
}
```

Создание объекта-исключения может проводиться в произвольном месте программы обычным образом, как для всех объектов, но при этом возбуждения исключения не происходит.

Программное возбуждение исключительной ситуации производится с помощью оператора `throw`, после которого указывается оператор создания объекта-исключения:

```
throw new ТипИсключения();
```

Например:

```
throw new WrongPasswordException();
```

Если после частичной обработки требуется повторно возбудить исключительную ситуацию `e`, то используется вызов:

```
throw e;
```

Для проверяемых исключений всегда требуется явное возбуждение. При возбуждении исключения во время выполнения какого-либо метода прерывается основной ход программы и идет процесс обработки исключения. Его иногда называют всплыванием исключения (по аналогии со всплыванием пузырька). Если в методе исключение данного типа не перехватывается, то исполняется соответствующий блок `finally` (если он есть) и всплывание продолжается — происходит выход из текущего метода в метод более высокого уровня. Соответственно, исключение начинает обрабатываться на уровне этого метода. Если оно не перехватывается, то происходит выход на еще более высокий уровень и т. д.

Если в методе исключение данного типа перехватывается, всплывание прекращается. Если же ни на каком уровне исключение не перехвачено, то оно обрабатывается исполняющей средой. При этом выполняются действия, предусмотренные в конструкторе исключения, а затем система выводит служебную информацию о том, где и какого типа исключение возникло.

Совет

Исключительная ситуация — это особая, экстремальная ситуация, не планируемая заранее. Не следует использовать исключения в качестве конструкций, на которых основаны часто повторяющиеся в программе действия.

Попробуйте самостоятельно усовершенствовать класс `WrongPasswordException` так, чтобы сообщение об ошибке появлялось в виде диалогового окна.

Объявление метода, который может возбуждать исключительную ситуацию. Зарезервированное слово *throws*

Формат объявления функции, которая может возбуждать проверяемые исключительные ситуации, следующий:

```
Модификаторы Тип Имя(список параметров)
  throws ТипИсключения1, ТипИсключения2,..., ТипИсключенияN
```

```
{
   Тело функции
}
```

Аналогично объявляется конструктор, который может возбуждать проверяемые исключительные ситуации:

```
Модификаторы ИмяКласса(список параметров)
  throws ТипИсключения1, ТипИсключения2,..., ТипИсключенияN
{
   Тело конструктора
}
```

Слово `throws` означает "возбуждает исключительную ситуацию" (дословно — бросает).

Непроверяемые исключения генерируются и обрабатываются системой автоматически, как правило, приводя к завершению приложения. При этом их типы нигде не указываются, и слово `throws` в заголовке метода можно опустить.

Если в теле реализуемого метода присутствует вызов метода, который может возбуждать исключительную ситуацию, и это исключение не перехватывается, то в заголовке реализуемого метода требуется указывать соответствующий тип возбуждаемого исключения. Если же это исключение порождается внутри защищенного блока программного кода, и в каком-либо блоке `catch` перехватывается этот тип исключения или более общий (прародительский), то указывать в заголовке тип исключения не следует.

Что было бы, если бы в рассмотренном ранее примере у нас отсутствовал блок `catch(ArithmeticException e)`?

В этом случае при `s="0"` происходило бы прерывание выполнения подпрограммы с выходом наружу. При этом метод должен быть объявлен так, как в листинге 8.23.

Листинг 8.23. Задание метода, который может возбуждать исключение

```
void myETest1(String s,double y) throws ArithmeticException {
    double x, z=0;
    int i;
    try{
        x=Double.parseDouble(s);
        z=Math.sqrt(y/x);
        i=10/(int)x;
        System.out.println("x="+x+" z="+z+" i="+i);
    } catch(NumberFormatException  e){
        System.out.println("Ошибка преобразования из строки в число! ");
    }
    System.out.println("В конце вызова z="+z);
}
```

Вызов метода может выглядеть так:

```
private void jButton9ActionPerformed(java.awt.event.ActionEvent evt) {
    myETest1(jTextField1.getText(),5.0);
}
```

При этом вызов `myETest1("0",5.0)`, порождающий исключительную ситуацию `ArithmeticException`, приведет к выводу в окно консоли подробной информации о прохождении исключения в нашем приложении `GUI_application` при нажатии мышью на кнопку `jButton9` (листинг 8.24).

Листинг 8.24. Вывод в консоль стека исключений

```
Exception in thread "AWT-EventQueue-0" java.lang.ArithmeticException: / by zero
    at java_gui_example.GUI_application.myETest1(GUI_application.java:273)
    at java_gui_example.GUI_application.jButton9ActionPerformed(GUI_application.java:244)
    at java_gui_example.GUI_application.access$900(GUI_application.java:17)
    at java_gui_example.GUI_application$10.actionPerformed(GUI_application.java:131)
    at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1849)
    at javax.swing.AbstractButton$Handler.actionPerformed(AbstractButton.java: 2169)
    at javax.swing.DefaultButtonModel.fireActionPerformed(DefaultButtonModel. java:420)
    at javax.swing.DefaultButtonModel.setPressed(DefaultButtonModel.java:258)
    at javax.swing.plaf.basic.BasicButtonListener.mouseReleased(
                                                        BasicButtonListener.java:234)
    at java.awt.Component.processMouseEvent(Component.java:5488)
    at javax.swing.JComponent.processMouseEvent(JComponent.java:3126)
    at java.awt.Component.processEvent(Component.java:5253)
    at java.awt.Container.processEvent(Container.java:1966)
    at java.awt.Component.dispatchEventImpl(Component.java:3955)
    at java.awt.Container.dispatchEventImpl(Container.java:2024)
    at java.awt.Component.dispatchEvent(Component.java:3803)
    at java.awt.LightweightDispatcher.retargetMouseEvent(Container.java:4212)
    at java.awt.LightweightDispatcher.processMouseEvent(Container.java:3892)
    at java.awt.LightweightDispatcher.dispatchEvent(Container.java:3822)
    at java.awt.Container.dispatchEventImpl(Container.java:2010)
    at java.awt.Window.dispatchEventImpl(Window.java:1774)
    at java.awt.Component.dispatchEvent(Component.java:3803)
    at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
    at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:242)
    at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread. java:163)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```

Как мы уже знаем, в теле метода можно разместить оператор `throw`, возбуждающий исключительную ситуацию. В этом случае если объект-исключение не перехвачен, то в заголовке метода требуется указывать соответствующий тип возбуждаемого исключения. В частности, оператор `throw` может быть использован для порождения исключения другого типа после обработки перехваченного исключения в блоке `catch`.

Если в родительском классе задан метод, в заголовке которого указан тип какой-либо исключительной ситуации, а в классе-потомке этот метод переопределяется, то в переопределяемом методе также требуется указывать совместимый тип исключительной ситуации. При этом может быть указан либо тот же тип, либо тип исключительной си-

туации — *потомка* от данного типа. В противном случае на этапе компиляции выдается диагностика ошибки.

Пример (проверка пароля, введенного пользователем) иллюстрирует листинг 8.25.

Листинг 8.25. Возбуждение пользовательского исключения

```
class CheckPasswordDemo{
  private String password="";

  public String getPassword(){
     return password;
  }

  public void setPassword(){
     ...//реализация метода
  }

  public void checkPassword(String pass)
    throws WrongPasswordException {
     if(!pass.equals(password))
        throw new WrongPasswordException();
    }
  }
}
```

При вызове метода `checkPassword` в случае неправильного пароля, переданного в качестве параметра, возбуждается исключительная ситуация. Следует обратить внимание, что сравнение `pass!=password` всегда будет давать `true`, т. к. строки сравниваются как объекты (т. е. при сравнении == проверяется идентичность адресов в памяти, а не содержание строк).

Еще один момент, на котором следует остановиться: не используйте возбуждение исключительных ситуаций для нормального режима функционирования программы! Не заменяйте им блок `else` в операторе `if`!

Возбуждение исключения выполняется намного дольше, потребляет много ресурсов и при неудачном применении только ухудшает программу. Например, в нашем случае имело бы смысл при неправильном вводе пароля предусмотреть возможность еще двух попыток ввода в обычном режиме и только после третьей неудачной попытки возбуждать исключение.

8.9. Работа с файлами и папками

Концепция работы с файлами в Java включает две составляющие:

- работа с файлами и папками с помощью объектов типа `File`.

  Обеспечивает работу с именами файлов (проверка существования файла или папки с заданным именем, нахождение абсолютного пути по относительному и наоборот, проверка и установка атрибутов файлов и папок);

- работа с потоками ввода/вывода.

  Обеспечивает взаимодействие не только с файлами, но и с памятью, а также с различными устройствами ввода/вывода.

Работа с файлами и папками с помощью объектов типа *File*

Объекты типа `File` могут рассматриваться как абстракции, инкапсулирующие работу с именами файлов и папок. При этом папка считается разновидностью файла, обладающей особыми атрибутами.

Объект типа `File` создается с помощью конструкторов, имеющих следующие варианты:

- `File(Имя папки)`
- `File(Имя файла)`
- `File(Имя папки,Имя файла).`

При этом имена могут быть как короткими (локальными) без указания пути к файлу или папке, так и длинными (абсолютными) с указанием пути. В табл. 8.3 приведены важнейшие файловые операции, заданные в классе `File`. При этом файлы (папки) ищутся по имени в соответствии с правилами поиска файлов в операционной системе. Для платформы Windows символ \ в строках, соответствующих путям, следует заменять последовательностью символов \\.

Таблица 8.3. *Важнейшие файловые операции, инкапсулированные классом* `File`

Поле или метод	Что содержит или делает
Переменные класса	
`String pathSeparator`	Содержит строку с символом разделителя пути в операционной системе. Это `"/"` в UNIX-подобных системах и `"\"` в Windows
`char pathSeparatorChar`	Содержит символ разделителя пути в операционной системе. Это `'/'` в UNIX-подобных системах и `'\'` в Windows
`String separator`	Содержит строку с символом разделителя между именами файлов и файловых масок в операционной системе
`char separatorChar`	Содержит символ разделителя между именами файлов и файловых масок в операционной системе
Проверка параметров файла или папки	
`boolean exists()`	Возвращает `true`, когда файл (или папка) с заданным в конструкторе именем существует, иначе — `false`
`long length()`	Возвращает длину файла в байтах в случае, когда файл с заданным в конструкторе именем существует и не является папкой, иначе — `0L`
`boolean canRead()`	Возвращает `true`, когда файл (или папка) с заданным в конструкторе именем существует и доступен по чтению, иначе — `false`. (В UNIX-подобных системах существуют файлы, доступные только по записи.) Может возбуждать `SecurityException`
`boolean setReadOnly()`	Возвращает `true` в случае, когда файл (или папка) с заданным в конструкторе именем существует, и ему удалось установить статус "доступен только по чтению", иначе — `false`

Таблица 8.3 *(продолжение)*

Поле или метод	Что содержит или делает
`boolean canWrite()`	Возвращает `true`, когда файл (или папка) с заданным в конструкторе именем существует и доступен по записи, иначе — `false`. (В операционных системах существуют файлы, доступные только по чтению.) Может возбуждать `SecurityException`
`boolean isDirectory()`	Возвращает `true` в случае, когда файл или папка с заданным в конструкторе именем существует и является папкой, иначе — `false`
`boolean isFile()`	Возвращает `true`, когда файл или папка с заданным в конструкторе именем существует и является файлом, иначе — `false`
`boolean isHidden()`	Возвращает `true`, когда файл или папка с заданным в конструкторе именем существует и является скрытым, иначе — `false`. В UNIX-образных системах скрытыми являются файлы, имена которых начинаются с точки, в Windows — те, которые имеют атрибут `hidden` (скрытый)
`long lastModified()`	Возвращает время последней модификации файла, если он существует и доступен по чтению, иначе — `0L`. Время отсчитывается в миллисекундах, прошедших с 0 часов 1 января 1970 года (по Гринвичу)
`Boolean setLastModified( long time)`	Устанавливает время последней модификации файла. Возвращает `true`, если он существует и доступен по записи, иначе — `false`. Время отсчитывается в миллисекундах, прошедших с 0 часов 1 января 1970 года (по Гринвичу)
Путь и имя файла (папки)	
`String getName()`	Возвращает короткое имя файла или папки
`String getParent()`	Возвращает абсолютное имя родительской папки, т. е. папки, в которой находится файл (или папка), соответствующий файловому объекту
`String getAbsolutePath()`	Возвращает абсолютный путь к файлу или папке, включая имя файла. При этом если в имени файла в конструкторе была задана относительная адресация, то соответствующая часть пути сохраняется в возвращаемой строке
`String getCanonicalPath()`	Возвращает абсолютный путь к файлу или папке, включая имя файла. При этом если в имени файла в конструкторе была задана относительная адресация, то соответствующая часть пути заменяется в возвращаемой строке на канонический вариант адресации — без элементов относительной адресации. Возбуждает `IOException`, если канонический путь не может быть построен
`int compareTo(File f)`	Сравнивает имена файлов (папок), сопоставляемых текущему файловому объекту и объекту `f`. Возвращает 0 в случае, когда абсолютные имена файлов (папок) совпадают, иначе возвращает число, зависящее от разницы в длинах имен и кодов составляющих их символов. Сравнение зависимо от операционной системы — в UNIX-образных системах регистр символов имеет значение, в Windows — не имеет. Соответствие понимается абстрактно на уровне имен и путей (самих файлов может не существовать)

Таблица 8.3 *(продолжение)*

Поле или метод	Что содержит или делает
`boolean isAbsolute()`	Возвращает `true` в случае, когда адресация к имени файла (папки) текущего файлового объекта является абсолютной. Хотя может содержать элементы относительной адресации, т. е. не быть канонической
`boolean equals(Object obj)`	Возвращает `true` тогда и только тогда, когда текущий объект и параметр `obj` соответствуют одному и тому же файлу (папке) (с учетом правил о путях и регистрах символов, задаваемых операционной системой). Соответствие понимается абстрактно на уровне имен и путей (самих файлов может не существовать)
Создание/уничтожение/переименование файлов и папок	
`Boolean createNewFile()`	Предпринимает попытку создания файла или папки по имени, которое было задано в конструкторе объекта. В случае успеха возвращается `true`, иначе — `false`. Возбуждает `IOException`, если файл не может быть создан (например, уже существует)

`File createTempFile(`
`String prefix,`
`String suffix)`
`File createTempFile(`
`String prefix,`
`String suffix,`
`File folder)`
	Метод класса. Обеспечивает создание пустого файла (или папки), задаваемого коротким именем `prefix+suffix` в папке операционной системы, предназначенной для временных файлов. Возвращает ссылку на объект. Префикс должен быть не менее трех символов. Возбуждает `IOException`, если файл не может быть создан (например, уже существует)

`boolean mkdir()`	Предпринимает попытку создания папки по имени, которое было задано в конструкторе объекта. Возвращает `true` в случае успешного создания и `false` в других случаях
`boolean mkdirs()`	Предпринимает попытку создания папки по имени, которое было задано в конструкторе объекта, причем заодно создаются все папки, заданные в пути, если они не существовали. Возвращает `true` в случае успешного создания и `false` в других случаях
`boolean delete()`	Предпринимает попытку удаления файла или папки по имени, которое было задано в конструкторе объекта. Возвращает `true` в случае успешного удаления и `false` в других случаях
`boolean renameTo(File dest)`	Предпринимает попытку переименования файла или папки с имени, которое было задано в конструкторе объекта, на новое, задаваемое параметром `dest`. Возвращает `true` в случае успешного переименования и `false` в других случаях
Создание нового файлового объекта с помощью имеющегося	
`File getAbsoluteFile()`	Создает новый файловый объект по абсолютному пути, соответствующему текущему файловому объекту
`File getCanonicalFile()`	Создает новый файловый объект по каноническому пути, соответствующему текущему файловому объекту. Возбуждает `IOException`, если канонический путь не может быть построен
`File getParentFile()`	Создает новый файловый объект по абсолютному пути, соответствующему родительской папке для текущего файлового объекта

Таблица 8.3 *(окончание)*

Поле или метод	Что содержит или делает
Списки папок и файлов	

`String[] list()`
 	
`String[] list(`
`FilenameFilter filter)`
	Возвращает массив строк (список) коротких имен, находящихся в папке файлов и папок. Имена элементов, находящихся во вложенных папках, не показываются. Если файловый объект не соответствует существующей папке, то возвращает `null`. При наличии фильтра возвращаются только те имена, которые соответствуют маске фильтра

`File[] listFiles()`
 	
`File[] listFiles(`
`FilenameFilter filter)`
	Возвращает массив файловых объектов, соответствующих файлам и папкам, вложенным в ту папку, из файлового объекта которой делается вызов. Элементы, находящиеся во вложенных папках, не учитываются. Если текущий файловый объект не соответствует существующей папке, то возвращает `null`. При наличии фильтра возвращаются объекты только для тех имен, которые соответствуют маске фильтра

`File[] listRoots()`	Возвращает массив файловых объектов, соответствующих возможным на данном компьютере корневым папкам. В UNIX это папка с именем "/", в Windows — корневые папки всех возможных дисков

Листинг 8.26 иллюстрирует пример работы с файловыми объектами.

Листинг 8.26. Пример работы с файловыми объектами

```
File f1=new File(".."); // ". ", "/ ", "C:/../ "
System.out.println("getAbsolutePath():"+f1.getAbsolutePath());

try{
System.out.println("getCanonicalPath():"+f1.getCanonicalPath());
}
catch(Exception e){
System.out.println("Исключение от getCanonicalPath()");
}

System.out.println("exists():"+f1.exists());
System.out.println("canRead():"+f1.canRead());
System.out.println("canWrite(): "+f1.canWrite());
```

Выбор файлов и папок с помощью файлового диалога

При работе с файлами в подавляющем большинстве приложений требуется вызов файлового диалога. Откроем приложение 08_9_1_File и из палитры компонентов (правое верхнее окно среды разработки) перетащим на экранную форму первый компонент из палитры `Swing` — `JLabel` (метка). Повторим эту операцию еще раз. В первой метке будем показывать имя выбранного файла, а во второй — путь к этому файлу.

Для того чтобы вызвать файловый диалог, назначим обработчик события пункту файлового меню **openMenuItem** (Открыть) — подузел **[JFrame]/menuBar[JMenu]/fileMenu [JMenu]/openMenuItem[JMenuItem]**. Двойной щелчок по узлу **openMenuItem** приводит к автоматическому созданию заготовки обработчика **openMenuItemActionPerformed** и открытию редактора исходного кода.

Сначала мы создаем в приложении объект типа `JFileChooser`, соответствующий файловому диалогу.

Если в самом начале файла после строки с заданием имени пакета

```
package examples;
```

записать:

`import javax.swing.*;`, что желательно,

то в соответствующих местах кода не потребуется квалификация `javax.swing`.

Но в данном примере импорт не сделан намеренно, чтобы было видно, где используются классы данного пакета. Зададим в классе `GUI_application` поле `fileChooser`:

```
javax.swing.JFileChooser fileChooser = new javax.swing.JFileChooser();
```

Добавим в обработчик необходимый код (листинг 8.27).

Листинг 8.27. Пример вызова файлового диалога

```
 private void openMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
   if(fileChooser.showOpenDialog(null)!= fileChooser.APPROVE_OPTION){
      System.out.println("Отказались от выбора");
      return;
   }
   System.out.println("Нажали Open");
   jLabel1.setText(fileChooser.getSelectedFile().getName());
   jLabel2.setText(fileChooser.getSelectedFile().getParent());
}
```

В первой строке обработчика вызывается метод: `fileChooser.showOpenDialog(openMenuItem)`.

Он показывает диалог на экране. В качестве параметра должен быть задан родительский компонент — в этом качестве мы используем пункт меню **openMenuItem**.

Сравнение с переменной класса `APPROVE_OPTION` позволяет выяснить, была ли выбрана кнопка **Open** (Открыть) файлового диалога.

Следует обратить внимание на характерный прием — выход из подпрограммы с помощью оператора `return` в случае, когда не был осуществлен выбор файла. Неопытные программисты написали бы данный фрагмент так, как показано в листинге 8.28.

Листинг 8.28. Неправильный стиль проверки условия

```
if(fileChooser.showOpenDialog(openMenuItem)==
                              fileChooser.APPROVE_OPTION){
  System.out.println("Нажали Open");
  jLabel1.setText(fileChooser.getSelectedFile().getName());
```

```
  jLabel2.setText(fileChooser.getSelectedFile().getParent());
}
else
  System.out.println("Отказались от выбора");
```

На первый взгляд принципиальной разницы нет. Но при усовершенствовании программы код, соответствующий выбору файла с помощью диалога, может заметно разрастись, а код, соответствующий отказу от выбора, останется тем же.

В результате практически весь код обработчика во втором варианте окажется вложенным в оператор `if`. В таком коде трудно разбираться. В первом варианте оператор `if` обладает небольшой областью действия, что позволяет легко разобраться с относящимся к нему кодом.

Можно было бы перенести строку:

```
javax.swing.JFileChooser fileChooser = new javax.swing.JFileChooser()
```

в обработчик события.

В этом случае при каждом нажатии в работающем приложении пункта меню **Файл/Открыть** создавался бы новый объект-диалог. Такой код был бы работоспособен. Но формирование диалога является относительно долгим процессом, требующим большого количества ресурсов операционной системы. Поэтому лучше создавать в приложении глобальную переменную, которой назначен диалог. Помимо прочего, это позволяет в повторно открываемом диалоге оказываться в той же папке, где происходил последний выбор файла.

Перед вызовом диалога можно программно установить папку, в которой он будет открываться:

```
File folder=...;
fileChooser.setCurrentDirectory(folder);
```

Диалог сохранения файла открывается аналогично — `showSaveDialog`.

Практически всегда при использовании файловых диалогов требуется задавать фильтр, по которому просматриваются файлы. В подавляющем большинстве случаев требуется определенное расширение в имени файла. К сожалению, компонент `JFileChooser` до сих пор не поддерживает встроенной возможности настраивать фильтр — для этого требуется создание специального класса.

Листинг 8.29 иллюстрирует пример простейшего класса фильтра для файлового диалога.

Листинг 8.29. Пример простейшего класса фильтра для файлового диалога

```
package example;

import java.io.*;


public class SimpleFileFilter extends javax.swing.filechooser.FileFilter{
  String ext;
```

```
 SimpleFileFilter(String ext){
   this.ext=ext;
 }

 public boolean accept(File f){
   if(f == null)
      return false;
   if(f.isDirectory()){
      return true ;
   }
   else
     return (f.getName().endsWith(ext));
 }
}

/**
 * Описание фильтра, возникающее в строке фильтра
 * @see FileView#getName
 */
   public  String getDescription(){
       return "Text files (.txt)";
   }

}
```

Использование приведенного класса следующее:

задается глобальная переменная:

```
javax.swing.filechooser.FileFilter fileFilter = new
                                                SimpleFileFilter(".txt");
```

после чего она задействуется в обработчике события:

```
fileChooser.addChoosableFileFilter(fileFilter);
```

Данный оператор добавляет фильтр в выпадающий список возможных фильтров (масок) файлового диалога. Если добавляется уже существующий фильтр, то операция добавления игнорируется.

В примере 08_9_2_File, имеющемся на прилагаемом к книге DVD, проиллюстрировано действие рассмотренного простого фильтра.

По умолчанию отображается последний из добавленных фильтров. Если нужно показать другой фильтр, который уже был добавлен в список фильтров диалога, то следует вызвать оператор, который делает добавленный фильтр текущим, т. е. видимым при открытии диалога:

```
fileChooser.setFileFilter(fileFilter);
```

Кроме стандартных диалогов открытия и сохранения файлов имеется возможность вызова диалога методом `showCustomDialog` с дополнительными программно-задаваемыми элементами.

Также существует возможность выбирать несколько файлов. Для этого до вызова диалога требуется задать разрешение на такой выбор:

```
fileChooser.setMultiSelectionEnabled(true);
```

Листинг 8.30 иллюстрирует последовательность получения массива выбранных файлов после вызова диалога.

Листинг 8.30. Вывод в строку имен выбранных файлов при множественном выборе

```
java.io.File[] files = fileChooser.getSelectedFiles();
if (files != null && files.length > 0) {
   String filenames = "";
   for (int i=0; i<files.length; i++) {
        filenames = filenames + "\n" + files[i].getPath();
   }
}
```

Можно выбрать папку, а не файл. В этом случае следует задать режим, когда позволяется выбирать только папки:

```
fileChooser.setFileSelectionMode(fileChooser.DIRECTORIES_ONLY);
```

либо и файлы, и папки:

```
fileChooser.setFileSelectionMode(fileChooser.FILES_AND_DIRECTORIES);
```

Возврат в обычный режим:

```
fileChooser.setFileSelectionMode(fileChooser.FILES_ONLY);
```

Просмотреть в выбранной папке список файлов той папки, с которой связана переменная `File folder`, позволяет вызов вида:

```
String[] filenames= folder.list(filter);
```

Получается массив строк с короткими именами файлов. Используется переменная `filter`, тип которой является классом, реализующим интерфейс `java.io.FilenameFilter`. О том, что такое интерфейсы, будет рассказано в *главе 9*.

В листинге 8.31 приведен пример простейшего класса фильтра имен `SimpleFilenameFilter`.

Листинг 8.31. Пример простейшего класса фильтра имен

```
package example;
import java.io.*;
public class SimpleFilenameFilter implements FilenameFilter{
    String ext;
    public SimpleFilenameFilter(String ext) {
      this.ext=ext;
    }

    public boolean accept(File dir,String fileName){
       return  ext==""||fileName.endsWith(ext);
    }
}
```

Листинг 8.32 содержит образец кода для показа имен файлов, содержащихся в выбранной папке, в компонент `jTextArea1` (текстовая область) типа `JTextArea`, позволяющий выводить произвольное число строк.

Листинг 8.32. Показ имен файлов, содержащихся в выбранной папке

```
String[] filenamesArray;
File folder=fileChooser.getSelectedFile();
SimpleFilenameFilter filter=new SimpleFilenameFilter("");
String filenames = "";
if(folder.isDirectory()){
    filenamesArray=folder.list(filter);
    for (int i=0; i<filenamesArray.length; i++) {
        filenames = filenames + "\n" + filenamesArray[i];
    }
    jTextArea1.setText(filenames);
}
```

Аналогичный образец для вывода выбранных в диалоге имен в режиме "мультиселект", позволяющем отмечать несколько файлов и/или папок иллюстрирует листинг 8.33.

Листинг 8.33. Показ имен файлов, выбранных в режиме множественного выбора

```
java.io.File[] files = fileChooser.getSelectedFiles();
if (files != null && files.length > 0) {
    String filenames = "";
    for (int i=0; i<files.length; i++) {
        filenames = filenames + "\n" + files[i].getPath();
    }
    jTextArea1.setText(filenames);
}
```

Приложение с рассмотренными вариантами кода приведено в примере 08_9_3_File на прилагаемом к книге DVD.

Работа с потоками ввода/вывода

Стандартные классы Java, инкапсулирующие работу с данными (оболочечные классы для числовых данных, классы `String` и `StringBuffer`, массивы и т. д.), не обеспечивают поддержку чтения этих данных из файлов или запись их в файл. Вместо этого применяется весьма гибкая и современная (но не очень простая) технология использования потоков (Streams).

Поток представляет накапливающуюся последовательность данных, поступающих из какого-то источника. Порция данных может быть считана из потока, при этом она из потока изымается. В потоке действует принцип очереди — "первым вошел — первым вышел".

Источник данных потока может быть как стационарным (файл, массив, строка), так и динамическим (другой поток). При этом в ряде случаев выход одного потока может служить входом другого. Поток можно представить себе как трубу, через которую перекачиваются данные, причем часто в таких трубах происходит обработка данных. Например, поток шифрования шифрует данные, полученные на входе, и при считывании из потока передает их в таком виде на выход. А поток архивации сжимает по соответ-

ствующему алгоритму входные данные и передает их на выход. Трубы потоков можно соединять друг с другом — выход одного со входом другого. Для этого в качестве параметра конструктора потока задается имя переменной, связанной с потоком — источником данных для создаваемого потока.

Буферизуемые потоки имеют хранящийся в памяти промежуточный буфер, из которого считываются выходные данные потока. Наличие такого буфера позволяет повысить производительность операций ввода/вывода, а также осуществлять дополнительные операции — устанавливать метки (маркеры) для какого-либо элемента, находящегося в буфере потока, и даже делать возврат считанных элементов в поток (в пределах буфера).

Абстрактный класс `InputStream` (входной поток) инкапсулирует модель входных потоков, позволяющих считывать из них данные. Важнейшие методы для данного класса приведены в табл. 8.4.

Таблица 8.4. *Методы класса* `InputStream`

Метод	Что делает
`int available()`	Возвращает текущее количество байтов, доступных для чтения из потока
`int read()`	Читает один байт из потока и возвращает его значение в виде целого, лежащего в диапазоне от 0 до 255. При достижении конца потока возвращает –1
`int read(byte[] b)` `int read(byte[] b, int offset, int count)`	Пытается прочесть `b.length` байтов из потока в массив `b`. Возвращает число реально прочитанных байтов. После достижения конца потока последующие считывания возвращают –1
`long skip(long count)`	Пытается пропускать (игнорировать) `count` байтов из потока (если `count` ≤0, то пропуска байтов нет). Возвращается реально пропущенное число байтов
`boolean markSupported()`	Возвращает `true` в случае, когда поток поддерживает операции `mark` и `reset`, иначе — `false`
`void mark(int limit)`	Ставит метку в текущей позиции начала потока. Используется для последующего вызова метода `reset`, с помощью которого считанные после установки метки данные возвращаются обратно в поток. Эту операцию поддерживают не все потоки (см. метод `markSupported`)
`void reset()`	Восстанавливает предшествующее состояние данных в начале потока, возвращая указатель начала потока на помеченный до этого меткой элемент (т. е. считанные после установки метки данные возвращаются обратно в поток). Попытка вызова при отсутствии метки или выходе ее за пределы лимита приводит к возбуждению исключения `IOException`. Эту операцию поддерживают не все потоки (см. методы `markSupported` и `mark`)
`void close()`	Закрывает поток. Последующие попытки чтения из этого потока приводят к возбуждению исключения `IOException`

Все методы класса `InputStream`, кроме `markSupported` и `mark`, возбуждают исключение `IOException` (оно возникает при ошибке чтения данных).

Абстрактный класс OutputStream (выходной поток) — модель выходных потоков, позволяющих записывать в них данные. Важнейшие методы для данного класса приведены в табл. 8.5.

Таблица 8.5. *Методы класса* `OutputStream`

Метод	Что делает
`void write(int b)`	Записывает один байт в поток. Благодаря применению типа `int` можно использовать в качестве параметра целочисленное выражение без приведения его к типу `byte`. Напомним, что в выражениях короткие целые типы автоматически преобразуются к типу `int`
`void write(byte[] b)` `void write(byte[] b, int offset, int count)`	Записывает в поток массив байтов. Если заданы параметры `offset` и `count`, то записывается не весь массив, а `count` байтов, начиная с индекса `offset`
`void flush()`	Форсирует вывод данных из выходного буфера и его очистку. Необходимость этой операции связана с тем, что в большинстве случаев данные уходят во внешний мир на запись не каждый раз после вызова `write`, а только после заполнения выходного буфера. Таким образом, если операции записи разделены паузой (большим интервалом времени) и требуется, чтобы уход данных из выходного буфера совершался своевременно, то после последнего вызова оператора `write`, предшествующего паузе, нужно вызывать оператор `flush`
`void close()`	Закрывает поток. Последующие попытки записи в этот поток приводят к возбуждению исключения `IOException`

Все методы этого класса возбуждают `IOException` в случае ошибки записи.

Абстрактные методы, заданные в классах `InputStream` и `OutputStream`, реализованы в классах-потомках.

Не все классы потоков являются потомками `InputStream/OutputStream`. Чтение строк (в виде массива символов) обеспечивают потомки абстрактного класса `java.io.Reader` (читатель) (в частности, чтение из файла — класс `FileReader`). Аналогично запись строк реализуют классы-потомки абстрактного класса `java.io.Writer` (писатель) (в частности, для записи массива символов в файл — класс `FileWriter`).

Имеется еще один важный класс для работы с файлами, не входящий в иерархию `InputStream/OutputStream` и `Reader/Writer`.

Это класс `RandomAccessFile` (файл с произвольным доступом), предназначенный для чтения и записи данных в произвольном месте файла. Такой файл с точки зрения класса `RandomAccessFile` представляет собой массив байтов, сохраненных на внешнем носителе. Класс `RandomAccessFile` не является абстрактным, поэтому можно создавать его экземпляры.

Все операции чтения и записи начинаются с позиции, задаваемой файловым указателем перед началом операции. После каждой такой операции файловый указатель сдвигается к концу файла на число позиций (байтов), соответствующих типу считанных данных. Если во время записи файловый указатель выходит за конец файла, то размер файла автоматически увеличивается.

Важнейшие методы для класса `RandomAccessFile` приведены в табл. 8.6.

***Таблица 8.6.** Важнейшие методы класса `RandomAccessFile`*

Метод	Что делает
Общие параметры файла	
`long getFilePointer()`	Возвращает значение файлового указателя — текущую позицию в файле считывающей/записывающей головки. Индексация начинается с нуля и дается в количестве байтов
`long length()`	Возвращает длину файла (в байтах)
`void setLength(long newLength)`	Изменяет длину файла — она устанавливается равной `newLength`. Если `newLength` меньше текущей длины, то файл укорачивается путем отбрасывания его конца. При этом если файловый указатель находился в отбрасываемой части, то он устанавливается в конец файла, иначе его позиция не меняется. Если `newLength` больше текущей длины, то файл увеличивает свой размер путем добавления в конец нового пространства, содержимое которого не гарантируется
`void close()`	Закрывает поток. Последующие попытки доступа к этому потоку приводят к возбуждению исключения `IOException`
Побайтовые операции установки позиции	
`void seek(long pos)`	Устанавливает файловый указатель в позицию `pos`
`int skipBytes(int n)`	Пропускает `n` байтов. Происходит передвижение считывающей/записывающей головки на `n` байтов (если `n>0`, то вперед к концу файла, если `n≤0`, то позиция головки не меняется). Возвращается число реально пропущенных байтов (оно может быть меньше `n`, например, из-за достижения конца файла)
Побайтовые операции чтения	
`int read()`	Читает один байт из потока и возвращает его значение в виде целого, лежащего в диапазоне от 0 до 255. При достижении конца потока возвращает –1
`int read(byte[] b)`	Пытается прочесть `b.length` байтов из потока в массив `b`. Возвращает число реально прочитанных байтов. После достижения конца потока последующие считывания возвращают –1
`int read(byte[] b, int offset, int count)`	Пытается прочесть `count` байтов из потока в массив `b`, начиная записывать их с позиции `offset`. Возвращает число реально прочитанных байтов. После достижения конца потока последующие считывания возвращают –1
`void readFully(byte[] b)`	Считывает из файла `b.length` байтов, начиная с текущей позиции файлового указателя, и заполняет весь буферный массив `b`
`void readFully(byte[] b, int offset, int count)`	Считывает из файла в массив `count` байтов, начиная записывать их с байта, имеющего позицию `offset`

Таблица 8.6 *(продолжение)*

Метод	Что делает
Побайтовые операции записи	
`void write(int b)`	Записывает в файл один байт `b` (то же, что `writeByte(b)`)
`void write(byte[] b)`	Записывает в файл весь массив байтов `b` (т. е. `b.length` байтов)
`void write(byte[] b, int offset, int count)`	Записывает в файл `count` байтов массива `b`, начиная с байта массива, имеющего индекс `offset`
Чтение одиночных значений примитивного типа	
`boolean readBoolean()`	Считывает значение `boolean`
`byte readByte()`	Считывает значение `byte`
`short readShort()`	Считывает значение `short`
`int readInt()`	Считывает значение `int`
`long readLong()`	Считывает значение `long`
`int readUnsignedByte()`	Считывает значение типа беззнаковый байт (тип отсутствует в Java) и преобразует в значение типа `int`
`int readUnsignedShort()`	Считывает значение типа беззнаковое короткое целое (тип отсутствует в Java) и преобразует в значение типа `int`
`char readChar()`	Считывает значение `char`
`float readFloat()`	Считывает значение `float`
`double readDouble()`	Считывает значение `double`
Запись одиночных значений примитивного типа	
`void writeBoolean(boolean v)`	Записывает в файл значение `boolean`
`void writeByte(int v)`	Записывает в файл значение `byte`. Благодаря использованию типа `int` можно применять в качестве параметра целочисленное выражение без приведения его к типу `byte`
`void writeShort(int v)`	Записывает в файл значение `short`. Благодаря использованию типа `int` можно применять в качестве параметра целочисленное выражение без приведения его к типу `short`
`void writeInt(int v)`	Записывает в файл значение `int`
`void writeLong(long v)`	Записывает в файл значение `long`
`void writeChar(int v)`	Записывает в файл значение `char`. Благодаря использованию типа `int` можно применять в качестве параметра целочисленное выражение без приведения его к типу `char`
`void writeFloat(float v)`	Записывает в файл значение `float`
`void writeDouble(double v)`	Записывает в файл значение `double`

Таблица 8.6 *(окончание)*

Метод	Что делает
Чтение отдельных строк	
`String readLine()`	Считывает из файла строку символов от текущей позиции файлового указателя до места переноса на новую строку (символ возврата каретки или последовательность `\n`) или до конца файла. При этом старшие байты получившихся в строке символов Unicode оказываются равными нулю, т. е. поддерживается только часть кодовой таблицы. Метод полезен для считывания текста, записанного в кодировке ANSI
`String readUTF`	Считывает из файла строку символов в кодировке UTF-8 от текущей позиции файлового указателя. Число последующих считываемых байтов строки в кодировке UTF-8 задается первыми двумя считанными байтами
Запись отдельных строк	
`void writeBytes(String s)`	Записывает в файл строки `s` как последовательности байтов, соответствующих символам строки. При этом каждому символу соответствует один записываемый байт, т. к. у каждого символа Unicode отбрасывается старший из двух байтов (записывается младший байт). Используется для записи символов в кодировке ANSI (см. также метод `writeChars`, в котором записываются оба байта)
`void writeChars(String s)`	Записывает в файл строки `s` как последовательности байтов, соответствующих символам строки. При этом каждому символу Unicode соответствует два записываемых байта
`void writeUTF(String s)`	Записывает в файл строки `s` как последовательности байтов, соответствующих символам строки в кодировке UTF-8

Все методы, заданные в классе `RandomAccessFile` (кроме `getChannel()`), возбуждают исключение `IOException` (оно возникает при ошибке записи или ошибке доступа к данным).

В конструкторе класса требуется указать два параметра: первый — имя файла или файловую переменную, второй — строку с модой доступа к файлу.

Имеются следующие варианты моды доступа:

- `"r"` — от read (только читать);
- `"rw"` — от read and write (читать и писать);
- `"rws"` — от read and write synchronously (читать и писать с поддержкой синхронизации) (про понятие синхронизации *см. главу 10*);
- `"rwd"` — от read and write to device (читать и писать с поддержкой синхронизации устройства).

Например:

```
java.io.RandomAccessFile rf1=new java.io.RandomAccessFile("q.txt","r");
java.io.RandomAccessFile rf2=new java.io.RandomAccessFile(file,"rw");
```

Рассмотрим примеры, иллюстрирующие работу с потоками.

В листинге 8.34 приведен пример чтения текста из файла.

Листинг 8.34. Пример чтения текста из файла с помощью файловых потоков

```
File file;
javax.swing.JFileChooser fileChooser=new javax.swing.JFileChooser();
javax.swing.filechooser.FileFilter fileFilter=
                                        new SimpleFileFilter(".txt");

private void openMenuItemActionPerformed(java.awt.event.ActionEvent evt) {
    fileChooser.addChoosableFileFilter(fileFilter);
    if(fileChooser.showOpenDialog(null)!=fileChooser.APPROVE_OPTION){
        return;//Нажали Cancel
    }
    file = fileChooser.getSelectedFile();
    try{
        InputStream fileInpStream=new FileInputStream(file);
        int size=fileInpStream.available();
        fileInpStream.close();
        char[] buff=new char[size];
        Reader fileReadStream=new FileReader(file);
        int count=fileReadStream.read(buff);
        jTextArea1.setText(String.copyValueOf(buff));
        javax.swing.JOptionPane.showMessageDialog(null,
                "Прочитано "+ count+" байтов");

        fileReadStream.close();
    } catch(Exception e){
        javax.swing.JOptionPane.showMessageDialog(null,
                "Ошибка чтения из файла \n"+file.getAbsolutePath());
    }

}
```

Переменной `fileInpStream`, которая будет использована для работы потока `FileInputStream`, мы задаем тип `InputStream`. Это очень характерный прием при работе с потоками, позволяющий при необходимости заменять в дальнейшем входной поток с `FileInputStream` на любой другой без изменения последующего кода. После выбора имени текстового файла с помощью файлового диалога мы создаем файловый поток:

```
fileInpStream=new FileInputStream(file);
```

Функция `fileInpStream.available()` возвращает число байтов, которые можно считать из файла. После ее использования поток `fileInpStream` нам больше не нужен и мы его закрываем. Поток `FileReader` не поддерживает метод `available()`, поэтому нам пришлось задействовать поток типа `FileInputStream`.

Массив `buff` служит буфером, в который мы считываем весь файл.

Если требуется читать файлы большого размера, то используют буфер фиксированной длины и в цикле считывают данные из файла блоками, равными длине буфера, до тех

пор, пока число считанных байтов не окажется меньше размера буфера. Это означает, что мы дошли до конца файла. В таком случае выполнение цикла следует прекратить. При такой организации программы нет необходимости в методе `available()` и потоке типа `FileInputStream`.

В операторе

```
Reader fileReadStream=new FileReader(file);
```

указан тип `Reader`, а не `FileReader`, по той же причине, по которой до этого был `InputStream`, а не `FileInputStream`.

Отметим, что чтение с помощью этого объекта строк с национальной кодировкой правильно работает только для строк ANSI. При использовании для записи и чтения методов `readUTF` и `writeUTF` объекта `RandomAccessFile` проблем с национальными кодировками не возникает.

Построчное чтение из файла осуществляется аналогично, но содержимое блока `try` следует заменить на код листинга 8.35.

Листинг 8.35. Построчное чтение из файла

```
FileReader filReadStream=new FileReader(file);
BufferedReader bufferedIn=new BufferedReader(filReadStream);
String s="",tmpS="";
while((tmpS=bufferedIn.readLine())!=null)
    s+=tmpS+"\n";
jTextArea1.setText(s);
bufferedIn.close();
```

Пример записи текста в файл очень похож на пример чтения, но гораздо проще, т. к. не требуется вводить промежуточные буферы и потоки (листинг 8.36).

Листинг 8.36. Запись текста в файл

```
private void saveAsMenuItemActionPerformed(
                                        java.awt.event.ActionEvent evt) {
    fileChooser.addChoosableFileFilter(fileFilter);
    if(fileChooser.showSaveDialog(null)!=fileChooser.APPROVE_OPTION){
        return;//Нажали Cancel
    }
    file = fileChooser.getSelectedFile();
    try{
        Writer filWriteStream=new FileWriter(file);
        filWriteStream.write(jTextArea1.getText() );
        filWriteStream.close();
    } catch(Exception e){
        javax.swing.JOptionPane.showMessageDialog(null,
                    "Ошибка записи в файл \n"+file.getAbsolutePath());
    }
}
```

Рассмотренная работа с файловыми потоками приведена на прилагаемом DVD в примере 08_9_4_FileStream.

Краткие итоги по главе

- В Java *массивы являются объектами*, но особого рода — их объявление отличается от объявления других видов объектов. Переменная типа `массив` ссылочная. Массивы Java динамические — память под них выделяется с помощью оператора `new` или создания безымянного массива, или присваивания ссылки на уже существующий объект-массив. Изменить длину уже существующего массива нельзя, хотя переменной типа `массив` (если в этом типе не указана длина) можно задать ссылку на массив новой длины.
- Двумерный массив представляет собой массив ячеек, каждая из которых имеет тип `одномерный массив`. Трехмерный массив — это массив двумерных массивов. Соответствующим образом они и задаются.
- Копировать массивы лучше с помощью метода `System.arraycopy`. Быстрое заполнение массива одинаковыми значениями может осуществляться методом `Arrays.fill` (класс `Arrays` расположен в пакете `java.util`). Поэлементное сравнение массивов следует выполнять с помощью метода `Arrays.equals` (сравнение на равенство содержимого массивов) либо `Arrays.deepEquals` (глубокое сравнение, т. е. на равенство содержимого, а не ссылок — на произвольном уровне вложенности). Сортировку (упорядочение по значениям) массива выполняет метод `Arrays.sort`.
- *Коллекции* (Collections) — "умные" массивы с динамически изменяемой длиной, поддерживающие ряд важных дополнительных операций по сравнению с обычными массивами. Доступ к элементам коллекции в общем случае не может осуществляться по индексу, т. к. не все коллекции поддерживают индексацию элементов. Эту функцию осуществляют с помощью специального объекта — итератора (iterator). У каждой коллекции имеется свой итератор, который умеет с ней работать.
- Класс `String` инкапсулирует действия со строками. Объект типа `String` — это строка, состоящая из произвольного числа символов от 0 до $2 \cdot 10^9$. Литерные константы типа `String` представляют собой последовательности символов, заключенные в двойные кавычки. Метод `toString()` класса `Object` обеспечивает строковое представление любого объекта.
- Строки типа `String` неизменяемые объекты — при каждом изменении содержимого строки создается новый объект-строка. Сделать работу с многочисленными присваиваниями более эффективной позволяют классы `StringBuffer` и `StringBuilder`.
- Иногда необходимы элементы, которые не являются ни числами, ни строками, но ведут себя как имена элементов и одновременно обладают порядковыми номерами (например, названия месяцев или дней недели). В этих случаях используют перечисления.
- Вывод графики осуществляется с помощью методов объекта типа `java.awt.Graphics`. Чтобы результаты вывода не пропадали, в классе приложения требуется переопределить метод `paint`, вызываемый при отрисовке, а также обработчик события `ComponentResized`.

- Исключительные ситуации в Java являются объектами. Их типы — это классы-потомки объектного типа `Throwable`. От `Throwable` наследуются классы `Error` (ошибка) и `Exception` (исключение). Экземпляры класса `Error` — непроверяемые исключительные ситуации, которые невозможно перехватить в блоках `catch`. Экземпляры класса `Exception` и его потомков — проверяемые исключительные ситуации (кроме одного потомка — класса `RuntimeException` и его потомков).
- Непроверяемые исключения генерируются и обрабатываются системой автоматически, как правило, приводя к завершению приложения. При этом их типы нигде не указываются, а слово `throws` в заголовке метода указывать не нужно. Если же в теле реализуемого метода присутствует вызов метода, который может возбуждать исключительную ситуацию, и это исключение не перехватывается, то в заголовке реализуемого метода требуется указывать соответствующий тип возбуждаемого исключения.
- Объекты типа `File` обеспечивают работу с именами файлов и папок (проверку существования файла или папки с заданным именем, нахождение абсолютного пути по относительному и наоборот, проверку и установку атрибутов файлов и папок).
- При работе с файлами в подавляющем большинстве приложений требуется вызов файлового диалога `JFileChooser` (пакет `javax.swing`).
- Потоки ввода/вывода обеспечивают работу не только с файлами, но и с памятью, а также с различными устройствами ввода/вывода. Соответствующие классы расположены в пакете `java.io`. Абстрактный класс `InputStream` (входной поток) инкапсулирует модель входных потоков, позволяющих считывать из них данные. Абстрактный класс `OutputStream` (выходной поток) инкапсулирует модель выходных потоков, позволяющих записывать в них данные.
- Чтение строк (в виде массива символов) выполняют потомки абстрактного класса `Reader` (читатель) (в частности, чтение из файла — класс `FileReader`). Аналогично для записи строк служат классы-потомки абстрактного класса `Writer` (писатель) (в частности, для записи массива символов в файл — класс `FileWriter`).
- Имеется еще один важный класс для работы с файлами — `RandomAccessFile` (файл с произвольным доступом), предназначенный для чтения и записи данных в произвольном месте файла. Такой файл с точки зрения класса `RandomAccessFile` представляет массив байтов, сохраненный на внешнем носителе. Класс `RandomAccessFile` не относится к абстрактным, поэтому можно создавать его экземпляры.

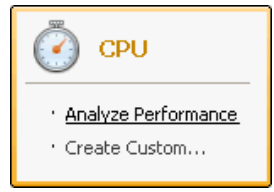
Типичные ошибки

- Забывают, что ячейки в массиве имеют индексы, начинающиеся с нуля, и что для массива длиной `n` индекс последнего элемента равен `n-1`.
- При неаккуратном использовании анонимного массива для задания начальных значений массива создается иррегулярный массив без диагностики ошибки на этапе компиляции. Ошибку можно отследить только во время выполнения приложения.
- Очень распространенная ошибка — попытка сравнения строк с помощью оператора `==`. Его следует заменять методом `equals`.

- Путают неинициализированную строковую переменную (содержащую ссылку `null`) с пустой строкой, представляющей собой полноценный объект типа `String`.

Задания

- Создать приложение Java с графическим пользовательским интерфейсом, где предусмотреть перечисление `Spring` (весна), в котором отражены весенние месяцы, а также кнопки **m2=m1** и **Вывести значение m2**. Задать две переменные типа `Spring` — `m1` и `m2` (переменную `m1` — инициализированной со значением `April`, переменную `m2` — не инициализированной). При нажатии на кнопку **m2=m1** переменной `m2` должно присваиваться значение `m1`. При нажатии на кнопку **Вывести значение m2** должно выводиться значение переменной `m2`.
- Написать пример с графическим пользовательским интерфейсом для записи численных данных в файл и для считывания их из файла с помощью классов `FileInputStream` и `FileOutputStream` (потомков `InputStream` и `OutputStream`).
- Написать пример с графическим пользовательским интерфейсом для записи численных данных в файл и для считывания их из файла с помощью класса `RandomAccessFile`.
- Написать пример с графическим пользовательским интерфейсом для записи строковых данных в файл из компонента `JTextArea` и для считывания их из файла с помощью класса `RandomAccessFile`.
- Проверить работу с русским языком при записи и чтении текста в различных кодировках (ANSI, Unicode, UTF-8). В этом и двух предыдущих примерах реализовать диалоги выбора файлов.
- Написать пример сложения, вычитания, умножения матриц с числом строк и столбцов, задаваемых пользователем. Для визуального представления матриц использовать компоненты `JTable`. Действия проводить с помощью массивов чисел типа `double`.
- Усложнить пример, добавив возможность сохранения каждой из трех матриц в текстовом файле, а также возможность загрузки первых двух матриц из текстового файла.



ГЛАВА 9

Интерфейсы и композиция

9.1. Проблемы множественного наследования классов. Интерфейсы

Достаточно часто требуется совмещать в объекте поведение, характерное для двух или более независимых иерархий. Но еще чаще необходимо писать единый полиморфный код для объектов из таких иерархий в случае, когда эти объекты обладают схожим поведением. Как мы знаем, за поведение объектов отвечают методы. Это значит, что в полиморфном коде для объектов из разных классов нужно вызывать методы, имеющие одинаковую сигнатуру, но разную реализацию. *Унарное наследование*, которое мы изучали до сих пор и при котором у класса может быть только один прародитель, не обеспечивает такой возможности. При унарном наследовании нельзя ввести переменную, которая могла бы ссылаться на экземпляры из разных иерархий, т. к. она должна иметь тип, совместимый с базовыми классами этих иерархий.

В C++ для решения таких задач используется *множественное наследование*. Оно означает, что у класса может быть не один непосредственный прародитель, а два или более. В этом случае совместимость с классами из разных иерархий обеспечивается созданием класса, наследующего от необходимого числа классов-прародителей.

Но при множественном наследовании классов возникает ряд трудноразрешимых проблем, поэтому в Java оно не поддерживается. Основные причины отказа от множественного наследования классов: наследование ненужных полей и методов, а также конфликты совпадающих имен из разных ветвей наследования.

В частности, существует так называемое *ромбовидное наследование*, когда у класса `A` имеются наследники `B` и `C`, а от них наследуется класс `D`. Поэтому класс `D` получает поля и методы, имеющиеся в классе `A`, в удвоенном количестве — один комплект по линии родителя `B`, другой — по линии родителя `C`.

Конечно, в C++ имеются средства для преодоления указанных трудностей. Например, возможность различать имена полей и методов, доставшихся от разных прародителей, с помощью дополнительного указания имени класса, из которого берется поле или метод.

А проблемы ромбовидного наследования в значительной степени снимаются применением так называемых виртуальных классов, благодаря чему при ромбовидном наследо-

вании потомкам достается только один комплект членов класса A. Но в результате заметно усложняется логика работы с классами и объектами, что вызывает логические ошибки, не отслеживаемые компилятором. Поэтому в Java реализовано *множественное наследование с помощью интерфейсов*, лишенное практически всех указанных недостатков.

Интерфейсы — важнейшие элементы языков программирования, применяемые как для написания полиморфного кода, так и для межпрограммного обмена. Концепция интерфейсов Java была первоначально заимствована из языка Objective-C, но в дальнейшем развивалась и дополнялась самостоятельно. В настоящее время она фактически стала основой объектного программирования в Java, заменив концепцию множественного наследования, используемого в C++.

Интерфейсы являются специальной разновидностью полностью абстрактных классов (т. е. таких классов, в которых вообще нет реализованных методов — все методы абстрактные). Полей данных в них также нет, но можно задавать константы (неизменяемые переменные класса). Класс в Java должен быть наследником одного класса-родителя и может быть наследником произвольного числа интерфейсов. Сами интерфейсы тоже могут наследоваться от интерфейсов, причем также с разрешением на множественное наследование.

Отсутствие в интерфейсах полей данных и реализованных методов снимает почти все проблемы множественного наследования и обеспечивает изящный инструмент для написания полиморфного кода. Например, все коллекции обладают методами `add`, `addAll`, `clear`, `contains` и рядом других, это обеспечивается тем, что их классы являются наследниками интерфейса `Collection`. Аналогично, все классы итераторов — наследники интерфейса `Iterator` и т. д.

Имеется еще одно применение интерфейсов, которое появилось в JDK 5, — это использование их в качестве метки, маркера (marker). Можно пометить ряд методов с помощью конструкции `@ИмяИнтерфейса`, чтобы компилятор мог их обнаружить и задействовать в определенных целях. Например, методы, помеченные спецификатором `@Action`, где `Action` — интерфейс, заданный в пакете `application`, могут играть роль обработчиков событий для объекта `action` *(см. разд. 13.5)*.

Декларация интерфейса очень похожа на декларацию класса (листинг 9.1).

Листинг 9.1. Декларация интерфейса

```
МодификаторВидимости interface ИмяИнтерфейса
  extends ИмяИнтерфейса1, ИмяИнтерфейса2,..., ИмяИнтерфейсаN{
  декларация констант;
  декларация заголовков методов;
}
```

У интерфейсов может быть только два варианта видимости. Если в качестве модификатора видимости задано слово `public`, то интерфейс общедоступный. Если же модификатор отсутствует, то интерфейсу обеспечивается видимость по умолчанию — пакетная. В списке прародителей, расширением которых является данный интерфейс, указываются прародительские интерфейсы `ИмяИнтерфейса1`, `ИмяИнтерфейса2` и т. д. Если этот список пуст, то интерфейс не имеет прародителя.

Для имен интерфейсов в Java нет специальных правил, за исключением того, что их имена (как и других объектных типов) принято начинать с заглавной буквы. Автор предпочитает добавлять к именам интерфейсов префикс `I` (от слова Interface), чтобы их можно было легко отличать от имен классов. А переменной типа `интерфейс` ставится префикс `i`. Например, интерфейс получит имя `IScalable`, а переменная — имя `iScalable`.

Объявление константы осуществляется почти так же, как в классе:

```
МодификаторВидимости Тип ИмяКонстанты = значение;
```

Необязательным модификатором видимости может быть слово `public`. Либо модификатор должен отсутствовать, но при этом видимость также считается `public`, а не пакетной. Еще одно отличие от декларации в классе: при задании полей в интерфейсе все они автоматически считаются окончательными (модификатор `final`), т. е. без права изменения, да к тому же являющимися переменными класса (модификатор `static`). Сами модификаторы `static` и `final` при этом ставить не нужно.

Декларация метода в интерфейсе очень похожа на декларацию абстрактного метода в классе — указывается только заголовок метода:

```
МодификаторВидимости Тип ИмяМетода(списокПараметров)
 throws списокИсключений;
```

В качестве модификатора видимости, как и в предыдущем случае, можно использовать слово `public`. При отсутствии модификатора видимость также считается `public`, а не пакетной. В списке исключений через запятую перечисляют типы проверяемых исключений (потомки `Exception`), которые может возбуждать метод. Часть `throws списокИсключений` необязательная. При задании в интерфейсе все методы автоматически считаются общедоступными (`public`) и абстрактными (`abstract`).

Листинг 9.2 иллюстрирует пример задания интерфейса.

Листинг 9.2. Пример задания интерфейса

```
package figures_pkg;

public interface IScalable {
    public int  getSize();
    public void setSize(int newSize);
}
```

Класс можно наследовать от одного родительского класса и от произвольного количества интерфейсов. Но слово `extends` при наследовании от интерфейса заменяют зарезервированным словом `implements` (реализует). Говорят, что класс-наследник интерфейса реализует соответствующий интерфейс, т. к. он обязан реализовать все его методы. Это гарантирует, что объекты для любых классов, наследующих некий интерфейс, могут вызывать методы этого интерфейса, что позволяет писать полиморфный код для объектов из разных иерархий классов. При реализации возможно добавление новых полей и методов, как и при обычном наследовании. Поэтому можно считать, что это просто один из вариантов наследования, обладающий некоторыми особенностями.

Уточнение

В абстрактных классах, реализующих интерфейс, реализации методов может не быть — в этом случае наследуется только декларация абстрактного метода из интерфейса.

Замечание

Интерфейс также может реализовать (implements) другой интерфейс, если в нем при задании типа использовался *шаблон* (generic declaration — параметризуемый тип). Но эта тема выходит за пределы данного учебного пособия.

В наследнике класса, реализующего интерфейс, можно переопределить методы этого интерфейса. Повторное указание в списке родителей интерфейса, который уже был унаследован кем-то из прародителей, разрешено как в интерфейсах, так и в классах, реализующих интерфейсы.

Реализации у сущности типа `интерфейс` не бывает, как и для абстрактных классов (т. е. экземпляров интерфейсов не бывает). Но, как и для абстрактных классов, можно вводить переменные типа `интерфейс`, которые могут ссылаться на объекты, принадлежащие классам, реализующим соответствующий интерфейс (т. е. классам-наследникам этого интерфейса).

Правила совместимости таковы: переменной типа `интерфейс` можно присваивать ссылку на объект любого класса, реализующего этот интерфейс.

С помощью переменной типа `интерфейс` разрешается вызывать только методы, декларированные в данном интерфейсе, а не любые методы данного объекта (если, конечно, не использовать приведение типа).

Основное назначение переменных интерфейсного типа — вызов с их помощью методов, продекларированных в соответствующем интерфейсе. Если такой переменной назначена ссылка на объект, то можно гарантировать, что из этого объекта разрешено вызывать данные методы, независимо от того, к какому классу принадлежит объект.

Ситуация очень схожа с полиморфизмом на основе виртуальных и динамических методов объектов. Но гарантией работоспособности служит одинаковость сигнатуры методов в разных, а не в одной иерархии, благодаря совпадению с декларацией одного и того же интерфейса. Методы обязательно должны быть методами объектов — полиморфизм на основе методов класса невозможен, т. к. эти методы вызываются путем статического связывания (на этапе компиляции).

9.2. Отличия интерфейсов от классов. Проблемы наследования интерфейсов

Сформулируем основные отличительные черты интерфейсов:

- не бывает экземпляров типа `интерфейс`, т. е. экземпляров интерфейсов, реализующих тип интерфейса (что очевидно, если вспомнить, что интерфейс является разновидностью полностью абстрактного класса, а у абстрактных классов экземпляров не бывает);
- список элементов интерфейса может включать только методы и константы, поля данных недопустимы;

- элементы интерфейса всегда имеют тип видимости `public` (в том числе без явного указания). Не разрешены модификаторы видимости, отличные от `public`;
- в интерфейсах не бывает конструкторов и деструкторов;
- методы, продекларированные в интерфейсе, являются абстрактными по умолчанию, но разрешается явным образом записывать модификатор `abstract`; методы не могут иметь модификаторы `static`, `native`, `synchronized`, `final`, `private`, `protected`;
- интерфейс, как и класс, наследует все методы прародителя, однако только на уровне абстракций, без реализации методов (т. е. интерфейс наследует лишь обязательность реализации этих методов в классе, поддерживающем данный интерфейс);
- наследование через интерфейсы может быть множественным. В заголовке интерфейса можно указать, что интерфейс наследуется от одного или нескольких прародительских интерфейсов;
- реализация интерфейса может быть только в классе; если при этом класс не абстрактный, он должен реализовать все методы интерфейса;
- наследование класса от интерфейсов также может быть множественным.

Кроме указанных отличий имеется еще одно, связанное с проблемой множественного наследования. В наследуемых классом интерфейсах могут содержаться методы, имеющие одинаковую сигнатуру (хотя, возможно, и отличающиеся контрактом). Как выбрать в классе или при вызове из объекта тот или иной из этих методов? Ведь, в отличие от перегруженных методов, компилятор не сможет их различить. Такая ситуация называется конфликтом имен.

Аналогичная ситуация может возникнуть и с константами, хотя, в отличие от методов, реально этого почти никогда не происходит.

При использовании констант из разных интерфейсов необходима квалификация имени константы именем соответствующего интерфейса (аналогично разрешению конфликта имен в случае пакетов). Пример приведен в листинге 9.3.

Листинг 9.3. Задание в интерфейсе константы

```
public interface I1 {
    Double PI=3.14;
}

public interface I2 {
    Double PI=3.1415;
}

class C1 implements I1,I2 {
 void m1(){
   System.out.println("I1.PI="+ I1.PI);
   System.out.println("I2.PI="+ I2.PI);
  }
}
```

Но для методов такой способ различения имен запрещен. Поэтому нельзя наследовать методы, имеющие совпадающие сигнатуры, но разные контракты. Если сигнатуры различаются, то проблем нет, и используется перегрузка.

Методы, объявленные в интерфейсе, могут возбуждать проверяемые исключения. Контракты методов считаются совместимыми в случае, когда контракты отличаются только типом возбуждаемых исключительных ситуаций, причем классы этих исключений лежат в одной иерархии. При этом в классе, реализующем интерфейс, метод должен быть объявлен как возбуждающий совместимый с интерфейсом тип исключения (т. е. либо такой же, как в прародительском интерфейсе, либо являющийся наследником этого типа).

Подведем некоторый итог.

В том случае, если класс `A2` на уровне абстракций ведет себя так же, как класс `A1`, но, кроме того, обладает дополнительными особенностями поведения, следует использовать наследование (т. е. считать класс `A2` наследником класса `A1`). Действует правило:

"`A2 есть A1`" (`A2 is A1`).

Если для нескольких классов `A1`, `B1`,... из разных иерархий можно на уровне абстракций выделить общность поведения, то следует задать интерфейс `I`, который описывает эти абстракции поведения, а классы задать как наследующие этот интерфейс. Таким образом, действует правило:

"`A1, B1,... есть I`" (по-английски это звучит как "`A1, B1,... is I`").

Множественное наследование в Java может быть двух видов:

- только от интерфейсов, без наследования реализации;
- от класса и от интерфейсов, с наследованием реализации от прародительского класса.

Если класс-прародитель унаследовал какой-либо интерфейс, то все его потомки также будут наследовать этот интерфейс.

9.3. Пример работы с интерфейсами

В листинге 9.4 приведен пример абстрактного класса, являющегося наследником `Figure` и реализующего указанный ранее интерфейс `IScalable`.

Листинг 9.4. Реализация в абстрактном классе `ScalableFigure` интерфейса `IScalable`

```
package figures_pkg;

public abstract class ScalableFigure extends Figure implements IScalable {
   private int size;

   public int getSize() {
       return size;
   }

   public void setSize(int size) {
      this.size=size;
   }
}
```

В качестве наследника приведем код класса `Circle` (листинг 9.5).

Листинг 9.5. Класс `Circle` — наследник `ScalableFigure`

```
package figures_pkg;
import java.awt.*;

public class Circle extends ScalableFigure {

    public   Circle(Graphics g,Color bgColor, int r){
        setGraphics(g);
        setBgColor(bgColor);
        setSize(r);
    }

    public  Circle(Graphics g,Color bgColor){
        setGraphics(g);
        setBgColor(bgColor);
        setSize( (int)Math.round( Math.random()*40 ) );
    }

    public void show(){
        Color oldC=getGraphics().getColor();
        getGraphics().setColor(Color.BLACK);
        getGraphics().drawOval( getX(), getY(), getSize(), getSize() );
        getGraphics().setColor(oldC);
    }

    public void hide(){
        Color oldC=getGraphics().getColor();
        getGraphics().setColor(getBgColor());
        getGraphics().drawOval( getX(), getY(), getSize(), getSize() );
        getGraphics().setColor(oldC);
    }

}
```

Пример наследования интерфейса от интерфейса иллюстрирует листинг 9.6.

Листинг 9.6. Наследование интерфейса от интерфейса

```
package figures_pkg;

public interface IStretchable extends IScalable{
    double getAspectRatio();
    void setAspectRatio(double aspectRatio);
    int getWidth();
    void setWidth(int width);
    int getHeight();
    void setHeight(int height);
}
```

Интерфейс IScalable описывает методы объекта, способного менять свой размер (size). При этом отношение ширины к высоте (AspectRatio — отношение сторон) у фигуры не меняется. Интерфейс IStretchable описывает методы объекта, способного менять не только свой размер, но и растягиваться (изменять отношение ширины к высоте).

К интерфейсам применимы как оператор instanceof, так и приведение типов. Например, листинг 9.7 содержит фрагмент кода для изменения случайным образом размера объекта с помощью интерфейса IScalable.

Листинг 9.7. Использование для интерфейса оператора instanceof и приведения типа

```
Object object;
...
object= new Circle(...);//конструктор создает окружность
...
if(object instanceof IScalable){
    ((IScalable) object).setSize( (int)(Math.random()*80) );
}
```

Все очень похоже на использование класса ScalableFigure (листинг 9.8).

Листинг 9.8. Использование для класса оператора instanceof и приведения типа

```
ScalableFigure figure;
...
figure = new Circle(...);//конструктор создает окружность
...
if( figure instanceof ScalableFigure){
    figure.hide();
    ((ScalableFigure)figure).setSize( (int)(Math.random()*80) );
    figure.show();
}
```

Но если во втором случае переменная figure имеет тип Figure, т. е. связанный с ней объект обязан быть фигурой, то на переменную object такое ограничение не распространяется. Зато фигуру можно скрывать и показывать, а для переменной типа Object это можно делать только после проверки, что object является экземпляром (т. е. instanceof) класса Figure.

Аналогичный код можно написать для переменной типа IScalable (листинг 9.9).

Листинг 9.9. Использование переменной интерфейсного типа

```
IScalable scalableObj;
...
scalableObj = new Circle(...);//конструктор создает окружность
...
scalableObj.setSize( (int)( Math.random()*80 ) );
```

Заметим, что присваивание Object object= new Circle(...) разрешено, т. к. Circle — наследник Object.

Аналогично, присваивание `Figure figure = new Circle(...)` разрешено, т. к. `Circle` — наследник `Figure`.

И, наконец, присваивание `scalableObj = new Circle(...)` разрешено, т. к. `Circle` — наследник `IScalable`.

При замене в коде `Circle(...)` на `Dot(...)` мы получили бы правильный код в первых двух случаях, а вот присваивание `scalableObj = new Dot(...)` вызвало бы ошибку компиляции, т. к. класс `Dot` не реализует интерфейс `IScalable`, т. е. не является его потомком.

9.4. Композиция — еще одна альтернатива множественному наследованию

Как уже говорилось ранее, наследование затрагивает один из важных аспектов, присущих объектам, — поведение. Причем оно относится не к самим объектам, а к классам. Но имеется и другой аспект, присущий объектам, — внутреннее устройство. При наследовании это обстоятельство скорее скрывается, нежели подчеркивается: наследники должны быть устроены так, чтобы отличие в их устройстве не сказывалось на абстракциях их поведения.

Композиция — это такое устройство объекта, когда он либо состоит из других объектов, предназначенных для того, чтобы быть его частями (отношение агрегации или включения), либо находится с ними в отношении ассоциации (использования независимых объектов).

Таким образом, композиция — это объединение частей в единую систему.

Если наследование характеризуется отношением "`is-a`" (это есть, является), то композиция характеризуется отношением "`has-a`" (имеет в своем составе, состоит из) и "`use-a`" (использует).

Важность композиции связана с тем, что она позволяет *объединять отдельные части в единую более сложную систему*. Причем описание и испытание работоспособности отдельных частей можно делать независимо от других частей, а тем более независимо от всей сложной системы.

В качестве примера агрегации можно привести классический пример — автомобиль. Он *состоит из* корпуса, колес, двигателя, карбюратора, топливного бака и т. д. Каждая из этих частей, в свою очередь, состоит из более простых деталей и т. д. до того уровня, когда деталь можно считать единым целым, не включающим в себя другие объекты.

Шофер также является неотъемлемой частью автомобиля, но вряд ли можно считать, что автомобиль состоит из шофера и других частей. Однако можно сказать, что у автомобиля обязательно должен быть шофер, либо нужно говорить, что шофер *использует* автомобиль. Отношение объекта `автомобиль` и объекта `шофер` гораздо слабее, чем агрегация, но все-таки весьма сильное — это композиция в узком смысле этого слова.

И наконец, отношение автомобиля с находящимися в нем сумками или другими посторонними предметами — это ассоциация (т. е. отношение независимых предметов, которые на некоторое время образовали единую систему). В таких случаях говорят, что автомобиль *используют* для того, чтобы отвезти предметы по нужному адресу.

С точки зрения программирования на Java композиция любого вида — это наличие в объекте поля ссылочного типа. Вид композиции определяется условиями создания связанного с этой ссылочной переменной объекта и изменения этой ссылки. Если такой вспомогательный объект создается одновременно с главным объектом и умирает вместе с ним — это агрегация. В противном случае это или композиция в узком смысле слова, или ассоциация.

Композиция во многих случаях может служить альтернативой множественному наследованию, причем именно в тех ситуациях, когда наследование интерфейсов не работает. Так бывает, когда необходимо унаследовать от двух или более классов их поля и методы.

Приведем пример.

Пусть у нас имеются: класс `Car` (Автомобиль), класс `Driver` (Шофер) и класс `Speed` (Скорость), и пусть это совершенно независимые классы. Зададим класс `MovingCar` (Движущийся автомобиль):

```
public class MovingCar extends Car{
  Driver driver;
  Speed  speed;
  ...
}
```

Отличительной чертой объектов `MovingCar` будет то, что они включают в себя не только особенности *поведения автомобиля*, но и все особенности объектов типа `Driver` и `Speed`.

Например, автомобиль знает своего водителя: если у нас имеется объект `movingCar`, то `movingCar.driver` обеспечит доступ к объекту `Водитель` (если, конечно, ссылка не равна `null`), в результате чего можно будет пользоваться общедоступными (и только!) методами этого объекта. То же относится к полю `speed`. И нам не придется строить гибридный класс-монстр, в котором от родителей `Car`, `Driver` и `Speed` унаследованы поля и методы по механизму множественного наследования (нечто вроде машино-кентавра, где шофера скрестили с автомобилем). Причем в случае композиции не нужно заниматься реализацией в классе-наследнике интерфейсов, описывающих взаимодействие автомобиля с шофером и измерение/задание скорости.

Но у композиции имеется заметный недостаток: для получившегося класса существует ограничение при использовании полиморфизма, т. к. он не является наследником классов `Driver` и `Speed`. Поэтому полиморфный код, написанный для объектов типа `Driver` и `Speed`, для объектов типа `MovingCar` работать не будет. И хотя такой код будет функционировать для соответствующих полей `movingCar.driver` и `movingCar.speed`, это не всегда помогает (например, если объект должен помещаться в список). Тем не менее, часто композиция — гораздо более удачное решение, чем множественное наследование.

Таким образом, сочетание множественного наследования интерфейсов и композиции в подавляющем большинстве случаев является полноценной альтернативой множественному наследованию классов.

Краткие итоги по главе

- Интерфейсы используются для написания полиморфного кода для классов, лежащих в различных, никак не связанных друг с другом иерархиях.
- Интерфейсы описываются аналогично абстрактным классам. Так же, как абстрактные классы, они не могут иметь экземпляров. Но, в отличие от абстрактных классов, интерфейсы не могут иметь полей данных (за исключением констант), а также не могут иметь реализации никаких своих методов.
- Интерфейс определяет методы, которые должны быть реализованы классом-наследником этого интерфейса.
- Хотя экземпляров типа `интерфейс` не бывает, но могут существовать переменные типа `интерфейс`. Такая переменная — это ссылка. Она дает возможность ссылаться на объект, чей класс реализует данный интерфейс.
- С помощью переменной типа `интерфейс` разрешается вызывать только методы, декларированные в данном интерфейсе, а не любые методы данного объекта.
- *Композиция* — это такое устройство объекта, когда он либо состоит из других объектов, предназначенных для того, чтобы быть его частями (отношение агрегации или включения), либо находится с ними в отношении ассоциации (использования независимых объектов). Композиция позволяет объединять отдельные части в единую более сложную систему.
- Наследование характеризуется отношением: "`is-a`" (это есть, является), а композиция — отношением "`has-a`" (имеет в своем составе, состоит из) и "`use-a`" (использует).
- Сочетая множественное наследование интерфейсов и композицию, в подавляющем большинстве случаев можно заменить множественное наследование классов.

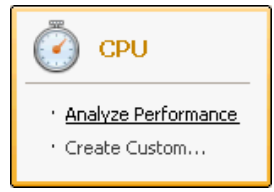
Типичные ошибки

- После настройки ссылки, хранящейся в переменной типа `интерфейс`, на объект какого-либо класса, реализующего этот интерфейс, пытаются вызвать поле или метод данного объекта, не объявленные в интерфейсе. Для такого вызова требуется приведение типа, причем до него рекомендуется проверить соответствие объекта этому типу.

Задания

- Написать реализацию класса `Square` — наследника `ScalableFigure`. Класс должен располагаться в пакете `AdditionalFigures`.
- В пакете `AdditionalFigures` задать интерфейс `IScalable`.
- В качестве класса, реализующего этот интерфейс, написать абстрактный класс `StretchableFigure`. Класс должен располагаться в пакете `AdditionalFigures`.
- Написать реализацию класса `Rectangle` — наследника `StretchableFigure`. Класс должен располагаться в пакете `AdditionalFigures`.

- Написать приложение, в котором в зависимости от выбранной радиокнопки создается и отрисовывается на панели в произвольном месте, не выходящем за пределы панели, точка, окружность, квадрат или прямоугольник. При нажатии на кнопки **Создать объект**, **show**, **hide**, **moveTo** должны выполняться соответствующие методы для последнего созданного объекта.
- Усложнить копию данного приложения, добавив на форму компонент с прокручивающимся или выпадающим списком, содержащим имена объектов. Имя объекта должно состоять из имени, соответствующего типу, и порядкового номера (`dot1`, `circle3` и т. п.). При нажатии на кнопки **Создать объект**, **show**, **hide**, **moveTo** должны выполняться соответствующие методы для объекта, выделенного в списке.
- Добавить кнопки **Изменить размер** и **Растянуть объект**. В случае если объект поддерживает интерфейс `ScalableFigure`, то при нажатии первой из них он должен менять размер. Если он поддерживает интерфейс `StretchableFigure`, то при нажатии второй кнопки должен растягиваться или сплющиваться в зависимости от значения в соответствующем пункте ввода.
- В пакете `AdditionalFigures` написать интерфейс `IBordered`, обеспечивающий поддержку методов, необходимых для рисования границы (`border`) заданной ширины и цвета вокруг графического объекта. Реализовать этот интерфейс в классах `BorderedCircle`, `BorderedSquare`, `BorderedRectangle`.



ГЛАВА 10

Многопоточное программирование и многоядерные системы

10.1. Потоки выполнения (threads) и синхронизация

В многозадачных операционных системах (MS Windows, Linux и др.) программу, выполняющуюся под управлением операционной системы (ОС), принято называть приложением операционной системы (application) или процессом (process). Обычно в ОС параллельно (или псевдопараллельно — в режиме разделения процессорного времени) выполняется большое число процессов. Для выполнения процесса на аппаратном уровне поддерживается независимое от других процессов виртуальное адресное пространство. Попытки процесса выйти за пределы адресов этого пространства отслеживаются аппаратно.

Такая модель удобна для разграничения независимых программ. Однако во многих случаях она не подходит и приходится использовать подпроцессы (subprocesses) или (более употребительное название) threads. Дословный перевод слова threads — нити. Иногда их называют легковесными процессами (lightweight processes), т. к. при прочих равных условиях они потребляют гораздо меньше ресурсов, чем процессы. Мы будем употреблять термин "потоки выполнения", поскольку термин multithreading (работа в условиях существования нескольких потоков) на русский язык гораздо точнее переводится как *многопоточность*. Следует соблюдать аккуратность, чтобы не путать threads с потоками ввода/вывода (streams).

Потоки выполнения отличаются от процессов тем, что находятся в адресном пространстве своего родительского процесса. Они выполняются параллельно (псевдопараллельно), но в отличие от процессов легко могут обмениваться данными в пределах общего виртуального адресного пространства (т. е. у них могут иметься общие переменные, в том числе массивы и объекты).

В приложении всегда имеется главный (основной) поток выполнения. Поток может создавать дочерние потоки. При этом если родительский поток закрывается, то закрываются все его обычные (пользовательские) дочерние потоки. Кроме пользовательских потоков возможно создание потоков-демонов (daemons), которые могут продолжать работу и после окончания работы своего родительского потока выполнения. Выполнение приложения заканчивается в случае, когда завершены все пользовательские потоки.

При этом все потоки-демоны завершаются без каких-либо ожиданий и предупреждений.

Любая программа Java неявно использует потоки выполнения. В главном потоке виртуальная Java-машина (JVM) запускает метод `main` приложения, а также все методы, вызываемые из него. Главному потоку автоматически дается имя `"main"`. Кроме главного потока в фоновом режиме (с малым приоритетом) запускается дочерний поток, занимающийся сборкой мусора. Виртуальная Java-машина автоматически стартует при запуске на компьютере хотя бы одного приложения Java и завершает работу в случае, если у нее на выполнении остаются только потоки-демоны.

В Java каждый поток выполнения рассматривается как объект. Но интересно то, что в Java каждый объект, даже не имеющий никакого отношения к классу `Thread`, может работать в условиях многопоточности, поскольку в классе `Object` определены методы объектов, предназначенные для взаимодействия объектов в таких условиях. Это `notify()` (оповестить), `notifyAll()` (оповестить всех), `wait()` (ждать), `wait(timeout)` (ждать до истечения тайм-аута). Об этих методах будет рассказано далее.

10.2. Преимущества и проблемы при работе с потоками выполнения

Почему бывают нужны потоки выполнения? Представьте себе программу управления спектрометром, в которой одно табло должно показывать время, прошедшее с начала измерений, второе — число импульсов со счетчика установки, третье — длину волны, для которой в данный момент идут измерения. Кроме того, в фоновом режиме должен отрисовываться получающийся после обработки данных спектр. Возможны две идеологии работы программы — последовательная и параллельная. При последовательном подходе во время выполнения алгоритмов, связанных с показом информации на экране, следует время от времени проверять, не пришли ли новые импульсы со счетчиков, и не нужно ли установить спектрометр на очередную длину волны. Кроме того, во время обработки данных и отображения спектра следует через определенные промежутки обновлять табло времени и счетчиков.

В результате код программы перестает быть структурным. Нарушается принцип инкапсуляции — "независимые вещи должны быть независимы". Независимые по логике решаемой проблемы алгоритмы оказываются перемешаны друг с другом. Такой код получается ненадежным и плохо модифицируемым. Однако он относительно легко отлаживается, поскольку последовательность выполнения операторов программы однозначно определена.

В параллельном варианте для каждого из независимых алгоритмов запускают свой поток выполнения, и ему задают необходимый приоритет. В нашем случае один (или более) поток занимается измерениями; второй — показывает время, прошедшее с начала измерений; третий — число импульсов со счетчика; четвертый — длину волны; пятый поток рисует спектр. Каждый из них занят своим делом и не вмешивается в другой. Связь между потоками идет только через данные, которые первый поток поставляет остальным.

Несмотря на изящество параллельной модели, у нее имеются очень неприятные недостатки.

Во-первых, негарантированное время отклика. Например, при анимации графических объектов может наблюдаться сбой равномерности движения объекта.

Во-вторых, отладка программ, работающих параллельно, с помощью традиционных средств практически невозможна. Если при каком-то сочетании условий по времени происходит ошибка, то воспроизвести ее обычно оказывается невозможно, поскольку воссоздать те же временные интервалы крайне сложно. Поэтому требуется применять специальные технологии с регистрацией всех потоков данных в файлы с отладочной информацией. Для этих целей, а также для нахождения участков кода, вызывающих наибольшую трату времени и ресурсов при выполнении приложения, предусмотрены специальные программы — отладчик (debugger), профилировщик (profiler) — *см. главу 17*. В NetBeans 6.5 отладчик стал работать в многопоточном режиме, что значительно расширило его возможности.

10.3. Синхронизация по ресурсам и событиям

Использование потоков выполнения порождает целый ряд проблем не только в плане отладки программ, но и при организации взаимодействия между разными потоками.

Синхронизация по ресурсам

Если разные потоки получают доступ к одним и тем же данным, причем один из них или они оба меняют эти данные, то для них требуется установить разграничение доступа — пока один поток меняет данные, второй не должен иметь права их читать или менять, он должен дожидаться окончания доступа к данным первого потока. Говорят, что осуществляется *синхронизация* потоков. В Java для этих целей служит оператор `synchronize` (синхронизировать). Такой тип синхронизации называется *синхронизацией по ресурсам* и обеспечивает блокировку данных на то время, которое необходимо потоку для выполнения тех или иных действий.

В Java такого рода блокировка осуществляется на основе концепции *мониторов* и заключается в следующем. Под монитором понимается некая управляющая конструкция, обеспечивающая монопольный доступ к объекту. Если во время выполнения синхронизованного метода объекта другой поток попробует обратиться к методам или данным этого объекта, он будет заблокирован до тех пор, пока не закончится выполнение синхронизованного метода. При запуске синхронизованного метода говорят, что объект *входит в монитор*, при завершении — что объект *выходит из монитора*. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Имеются два способа синхронизации по ресурсам: синхронизация объекта и синхронизация метода.

Синхронизация объекта `obj1` (его иногда называют *объектом действия*) осуществляется следующим образом:

```
synchronized(obj1) оператор;
```

Листинг 10.1 иллюстрирует пример.

Листинг 10.1. Синхронизация объекта

```
synchronized(obj1){
   ...
   m1(obj1);
   ...
   obj1.m2();
   ...
}
```

В данном случае в качестве синхронизованного оператора выступает участок кода в фигурных скобках. Во время выполнения этого участка доступ к объекту `obj1` блокируется для всех других потоков. Это означает, что пока будет выполняться вызов оператора, вызовы любого синхронизованного метода или синхронизованного оператора для этого объекта будут приостановлены до окончания работы оператора.

Данный способ синхронизации обычно применяется для экземпляров классов, разработанных без расчета на работу в режиме многопоточности.

Второй способ синхронизации по ресурсам используется при разработке класса, рассчитанного на взаимодействие в многопоточной среде. При этом методы, критичные к атомарности операций с данными (обычно требующие согласованного изменения нескольких полей данных), объявляются как синхронизованные с помощью модификатора `synchronized` (листинг 10.2).

Листинг 10.2. Синхронизация метода

```
public class ИмяКласса{
   ...
   public synchronized тип метод(...){
      ...
   }
}
```

Вызов данного метода из объекта приведет к вхождению данного объекта в монитор (листинг 10.3).

Листинг 10.3. Пример синхронизации метода

```
public class C1{
   public synchronized void m1(){
   }
}
C1 obj1=new C1();
obj1.m1();
```

Пока будет выполняться вызов `obj1.m1()`, доступ из других потоков к объекту `obj1` окажется заблокирован — выполнение вызова любого синхронизованного метода или синхронизованного оператора для этого объекта будет приостановлено до окончания работы метода `m1()`.

Если синхронизованный метод является не методом объекта, а методом класса, то при вызове метода в монитор входит класс, и приостановка до окончания работы метода будет относиться ко всем вызовам синхронизованных методов данного класса.

Итак, возможна синхронизация как целого метода (но только при задании его реализации в классе), так и отдельных операторов. Иногда синхронизованную область кода (метод или оператор) называют *критической секцией* кода.

Синхронизация по событиям

Кроме синхронизации по данным существует *синхронизация по событиям*, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основные операции при таком типе синхронизации: `wait` (ждать) и `notify` (оповестить).

В Java синхронизацию по событиям обеспечивают следующие методы, заданные в классе `Object` и наследуемые всеми остальными классами:

- `void wait()` — поток, внутри которого какой-либо объект вызвал данный метод (владелец монитора), переводится в состояние ожидания. Поток приостанавливает работу своего метода `run()` вплоть до поступления объекту, вызвавшему приостановку (засыпание) потока, уведомления `notify()` или `notifyAll()`. При неправильной попытке разбудить поток соответствующий код компилируется, но при запуске вызывает появление исключения `llegalMonitorStateException`;
- `void wait(long millis)` — то же самое, но ожидание длится не более `millis` миллисекунд;
- `void wait(long millis, int nanos)` — то же самое, но ожидание длится не более `millis` миллисекунд и `nanos` наносекунд;
- `void notify()` — оповещение, приводящее к возобновлению работы потока, ожидающего выхода данного объекта из монитора. Если таких потоков несколько, то выбирается один из них (какой — зависит от реализации системы);
- `void notifyAll()` — оповещение, приводящее к возобновлению работы всех потоков, ожидающих выхода данного объекта из монитора.

Метод `wait` для любого объекта `obj` нужно использовать следующим образом: необходимо организовать цикл `while`, в котором следует выполнить оператор `wait` (листинг 10.4).

Листинг 10.4. Схема использования метода `wait`

```
synchronized(obj){
   while(not условие)
     obj.wait();
   ...//выполнение операторов после того, как условие стало true
}
```

При этом не придется беспокоиться, что цикл `while` постоянно выполняется и занимает много ресурсов процессора. Этого не происходит, поскольку после вызова `obj.wait()` поток, в котором находится указанный код, "засыпает" и перестает занимать ресурсы

процессора. При этом метод `wait` на время сна потока снимает блокировку с объекта `obj`, задаваемую оператором `synchronized(obj)`, позволяя другим потокам обращаться к объекту с вызовом `obj.notify()` или `obj.notifyAll()`.

10.4. Класс *Thread* и интерфейсы *Runnable* и *Callable*. Создание и запуск потока выполнения

Имеются три способа создания класса, экземплярами которого будут потоки выполнения:

- унаследовать класс от `java.lang.Thread`;
- реализовать интерфейс `java.lang.Runnable` (Запускаемый).

  Этот интерфейс имеет декларацию единственного метода `public void run()`, который обеспечивает последовательность действий при работе потока. При этом класс `Thread` уже реализует интерфейс `Runnable`, но с пустой реализацией метода `run()`. Так что при создании экземпляра `Thread` создается поток, который ничего не делает. Поэтому в потомке нужно переопределить метод `run()`. В нем следует написать реализацию алгоритмов, которые должны выполняться в данном потоке. Отметим, что после выполнения метода `run()` поток прекращает существование — умирает;
- реализовать интерфейс `java.util.concurrent.Callable` (Вызываемый).

  Этот интерфейс подобен `Runnable`, но его единственный метод `call()` может возвращать значение и вызывать проверяемые исключения.

Сначала рассмотрим первый способ, когда мы наследуем класс от класса `Thread`, переопределив метод `run()`.

Объект-поток создается с помощью конструктора. Имеются несколько перегруженных вариантов конструкторов, самый простой из них — с пустым списком параметров.

Например, в классе `Thread` их заголовки выглядят так:

- `public Thread()` — конструктор по умолчанию. Поток получает имя `system`;
- `public Thread(String name)` — поток получает имя, содержащееся в строке `name`.

Также можно создавать потоки в группах. Но в связи с тем, что данная технология устарела и не нашла широкого распространения, то о группах потоков выполнения в данном учебном пособии рассказываться не будет.

В классе-потомке можно вызывать конструктор по умолчанию (без параметров) либо задавать свои конструкторы, используя вызовы прародительских с помощью вызова `super(список параметров)`. Из-за отсутствия наследования конструкторов в Java приходится в наследнике заново задавать конструкторы с той же сигнатурой, что и в классе `Thread`. Это является простой, но утомительной работой. Именно поэтому обычно предпочитают способ задания класса с реализацией интерфейса `Runnable`, о чем будет рассказано чуть позже.

Листинг 10.5 иллюстрирует создание и запуск потока.

Листинг 10.5. Первый способ создания и запуска потока

```
public class T1 extends Thread{
   public void run(){
     ...
   }
   ...
}

Thread thread1= new T1();
thread1.start();
```

Второй способ создания класса, экземплярами которого будут потоки выполнения, — использование класса, в котором реализован интерфейс `java.lang.Runnable`. Этот интерфейс, как уже говорилось, имеет единственный метод `public void run()`. Реализовав его в классе, можно создать поток с помощью перегруженного варианта конструктора `Thread` (листинг 10.6).

Листинг 10.6. Второй способ создания и запуска потока

```
public class R1 implements Runnable{
   public void run(){
     ...
   }
   ...
}

Thread thread1= new Thread(new R1());
thread1.start();
```

Обычно таким способом пользуются гораздо чаще, т. к. в разрабатываемом классе не приходится заниматься дублированием конструкторов класса `Thread`. Кроме того, этот способ можно применять в случае, когда уже имеется класс, принадлежащий иерархии, в которой базовым классом не является `Thread` или его наследник, и мы хотим задействовать этот класс для работы внутри потока. В результате от этого класса получаем метод `run()`, в котором реализован нужный алгоритм, и этот метод работает внутри потока типа `Thread`, обеспечивающего необходимое поведение в многопоточной среде. Однако в данном случае несколько затрудняется доступ к методам из класса `Thread`.

Например, чтобы вывести информацию о приоритете потока, при первом способе в методе `run()` нужно написать оператор:

```
System.out.println("Приоритет потока="+this.getPriority());
```

А при втором способе приходится это делать в несколько этапов. Во-первых, при задании класса нам следует добавить в объекты типа `R1` поле `thread` (листинг 10.7).

Листинг 10.7. Пример реализации интерфейса `Runnable`

```
public class R1 implements Runnable{
    public Thread thread;
```

```
    public void run() {
        System.out.println("Приоритет потока="+thread.getPriority());
    }
}
```

С помощью этого поля мы будем добираться до объекта-потока. Но теперь после создания потока нужно не забыть установить для этого поля ссылку на созданный объект-поток. В итоге, создание и запуск потока будет выглядеть так:

```
R1 r1=new R1();
Thread thread1=new Thread(r1, "thread1");
r1.thread=thread1;
thread1.start();//либо, что то же, r1.thread.start()
```

Через поле `thread` мы можем получать доступ к потоку и всем его полям и методам в алгоритме, написанном в методе `run()`. Указанные ранее дополнительные действия — это всего три лишних строчки программы (первая — `R1 r1=new R1();`, вторая — `r1.thread=thread1;`, третья — объявление в классе `R1 – public Thread thread;`).

В классе `Thread` имеются следующие перегруженные варианты конструктора с параметром типа `Runnable`:

- `public Thread(Runnable target)` — с именем `system` по умолчанию;
- `public Thread(Runnable target, String name)` — с заданием имени.

Также имеются варианты с заданием группы потоков.

Третий способ создания класса, экземплярами которого будут потоки выполнения, — использование класса, в котором реализован интерфейс `java. util.concurrent.Callable`, содержащий декларацию единственного метода `call()`.

Этот вариант появился в JDK 5. Интерфейс `Callable` задан как `Callable<V>`, где в треугольных скобках указан так называемый параметризуемый тип. Метод `call` имеет тип возвращаемого значения `V`. В классе, реализующем такой интерфейс, после слов `implements Callable` в треугольных скобках вместо `V` следует указать тип возвращаемого методом `call` значения.

10.5. Поля и методы, заданные в классе *Thread*

В классе `Thread` имеется ряд полей данных и методов, о которых необходимо иметь представление при работе с потоками.

Важнейшие константы и методы класса `Thread`.

- `MIN_PRIORITY` — минимально возможный приоритет потоков. Зависит от операционной системы и версии JVM. На компьютере автора оказался равен 1.
- `NORM_PRIORITY` — нормальный приоритет потоков. Главный поток создается с нормальным приоритетом, а затем приоритет можно изменить. На компьютере автора приоритет оказался равен 5.
- `MAX_PRIORITY` — максимально возможный приоритет потоков. На компьютере автора оказался равен 10.

- `static int activeCount()` — возвращает число активных потоков приложения.
- `static Thread currentThread()` — возвращает ссылку на текущий поток.
- `static void sleep(long millis)` — вызывает приостановку (засыпание) потока на `millis` миллисекунд. При этом все блокировки (мониторы) потока сохраняются. Перегруженный вариант `sleep(long millis, int nanos)`, где параметр `nanos` задает число наносекунд. Досрочное пробуждение осуществляется методом `interrupt()` с возбуждением исключения `InterruptedException`.
- `static void yield()` (поступиться правами) — вызывает временную приостановку потока с передачей права другим потокам выполнять необходимые им действия.
- `static boolean holdsLock(Object obj)` — возвращает `true` в случае, когда текущий поток блокирует объект `obj`.
- `static boolean interrupted()` — возвращает состояние статуса прерывания текущего потока, после чего устанавливает его в значение `false`.

Важнейшие методы объектов типа `Thread`.

- `void run()` — метод, который обеспечивает последовательность действий во время жизни потока. В классе `Thread` задана его пустая реализация, поэтому в классе потока он должен быть переопределен. После выполнения метода `run()` поток умирает.
- `void start()` — вызывает выполнение текущего потока, в том числе запуск его метода `run()` в нужном контексте. Может быть вызван всего один раз.
- `void setDaemon(boolean on)` — в случае `on==true` устанавливает потоку статус демона, иначе — статус пользовательского потока.
- `boolean isDaemon()` — возвращает `true` в случае, когда текущий поток является демоном.
- `long getId()` — возвращает уникальный идентификатор потока. Уникальность относится только ко времени жизни потока — после его завершения (смерти) данный идентификатор может быть присвоен другому создаваемому потоку.
- `String getName()` — возвращает имя потока, которое ему было задано при создании или методом `setName`.
- `void setName(String name)` — устанавливает новое имя потока.
- `int getPriority()` — возвращает приоритет потока.
- `void setPriority(int newPriority)` — устанавливает приоритет потока.
- `void checkAccess()` — осуществляет проверку из текущего потока на позволительность доступа к другому потоку. Если поток, из которого идет вызов, имеет право на доступ, то метод не делает ничего, иначе возбуждает исключение `SecurityException`.
- `String toString()` — возвращает строковое представление объекта потока, в том числе его имя, группу, приоритет.
- `void interrupt()` — прерывает сон потока после вызовов `wait(...)` или `sleep(...)`, устанавливая ему статус прерванного (статус прерывания равен `true`). При этом возбуждается проверяемая исключительная ситуация `InterruptedException`.
- `boolean isInterrupted()` — возвращает текущее состояние статуса прерывания потока без изменения значения статуса.

- `void join()` ("слияние") — переводит поток, из которого осуществляется вызов, в режим ожидания завершения (смерти) потока, для которого вызван метод `join()`. Например, если мы осуществим в потоке `thread1` вызов `thread2.join()`, выполнение потока `thread1` будет приостановлено до тех пор, пока не завершится поток `thread2`. Это ожидание (выполнение метода `join()`) может длиться сколь угодно долго, если соответствующий поток на момент вызова метода `join()` заблокирован (т. е. если в нем выполняется синхронизованный метод или он ожидает завершения синхронизованного метода).

  Перегруженный вариант `join(long millis)` — ожидать завершения потока в течение `millis` миллисекунд. Вызовы `join(0)` и `join()` эквивалентны. Еще один перегруженный вариант `join(long millis, int nanos)`, где параметр `nanos` задает число наносекунд. Ожидание смерти может быть прервано другим потоком с помощью метода `interrupt()` с возбуждением исключения `InterruptedException`. Метод `join()` — аналог функции `join` в UNIX — служит обычно для завершения главным потоком работы всех дочерних пользовательских потоков ("слияния" их с главным потоком).

- `boolean isAlive()` — возвращает `true` в случае, когда текущий поток жив (не умер). Отметим, что даже если поток завершился, от него остается объект-призрак, отвечающий на запрос `isAlive()` значением `false` (т. е. сообщающий, что объект умер).

10.6. Работа многопоточных приложений в многопроцессорных и многоядерных системах

Причины перехода к многопроцессорным и многоядерным системам

К 2007 году все ведущие разработчики процессоров перешли на многоядерную технологию. При этом в одном корпусе процессора располагают несколько процессорных ядер, способных независимо выполнять вычисления, но имеющих доступ к одной и той же общей памяти и одним и тем же внешним шинам и сигналам.

Переход к новой архитектуре процессоров связан с тем, что уже достигнут физический предел частоты работы процессоров. Повышать быстродействие процессоров за счет дальнейшего повышения тактовой частоты, как это делалось на протяжении всей предыдущей истории компьютерной техники, больше нельзя. Причем ограничивающим фактором является скорость света, которая нам обычно кажется такой большой! Покажем это путем простого расчета.

Расстояние между выводами процессоров составляет примерно 3 см. Сколько-нибудь заметно сократить это расстояние нельзя по двум причинам.

Во-первых, из-за невозможности дальнейшего уменьшения размеров транзисторов, из которых состоят микросхемы. Даже если удается ценой огромных технологических усилий уменьшить размер транзисторов, все равно разработчики процессоров предпочитают сохранять размер процессора, но увеличивать число элементов в нем.

Во-вторых, имеется проблема отвода тепла от процессора. Существующие материалы не позволяют уменьшать площадь теплоотвода, а заметно снизить тепловыделение невозможно, хотя к этому и стремятся.

Скорость света в вакууме составляет 300 000 км/с = $3 \cdot 10^{10}$ см/с, или, что то же самое, 30 см/нс (1 нс = 10^{-9} с). Это максимальная скорость распространения любых сигналов, существующая в природе. В металлах электромагнитные сигналы распространяются примерно в три раза медленнее, так что мы получаем 10 см/нс. Таким образом, от одного вывода процессора до противоположного сигнал дойдет за t = 0,3 нс. В результате максимальная тактовая частота, на которой процессор еще может работать как синхронное устройство, где на всех выводах микросхемы сигналы появляются почти одновременно, может быть оценена как $1/t$ = 3,3 ГГц.

Процессор с большей тактовой частотой будет иметь на разных выводах сигналы, не соответствующие друг другу (это так называемые гонки сигналов). Дальнейшее повышение тактовой частоты не приведет к увеличению быстродействия процессора — придется ждать, пока сигналы не дойдут до каждого вывода микросхемы.

Конечно, наша оценка весьма грубая, и внутри процессоров принимают специальные меры борьбы с гонками — стараются сделать путь сигналов ко всем ножкам одинаковым и вводят конвейеры и элементы асинхронной архитектуры, так что одни части процессора могут выполнять некоторые операции без привязки к сигналам в других частях процессора. Тем не менее, полученная цифра оказывается весьма близкой к реальности. Таким образом, увеличивать скорость работы процессоров за счет дальнейшего повышения тактовой частоты нельзя.

Какие имеются альтернативные варианты повышения вычислительных мощностей компьютеров?

Распределенные системы, состоящие из компьютеров, связанных через Интернет, кластеры параллельных вычислений, в которых совместно работают несколько компьютеров, и многопроцессорные системы, где на одной материнской плате расположено несколько процессоров, на первый взгляд, позволяют заметно увеличить вычислительные мощности путем простого добавления процессоров.

Кажется, что чем больше процессоров, тем значительнее вычислительные мощности. В 10 раз больше процессоров — в 10 раз быстрее считает, в 100 раз больше — в 100 раз быстрее считает и т. д. Но это очень наивная точка зрения — в подавляющем большинстве случаев она не соответствует действительности. Способность вычислительной системы увеличивать вычислительную мощность с ростом числа занятых процессоров называется *масштабируемостью* системы. Построение хорошо масштабируемых систем является сложной и не всегда решаемой задачей.

Вычислительные задачи, требующие большой процессорной мощности, можно разделить на две категории.

Первая категория — это задачи специального вида, решаемые с помощью не связанных или почти не связанных друг с другом вычислений. Как, например, шахматные задачи, где каждый вариант хода и все последующие ходы рассматриваются независимо от других вариантов ходов и всех последующих за ними ходов. Другой пример — серверные системы, обрабатывающие независимые запросы большого числа пользователей. Только для этой категории использование распределенных, кластерных и многопроцессорных систем может дать заметное повышение производительности. И чем меньше обмен данными между различными частями задачи, тем более эффективно распараллеливание ее выполнения на далеко отстоящих друг от друга процессорах. При этом физическое расстояние между процессорами играет большую роль — чем оно меньше и

чем больше пропускная способность канала обмена, тем слабее уменьшение быстродействия из-за обмена данными.

Вторая категория — это задачи общего вида, решаемые с помощью связанных друг с другом вычислений. При этом часть данных может обрабатываться параллельно, но пока не будут получены некие промежуточные результаты, дальнейшие вычисления идти не могут. При этом часто возникают ситуации, когда один или несколько вычислительных потоков вынуждены ждать другой поток до тех пор, пока он не поставит им необходимые данные. Для этой категории задач применение многопроцессорных компьютеров (а тем более кластеров параллельных вычислений или распределенных систем) часто не только не приводит к повышению быстродействия, но даже уменьшает его. Ведь при работе параллельных систем имеются вычислительные накладные расходы на обмен данными. И чем больше процессоров задействовано, тем они больше. Так что они могут стать чрезвычайно существенными для главного процессора, который должен выдать окончательный результат. А задержки ответов от разных процессоров приводят к резкому замедлению работы системы. В результате становится ясным, что многопроцессорные системы позволяют решить проблему повышения быстродействия при решении задач, сильно связанных по данным, только в том случае, когда расстояние между процессорами мало. И чем меньше это расстояние, тем больший эффект будет достигнут в параллельных вычислениях общего вида. В этом плане кластерные системы применимы для более широкой категории задач, чем распределенные, многопроцессорные системы — для более широкой категории задач, чем кластерные. И, наконец, многоядерные системы, где расстояние между процессорами минимально возможное (т. к. они расположены на одном кристалле), пригодны для решения самого широкого класса задач параллельных вычислений.

Когда мы говорим про параллельные вычисления, то имеются в виду параллельные алгоритмы обработки данных. Это совсем не обязательно численные расчеты.

Итак, многоядерная архитектура процессоров — единственно возможное решение, позволяющее повысить быстродействие современных процессоров для всех категорий задач, а не только для слабосвязанных по данным. В 2007 году корпорации Intel и AMD массово выпускали двухъядерные процессоры для персональных компьютеров. В 2008 году эти корпорации наладили выпуск 4-ядерных процессоров для персональных компьютеров, в 2009 году предполагается выпуск 6-ядерных. В ближайшие годы Intel планирует довести число ядер до 128.

Аналогичная ситуация наблюдается и для серверных систем. Например, корпорация Sun Microsystems выпускает 8-ядерные серверные процессоры Niagara.

Таким образом, многоядерная архитектура в ближайшие годы позволит заметно увеличить производительность компьютеров, но только в том случае, если программное обеспечение даст возможность эффективно распараллелить вычисления.

Пример многопоточной программы

Каждое из ядер работает независимо, но чем меньше им приходится обмениваться данными, тем больше эффект от наличия нескольких ядер. Если приложение написано так, что все действия выполняются строго последовательно, то эффект от других ядер или процессоров пропадает. Точнее, он все-таки есть, но минимален — современные опе-

рационные системы задействуют другие ядра или процессоры для выполнения других процессов.

Как мы уже знаем, наибольшее препятствие на пути повышения производительности параллельных вычислений — обмен данными. Задержки при обмене данных имеют две причины: первая — время доступа к чужим данным, вторая — ожидание процессами или потоками данных информации от другого процесса или потока.

Время доступа к данным складывается из времени их преобразования при передаче от процессора к процессору и времени физической доставки данных. Время физической доставки данных (распространения сигналов) минимально для многоядерных систем. Время преобразования данных определяется протоколом, позволяющим осуществлять обмен данными. Так, при обмене данными между процессами, выполняемыми на разных процессорах или ядрах, требуется осуществлять много промежуточных действий, чтобы данные попали от одного процесса к другому. А вот в многопоточных системах обмен данными не требует дополнительных затрат — данные располагаются в том же адресном пространстве. В результате можно эффективно использовать быструю буферную память (кэш-память) разного уровня, позволяющую уменьшать время доступа к данным.

В связи с этим программирование в многопоточной среде признано наиболее перспективной моделью параллелизации программ в многоядерных системах и становится одним из важнейших направлений развития программных технологий. Модель многопоточности Java позволяет весьма элегантно реализовать преимущества многоядерных процессорных систем. Во многих случаях программы Java, написанные с использованием многопоточности, эффективно распараллеливаются автоматически на уровне виртуальной машины — без изменения не только исходного, но даже скомпилированного байт-кода приложений.

Но программирование в многопоточной среде — это сложное и ответственное занятие, требующее очень высокой квалификации. Многие алгоритмы, кажущиеся простыми, естественными и надежными, в многопоточной среде оказываются неработоспособными, из-за чего способы решения даже самых простых задач становятся необычными и запутанными, не говоря уже о проблемах, возникающих при решении сложных задач. В связи с этим автор рекомендует на начальном этапе не увлекаться многопоточностью, а только ознакомиться с данной технологией.

Тем, кто все же захочет заняться таким программированием, следует сначала прочитать главу 9 в книге Джошуа Блоха [11]. Для решения некоторых проблем многопоточного программирования в JDK 5 были добавлены пакеты `java.util.concurrent` и `java.util.concurrent.atomic`. Последний из них содержит набор классов, обеспечивающих атомарность (неделимость, обеспечивающую синхронное изменение отдельных байтов ячейки или полей объекта) операций с различными типами данных.

Тем не менее, приведем пример многопоточного приложения `10_6_Threads` и рассмотрим его работу на многоядерном компьютере.

Сначала приведем исходный код для последовательного варианта вычислений, который нам уже хорошо знаком по *главе 5*.

Класс `MiddleRectangles` предназначен для вычисления с помощью метода `integral` интеграла от функции, задаваемой объектом `fnc`, передаваемым в метод в качестве парамет-

ра. Вызов `fnc.f` и представляет собой функцию, которую необходимо интегрировать. Использовать объект приходится из-за того, что в Java не предусмотрена передача функций в качестве параметров (листинг 10.8).

Листинг 10.8. Передача функции в качестве параметра с помощью объекта

```
/**
 *
 * @author monakhov
 */
public class MiddleRectangles {



    /**
     * Creates a new instance of MiddleRectangles
     */
    //    public MiddleRectangles() {
    //    }

    public static double integral(double a,double b, double n, Fnc fnc) {

        double r=0;
        double h=(b-a)/n;
        for(int i=0;i<n;i++)
            r=r+fnc.f(a+i*h+h/2);
        return r*h;
    }

}
```

В классе приложения к вычислению интеграла относится код листинга 10.9.

Листинг 10.9. Вычисление интеграла в однопоточном режиме

```
Fnc fnc;
double a=0,b=10;
int n=1000;
double result=0;

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    a=Double.parseDouble(jTextField2.getText());
    b=Double.parseDouble(jTextField3.getText());
    n=Integer.parseInt(jTextField4.getText());
    if( jRadioButton1.isSelected() )  fnc=new Fnc();
    if( jRadioButton2.isSelected() )  fnc=new FncX2();
    if( jRadioButton3.isSelected() )  fnc=new FncSin();
    if( jRadioButton4.isSelected() )  fnc=new FncSqrt();
    long t0=System.currentTimeMillis();
    result=MiddleRectangles.integral(a,b,n,fnc);
```

```
    long t=System.currentTimeMillis()-t0;
    jLabel4.setText("integral="+result);
    jLabel8.setText("t="+t+" ms");
}
```

Вычисления проводятся при нажатии на кнопку `jButton2`. В зависимости от того, какая из радиокнопок выбрана, создается объект, связанный с той или другой функцией.

Теперь перейдем от последовательного алгоритма к многопоточному. Идеология параллелизации нашего приложения будет следующая. Интегрирование представляет собой суммирование значений, соответствующих каждому из интервалов разбиения. Поэтому можно разделить интервал, где проводится интегрирование, на столько частей, сколько у нас потоков. И каждый поток будет заниматься интегрированием на своем участке. После вычисления очередным потоком частичной суммы его результат добавится в переменную, где и будет получен окончательный результат, а сам поток должен умереть. Когда все потоки возвратят свои результаты, то будет получен окончательный ответ, который следует вывести на экран.

Для начала зададим класс `Integrator`. Объект этого класса будет использоваться для того, чтобы в переменной `result` суммировать результаты интегрирования (листинг 10.10).

Листинг 10.10. Класс `Integrator`, предназначенный для суммирования результатов

```
package examples;
/**
 *
 * @author monakhov
 */

public class Integrator {

public double result=0;
public int activeThreadsCount=0;
    /**
     * Creates a new instance of Integrator
     */
    public Integrator() {
      super();
    }

}
```

Поток выполнения создадим на основе класса `ThreadedCalc`, наследника класса `Thread`. Конструктор этого класса создает поток выполнения и через список параметров передает в него информацию о границах интегрирования и числе интервалов для данного потока, а также ссылку на объект интегрируемой функции и ссылку на объект `integrator` (в поле `result` которого каждый поток добавляет свои результаты интегрирования) (листинг 10.11).

Листинг 10.11. Класс потока выполнения

```
/*
 * ThreadedCalc.java
 *
 */

package examples;

/**
 *
 * @author monakhov
 */

public class ThreadedCalc extends Thread{
    /** Creates a new instance of ThreadedCalc */
    double result=0;
    double a;
    double b; double n;
    Fnc fnc;
    Integrator integrator;

    public ThreadedCalc(double a,double b, double n, Fnc fnc,
                        Integrator integrator) {
      super();
      this.a=a;
      this.b=b;
      this.n=n;
      this.fnc=fnc;
      this.integrator=integrator;
      integrator.activeThreadsCount++;
    }

   public void run(){
     result=MiddleRectangles.integral(a,b,n,fnc);
     synchronized(integrator){
      integrator.notify();
      integrator.result+=result;
      integrator.activeThreadsCount--;
     }
   }
}
```

Последний оператор конструктора `ThreadedCalc` заслуживает особого внимания — в нем идет увеличение счетчика потоков `activeThreadsCount`, имеющегося в объекте `integrator`.

У нас этот оператор записан без синхронизации в виде:

```
integrator.activeThreadsCount++;
```

поскольку в нашей программе потоки создаются по очереди, а параллельную работу они начинают уже потом, когда все потоки созданы.

Но классы разрабатываются не для решения какой-то одной конкретной задачи, а для многократного использования при решении различных сходных задач. Поэтому лучше перестраховаться и предусмотреть синхронизацию при доступе к объекту, задействованному разными потоками:

```
synchronized(integrator){
      integrator.activeThreadsCount++;
}
```

Следует остановиться на еще одном моменте — атомарности и неатомарности операций. Атомарными называются такие операции, которые не могут быть прерваны или выполнены не до конца. В Java атомарными являются операции присваивания ссылок или значений переменным типа `boolean`, `int` или `float`, а также операции чтения значений соответствующих типов. Операция инкремента не атомарная, т. к. она представляет собой последовательность считывания значения и его изменения. В результате при работе в многопоточном режиме для нее нужна синхронизация.

Более того, необходимо задать в классе `Integrator` поле `activeThreadsCount` с модификатором `volatile`, чтобы компилятор не мог выполнять с ним оптимизирующую компиляцию в программах, где доступ к этому полю будет идти из нескольких потоков. В ряде случаев такая оптимизация (перестановка местами строк кода, вынос неизменяемой величины из цикла и т. д.) может приводить к совершенно неправильным результатам. Например, если имеется фрагмент кода

```
integrator.activeThreadsCount=1;
for(i=0;i<=n;i++){
    выполнять действия, не связанные с integrator.activeThreadsCount
};
count = integrator.activeThreadsCount;
```

оптимизирующий компилятор присвоит переменной `count` значение 1, хотя к этому моменту значение поля `activeThreadsCount` уже может измениться. Модификатор `volatile` покажет компилятору и виртуальной машине, что данная переменная может быть изменена другими потоками, и переменной `count` будет присвоено значение поля, считанное из памяти в правильный момент времени. Кроме того, для повышения скорости работы потоков Java-машина может неявным для программиста образом создавать копии переменных для разных потоков — каждому потоку свою копию. Модификатор `volatile` гарантирует, что для переменной не будет создаваться таких копий.

В JDK 5 в пакете java.util.concurrent.atomic появилось большое количество оболочечных классов, поддерживающих атомарные операции.

Вернемся теперь к анализу листинга 10.11. Метод `run` выполняет основную задачу потока — интегрирование. Результат вычисляется и сохраняется в поле `result` потока. После этого идет синхронизованное обращение к объекту `integrator` для добавления в него полученного потоком результата и уменьшения на единицу счетчика активных потоков. Но перед этим объект следует разбудить методом `notify()`, т. к. после запуска всех потоков в основной программе (см. чуть ниже) идет цикл ожидания `integrator.wait()` до тех пор, пока не выполнится условие `activeThreadsCount==0`.

Теперь рассмотрим обработчик нажатия на кнопку `jButton2` — `Создать потоки и вычислить интеграл` (листинг 10.12).

Листинг 10.12. Создание потоков и многопоточное вычисление интеграла

```
Integrator integrator;

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    double aForThread,bForThread,h,step;
    int nForThread,threadCount;
    threadCount=(Integer)(jSpinner1.getValue());
    n=Integer.parseInt(jTextField4.getText());
    nForThread=n/threadCount;
    n=nForThread*threadCount;
    jLabel7.setText(nForThread+" intervals for 1 thread");
    h=(b-a)/n;
    step=h*nForThread;
    jTextField4.setText(n+"");
    ThreadedCalc[] threads=new ThreadedCalc[threadCount];
    ThreadedCalc thread;
    integrator=new Integrator();
    long t0=System.currentTimeMillis();
    for(int i=0;i<threadCount;i++){
          aForThread=a+i*step;
          bForThread=aForThread+step;
          if( jRadioButton1.isSelected() )
               fnc=new Fnc();
          if( jRadioButton2.isSelected() )
               fnc=new FncX2();
          if( jRadioButton3.isSelected() )
               fnc=new FncSin();
          if( jRadioButton4.isSelected() )
               fnc=new FncSqrt();
          thread=new ThreadedCalc(aForThread,bForThread,nForThread,fnc,
                                                              integrator);
          threads[i]=thread;
    }

    for(int i=0;i<threadCount;i++){
          threads[i].start();
    }

    synchronized(integrator){
        while(!(integrator.activeThreadsCount==0) )
            try{ integrator.wait();
            }catch(Exception e){
                  JOptionPane.showMessageDialog(null,
                  "activeThreadsCount="+integrator.activeThreadsCount+
                   ""+e.getMessage(),"",JOptionPane.INFORMATION_MESSAGE);
            }

    }
```

```
    long t=System.currentTimeMillis()-t0;
    jLabel8.setText("t="+t+" ms");
    jLabel6.setText(integrator.result+"");
}
```

Переменная `n` задает число интервалов интегрирования, а `threadCount` — число потоков. Сначала мы считываем число потоков либо по показаниям спиннера `jSpinner1` — компонента, позволяющего изменять показываемое в поле редактирования число, либо путем ввода с клавиатуры, либо щелчками мыши по стрелкам компонента. Стрелка вверх задает увеличение числа на единицу, стрелка вниз — уменьшение на единицу.

Число интервалов на один поток рассчитывается как:

```
NForThread = n/threadCount.
```

Правда, если при целочисленном делении получится остаток, то потребуются дополнительные меры. Можно было бы для первого или для последнего потока задавать число интервалов, отличающееся от числа для других потоков. Но мы для простоты изменим общее число интервалов интегрирования, задавая его как `n=(int)(nForThread*threadCount).`

Для того чтобы хранить наши потоки, можно использовать массив `threads`, динамически задавая нужное число элементов для этого массива. При этом требуется сначала в цикле создать все потоки, и только потом в другом цикле запустить их, как показано в приведенном ранее примере. Но в примере 10_6_Treads, находящемся на прилагаемом DVD, мы сразу после создания потока будем запускать его на выполнение, и даже откажемся от создания массива `threads`, заменив все действия с потоками в данном обработчике единственной строкой:

```
new
  ThreadedCalc(aForThread,bForThread,nForThread,fnc,integrator)).start();
```

После цикла, в котором осуществляется запуск потоков, идет наиболее интересный фрагмент программы, повторим его и посмотрим, что там делается (листинг 10.13).

Листинг 10.13. Накопление результатов из разных потоков

```
synchronized(integrator){
    while(!(integrator.activeThreadsCount==0) )
        try{ integrator.wait();
        }catch(Exception e){
                    JOptionPane.showMessageDialog(null,
                     "activeThreadsCount="+integrator.activeThreadsCount+
                     "  "+e.getMessage(),"",
                    JOptionPane.INFORMATION_MESSAGE);
        }
}
```

Наши потоки работают, вычисляя интегралы, — каждый для своего участка. После вычисления очередным потоком своей части интеграла он добавляет этот результат к значению, хранящемуся в поле `result` объекта `integrator`, и заканчивает работу (умирает). Мы заранее не знаем, когда потоки закончат работу. Но в поле `integrator.activeThreadsCount` хранится число работающих потоков. Первоначально оно равно `threadCount`, но перед

смертью каждый поток уменьшает это число на единицу. Таким образом, после окончания работы всех потоков оно станет равно нулю.

Значит, нам надо проверять, когда выполнится условие `integrator.activeThreadsCount==0` (это признак того, что в поле `integrator.result` получен окончательный результат). Доступ к объекту `integrator` обязательно должен быть синхронизированным, чтобы можно было вызвать метод `integrator.wait()` и предотвратить попытки доступа к объекту в то время, когда с ним выполняет операцию записи какой-нибудь из потоков.

Таким образом, мы проверяем в цикле, не наступило ли условие окончания вычислений. Если нет, то приказываем объекту заснуть и ожидать, пока какой-нибудь поток не разбудит его методом `notify()` или `notifyAll()`.

Отметим, что метод `integrator.wait()` обязательно следует заключать в блок `try...catch`.

Работа многопоточного приложения на многоядерном компьютере

Работа приложения 10_6_Threads, описанного в предыдущем разделе, проверялась на двухъядерном портативном компьютере (ноутбуке) Dell Inspiron MXC061 с процессором Intel Core 2 Duo T5300, 1,73 ГГц, 1,5 Гб ОЗУ, под операционной системой Microsoft Windows XP Home Edition SP2.

Результаты испытаний для различных функций и различного числа потоков для данных условий приведены в табл. 10.1 и 10.2.

Таблица 10.1. *Время вычисления интеграла для 100 млн интервалов, с*

Число дочерних потоков	Функция			
	x	x*x	sqrt(x)	sin(x)
0	1,5	1,6	3,8	12,7
1	1,5	1,6	3,8	12,7
2	0,8	0,9	2,0	6,9
10	0,9	0,8	1,9	6,5
100	0,9	0,8	2,0	6,5
1 000	1,0	1,0	2,2	7,3
10 000	3,2	3,2	4,3	9,6
100 000	26	27	28	32

Таблица 10.2. *Эффективность по отношению к однопоточному режиму, раз*

Число дочерних потоков	Функция			
	x	x*x	sqrt(x)	sin(x)
0	1,0	1,0	1,0	1,0
1	1,0	1,0	1,0	1,0

Таблица 10.2 *(окончание)*

Число дочерних потоков	Функция			
	x	x*x	sqrt(x)	sin(x)
2	1,9	1,8	1,9	1,8
10	1,7	2,0	2,0	2,0
100	1,7	2,0	1,9	2,0
1 000	1,5	1,6	1,7	1,7
10 000	0,5	0,5	0,9	1,3
100 000	0,06	0,06	0,14	0,4

Аналогичные результаты были получены при использовании четырехъядерного процессора. Скорость увеличивалась в два раза для двух потоков; в три раза для трех потоков; в четыре раза для числа потоков, кратных четырем или лежащих в интервале от 16 до 100. Далее начинается постепенное уменьшение эффективности, вплоть до 2,7 при 1000 потоках и 0,5 при 10 000 потоках.

На основании полученных результатов можно констатировать:

- скорость выполнения приложения с одним дочерним потоком практически не отличается от скорости выполнения без создания дочерних потоков;
- при создании от 2 до 100 потоков скорость работы приложения на двухъядерном процессоре почти удваивается — лежит в пределах от 1,7 до 2,0 по сравнению с однопоточным режимом;
- при создании от 16 до 100 потоков скорость работы приложения на четырехъядерном процессоре почти учетверяется — лежит в пределах от 3,6 до 4,0 по сравнению с однопоточным режимом;
- для двухъядерного процессора пик скорости для большинства функций наблюдается не при двух потоках, как можно было бы предположить, а при 10—100 потоках;
- для четырехъядерного процессора пик скорости для большинства функций наблюдается при четырех, восьми и двенадцати потоках;
- заметная потеря производительности наблюдается при числе потоков, превышающем 1000.

На основании этого можно сделать следующие выводы:

- создание потоков выполнения в версии Java, соответствующей JDK 6, происходит очень эффективно по времени;
- при написании многопоточных приложений можно создавать большое число потоков (десятки и сотни) практически без потери производительности;
- при разработке программ для многоядерных процессоров можно создавать приложения с числом потоков, в десятки раз превышающим число ядер — это не только не приведет к заметной потере производительности, но даже может увеличить ее.

10.7. Синхронизация на основе интерфейсов *Lock* и *Condition*

Интерфейс *Lock*

В JDK 5 появился новый механизм синхронизации, основанный на использовании интерфейса `java.util.concurrent.locks.Lock`. Слово `lock` означает "блокиратор", "замок". Поток может поставить блокировку, осуществляемую с помощью объекта-блокиратора, являющегося экземпляром класса, реализующего интерфейс `Lock`. Как правило, только один поток имеет доступ к блокированным ресурсам (до тех пор, пока блокировка не снята). Данный механизм гораздо более гибкий по сравнению с механизмом синхронизации. Например, он позволяет ставить и снимать блокировки на разделяемые ресурсы в любой момент времени при выполнении необходимых условий, а также делать попытки установить блокировки, что снимает огромное число проблем взаимных блокировок, непреодолимых в рамках идеологии синхронизации. Но данный механизм намного сложнее синхронизации, т. к. для его применения требуется реализовывать методы, декларируемые интерфейсом `Lock`:

`lock`, `lockInterruptibly`, `tryLock`, `newCondition`, `unlock`.

Наиболее употребительные методы: `tryLock` и `unlock`. Листинг 10.14 иллюстрирует механизм блокировки с помощью объекта-блокиратора `lock`.

Листинг 10.14. Блокировка с помощью объекта-блокиратора `lock`

```
Lock lock = new MyLock();
if (lock.tryLock()) {
    try {
        ... — выполнение необходимых действий
    } finally {
        lock.unlock();
    }
} else {
    ... — выполнение необходимых действий
}
```

Назначение методов интерфейса `Lock` приведено в табл. 10.3.

***Таблица 10.3.** Методы интерфейса `Lock`*

Метод	Перевод названия	Что делает
`void lock()`	Блокировать	Ставит блокировку. Если поток, на который ставится блокировка, недоступен (например, блокирован другим потоком), то поток, пытающийся поставить блокировку, спит до тех пор, пока блокировка не будет поставлена. Блокировка снимается методом `unlock()`

Таблица 10.3 *(окончание)*

Метод	Перевод названия	Что делает
`void lockInterruptibly() throws InterruptedException`	Блокировать с возможностью прерывания	Ставит блокировку. Если поток, на который ставится блокировка, недоступен (например, блокирован другим потоком), то поток, пытающийся поставить блокировку, спит до тех пор, пока блокировка не будет поставлена, либо пока он не будет прерван методом `interrupt()` (см. класс `Thread`). При таком прерывании возбуждается исключение `InterruptedException`
`Condition newCondition() throws UnsupportedOperationException`	Новое условие	Возвращает новый экземпляр класса `Condition`. При переходе от идеологии мониторов к идеологии блокировок методы интерфейса `Condition` заменяют методы `wait`, `notify` и `notifyAll` класса `Object`
`boolean tryLock()`	Попытаться блокировать	Пытается осуществить блокировку. Возвращает `true` в случае удачи, `false` — если осуществить блокировку не удалось (например, из-за того что другим потоком уже поставлена блокировка)
`boolean tryLock(long time,TimeUnit unit) throws InterruptedException`	Попытаться блокировать	Пытается осуществить блокировку в течение заданного времени. Возвращает `true` в случае удачи, `false` — если осуществить блокировку не удалось (например, из-за того что другим потоком уже поставлена блокировка). При прерывании текущего потока во время его попытки блокировать ресурс возбуждается исключение `InterruptedException`
`void unlock()`	Разблокировать	Снимает блокировку

Метод `tryLock(long time, TimeUnit unit)` требует задания времени тайм-аута в качестве первого параметра и единицы измерения этого времени — в качестве второго. В классе `java.util.concurrent.TimeUnit` задано несколько констант для возможных единиц измерения времени (листинг 10.15).

Листинг 10.15. Константы, задающие единицы измерения времени

```
NANOSECONDS
MICROSECONDS
MILLISECONDS
SECONDS
MINUTES
HOURS
DAYS
```

В JDK 5 наряду с интерфейсом `Lock` появился класс `ReentrantLock`, реализующий данный интерфейс и заданный в пакете `java.util.concurrent.locks`. Он реализует все базовые возможности блокираторов, а также имеет некоторые дополнительные возможности. В большинстве случаев данный класс может быть использован для создания блокираторов.

Для помощи в работе с технологией блокировки предназначен класс `java.util.concurrent.locks.LockSupport`, предоставляющий ряд полезных методов, в частности метод `park` (парковать), обеспечивающий парковку текущего потока (особый вариант блокировки), и метод `unpark` (распарковать), снимающий эту блокировку.

Приведем пример из документации NetBeans для класса `FIFOMutex` (блокировщик очереди `FIFO` "First In — First Out" (первый вошел — первый вышел)), реализующего схему построения класса для поддержки интерфейса `Lock`. Правда, реализованы только методы `lock()` и `unlock()`, и это именно черновая схема работы. В этот пример автор внес небольшие исправления — добавил импорт пакетов и в последнем операторе добавил приведение типа к `Thread` (листинг 10.16).

Листинг 10.16. Исправленный пример класса `FIFOMutex`

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;

class FIFOMutex {
   private final AtomicBoolean locked = new AtomicBoolean(false);
   private final Queue waiters = new ConcurrentLinkedQueue();

   public void lock(){
     Boolean wasInterrupted = false;
     Thread current = Thread.currentThread();
     waiters.add(current);//добавить в конец очереди потоков

     //Парковать ("усыпить") текущий поток и ждать, пока он не станет
     //первым в очереди, или если он первый, но еще не заблокирован
     while (waiters.peek() != current ||
            !locked.compareAndSet(false, true)){
        LockSupport.park(this);
        if (Thread.interrupted()) //игнорировать прерывания текущего
          wasInterrupted = true;  //потока во время ожидания,
     }                            //но установить флаг

     waiters.remove();//удалить из очереди потоков
     if (wasInterrupted)
       current.interrupt();
   }

   public void unlock() {
     locked.set(false);
```

```
      LockSupport.unpark((Thread)waiters.peek());//распарковать первый
    }                                            //из очереди потоков

}
```

Экземпляры данного класса — разделяемые объекты — доступ к каждому из которых возможен из разных потоков. Эти объекты являются блокираторами — при вызове метода `lock` они блокируют тот поток, из которого осуществляется этот вызов. Вызов метода `unlock` для данного блокиратора, осуществленный из какого-либо другого потока, разблокирует данный поток (точнее, поток, стоящий в "голове" очереди).

Исходный код данного класса присутствует в проекте 10_7_Lock.

Рассмотрим этот пример подробнее.

Сначала идет импорт пакетов, необходимых для работы класса. Поле `locked` используется для проверки, блокирован ли поток. Оно имеет тип `AtomicBoolean`, появившийся в JDK 5 для поддержки атомарных (неделимых) операций, как и другие типы из пакета `java.util.concurrent.atomic`.

Переменная `waiters` (ожидающие) типа `Queue` (очередь) служит для создания очереди блокируемых потоков. Экземпляр реально создаваемой очереди имеет тип `ConcurrentLinkedQueue` — это класс, появившийся в JDK 5 для поддержки коллекций в условиях конкурирующей записи в коллекцию элементов из разных потоков выполнения.

Метод `lock`, вызываемый разделяемым объектом из какого-либо потока, занимается блокировкой текущего потока (т. е. того потока, из которого этот метод был вызван). С помощью переменной `waiters` текущий поток добавляется в список блокируемых потоков.

Вызов `waiters.peek()` обеспечивает возврат ссылки на первый элемент коллекции (`head` — "голову" очереди) (т. е. на тот элемент, который первым был добавлен в очередь).

Далее идет цикл, в котором сначала вызывается метод `LockSupport.park(this)`. Этот метод усыпляет текущий поток до тех пор, пока какой-либо другой поток не разбудит его методом `unpark` либо пока другой поток не прервет его методом `interrupt`.

Затем в описываемом цикле идет проверка — не был ли поток прерван (т. е. установлено ли состояние `Thread.interrupted()==true`). В этом случае не производится выхода из цикла, а просто устанавливается флаг `wasInterrupted`, в котором запоминается, что поток был прерван.

Условие завершения цикла следует рассмотреть отдельно.

Первая возможность, при которой цикл завершается — если текущий поток становится первым в очереди.

Вторая — если функция `locked.compareAndSet(false, true)` возвращает `true`.

Функция `atomicBool.compareAndSet(expectedValue,newValue)` возвращает `true` в случае, когда значение переменной `atomicBool` совпадает со значением `expectedValue`. При этом переменная `atomicBool` принимает новое значение `newValue`.

Таким образом, второй вариант выхода из цикла происходит в случае, когда переменная `locked` имеет значение `false`, т. е. когда поток разблокирован.

Оператор `waiters.remove()` удаляет головной элемент (текущий поток) из очереди потоков, блокируемых с помощью данного объекта-блокиратора, после чего в случае, если поток был выведен из состояния парковки (сна) прерыванием, нужно вызвать метод `interrupt()`, прерывающий поток выполнения. Во время сна прерывания потока не происходит, а устанавливается состояние `interrupted`.

Метод `unlock` совсем прост — переменная `locked` устанавливается в `false` и первый из очереди блокируемых потоков разблокируется.

Интерфейс *Condition*

Интерфейс `Condition` (условие), появившийся в JDK 5, как уже говорилось, предназначен для более современного варианта работы по событиям, чем методы `wait`, `notify` и `notifyAll`, заданные в классе `Object`. Данный интерфейс предназначен для совместной работы с интерфейсом `Lock`.

Интерфейс имеет методы, задающие условия сна и пробуждения потоков, перечисленные в табл. 10.4.

Таблица 10.4. *Методы интерфейса* `Condition`

Метод	Перевод названия	Что делает
`void await()` `throws InterruptedException`	Ожидать	Переводит текущий поток в режим ожидания (сна), из которого его можно вывести методами `signal()`, `signalAll()` или прерыванием потока — `interrupt()`. Но также возможно фальшивое пробуждение (хотя и не с очень большой вероятностью), когда потоку никто не посылал сигналов. Предполагается, что перед вызовом данного метода поток должен быть блокирован с помощью какого-либо блокиратора
`boolean await(long time, TimeUnit unit)`	Ожидать	То же, но с окончанием сна через число единиц времени, задаваемых первым параметром. Что это за единицы, задает второй параметр метода. Возвращает `true` в случае, если пробуждение наступило по достижении заданного времени, а если раньше, то возвращается `false`
`long awaitNanos(long nanosTimeout)`	Ожидать, наносекунд	То же, но с окончанием сна через число наносекунд, задаваемых параметром. Возвращается число наносекунд, оставшихся до истечения заданного тайм-аута — на случай фальшивого пробуждения для организации дальнейшего ожидания. В случае нормального завершения метода возвращаемое число равно нулю или отрицательно
`void awaitUninterruptibly()`	Ожидать с запретом прерывания	То же, но без разрешения вывода из сна прерыванием потока

Таблица 10.4 *(окончание)*

Метод	Перевод названия	Что делает
`boolean awaitUntil(Date deadline)`	Ожидать до тех пор, пока	То же, но с окончанием сна по достижении даты, задаваемой параметром. Возвращает `true` в случае, если пробуждение наступило по достижении заданной даты, а если раньше, то возвращается `false`
`void signal()`	Сигнализировать	Будит один спящий поток. При этом поток гарантированно является блокированным каким-либо блокиратором
`void signalAll()`	Сигнализировать всем	Будит все спящие потоки. При этом потоки гарантированно являются блокированными какими-либо блокираторами

В документации по интерфейсу `Condition` приводится типичный пример использования метода `awaitUntil` (листинг 10.17).

Листинг 10.17. Типичный пример использования метода `awaitUntil`

```
synchronized boolean aMethod(Date deadline) {
   boolean stillWaiting = true;
   while (!условие_которого_ждем) {
     if (stillwaiting)
         stillWaiting = theCondition.awaitUntil(deadline);
      else
        return false;
   }
   // ...
}
```

В этом примере `theCondition` — экземпляр класса, реализующего интерфейс `Condition`. Возвращаемое методом `aMethod` значение позволяет узнать, была ли достигнута заданная дата `deadline` или пробуждение произошло досрочно.

Аналогичный пример имеется и для метода `awaitNanos` (листинг 10.18).

Листинг 10.18. Типичный пример использования метода `awaitNanos`

```
synchronized boolean aMethod(long timeout, TimeUnit unit) {
   long nanosTimeout = unit.toNanos(timeout);
   while (!условие_которого_ждем) {
     if (nanosTimeout > 0)
         nanosTimeout = theCondition.awaitNanos(nanosTimeout);
      else
        return false;
   }
   // ...
}
```

Краткие итоги по главе

- Программу, выполняющуюся под управлением операционной системы, называют *процессом* (process) или приложением. У каждого процесса свое адресное пространство. *Потоки выполнения* (threads) отличаются от процессов тем, что выполняются в адресном пространстве своего родительского процесса. Потоки выполняются параллельно (псевдопараллельно), но в отличие от процессов легко могут обмениваться данными в пределах общего виртуального адресного пространства, т. е. у них могут иметься общие переменные, в том числе массивы и объекты.
- В приложении всегда есть главный (основной) поток. Если он закрывается, то закрываются все остальные пользовательские потоки приложения. Кроме них возможно создание потоков-демонов, которые могут продолжать работу и после окончания работы главного потока.
- Любая программа Java неявно использует потоки. В главном потоке виртуальная Java-машина (JVM) запускает метод main приложения, а также все методы, вызываемые из него. Главному потоку автоматически дается имя "main".
- Если разные потоки получают доступ к одним и тем же данным, причем один из них или они оба меняют эти данные, то для них требуется обеспечить установку разграниченного доступа. Пока один поток меняет данные, второй не должен иметь права их читать или менять — он должен дожидаться окончания доступа к данным первого потока. Говорят, что осуществляется *синхронизация* потоков. В Java для этих целей служит оператор synchronize (синхронизировать). Иногда синхронизованную область кода (метод или оператор) называют *критической секцией* кода.
- При запуске синхронизованного метода говорят, что объект *входит в монитор*, при завершении — что объект *выходит из монитора*. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.
- Существуют два способа синхронизации по ресурсам: синхронизация объекта и синхронизация метода.
- Синхронизация объекта obj1 при вызове несинхронизованного метода:

```
synchronized(obj1) оператор;
```

- Синхронизация метода с помощью модификатора synchronized при задании класса:

```
public synchronized тип метод(...){...}
```

- Кроме синхронизации по данным предусмотрена *синхронизация по событиям*, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основные операции при таком типе синхронизации: wait (ждать) и notify (оповестить).
- Атомарными называются такие операции, которые не могут быть прерваны или выполнены не до конца. В Java атомарными являются операции присваивания ссылок или значений переменным типа boolean, int или float, а также операции чтения значений соответствующих типов.
- Возможны три способа создания класса, экземплярами которого будут потоки: унаследовать класс от java.lang.Thread, реализовать интерфейс java.lang.Runnable, реализовать интерфейс java.lang.Callable.

- Интерфейс `java.lang.Runnable` имеет декларацию единственного метода `public void run()`, который обеспечивает последовательность действий при работе потока. Класс `Thread` уже реализует интерфейс `Runnable`, но с пустой реализацией метода `run()`. Поэтому в потомке `Thread` нужно переопределить метод `run()`.
- Интерфейс `java.lang.Callable` имеет декларацию единственного метода `call()`, который обеспечивает последовательность действий при работе потока. У этого метода тип возвращаемого значения является параметризуемым.
- Создание потоков выполнения в JDK 6 происходит очень эффективно по времени.
- При написании многопоточных приложений можно создавать большое число потоков (десятки и сотни) практически без потери производительности.
- Для многоядерных процессоров можно разрабатывать приложения с числом потоков, в десятки раз превышающим число ядер — это не только не приведет к заметной потере производительности, но даже может увеличить ее.
- В JDK 5 появился новый механизм синхронизации по данным и по событиям, основанный на интерфейсах `Lock` и `Condition`.
- В JDK 5 появился класс `java.util.concurrent.locks.ReentrantLock`, реализующий интерфейс `Lock`. Он обеспечивает поддержку всех базовых возможностей блокираторов, а также имеет некоторые дополнительные возможности.

Типичные ошибки

- При написании многопоточной программы забывают предусмотреть синхронизацию при доступе к совместно используемым разными потоками данным, из-за чего во время считывания данных одним потоком другой их меняет. Самый наглядный вариант — считывание элементов массива, когда второй поток меняет элементы этого массива. Второй пример — считывание одним потоком значения поля объекта в то время, когда другой поток меняет состояние этого объекта (в том числе, возможно, объект сам меняет по очереди значения нескольких своих полей из-за сделанного вторым потоком изменения).
- Реализуют неправильный алгоритм синхронизации, в результате которого потоки блокируют друг друга, бесконечно ожидая завершения синхронизованных методов друг от друга (вариант "смертельной блокировки", тупика — deadlock).
- Используют синхронизацию там, где этого делать не нужно, в результате некоторые потоки бесполезно простаивают, ожидая завершения синхронизованного метода.
- Не учитывают, что математические операции не являются атомарными, и поэтому требуют синхронизации.
- Не учитывают, что только либо чтение, либо запись переменной, имеющей тип `boolean`, `int` или `float`, является атомарной операцией. А операции, совмещающие чтение и запись (`++`, `--`, `+=`, `-=`, `*=`, `/=`) с величинами этих типов, не атомарные.
- Не учитывают, что операции со значениями типа `long` или `double` *не являются атомарными* и обязательно требуют синхронизации.
- Не ставят модификатор `volatile`, т. к. не учитывают, что оптимизирующий компилятор может проводить модификацию кода, совершенно оправданную в однопоточных программах, но неприемлемую для многопоточных.

- Опускают модификатор `volatile`, поскольку не знают, что Java-машина может создавать копии переменных для разных потоков.

Задания

- Написать приложение, в котором используются потоки. Класс потока задать как наследник `Thread`.

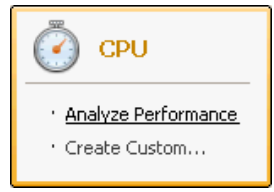
  Класс потока должен обеспечивать в методе `run` построчный несинхронизированный вывод в консольное окно чисел от 1 до 100 порциями по 10 чисел в строке, разделенных пробелами, причем перед каждой такой порцией должна стоять надпись "`Thread 1:`" для первого потока, "`Thread 2:`" — для второго и т. д.

  Для вывода строки задать в классе метод `print10`.

  В приложении при нажатии на первую кнопку должны создаваться два или более потоков, а при нажатии на вторую они должны стартовать.

- Усовершенствовать приложение, обеспечив синхронизацию за счет объявления вызова `print10` в методе `run` синхронизированным.

- Создать копию класса потока, отличающуюся от первоначальной тем, что выводятся числа от 101 до 200, класс должен быть определен как реализующий интерфейс `Runnable`, а метод `print10` задан как синхронизированный. Добавить в приложение создание и старт потоков — экземпляров данного класса.



ГЛАВА 11

Введение в сетевое программирование

В этой главе рассматриваются вопросы использования Java в программировании клиентских и серверных приложений при работе в WWW. Сначала дается краткая информация по языку HTML, без знания которого разговор о программировании в WWW невозможен. Далее приводятся примеры написания апплетов — самого первого варианта применения Java в WWW. И, наконец, рассматриваются основы серверных Java-технологий — более сложных, но и заметно более важных в практическом плане.

11.1. Краткая справка по языку HTML

Хотя сам HTML не является языком программирования, но он составляет неотъемлемую часть языков программирования WWW, поскольку именно в документы, написанные с помощью конструкций этого языка, вставляются апплеты Java, Flash-анимации и мультимедийные клипы. А серверные приложения Java генерируют HTML-документы, отсылаемые на компьютер клиента для их показа пользователю компьютера. Более гибкий, но и более сложный язык, основанный на тех же принципах, что и HTML, — XML. Но в данной книге мы не будем излагать материал, относящийся к XML, т. к. на начальном этапе WWW-программирования с использованием Java вполне достаточно базовых знаний по языку HTML.

Система WWW и язык HTML

World Wide Web (Всемирная паутина) или, как ее обычно называют, WWW — это распределенная компьютерная система, основанная на гипертекстовых документах. Информация в ней хранится на компьютерах с соответствующим программным обеспечением (серверах), объединенных в глобальную сеть. Она включает в себя не только текст, но и возможность выполнения определенных действий при выборе специально отмеченных участков текста (так называемый гипертекст), а также графику, видео, звук (средства мультимедиа). Эта информация содержится в виде HTML-документов, которые могут включать ссылки на другие документы, хранящиеся как на том же самом, так и на другом сервере. На экране компьютера гиперссылки выглядят как выделенные другим цветом и/или подчеркиванием участки текста или рисунки (графические изображения).

С помощью гиперссылок, называемых также гипертекстовыми связями, пользователь может автоматически связаться с соответствующим источником информации в сети и получить на экране своего компьютера документ, на который была сделана ссылка. В большинстве случаев гиперсвязи выбирают щелчком кнопки мыши на участке текста с гиперсвязью. При этом компьютер посылает через сеть по протоколу HTTP (HyperText Transport Protocol) запрос серверу, хранящему файл с необходимым документом. Сервер, получив запрос, посылает клиенту этот файл или сообщение об отказе (если требуемый документ по тем или иным причинам недоступен).

HTML-документы просматривают с помощью браузеров — программ просмотра WWW-документов (WWW-browsers). В настоящее время получили распространение десятки таких программ, но наиболее известные и развитые — Microsoft Internet Explorer, Mozilla (в том числе один из его клонов, Firefox), Opera, а также уже сошедший со сцены Netscape Navigator.

WWW-документ, как уже отмечалось, содержит форматированный текст, графику и гиперсвязи с различными ресурсами Интернета. Чтобы реализовать все эти возможности, был разработан специальный компьютерный язык — HyperText Markup Language (HTML) — *язык разметки гипертекста*. Гипертекст (сверхтекст) — это текст, содержащий дополнительные возможности, в частности гиперссылки.

Существует несколько версий языка HTML, самая современная на данный момент — HTML 4.01 — принята в виде рекомендации Консорциума W3C (World Wide Web Consortium), отвечающего за развитие языка HTML и других WWW-технологий. XML-версия языка HTML, называемая XHTML, пока не нашла широкого распространения. Наиболее употребляемая версия при создании простых WWW-документов — это HTML 3.2. Создано много сред, позволяющих интерактивно создавать HTML-документы. Тем не менее, даже в этом случае полезно знать основные принципы устройства HTML-документов и имеющиеся в этом языке средства разметки.

Документ, написанный на языке HTML, представляет собой текстовый файл, содержащий собственно текст, несущий информацию пользователю, и теги разметки (murkup tags). Теги представляют собой определенные последовательности символов, заключенные между угловыми скобками (< >).

Программа просмотра располагает текст на экране дисплея согласно задаваемой тегами разметке, а также включает в него рисунки, хранящиеся в отдельных графических файлах, и формирует гиперсвязи с другими документами или ресурсами Интернета. После тега через пробел, вплоть до закрывающего символа >, может следовать один или несколько параметров *(атрибутов)* с задаваемыми для этих параметров значениями.

Файл на языке HTML имеет расширения html или htm. Он приобретает облик WWW-документа только тогда, когда просматривается в специальной программе просмотра — браузере.

Текст в HTML может включать любую последовательность символов, за исключением следующих:

```
<
>
&
"
```

Вместо них должны присутствовать комбинации:

```
&lt;
&gt;
&amp;
&quot;
```

Точка с запятой присутствует обязательно — ею заканчивается кодирующая последовательность.

Символы табуляции и перехода на новую строку эквивалентны пробелу, а несколько этих символов и пробелов подряд (в любой комбинации) — одному пробелу.

Для вставки в текст значимого пробела используется комбинация:

```
 
```

Она обозначает так называемый неразрывный пробел (non-breaking space), который не считается разделителем между словами. Поэтому слова, разделенные этим пробелом, хоть и пишутся через пробел, но на новую строку переносятся вместе, а не по раздельности.

Теги и их атрибуты

Теги предназначены для форматирования и разметки документа. Теги бывают парные (контейнеры) и непарные. Действие парного тега начинается с открывающего тега `<имяТега>` и заканчивается при встрече соответствующего ему закрывающего `</имяТега>`, признаком которого является символ `/`.

Например:

```
<html>Это html документ </html>
```

Непарный тег вызывает единичное действие в том месте, где он встречается. Например, тег `<br>` вызывает перевод текста на новую строку.

Исключением из правила, гласящего об эквивалентности любого числа пробелов, табуляций и переходов на новую строку одному пробелу, является текст внутри контейнера `<pre> </pre>`. Этот текст показывается так же, как он был предварительно отформатирован в обычном текстовом редакторе с использованием моноширинного шрифта, а все пробелы и переносы на новую строку значимые. Однако внутри данного контейнера могут действовать другие теги разметки.

Внутри тега, кроме его имени, могут находиться *атрибуты*, задающие дополнительные параметры, необходимые для действия тега.

Например, тег

```
<img src="MyFile.gif" width=100 height=40>
```

обеспечивает показ в данном месте текста изображения из файла с именем MyFile.gif, а ширина и высота изображения задается 100 на 40 точек (пикселов) соответственно. При этом атрибут `src`, задающий имя файла, обязательный, а `width` и `height` — необязательные (могут отсутствовать).

Регистр, в котором набраны имена тегов или атрибутов (но не значения атрибутов!), в рамках HTML не имеет значения. В ранних версиях HTML во многих программах,

работающих с HTML-документами (редакторах, конвертерах из одного формата текста в другой), оказывалось предпочтение верхнему регистру (заглавные буквы). Но после появления стандарта XHTML для обеспечения максимальной совместимости лучше использовать нижний регистр (маленькие, строчные буквы).

Типичный HTML-документ имеет заголовок и тело.

Начало документа отмечается тегом `<html>` и заканчивается тегом `</html>`.

Заголовок документа:

```
<head>Текст, включающий служебную информацию о документе </head>
```

Он никак не отображается на экране компьютера при просмотре HTML-файла, за исключением названия документа, помещаемого между тегами заголовка и заключенного в контейнере:

```
<title>Текст заголовка</title>
```

Это название обычно выводится как заголовок окна браузера, в котором отображается файл.

Важнейшая служебная информация — кодировка документа — задается следующим образом:

```
<meta http-equiv="content-type" content="text/html; charset=Windows-1251">
```

(русскоязычная кодировка ISO 1251);

```
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
```

(кодировка UTF-8) и т. д.

Тело определяет видимую часть документа:

```
<body>
Это html-документ, содержащий какой-то текст.
</body>
```

На дисплее будет выведено:

```
Это html-документ, содержащий какой-то текст.
```

Часть текста или участок изображения в HTML-документах может ссылаться на другой текст внутри того же самого документа или на другой документ в Интернете. Такая связь называется гипертекстовой связью (hypertext link), гиперсвязью, гипертекстовой ссылкой или гиперссылкой. Она выделяется подчеркиванием и цветом.

При ссылке на документ, находящийся на другом сервере, необходимо указать адрес (URL — Uniform Recourses Location) этого документа: сетевой адрес сервера и путь к этому документу на сервере. Если документ находится на том же сервере, но в другой папке, достаточно указать только путь к этой папке.

Подавляющее большинство документов, на которые делается ссылка, находится на том же сервере и в той же папке. При этом гипертекстовая ссылка имеет вид:

```
<a href="имя_файла">текст_ссылки</a>
```

Здесь:

- `имя_файла` — это имя файла (с указанием расширения), содержащего документ, на который делается ссылка;
- `текст_ссылки` — это якорь, т. е. участок текста, который будет выделен, как связанный с гиперсвязью.

В общем случае перед именем файла ставится URL-адрес сервера и полный путь к файлу на сервере.

Якорь задается в виде:

```
<a href="имя_файла#имя_метки">текст</a>
```

Язык HTML позволяет ссылаться не только на документы целиком, но и на отдельные части конкретного документа. В этом случае та часть документа, на которую делается ссылка, помечается меткой:

```
<a name="имя_метки">участок документа</a>
```

Здесь:

- `имя_метки` — произвольное имя метки, на которую должен быть переход (уникальное для данного документа);
- `имя_файла` — имя файла (вместе с расширением и путем), содержащего документ, на какое-либо место которого осуществляется ссылка.

Отметим, что переход производится в начало параграфа, в котором расположена мишень. Поэтому делению текста на параграфы при наличии гиперсвязей внутри документа нужно уделять особое внимание.

Замечание

Браузеры имеют специальное средство, уменьшающее загрузку компьютерных сетей, — буферизацию принятых сообщений. Этот буфер хранится некоторое время (задаваемое в параметрах браузера — обычно дни или недели) на жестком диске. Те адреса, по которым вы еще "не ходили", показываются одним цветом (по умолчанию синим), а по которым "ходили" (и документы еще хранятся в буфере) — другим (по умолчанию малиновым).

Рассмотрим теперь важнейшие теги разметки документа, задающие его внешний вид при просмотре в браузере.

Параграф осуществляет показ одной пустой строки и перевод каретки перед началом заключенного в контейнере текста. Внутри параграфа возможно выравнивание:

```
<p>без выравнивания текста</p>
<p align="left">выравнивание текста по левому краю</p>
<p align="right">выравнивание текста по правому краю</p>
<p align="center">выравнивание текста по центру</p>
```

Закрывающий тег `</p>` необязателен.

Горизонтальная раздельная черта `<hr>` осуществляет показ с новой строки горизонтальной разделительной черты. После нее с новой строки начинается новый параграф текста.

Заголовки используются для выводов заголовков и подзаголовков. Значение уровня заголовка может быть от 1 до 6 (листинг 11.1).

Листинг 11.1. Заголовки разных уровней

```
<h1>Заголовок первого уровня</h1>
<h2>Заголовок второго уровня</h2>
<h3>Заголовок третьего уровня</h3>
<h4>Заголовок четвертого уровня</h4>
<h5>Заголовок пятого уровня</h5>
<h6>Заголовок шестого уровня</h6>
```

Листинг 11.2 иллюстрирует задание *нумерованных списков*.

Листинг 11.2. Нумерованный список

```
<ol>
  <li>...</li>
  <li>...</li>
  <li>...</li>
  ...
</ol>
```

Закрывающий тег `</li>` необязателен, но желателен.

В листинге 11.3 приведен вид нумерованного списка в HTML-файле (исходный код документа) и на дисплее (показ документа в браузере).

Листинг 11.3. Задание нумерованного списка

В файле

```
Курсовые в срок сдали следующие студенты:
      <ol>
        <li>Иванов</li>
        <li>Петров</li>
        <li>Сидоров</li>
       </ol>
```

На дисплее

```
Курсовые в срок сдали следующие студенты:
1.Иванов
2.Петров
3.Сидоров
```

Листинг 11.4 иллюстрирует вид *ненумерованного списка* в HTML-файле и на дисплее.

Листинг 11.4. Задание ненумерованного списка

В файле

```
Курсовые в срок сдали следующие студенты:
<ul>
  <li>Иванов</li>
  <li>Петров</li>
  <li>Сидоров</li>
</ul>
```

На дисплее

Курсовые в срок сдали следующие студенты:

- Иванов
- Петров
- Сидоров

В HTML существуют следующие основные стили форматирования текста:

- полужирный текст (bold):

```
<b>Полужирный текст (bold)</b>
```

- наклонный текст (italics):

```
<i>Наклонный текст (italics)</i>
```

- большой размер шрифта:

```
<big>Большой размер шрифта</big>
```

- маленький размер шрифта:

```
<small>Маленький размер шрифта</small>
```

- нижние индексы$_{\text{(subscript)}}$:

```
<sub>(subscript)</sub>
```

- верхние индексы$^{\text{(superscript)}}$:

```
<sup>(superscript)</sup>
```

Имя тега задается либо по первой букве названия стиля, либо по части этого названия, либо по полному англоязычному названию (для коротких названий).

В HTML имеется возможность показа предварительно отформатированного текста. Текст внутри контейнера `<pre>...</pre>` выводится моноширинным шрифтом, все символы имеют одинаковую ширину, все пробелы и переходы на новую строку показываются без игнорирования. Несмотря на то, что этот тег признан устаревшим, он очень удобен и поддерживается большинством современных браузеров.

Язык HTML позволяет вставлять в текст изображения, хранимые в отдельных графических файлах. Тег вывода изображения имеет следующий вид:

```
<img src="имя_файла" width=ширина height=высота border=ширина_рамки
     hspace=отступ_вертикальный vspace=отступ_горизонтальный
     alt=текстовое_сообщение>
```

где:

- `имя_файла` — имя графического файла (с указанием расширения), содержащего изображение;
- `ширина` — ширина изображения;
- `высота` — высота изображения;
- `ширина_рамки` — ширина рамки вокруг изображения;

- текстовое_сообщение — текст, который виден на месте изображения, пока идет его загрузка, или в режиме, когда в браузере отключен показ изображений.

Все размеры задаются в пикселах (точках экрана). Если реальный размер изображения не совпадает с заданным в атрибутах width и height, то при показе изображение масштабируется до этих размеров.

Атрибут border не обязателен, но желателен, если с картинкой ассоциирована гиперсвязь.

Атрибуты hspace и vspace задают отступы от картинки по вертикали и горизонтали для текста или других картинок.

Рекомендуется всегда указывать ширину и высоту изображения, в противном случае программа просмотра будет вынуждена перед выводом изображения документа на экран загрузить как весь текстовой файл, так и все файлы с изображениями, что занимает много времени. Если же атрибуты width и height указаны, то текст покажется сразу, а изображения будут показываться по мере подкачивания по сети. Кроме того, объем текстовых файлов, как правило, намного меньше, чем графических, поэтому они появляются на экране гораздо быстрее.

Для того чтобы изображение служило гиперссылкой, достаточно поместить тег img внутрь тега <a href="адрес"> следующим образом:

```
<a href=... ><img src=...></a>
```

В HTML-документах можно задавать *таблицы*. Каждая таблица должна начинаться тегом <table>, а если для таблицы требуется внешняя рамка, то с параметром border (возможны варианты: border или border=ширина_рамки), и заканчиваться тегом </table>.

Таблицы задаются построчно. Каждая строка начинается тегом <tr> и заканчивается тегом </tr>. Каждая графа (т. е. ячейка, клетка) в строке с данными должна начинаться тегом <td> и заканчиваться тегом </td> (этот тег необязателен, но желателен). При этом ширина столбцов подбирается автоматически по максимальной ширине одной из клеток столбца. В таблицы так же можно вставлять гипертекстовые ссылки, произвольным образом отформатированный текст, рисунки и т. п.

В листинге 11.5 приведен общий вид таблицы.

Листинг 11.5. Задание таблицы

```
<table border>
    <tr> <td>...</td> <td>...</td> <td>...</td> <td>...</td> </tr>
    <tr> <td>...</td> <td>...</td> <td>...</td> <td>...</td> </tr>
    ...
</table>
```

Эта таблица содержит две строки и четыре графы (столбца). Поскольку в HTML перевод на новую строку равнозначен пробелу, а лишние пробелы игнорируются, то текст в HTML-документах обычно форматируют с помощью переводов на новую строку так, чтобы не было слишком длинных строк. Поэтому таблицу из листинга 11.5 можно записать в виде листинга 11.6.

Листинг 11.6. Другой вариант форматирования текста при задании таблицы

```
<table border>
<tr>
 <td>...</td>
 <td>...</td>
 <td>...</td>
 <td>...</td>
</tr>
<tr>
 <td>...</td>
 <td>...</td>
 <td>...</td>
 <td>...</td>
</tr>
</table>
```

Вид документа при просмотре файла HTML-браузером (browser), естественно, не изменится.

Если в таблице нужны заголовки, они задаются тегами `<hr>...</hr>`.

Окна подключаемых модулей (подключаемых объектов, plug-ins) задаются контейнером `<object>` (листинг 11.7).

Листинг 11.7. Задание окна подключаемых модулей (plugins)

```
<object
 атрибуты
>
 <param name=имя1 value=значение1>
 <param name=имя2 value=значение2>
 ...
 <param name=имяN value=значениеN>
Альтернативный_текст
</object>
```

Такими объектами могут быть апплеты Java, мультимедийные клипы и т. п.

`Альтернативный_текст` — это текст, который будет виден в браузерах, не поддерживающих работу с объектами данного типа.

Математические и специальные символы, греческие буквы

Кодирование математических символов в HTML приведено в табл. 11.1 (хотя слабые браузеры могут показывать только часть из них).

Кодирование специальных символов в HTML приведено в табл. 11.2.

Кодирование греческих букв в HTML приведено в табл. 11.3.

Таблица 11.1. *Кодирование математических символов*

∂	∂	∏	∏	∑	∑	√	√	∞	∞
∩	∩	∫	∫	±	±	≡	≡	≈	≈
≠	≠	<	<	>	>	≤	≤	≥	≥
≠	≠	ˆ	ˆ	˜	˜	Ø	Ø	ø	ø
÷	÷								

Таблица 11.2. *Кодирование специальных символов*

°	°	•	•	…	…	′	′	″	″
‾	‾	™	™	©	©	®	®	♦	♦
◊	◊	—	—	–	–	’	’	†	†
‡	‡	"	«	"	»				

Таблица 11.3. *Кодирование греческих букв с помощью кодов символов*

Α	Α	Β	Β	Γ	Γ	Δ	Δ	Ε	Ε
Ζ	Ζ	Η	Η	Θ	Θ	Ι	Ι	Κ	Κ
Λ	Λ	Μ	Μ	Ν	Ν	Ξ	Ξ	Ο	Ο
Π	Π	Ρ	Ρ	Σ	Σ	Τ	Τ	Υ	Υ
Φ	Φ	Χ	Χ	Ψ	Ψ	Ω	Ω		
α	α	β	β	γ	γ	δ	δ	ε	ε
ζ	ζ	η	η	θ	θ	ι	ι	κ	κ
λ	λ	μ	μ	ν	ν	ξ	ξ	ο	ο
π	π	ρ	ρ	ς	ς	σ	σ	τ	τ
υ	υ	φ	φ	χ	χ	ψ	ψ	ω	ω

Существует еще один вариант задания греческих букв: через кодирующую последовательность вида `&имяБуквы;` (табл. 11.4).

Например:

α это `α`

β это `β`

γ это `γ`

и т. д.

Заглавные греческие буквы получаются так:

Α это `Α`

Β это `Β`

Γ это `Γ`

и т. д.

Таблица 11.4. *Кодирование греческих букв с помощью названий букв*

Α	Α	Β	Β	Γ	Γ	Δ	Δ	Ε	Ε
Ζ	Ζ	Η	Η	Θ	Θ	Ι	Ι	Κ	Κ
Λ	Λ	Μ	Μ	Ν	Ν	Ξ	Ξ	Ο	Ο
Π	Π	Ρ	Ρ	Σ	Σ	Τ	Τ	Υ	Υ
Φ	Φ	Χ	Χ	Ψ	Ψ	Ω	Ω		
α	α	β	β	γ	γ	δ	δ	ε	ε
ζ	ζ	η	η	θ	θ	ι	ι	κ	κ
λ	λ	μ	μ	ν	ν	ξ	ξ	ο	ο
π	π	ρ	ρ	σ	σ	τ	τ	υ	υ
φ	φ	χ	χ	ψ	ψ	ω	ω		

11.2. Апплеты

Структура апплета

Напомним некоторые вещи, о которых рассказывалось в *главе 1*.

Апплет — это специализированная программа Java с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера. Как правило, апплеты встраивают в HTML-документы (наиболее распространенный вид WWW-документов).

Между приложениями (applications) и апплетами (applets) Java имеется принципиальное различие. Приложение запускается непосредственно с компьютера пользователя и имеет доступ ко всем ресурсам компьютера наравне с любыми другими программами. Апплет же загружается из WWW с постороннего сервера, причем из-за самой идеологии WWW-сайт, с которого загружен апплет, в общем случае не может быть признан надежным. А вот сам апплет способен передавать данные на любой сервер в WWW — все зависит от алгоритма, заложенного создателем апплета. Поэтому, чтобы избежать риска утечки конфиденциальной информации с компьютера пользователя или застраховаться от совершения враждебных действий, у апплетов убраны многие возможности, имеющиеся у приложений.

Работу с апплетами поддерживает стандартная библиотека классов (core library), расположенная в пакете `java.applet` для обычных апплетов, а также класс `javax.swing.JApplet` для апплетов, использующих компоненты `Swing`, и/или библиотека Sun JFC (Java Foundation Classes).

Для создания обычного апплета требуется задать класс, являющийся наследником класса `java.applet.Applet`, который сам выступает наследником класса `java.awt.Panel`.

В классе апплета требуется переопределить ряд методов (листинг 11.8).

Листинг 11.8. Задание класса апплета

```
public class Applet1 extends java.applet.Applet{

    public void init(){
     //Инициализация перед началом работы.
     //Вызывается один раз после загрузки апплета
     //перед первым выполнением метода start()
    }

    public void start(){
     //Обеспечивает основную функциональность апплета.
     //В первый раз вызывается после загрузки апплета
     //и его инициализации методом init().
     //Затем вызывается каждый раз при заходе пользователя
     //на HTML-страницу с апплетом.
    }

    public void update(java.awt.Graphics g){
     //Форсирование перерисовки апплета с выполнением кода метода
    }

    public void paint(java.awt.Graphics g){
     //Исполняется каждый раз при перерисовке апплета.
     //Обеспечивает всю визуализацию в апплете
    }

    public String getAppletInfo(){
        return  "Справочная информация об апплете";
    }

     public void stop(){
     //Приостанавливает выполнение апплета.
     //Исполняется каждый раз сразу после того, как пользователь
     //покидает HTML-страницу с апплетом.
    }

    public void destroy(){
     //Обычно не требует переопределения.
     //Предназначен для высвобождения ресурсов, захваченных апплетами.
     //Исполняется через негарантированный промежуток времени
     //каждый раз после вызова метода stop()
     //перед разрушением объекта апплета сборщиком мусора.
    }
}
```

Кроме методов, нуждающихся в переопределении, в классе `Applet` имеются методы, позволяющие проверять и задавать его состояние во время выполнения:

- `getSize()` — возврат размера апплета (ширину и высоту можно получить как `getSize().width` и `getSize().height`);

- `showStatus(String s)` — показ в строке статуса браузера сообщения `s`;
- `AppletContext getAppletContext()` — получение апплетом информации о документе, из которого был вызван данный апплет, а также о других апплетах того же документа;
- `add(Component comp)` — добавление компонента в апплет;
- `AudioClip getAudioClip(URL url)` — получение апплетом аудиоклипа по заданному WWW-адресу url. Создается объект Java, ссылающийся на данный аудиоклип;
- `URL getDocumentBase()` — получение апплетом адреса WWW-документа, из которого был вызван апплет.

Есть много других методов для работы с апплетами, большинство которых унаследовано от класса `Panel`.

При работе с апплетами возникает ряд трудностей. В свое время ни на одном из компьютеров с JDK 5, Windows XP SP1 и браузером MS Internet Explorer 6.0 автору не удалось запустить ни одного апплета, в том числе из примеров JDK (хотя поддержка Java была включена, и автор пытался менять самые разные установки системы). Но в программе appletviewer при компиляции апплетов программой javac все работало. Правда, под JDK 6 на тех же компьютерах апплеты заработали. В Internet Explorer отображение апплетов блокируется — выдается предупреждение о том, что активное содержимое может нанести вред компьютеру (хотя, как мы знаем, апплеты специально сконструированы так, чтобы это было невозможно). Так что требуется специально разрешать показ апплетов. В Mozilla FireFox 3.05 показывались только апплеты, помеченные в HTML-документах с помощью устаревшего тега `<applet>`, а при наличии современного тега `<object>` Mozilla FireFox блокировал показ апплета.

Аналогичные проблемы возникали и у студентов при попытке написать собственные апплеты. С каждой новой версией JDK работоспособность Java повышается, тем не менее, крупные фирмы предпочитают при работе в сетях использовать не апплеты, а заниматься обработкой со стороны сервера и отсылать результаты на компьютер клиента в готовом виде. Ведь у клиентов, работающих в WWW, могут быть компьютеры с различными версиями JDK, операционными системами и браузерами.

Ряд проблем связан с настройкой путей. К тому же гарантировать работоспособность одного и того же апплета в таком разнообразном окружении практически невозможно. Если же обработка идет со стороны сервера, то все упрощается. Со стороны клиента нужен только браузер, работающий со ставшими совершенно стандартными языками HTML и JavaScript. Браузер передает запрос с клиентского компьютера на сервер и там формируется новый HTML-документ для клиентского компьютера (при необходимости со сформированными на сервере графическими файлами).

Такая идеология позволяет усовершенствовать систему незаметно для пользователей и с сохранением полной совместимости. Например, новая версия JVM устанавливается на сервере, и проводятся все необходимые настройки и тестирование, после чего вместо старой версии системы снаружи становится доступна новая версия.

Примеры апплетов. Аннотация *@Override*

Ряд примеров апплетов с исходными кодами приведен в JDK в папке demo/applets. Но большинство из них слишком сложны. Поэтому мы рассмотрим более простые

примеры, находящиеся в проекте 11_2_applets. В нем содержатся исходные коды трех классов (`Applet1`, `Applet2` и `Applet3`), а также четыре HTML-файла, которые нужно открывать в браузере для просмотра работы апплетов. В дальнейшем файлы example1.html, example2.html, example3.html, example4.html и используемые в них классы апплетов мы будем называть примером 1, 2, 3 и 4 соответственно.

Рассмотрим содержимое этих файлов. Пример 1 иллюстрируют листинги 11.9 и 11.10.

Листинг 11.9. Исходный код класса апплета `Applet1`

```
/**
 * The Applet1 class demonstrates
 * applet code.
 * It displays "Hello World!" greeting.
 * Run example1.html to see working applet.
 */
public class Applet1 extends java.applet.Applet {
    @Override
    public void paint(java.awt.Graphics g) {
        g.drawString("Hello world!", 50, 50);
    }
}
```

Директива `@Override` — необязательная аннотация, поясняющая человеку, читающему текст программы, что данный метод переопределяет родительский. Одновременно данная директива является метаданными для компилятора. Если метод аннотирован символом `@Override`, но на самом деле не переопределяет родительский метод (например, имеет другую сигнатуру), то компилятор на основании этой аннотации выдает сообщение об ошибке.

Листинг 11.10. Содержимое HTML-документа example1.html, в который встроено окно апплета `Applet1`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
 <title>Пример апплета Hello world </title>
</head>

<body bgcolor=#C0C0C0>
<h3>Пример 1. Hello world</h3>

Если ваш браузер делает запрос о разрешении отображать активное содержимое, то нужно дать
такое разрешение, иначе не удастся увидеть апплет.

<p>Это пример запуска апплета с помощью тега object в HTML-документе:
<hr>
```

```
<object
 codebase="."
 code="Applet1.class"
 width=200
 height=150
>
Здесь в окне апплета должно выводиться приветствие: Hello world. Это альтернативный текст,
который будет виден в браузерах, не поддерживающих работу с апплетами.
</object>
</body>
</html>
```

Для просмотра апплета в среде NetBeans следует в окне **Projects** вызвать правой кнопкой мыши в дереве проекта для файла апплета **Applet1.java** контекстное меню и выбрать в нем пункт **Run File**. Либо перейти в редакторе кода в этот файл и вызвать в главном меню пункт **Run | Run File**, либо использовать комбинацию клавиш <Shift>+<F6>. Апплет запустится в окне программы appletViewer ("просмотрщик апплетов"), входящей в состав JDK и предназначенной для отладки работы с апплетами. Но правильный показ апплета в appletViewer еще не гарантирует его корректной работы в браузерах при просмотре в составе HTML-документа.

Самый простой способ запустить апплет в составе HTML-документа — вызвать в окне **Projects** правой кнопкой мыши в дереве проекта для HTML-файла контекстное меню и выбрать в нем пункт **View** ("просмотр"). Если в открывшемся под управлением NetBeans браузере апплет не будет правильно показан даже после разрешения просмотра "опасного" активного содержимого (в Internet Explorer), еще ничего не значит — лучше просматривать HTML-документ непосредственно в браузере. Просмотр с помощью пункта **View** правильно работает только для HTML-документов без активного содержимого.

Для правильного просмотра в браузере следует с помощью окна **Мой компьютер**, программы Far или другого навигатора перейти в папку 11_2_applets\build\classes и открыть в браузере файл example1.html. Обычно проще всего это сделать двойным щелчком мыши по имени файла в окне навигации по файлам и папкам. Следует отметить, что исправленные версии исходных кодов HTML-файлов при компиляции проекта автоматически копируются в папку build\classes вместе со скомпилированными файлами классов Java (.class).

При переносе апплета на другой компьютер или при размещении его на сервере для работы в WWW следует расположить файлы Applet1.class и example1.html в одной папке.

В начале документа указана спецификация

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

задающая версию HTML, которая должна использоваться для просмотра HTML-документа.

Строка

```
<meta http-equiv="content-type" content="text/html; charset= UTF-8">
```

служит для задания русской кодировки символов в варианте мультиязычной кодировки Unicode.

Строка:

```
<title>Пример апплета Hello world </title>
```

задает заголовок окна браузера, в котором будет показан документ.

Текст, ограниченный тегом `object`, задает параметры окна апплета при показе в браузере или в программе appletviewer.

Атрибут `codebase` задает базовый адрес для апплета и данных к нему. Все остальные пути к файлам, используемые апплетом (классов Java, данным), если в них не указан полный путь, отсчитываются от базового адреса. Если указана точка, значит, путь отсчитывается от той папки, в которой находится HTML-документ.

Значение атрибута `code` задает имя файла с апплетом.

Атрибуты `width` и `height` задают ширину и высоту окна апплета. Они отсчитываются в пикселах (точках экрана). Характерный размер экрана компьютеров составляет 1024 пиксела в ширину и 768 пикселов в высоту (для разрешения 1024×768).

В листингах 11.11 и 11.12 приведен пример 2 (для просмотра апплета нужно запустить файл example2.html). Здесь, кроме вывода текста, в окне рисуется эллипс. Класс Java этого примера (листинг 11.11) похож на класс первого примера (см. листинг 11.9), но в нем немного по-другому организован импорт имен, а также проиллюстрировано, как рисовать эллипс, задавать цвет эллипса и текста.

Листинг 11.11. Исходный код класса апплета `Applet2`

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Color;

/**
 * The Applet2 class demonstrates
 * applet code.
 * It displays an oval and  "Hello World!" greeting.
 * Run example2.html to see working applet.
 */
public class Applet2 extends Applet {

    @Override
    public void paint(Graphics g) {
        g.setColor(Color.green);
        g.fillOval(0, 0, 100, 50);
        g.setColor(Color.black);
        g.drawString("Hello world!", 50, 50);
    }
}
```

В листинге 11.12 в документе example2.html показаны два апплета, каждый из которых является экземпляром класса `Applet2`. Проиллюстрированы два разных тега (`<object ...>` и `<applet...>`) для отображения апплета в окне HTML-документа.

Листинг 11.12. Содержимое HTML-документа example2.html, в который встроены два окна апплета `Applet2`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
<head>
 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
 <title>Пример апплета. Hello world и эллипс </title>
</head>

<body bgcolor=#C0C0C0>
<h3>Пример 2. Hello world и эллипс</h3>

Если ваш браузер делает запрос о разрешении отображать активное содержимое, то нужно дать
такое разрешение, иначе не удастся увидеть апплет.
<p>Это пример запуска апплета с помощью тега object:

<hr>
<object
 codebase="."
 code="Applet2.class"
 width=200
 height=150
>

Здесь должен нарисоваться эллипс и приветствие: Hello world.
Альтернативный текст, который будет виден в браузерах, не поддерживающих работу
с апплетами
</object>

Другой вариант запуска апплета — с помощью устаревшего тега applet:<p>

<applet code=Applet2.class width=300 height=120>
alt="Ваш браузер понимает тег &lt;APPLET&gt;, но почему-то не показывает апплет!"
Ваш браузер полностью игнорирует тег &lt;APPLET&gt; !
</applet>
</body>
</html>
```

Файлы example2.html и example1.html (см. листинг 11.10) отличаются именем апплета, который запускается, а также наличием второго варианта запуска апплета — с помощью устаревшего тега `<applet>` (рис. 11.1).

Большинство браузеров и appletviewer показывают апплеты, заданные с помощью этого тега, хотя в спецификации HTML 4 он и признан устаревшим (deprecated).

Пример 3 (файлы Applet3.java и example3.html на CD) иллюстрирует, как в апплете показать график функции.

Пример 4 (файлы example4.html, Applet1.java, Applet2.java и Applet3.java на DVD) демонстрирует, как в HTML-документе показывать несколько разных апплетов. Заметим,

что в appletviewer HTML-документ не виден. Но если в нем показывается несколько апплетов (как в примерах 2 и 4), то каждый апплет появляется в своем независимом окне.

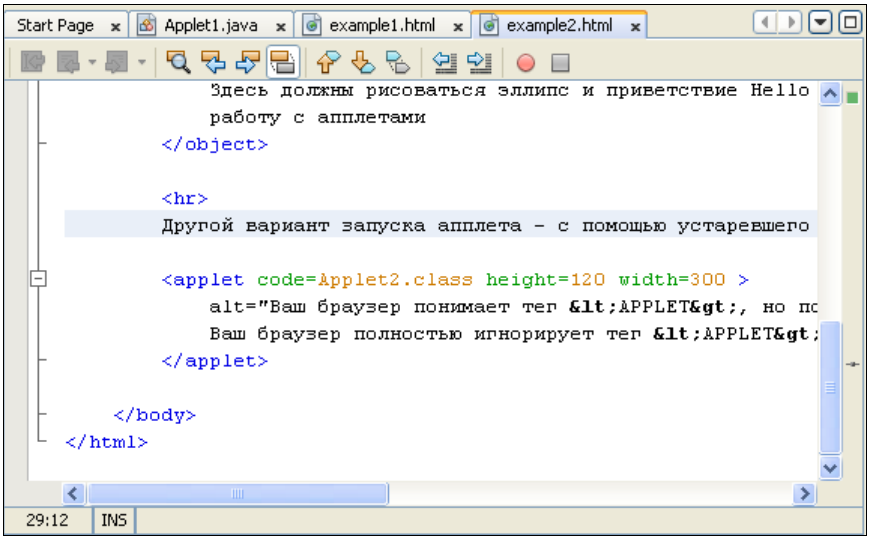


Рис. 11.1. Показ устаревшего тега `<applet>` в проекте NetBeans

В связи с тем, что ничего принципиально нового в этих примерах нет, а исходные коды соответствующих классов Java уже были приведены ранее, мы не будем рассматривать листинги для файлов example3.html и example4.html.

Создание проекта с апплетами

В данном разделе описывается процесс формирования проекта с апплетами, а также процедура добавления в проект HTML-файлов, что практически всегда бывает нужно и для серверных приложений.

Для разработки нового проекта NetBeans нужно в главном меню выполнить команду **File/New Project...**, после чего в появившейся форме выбрать пункт **Java/Java Class Library** и нажать кнопку **Next >**.

Появится форма, в которой можно уточнить имя и месторасположение проекта. Введем, к примеру, имя `11_2_Applet1`.

У нас появится пустой проект с именем `11_2_Applet1`.

Если мы хотим создать новый класс апплета, то следует щелкнуть правой кнопкой мыши по области **<default package>** в дереве проекта и в появившемся всплывающем меню выбрать **New/Other…/Java/JApplet**. Однако мы все равно будем менять код заготовки, поэтому можно провести создание проекта с апплетом, выбрав **New/Java Class**.

Появится форма, в которой необходимо задать имя создаваемого класса, а также можно указать пакет, в котором этот класс должен располагаться. Введем имя `Applet1`.

В нижней части формы дано предупреждение (warning) о том, что настоятельно не рекомендуется использовать пакет по умолчанию (`Warning: It is highly recommended that you do NOT place Java classes in the default package`). Но мы это предупреждение проигнорируем и нажмем кнопку **Finish**.

В редакторе появится исходный код, приведенный в листинге 11.13.

Листинг 11.13. Заготовка апплета

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author User
 */
public class Applet1 {

}
```

Закомментируем содержание заготовки: выделим весь текст (например, нажатием комбинации клавиш <Ctrl>+<A>) и нажмем на панели инструментов редактора исходного кода предпоследнюю кнопку с двумя зелеными наклонными линиями на фоне блока текста (**Comment** — комментировать).

После этого добавим в заготовку исходный код апплета, приведенный в примере 1 (см. листинг 11.9). Теперь нам необходимо добавить в проект HTML-документ. Для этого щелкнем по самой первой кнопке панели инструментов (зеленый плюсик на фоне пустой страницы), после чего появится диалог выбора типа создаваемого файла, нам необходим в нем пункт **Other...** (другое).

Этот диалог можно вызвать и через главное меню **File/New File...**, и щелчком правой кнопки мыши по области **<default package>** в дереве проекта, и последующим выбором в появившемся всплывающем меню пункта **New/Other...** (Новый/Другой...) (рис. 11.2).

В появившейся диалоговой форме **New File** (рис. 11.3) следует выбрать категорию **Other** (в группе **Categories**), затем тип файла **HTML File** (в группе **File Types**), потом нажать кнопку **Next >**.

Появится форма, в которой необходимо изменить имя файла и (при необходимости изменить значения по умолчанию) место его расположения. Мы зададим имя `index` для создания файла index.html (стандартное название головной страницы сайта). При расположении на сервере страница index.html открывается автоматически при заходе в папку, где она расположена (если не указан конкретный файл из этой папки).

После нажатия кнопки **Finish** появится заготовка HTML-документа, представленная в листинге 11.14.

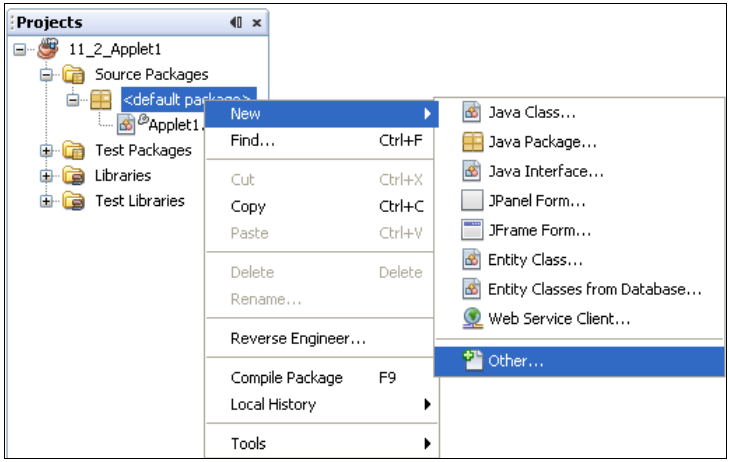


Рис. 11.2. Создание в проекте нового HTML-файла с помощью всплывающего меню, шаг 1

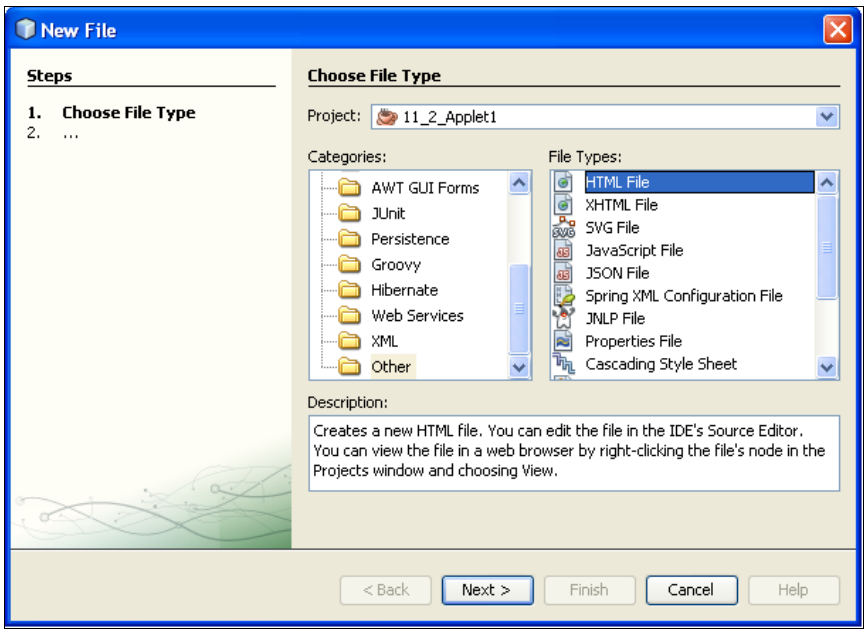


Рис. 11.3. Создание в проекте нового HTML-файла, шаг 2

Листинг 11.14. Заготовка HTML-документа, созданная средой NetBeans

```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    TODO write content
  </body>
</html>
```

Заменим содержимое тега <body> на то, которое нам нужно (см. листинг 11.10). После компиляции в папке Applet1\build\classes появятся файлы index.html и Applet1.class. Если запустить в браузере файл index.html, то увидим апплет — если, конечно, он покажется. Напомним, что лучше всего показывать апплеты с помощью тега <applet>, т. к. хоть он и считается устаревшим, но с ним работают практически все браузеры, в том числе FireFox.

Заметим, что теперь создание HTML-файла стало доступным из контекстного всплывающего меню, т. к. в нем появился пункт **HTML...**. Команды этого меню не заданы раз и навсегда — в них автоматически добавляются пункты, соответствующие последним по времени вызовам, которые заменяют редко используемые команды меню.

Апплет на основе формы Swing, позволяющей визуально проектировать его интерфейс, можно разработать с помощью создания файла типа JApplet Form. Для этого следует щелкнуть правой кнопкой мыши по области **<default package>** в дереве проекта и в появившемся всплывающем меню выбрать **New/Other…/Swing GUI Forms/JApplet Form**.

Редактор HTML-документов, стилей CSS и кода JavaScript

Редактор исходного кода NetBeans позволяет достаточно удобно редактировать HTML-файлы и стили CSS. Он не только обеспечивает выделение цветом тегов, атрибутов и значений атрибутов, что присуще многим редакторам, но еще и диагностирует ошибки при написании тегов, выделяя соответствующие участки оранжевой или красной волнистой линией и устанавливая в левом столбце оранжевые или красные квадраты с крестиком, сигнализирующие об ошибке в соответствующей строке (рис. 11.4).

Например, если мы в строке

```
</object>
```

исправим имя тега на </obgect>, сымитировав ошибку, то получим диагностику в двух местах:

- предупреждение в строке <object с сообщением: Unmatched teg (незавершенный тег), означающим, что имеется открывающий тег, а закрывающего для него тега нет;
- такое же сообщение об ошибке в испорченной строке </obgect> с сообщением о непарном закрывающем теге.

Сообщение об ошибке показывается при наведении курсора мыши на треугольник с восклицательным знаком, помечающего строку с предупреждением.

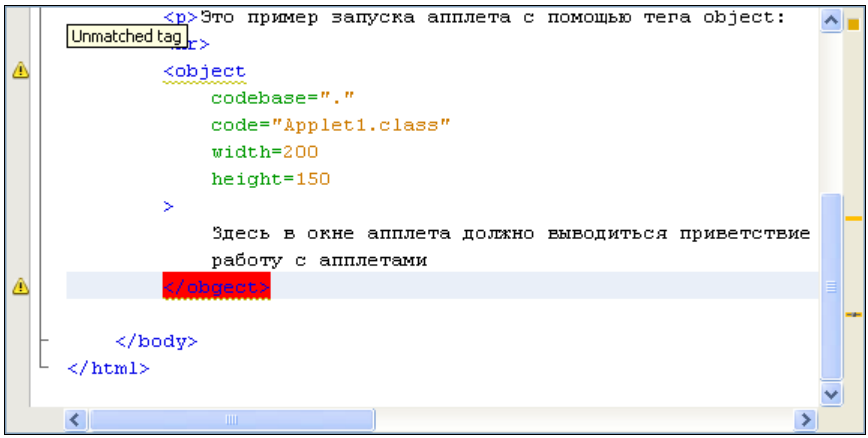


Рис. 11.4. Показ ошибок и предупреждений при редактировании HTML-файлов

В редакторе HTML-документов имеется еще одна удобная вещь — справка по тегам и их атрибутам. Для того чтобы вызвать справку по какому-нибудь тегу или атрибуту, следует установить на него курсор редактирования текста и нажать комбинацию клавиш <Ctrl>+<Пробел>, после чего появится контекстная справка (рис. 11.5 и 11.6).

Рис. 11.5. Контекстная справка по атрибуту `bgcolor`

В начале справки выводится список тегов, в котором выделена строка с нашим тегом. Далее идет окно справки по выбранному в списке тегу. Сначала идет формальное определение элемента и его атрибутов с помощью XML-нотации (эту часть справки не следует пытаться понять, а нужно просто пропустить). А вот дальше приведена полезная

информация — перечисляются допустимые атрибуты и описано их назначение. Сначала почему-то указываются устаревшие (deprecated), не рекомендуемые к использованию атрибуты (в теге <p> они отсутствуют), и только потом — действующие.

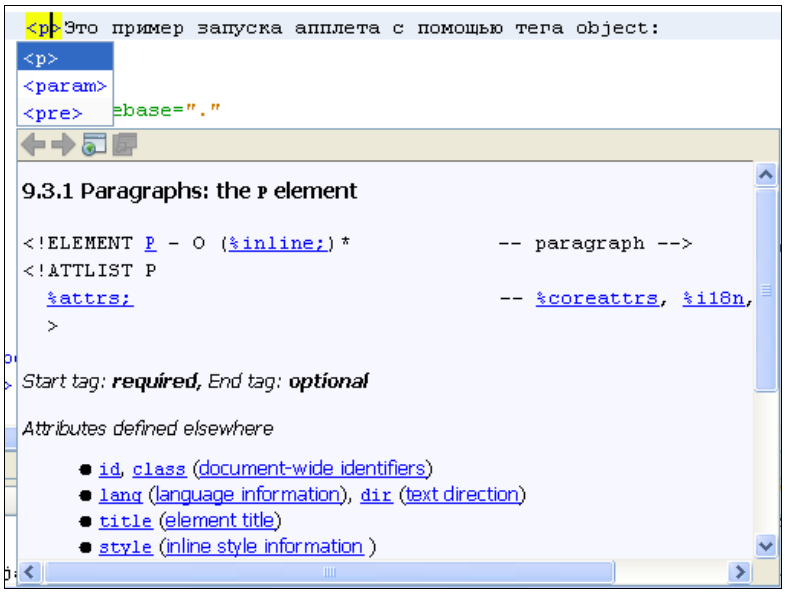


Рис. 11.6. Контекстная справка по тегу <p>

Если курсор установить после открывающей тег левой треугольной скобки и нажать комбинацию клавиш <Ctrl>+<Пробел>, то появится список допустимых тегов и подсказка по выделенному в этом списке тегу. При наборе с клавиатуры нового HTML-текста сразу после ввода такой скобки появляется список тегов даже без комбинации клавиш <Ctrl>+<Пробел>.

Аналогично, если после имени тега и следующего за ним пробела нажать комбинацию клавиш <Ctrl>+<Пробел>, то появится список допустимых атрибутов. А при наборе с клавиатуры нового HTML-текста этот список появляется сразу после ввода пробела после имени тега.

Заметим, что при редактировании HTML-документов можно автоматически форматировать исходный код так же, как при редактировании Java-файлов: с помощью комбинации горячих клавиш <Shift>+<Alt>+<F> или через пункт всплывающего меню **Format**.

Еще одна полезная особенность редактора — возможность использования палитры HTML-элементов (рис. 11.7).

Из этой палитры можно перетащить любой элемент в необходимое место HTML-документа. Откроется форма, в которой будет предложено уточнить значения параметров этого элемента, а после ввода этих параметров в HTML-тексте появятся необходимые теги и атрибуты. Например, при вставке таблицы это будет выглядеть так, как показано на рис. 11.8.

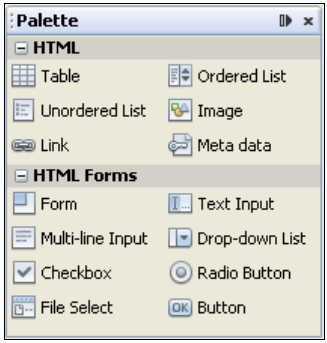


Рис. 11.7. Палитра HTML-элементов

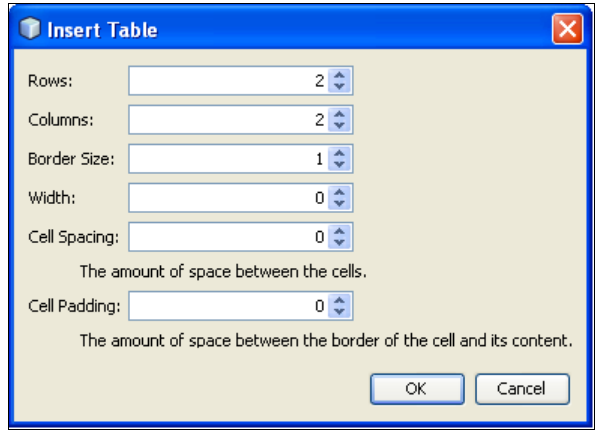


Рис. 11.8. Добавление в HTML-документ таблицы

После нажатия кнопки **OK** в HTML-документе появится заготовка таблицы, представленная в листинге 11.15.

Листинг 11.15. Заготовка HTML-таблицы, созданная средой NetBeans

```
<table border="1">
    <thead>
        <tr>
            <th></th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td></td>
            <td></td>
        </tr>
        <tr>
            <td></td>
            <td></td>
        </tr>
    </tbody>
</table>
```

Удивительно, но среди HTML-компонентов отсутствует апплет. По-видимому, Sun Microsystems в настоящее время не придает особого значения апплетам.

Совершенно так же, как HTML, редактируется код сценариев (script) языка JavaScript, причем редактор Java показывает и диагностирует многие ошибки в этом коде. И точно так же вызывается справка.

Например, код, приведенный в листинге 11.16, в режиме показа справки будет выглядеть так, как на рис. 11.9.

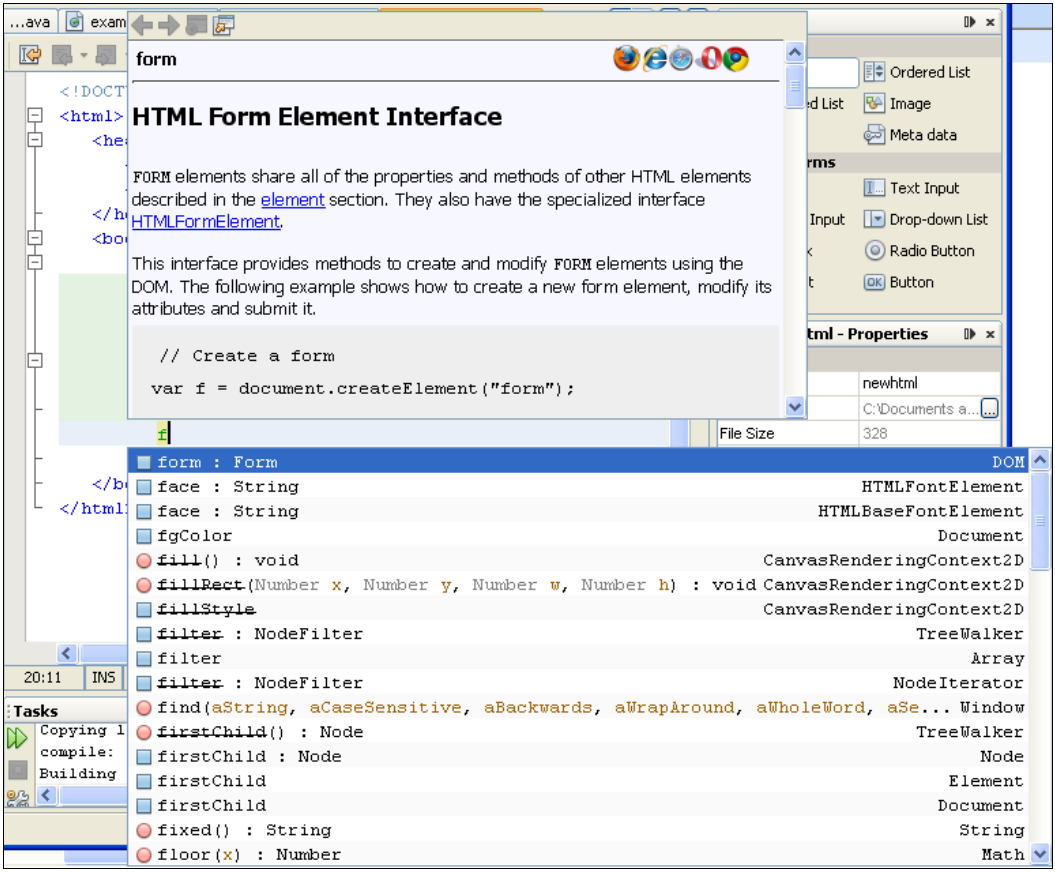


Рис. 11.9. Справка в режиме редактирования сценария JavaScript

Листинг 11.16. Пример сценария JavaScript в коде HTML

```
<!--
    Document   : newhtml
    Author     : V.Monakhov
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
   <head>
      <title></title>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
   </head>
   <body>
       <script >
          var x=0
          var y=sin(5.5*x)+1
          x=exp(y)
          function f(t){
```

```
            return t*t
           }
        </script>
    </body>
</html>
```

Отметим, что редактор не только показывает имеющиеся поля и методы JavaScript, но и дает информацию о том, какими браузерами и начиная с какой версии соответствующий элемент поддерживается (логотипы браузеров показываются в правом верхнем углу, а в тексте для многих полей и методов приводится таблица с указанием поддерживающих или не поддерживающих их браузеров).

Аналогичные возможности имеются и при использовании стилей CSS (их рассмотрение выходит за рамки данной книги).

В заключение отметим, что для просмотра отредактированного HTML-документа следует в окне **Projects** правой кнопкой мыши вызвать для него контекстное меню и выбрать пункт **View** (Просмотр).

Развитые возможности редактирования HTML-документов, сценариев JavaScript и стилей CSS делают NetBeans удобным средством подготовки WWW-страниц безотносительно к применению языка Java.

11.3. Сервлеты

Напомним, что сервлеты — это приложения Java, запускаемые на стороне сервера.

Пояснение

Слово *сервер* означает *обслуживающий* и применяется в двух различных смыслах. Во-первых, серверами называют компьютеры (обычно достаточно мощные), обслуживающие большое число пользователей. Во-вторых, серверами называют программы, позволяющие осуществлять те или иные функции такого обслуживания. Например, WWW-сервер позволяет осуществлять доступ через Интернет к находящимся на компьютере HTML-документам. Аналогично, FTP-сервер позволяет добираться через Интернет до файлов того же компьютера по протоколу FTP. Почтовый сервер позволяет получать и отсылать электронную почту со своим собственным набором пользователей и т. д. В нашем случае программный сервер (сервер приложений Java) должен поддерживать работу с сервлетами.

Как уже говорилось в *разд. 1.2*, сервлеты имеют возможности доступа к файловой системе и другим ресурсам сервера, а также к содержимому HTTP-запросов со стороны клиентского компьютера. А технология JSP основана на использовании специальных тегов в HTML- или XML-документах, которые позволяют серверу автоматически генерировать и компилировать сервлеты. В результате удается очень просто и удобно осуществлять обработку данных или элементов документа, а также внедрять в нужные места документа результаты обработки. Сервлеты автоматически генерируются всего один раз — перед первым выполнением обработки документа с тегами JSP на стороне сервера, поэтому код JSP выполняется достаточно быстро (но, конечно, требуется, чтобы была установлена соответствующая Java-машина).

Сам сервер — это программа, написанная в рамках технологии Java EE (Java Platform, Enterprise Edition). Модуль сервера, отвечающий за исполнение сервлетов, называется сервлет-контейнером.

Для дальнейшей работы требуется, чтобы на компьютере, где происходит разработка примеров, был установлен модуль NetBeans Web & Java EE и сервер GlassFish или Tomcat Apache *(см. разд. 2.2)*.

Для правильной работы серверных примеров необходимо, чтобы на компьютере была обеспечена поддержка работы с Интернетом. Но выходить в Интернет не нужно, осуществляется соединение **http://localhost:8080/**.

В данном имени сначала идет имя сервера localhost (локальный компьютер), после двоеточия указывается номер порта, который обычно служит для вызова различных служб на одном и том же аппаратном сервере. Для программного сервера GlassFish адрес порта по умолчанию 8080, для Tomcat Apache — 8084. Реального соединения с Интернетом не происходит — все коммуникации станут проходить локально.

Сервлеты — основа технологий JSP и JSTL, поэтому их изучение весьма важно. В наиболее простых серверных приложениях можно ограничиться сервлетами; в более сложных — использовать выражения JSP и сервлеты; в еще более сложных — применять JSTL; и наконец, в наиболее сложных случаях создавать собственные теги JSP.

Пример приложения, работающего с использованием сервлетов, приведен в листинге 11.17.

Листинг 11.17. Исходный код сервлета, выдающего на клиентском компьютере сообщение "Привет!"

```
/**
 * @author Vadim Monakhov
 */

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Hello extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
    throws IOException, ServletException
    {
       response.setContentType("text/html");
       response.setCharacterEncoding("UTF-8");
       PrintWriter out = response.getWriter();
       out.println("<html>");
       out.println("<head>");
       out.println("<meta http-equiv=\"content-type\"
                      content=\"text/html; charset=UTF-8\">");
       out.println("<title>Сгенерированный сервлетом документ</title>");
       out.println("</head>");
```

```
        out.println("<body>");
        out.println(
                "<h1>Пример документа, сгенерированного сервлетом</h1>");
        out.println("Привет!");
        out.println("</body>");
        out.println("</html>");
    }
}
```

В листинге 11.18 представлен пример HTML-файла, из которого вызывается сервлет, приведенный в листинге 11.17.

Листинг 11.18. Исходный код HTML-файла, из которого вызывается сервлет

```
<html>
 <head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
   <title>Пример доступа к сервлету</title>
 </head>
<body bgcolor="#FFFFFF">
  <a href="servlet/Hello">Запустить сервлет</a>
</body>
</html>
```

Отметим, что обычно в приводимых примерах вывод осуществляется на английском языке. При попытке вывода текста на русском или другом национальном языке возникают проблемы. Данный пример иллюстрирует работу с текстом в кодировке UTF-8. В редакторе кода NetBeans кодировка UTF-8 назначена по умолчанию.

Создание сервлета и конфигурирование сервера приложений

Чтобы отлаживать серверные приложения, нужно установить на компьютер программу по крайней мере одного WWW-сервера. На одном и том же компьютере может быть несколько программных WWW-серверов. Если в процессе установки среды NetBeans *(см. разд. 2.2)* сервер не был поставлен, для его добавления следует выйти из среды NetBeans, заново запустить инсталлятор NetBeans и добавить сервер GlassFish или Tomcat Apache *(см. разд. 2.2)*.

Рассмотрим процесс создания сервлета. Работающие варианты сервлетов и их исходные коды можно посмотреть, открыв пункты меню: **File/New Project.../Samples/Java Web/Servlet Examples**.

Большим достоинством среды NetBeans является то, что оригиналы всех примеров, открываемых в NetBeans, остаются в неприкосновенности — автоматически создаются копии примеров. Поэтому даже если вы внесете исправления в исходный код проекта с примером, это не повлечет изменений в новом проекте с таким же примером (рис. 11.10).

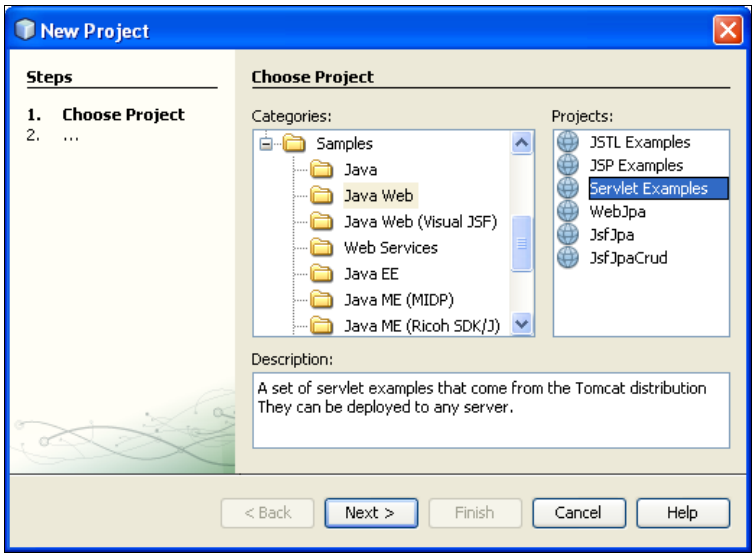


Рис. 11.10. Открытие примера с сервлетами

Пример разработан организацией The Apache Software Foundation (**http://www.apache.org/**), и его использование должно соответствовать лицензии, выставленной по адресу **http://www.apache.org/licenses/LICENSE-2.0**.

После нажатия кнопки **Next >**, ввода имени проекта и нажатия кнопки **Finish** будет создан пример с именем Servlet Examples.

После этого можно запустить приложение. Оно будет скомпилировано и автоматически размещено (deployed) на программном сервере GlassFish, после чего будет запущено в режиме имитации работы в WWW с сервером, имеющим WWW-адрес: **http://localhost** (локальный компьютер).

Внешний вид приложения показан на рис. 11.11 (из копии экрана убран длинный английский текст, чтобы было понятнее содержание примера).

В приложении приведен ряд примеров сервлетов. При нажатии на гиперссылку **Execute** (Выполнить) соответствующий сервлет выполняется, а на экране показывается сформированный им HTML-документ. Например, при нажатии на гиперссылку **Execute** около надписи **Hello World!** (первый из примеров) мы получим приветствие, представленное на рис. 11.12.

Следующий пример (нажатие на гиперссылку **Execute** около надписи **Request Info**) даст отклик, показанный на рис. 11.13.

Рассмотрим устройство первого примера, традиционно носящего название **Hello World!** (в нашем варианте его файл называется **HelloWorldExample.java** — рис. 11.14). Примерное содержимое исходного кода сервлета можно увидеть, перейдя по гиперссылке **Source** в HTML-документ с изображением исходного кода. Но это не настоящий код, а лишь HTML-документ, исходный код которого можно увидеть, сделав двойной щелчок в окне **Projects...** по узлу **Web Pages/helloworld.html**. А вот настоящий исходный код (листинг 11.19) можно просмотреть, сделав двойной щелчок в том же окне по узлу **Source Packages/<default package>/HelloWorldExample.java**. То же относится и к другим примерам.

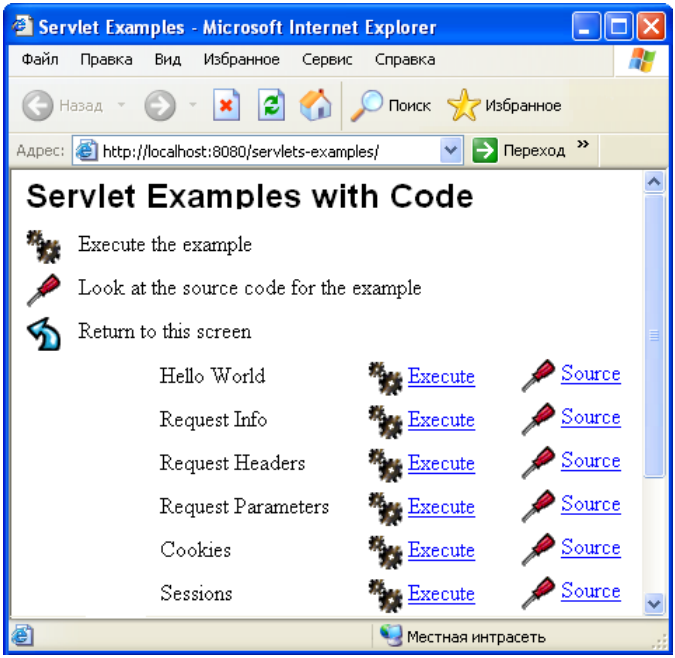


Рис. 11.11. Элементы клиентского экрана запущенного приложения с сервлетами

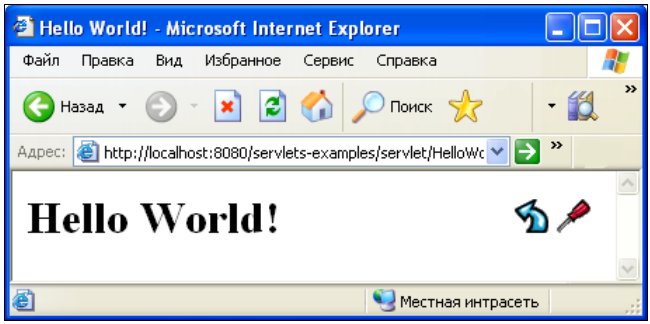


Рис. 11.12. Работа примера **Hello World!**

Request Information Example	
Method:	GET
Request URI:	/servlets-examples/servlet/RequestInfoExample
Protocol:	HTTP/1.1
Path Info:	null
Remote Address:	127.0.0.1

Рис. 11.13. Работа примера **Request Info**

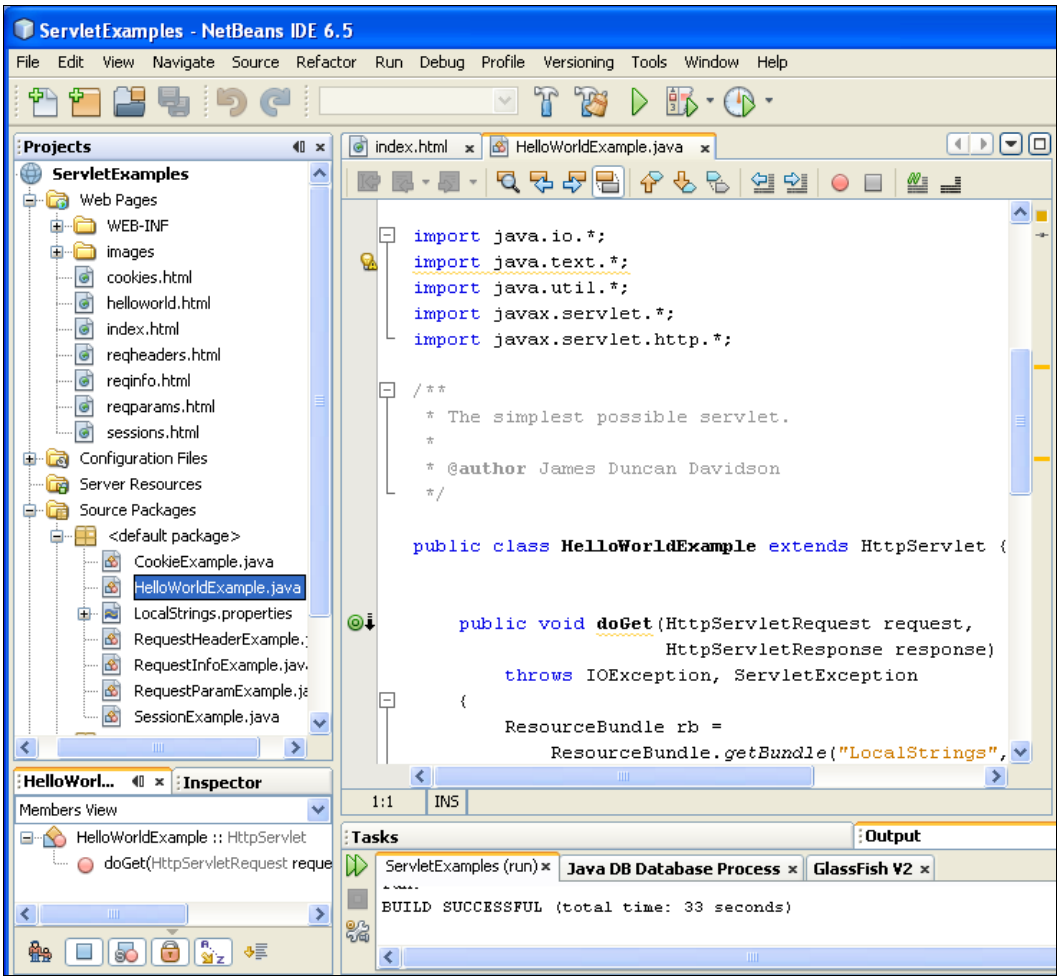


Рис. 11.14. Вид исходного кода примера **Hello World!**

Листинг 11.19. Исходный код сервлета Hello World!

```
/*
* Copyright 2004 The Apache Software Foundation
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*     http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
* See the License for the specific language governing permissions and
* limitations under the License.
*/
/* $Id: HelloWorldExample.java,v 1.3 2006/10/12 14:31:28 abadea Exp $
 *
 */

import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The simplest possible servlet.
 *
 * @author James Duncan Davidson
 */

public class HelloWorldExample extends HttpServlet {


    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        ResourceBundle rb =
            ResourceBundle.getBundle("LocalStrings",request.getLocale());
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

 String title = rb.getString("helloworld.title");

 out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");

// note that all links are created to be relative. this
// ensures that we can move the web application that this
// servlet belongs to to a different place in the url
// tree and not have any harmful side effects.

        // XXX
        // making these absolute till we work out the
        // addition of a PathInfo issue
```

```
 out.println("<a href=\"../helloworld.html\">");
        out.println("<img src=\"../images/code.gif\" height=24 " +
                  "width=24 align=right border=0 alt=\"view code\"></a>");
        out.println("<a href=\"../index.html\">");
        out.println("<img src=\"../images/return.gif\" height=24 " +
                    "width=24 align=right border=0 alt=\"return\"></a>");
        out.println("<h1>" + title + "</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Чтобы компилятор не выдавал предупреждений (они видны на рис. 11.14), автор добавил в примеры на прилагаемом DVD в нужные места директиву `@Override` и убрал лишние импорты.

Отметим особенность предупреждений и других сообщений NetBeans, помечаемых изображением слева от строки, для которой выдается сообщение: при наличии двух и более сообщений показывается пиктограмма-изображение, соответствующее первому сообщению (например, зеленый кружок для метода, переопределяющего прародительский), и справа от этой пиктограммы рисуется стрелка вниз (строка с декларацией метода `doGet` на рис. 11.14). Наведение курсора мыши на пиктограмму приводит к показу сообщения. Нажатие на стрелку приводит к отображению пиктограммы для следующего сообщения. Переход по пиктограммам циклический — после вывода последней пиктограммы опять показывается первая.

При редактировании файла SetCharacterEncodingFilter.java среда NetBeans выдает предупреждение во фрагменте, использующем переменную `encoding`. Строка

```
String encoding = selectEncoding(request);
```

помечается желтым треугольником с восклицательным знаком и лампочкой и выдается сообщение "local variable hides a field" (локальная переменная скрывает поле данных с таким же именем). Нажатие на пиктограмму приводит к неправильной рекомендации "Rename the field" (переименуйте поле данных), хотя, конечно, необходимо переименовывать переменную. Щелчок мышью в области предупреждения приводит к выделению в тексте программы всех мест, где встречается данная переменная (рис. 11.15). Причем изменение имени переменной в области ее декларации (обведена фиолетовой рамкой) сразу приводит к изменению имени этой переменной во всех местах программы.

NetBeans предоставляет удобную смену сервера, на котором запускается серверное приложение. В окне **Projects** вызовите контекстное меню щелчком правой кнопки мыши по имени проекта и выберите команду **Properties** (Свойства) (в самой нижней части меню).

В появившейся форме (рис. 11.16) в выпадающем списке **Server** производится выбор необходимого сервера для запуска приложения.

В поле **Context Path** (рис. 11.16) задается имя главной страницы сайта, которое, вообще говоря, не имеет отношения к имени приложения. Например, если мы изменим содержание этого поля для приложения 11_3_ServletExamples_2 с servlet-examples на ServletExamples, то сервлет будет запускаться с адреса **http://localhost:8080**

/ServletExamples/. А страница примера **HelloWorldExample** будет иметь адрес **http://localhost:8080/ServletExamples/servlet/HelloWorldExample** (при запуске примера на сервере Tomcat Apache адрес порта по умолчанию будет не 8080, а 8084).

```
public void doFilter(ServletRequest request, ServletRespon
                     FilterChain chain)
throws IOException, ServletException {

    // Conditionally select and set the character encoding
    if (ignore || (request.getCharacterEncoding() == null)
        String encoding = selectEncoding(request);
        if (encoding != null)
            request.setCharacterEncoding(encoding);
    }

// Pass control on to the next filter
    chain.doFilter(request, response);

}
```

Рис. 11.15. Переименование переменной, перекрывающей видимость поля данных

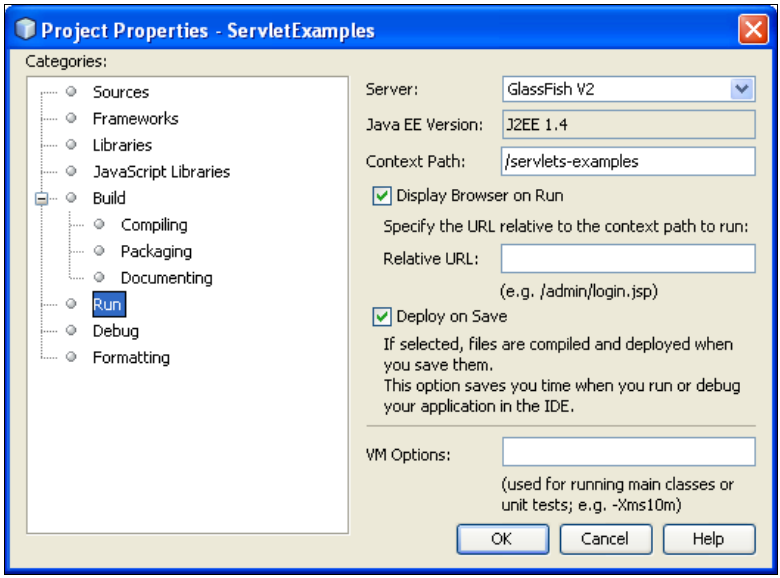


Рис. 11.16. Смена сервера для запуска приложения

Отметим, что серверы Tomcat Apache и GlassFish v3 стартуют гораздо быстрее, чем GlassFish v2, и запуск приложений из среды NetBeans на этих серверах также происходит быстрее. Поэтому может оказаться удобнее отлаживать серверные приложения с использованием Tomcat Apache или GlassFish v3. Тем более что по опыту автора сервер GlassFish v2 при работе с NetBeans оказался гораздо менее устойчивым.

Еще одно замечание: если при попытке повторного запуска сервлета под GlassFish v2 компилятор выдает ошибку

```
Deployment error,
```

то не спешите перезагружать среду NetBeans или даже компьютер. Сначала проверьте, не забыли ли вы закрыть окно работающего сервлета. Если нет — нужно закрыть проект, снова его открыть через меню **Open Recent Project** (Открыть недавний проект), сделать его главным проектом и запустить на выполнение. Простое повторное открытие проекта не помогает. После смены сервера с последующим возвратом к предыдущему серверу в ряде случаев возникают и более серьезные проблемы с запуском серверных приложений. В этом случае следует выполнить команду **Clean and build**, а если и это не поможет — закрыть среду NetBeans (при этом все запущенные серверы автоматически остановятся) и заново ее запустить.

Пункт меню **Tools/Servers** среды NetBeans позволяет увидеть установленные в системе программные серверы и задать ряд их параметров. Например, можно сменить пароль администратора, а для Apache Tomcat — задать номер порта, с которым работает сервер, платформу Java и ряд других параметров.

Локализация сервлетов — работа с языковыми форматами

Как можно заметить, пример, показанный в листинге 11.19, похож на приведенный в листинге 11.17, но хотя и претендует на то, что он простейший, усложнен ненужными комментариями. Кроме того, в нем отсутствуют строки, обеспечивающие поддержку национальных языков:

- команды задания кодировки UTF-8 при формировании строк:

```
response.setCharacterEncoding("UTF-8");
```

- команды задания кодировки UTF-8 в формируемом HTML-документе:

```
out.println("<meta http-equiv=\"content-type\"
             content=\"text/html; charset=UTF-8\">");
```

Однако здесь имеется одна новая полезная особенность — оператор:

```
String title = rb.getString("helloworld.title");
```

посредством которого можно обеспечить поддержку автоматической локализации приложения (показа элементов программы на нужном языке).

В окне **Projects** в пакете по умолчанию сразу после файла HelloWorldExample.java находится узел **LocalStrings.properties**. В нем показаны имена:

- **default language** (язык по умолчанию);
- **en** — английский;
- **es** — испанский;
- **fr** — французский.

В соответствующих этим узлам файлах содержатся пары с именем величины и ее значением:

```
Имя1=значение1
Имя2=значение2
...
```

Содержимое файла, соответствующего узлу **default language**, приведено в листинге 11.20. Полный текст листинга не приводится, окончание файла заменено многоточием.

Листинг 11.20. Содержимое файла свойств для узла *default language* (языковый формат по умолчанию)

```
helloworld.title=Hello World!

requestinfo.title=Request Information Example
requestinfo.label.method=Method:
requestinfo.label.requesturi=Request URI:
requestinfo.label.protocol=Protocol:
requestinfo.label.pathinfo=Path Info:
requestinfo.label.remoteaddr=Remote Address:
requestheader.title=Request Header Example

requestparams.title=Request Parameters Example
requestparams.params-in-req=Parameters in this request:
requestparams.no-params=No Parameters, Please enter some
requestparams.firstname=First Name:
requestparams.lastname=Last Name:
...
```

Для каждого национального языка международными стандартами предусмотрен свой алфавит и набор слов для обозначения тех или иных понятий. Принято говорить о том, что задан locale (локэйл — языковый формат). Обозначение языкового формата состоит из кода языка и кода страны, разделенных символом подчеркивания. Например, английский языковый формат в варианте США обозначается как en_US, а для Великобритании — как en_GB. Испанский языковый формат для Испании обозначается как es_ES, а для Мексики — как es_MX.

Русский языковый формат для России обозначается как ru_RU. Полное определение языкового формата включает еще и кодировку, которая указывается через точку. Пример полного определения: ru_RU.KOI8-R.

В зависимости от текущего языкового формата, установленного на компьютере (для Windows — в разделе **Язык и региональные стандарты**), приложение берет значения из того или иного файла. Если языковый формат не совпадает ни с одним из предусмотренных, то используется набор из узла **default language** (рис. 11.17). В большинстве случаев его содержимое делают совпадающим с файлом для английского языкового формата. Именно из этого файла у нас бралось значение, возвращаемое функцией `rb.getString("helloworld.title")` (см. листинг 11.19). Для французского языкового формата набор значений будет уже другим (листинг 11.21).

Листинг 11.21. Содержимое файла свойств для узла *fr* (французский языковый формат)

```
helloworld.title=Salut le Monde!

requestinfo.title=Exemple d''information sur la requête
requestinfo.label.method=Méthode:
```

```
requestinfo.label.requesturi=URI de requête:
requestinfo.label.protocol=Protocole:
requestinfo.label.pathinfo=Info de chemin:
requestinfo.label.remoteaddr=Adresse distante:

requestheader.title=Exemple d''information sur les entête de requête

requestparams.title=Exemple de requête avec paramêtres
requestparams.params-in-req=Paramêtres dans la requête:
requestparams.no-params=Pas de paramêtre, merci dans saisir quelqu'uns
requestparams.firstname=Prénom:
requestparams.lastname=Nom:
...
```

Поэтому на французском компьютере на экран вместо `Hello World!` выведется `Salut le Monde!`.

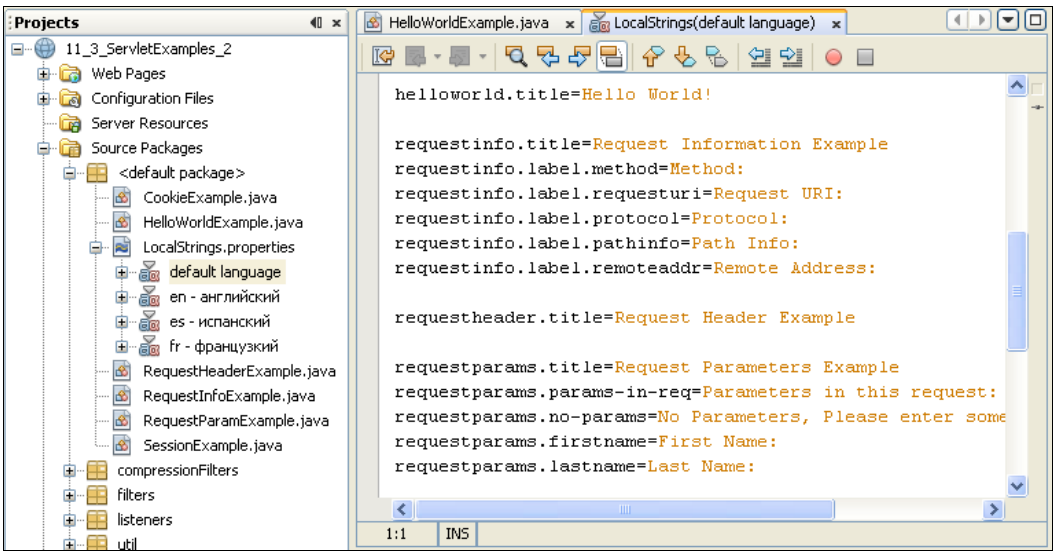


Рис. 11.17. Файлы LocalStrings.properties — набор **default language**

Правда, без установки кодировки UTF-8 как в Java, так и в заголовке HTML-документа при попытке локализации для большинства других языков ничего хорошего не выйдет. Например, если в качестве **default language** мы выберем русский язык и напишем все сообщения по-русски, то во время работы приложения вместо русских букв на экране отобразятся знаки вопросов.

Проект 11_3_ServletsExamples_2 доработан — в него для примера **HelloWorldExample** добавлены уже известные нам операторы, необходимые для правильной локализации приложения:

```
response.setCharacterEncoding("UTF-8");
out.println("<meta http-equiv=\"content-type\"
                  content=\"text/html; charset=UTF-8\">");
```

Имеется возможность расширить в примере число поддерживаемых языковых форматов. Например, добавим русский язык.

Для этого в дереве проектов щелкнем правой кнопкой мыши по пункту **LocalStrings.properties** и в появившемся всплывающем меню выберем пункт **Add Locale...** (Добавить языковый формат) (рис. 11.18).

Появится экранная форма, в которой в разделе **Predefined Locales** (Предопределенные языковые форматы) следует выбрать **ru_RU** и нажать кнопку **OK** (рис. 11.19).

После этого в списке **LocalStrings.properties** будут видны следующие узлы:

- **default language**
- **en — английский**
- **es — испанский**
- **fr — французский**
- **ru_RU — русский (Россия)**

Содержимое файла LocalStrings.properties/ru_RU — русский (Россия) будет скопировано с файла LocalStrings.properties/default language. Его можно двойным щелчком мыши открыть в окне редактора и отредактировать, вставив в правых частях присваиваний русские слова. Например, если мы сделаем замены, показанные в листинге 11.22, то в результате выполнения программы увидим в примере сервлета **Hello World!** то, что представлено на рис. 11.20.

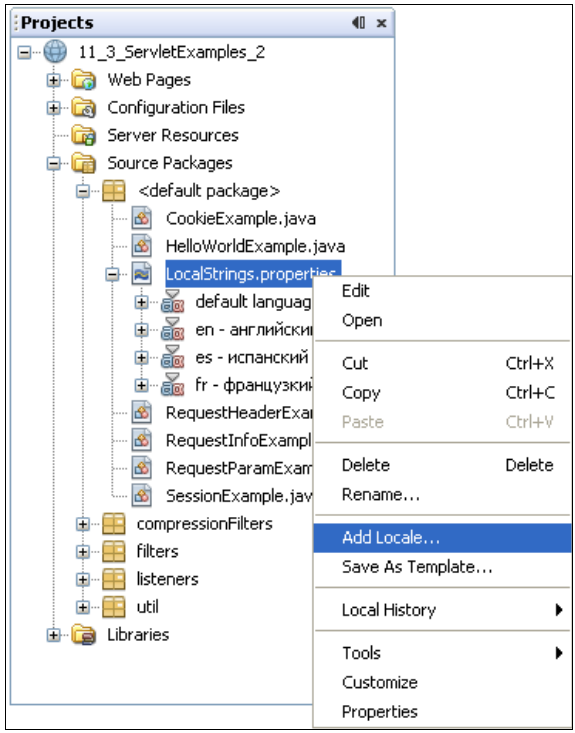


Рис. 11.18. Добавление поддержки приложением русского языкового формата, шаг 1

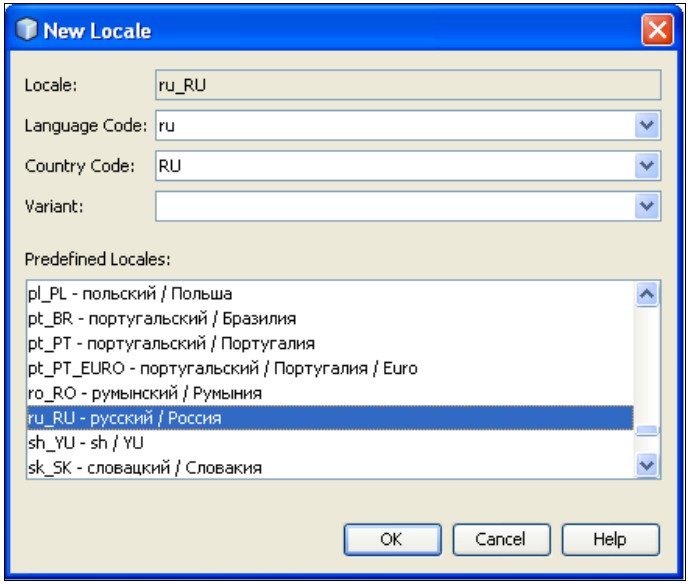


Рис. 11.19. Добавление поддержки приложением русского языкового формата, шаг 2

Листинг 11.22. Локализация примера HelloWorldExample

```
helloworld.title=Здравствуй, мир!
requestinfo.title=Пример информационного запроса
requestinfo.label.method=Метод:
requestinfo.label.requesturi=URI запроса:
requestinfo.label.protocol=Протокол:
requestinfo.label.pathinfo=Информация о пути:
requestinfo.label.remoteaddr=Удаленный адрес:
```

Рис. 11.20. Результат работы правильно локализованного примера **Hello World!**

Но в примере **Request Info** результат будет совсем не таким, как нам хочется (рис. 11.21).

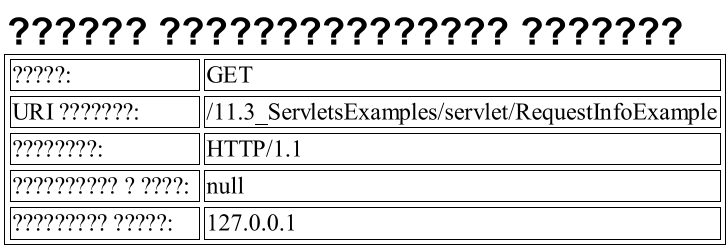


Рис. 11.21. Работа неправильно локализованного примера **Request Info**

Ведь мы ввели поддержку UTF-8 (а значит, и показ русских символов) в генерируемый сервером документ только в файле HelloWorldExample.java, а файл RequestInfoExample.java остался без изменений. Возможность внести в этот файл соответствующие усовершенствования предоставляется читателю.

Подробный анализ того, как в Java осуществляется работа с ресурсами приложения и локализацией, будет сделан в *главе 13*.

11.4. Технология JSP — Java Server Pages

Один из важнейших видов серверных программ — приложения на основе технологии JSP — Java Server Pages. Несмотря на то, что она опирается на использование сервлетов, JSP является самостоятельной технологией. Идеология JSP базируется на внедрении в HTML-документы специальных конструкций, позволяющих со стороны сервера программно обрабатывать данные и выводить в соответствующее место документа результат.

В NetBeans приведены примеры JSP Example и JSTL Example (см. рис. 11.10). Порядок примеров в мастере создания приложений прямо противоположный рассматриваемому нами — сначала предлагается применять JSTL, как наиболее развитое и простое средство, затем — JSP, как средство более низкого уровня, и только потом Servlet, как средство еще более низкого уровня. Мы придерживаемся обратного порядка, т. к. JSP использует сервлеты, а JSTL является надстройкой над JSP. Поэтому изучать эти технологии лучше в выбранном нами порядке.

Пример JSP Example иллюстрирует базовые конструкции JSP, его можно просмотреть, создав проект посредством команды **File/New Project.../Samples/Java Web/JSP Example**.

Замечание

В заготовке, на основе которой NetBeans создает пример, имеется ошибка. По какой-то причине файл Clock2.java попал в папку Web Pages/WEB-INF/jsp/applet. В сгенерированном примере его необходимо переместить в пакет по умолчанию в Source Packages.

Второй пример (JSTL Example) — это надстройка над JSP, специальный набор тегов JSP, созданный группой экспертов для облегчения разработки серверных приложений. Пример можно просмотреть, создав проект посредством команды **File/New Project.../Samples/Java Web/JSTL Example**. В прежних версиях NetBeans после создания каждого серверного примера требовалось назначать ему сервер, в NetBeans 6.5 такое назначение делается автоматически.

Оба этих примера тоже разработаны организацией The Apache Software Foundation (**http://www.apache.org/**), их использование также должно соответствовать лицензии **http://www.apache.org/licenses/LICENSE-2.0**.

Рассмотрим подробнее пример JSP Example (в работе он прост, а вот создать самостоятельно что-либо подобное будет совсем не просто).

Как и в предыдущем случае, после запуска приложения при нажатии на гиперссылку **Execute** в появившемся окне выполняется соответствующий пример — в данном случае запускается страница JSP (рис. 11.22). А при нажатии гиперссылки **Source** показывает-

ся HTML-страница с примерным видом исходного кода. Напомним, что это не сам исходный код, а просто HTML-страница, так что показанный в ней код может несколько отличаться от истинного.

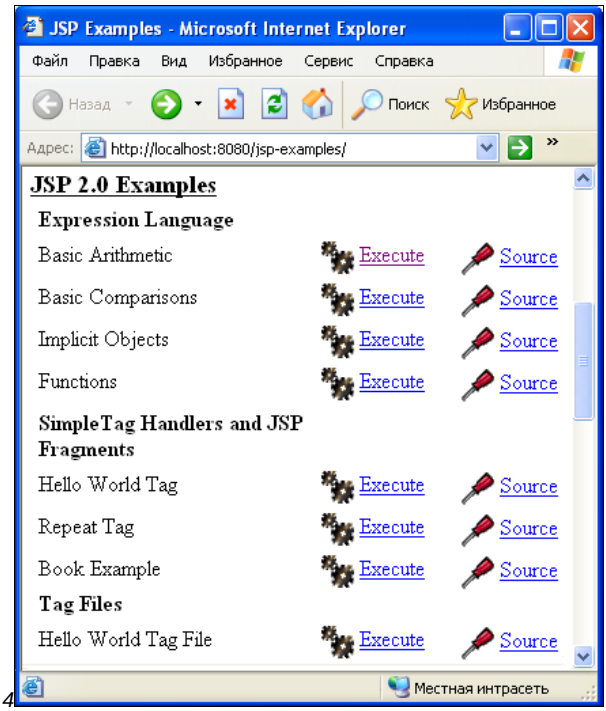


Рис. 11.22. Первая страница запущенного примера JSP

Страницы JSP представляют собой обычные HTML-документы, но имеющие расширение имени файла .jsp, а не .html или .htm. Это — заготовка HTML-документа, который будет показан пользователю-клиенту. При создании клиентского HTML-документа в этой заготовке выражения в фигурных скобках после знака доллара вычисляются, а в HTML-документ подставляется строковое представление результата.

Например, выражение вида **${1 + 2}** выдаст в соответствующее место документа символ **3**. Последовательность **\$** означает, что выражение **${...}** не вычисляется, а рассматривается как строка (рис. 11.23). Данный метод использования JSP весьма прост и не требует высокой квалификации и специальных знаний, чего нельзя сказать о применении тегов JSP — оно достаточно запутанно (не говоря уже о задании пользовательских тегов).

В JSP имеется ряд встроенных объектов Java, которым соответствуют теги с такими же именами:

- `request` — запрос. Тип объекта — класс, реализующий интерфейс `javax.servlet.http.HttpServletRequest`;
- `response` — ответ на запрос. Тип объекта — класс, реализующий интерфейс `javax.servlet.http.HttpServletResponce`;

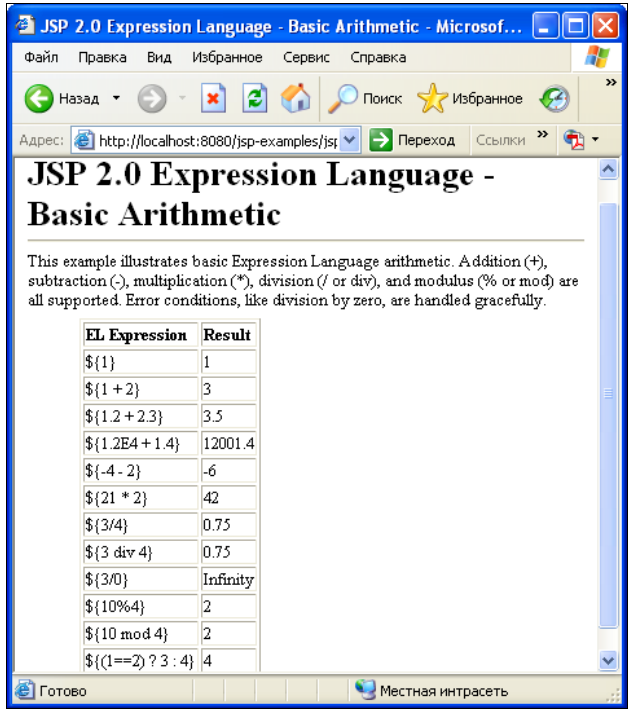


Рис. 11.23. Первый пример JSP-вычисления выражений

- `pageContext` — контекст страницы JSP. Обеспечивает доступ к пространству имен и элементам страницы (тегам, атрибутам). Тип объекта — класс `javax.servlet.jsp.PageContext`;
- `session` — сессия (сеанс связи клиента с сервером). Тип объекта — класс, реализующий интерфейс `javax.servlet.http.HttpSession`;
- `application` — приложение. Тип объекта — класс, реализующий интерфейс `javax.servlet.ServletContext`;
- `out` — выходной поток. Тип объекта — класс `javax.servlet.jsp.JspWriter`;
- `config` — объект конфигурации сервлета для текущей страницы. Тип объекта — класс, реализующий интерфейс `javax.servlet.ServletConfig`;
- `page` — объект, обрабатывающий запрос для данной страницы. Тип объекта — класс `Object`.

Данные объекты применяются в тегах JSP вида:

```
<%@ имяОбъекта параметр1=значение1 параметр2=значение2 ... %>
```

Пример:

```
<%@ page session=true import="java.util.*" %>
```

Имеется возможность задания сценариев (интерпретируемого кода) с помощью специальных определений вида:

```
<%@ код %>
```

где `код` — сценарий Java для работы с документом или вычислений.

Пример:

```
<%
for(int i=0; i<table.getEntries().getRows(); i++) {
   cal.Entry entr = table.getEntries().getEntry(i);
%>
```

В частности, разрешается задавать собственные теги вида:

```
<%@ имяБиблиотеки prefix="имяОбъекта" uri="путь к библиотеке" %>
```

после чего допустимы теги вида:

```
< имяОбъекта:оператор>
```

Так делается в примере **Hello World Tag** (файл Web Pages/ jsp2/simpletag/hello.jsp).

Исходный код наиболее важных фрагментов JSP-страницы этого примера с JSP-выводом следующий:

задание тега `mytag` с помощью декларации:

```
<%@ taglib prefix="mytag" uri="/WEB-INF/jsp2/jsp2-example-taglib.tld" %>
```

и его использование:

```
    <mytag:helloWorld/>
```

Листинг 11.23 иллюстрирует соответствующий класс Java.

Листинг 11.23. Класс `HelloWorldSimpleTag`

```
 public class HelloWorldSimpleTag extends SimpleTagSupport {
   @Override
   public void doTag() throws JspException, IOException {
       getJspContext().getOut().write( "Hello, world!" );
    }
}
```

Это сервлет-обработчик (handler) вызова тега `mytag`. Его текст находится в файле Source Packages/jsp2.examples.simpletag/HelloWorldSimpleTag.java, а полное имя класса `jsp2.examples.simpletag.HelloWorldSimpleTag`.

Обсудим некоторые особенности приведенных фрагментов кода.

В оригинальном примере JSP Example аннотация `@Override` перед заданием метода `doTag()` отсутствует, но лучше ее поставить — сразу станет понятно, что данный метод переопределяет родительский.

Класс `SimpleTagSupport` (его полное имя `javax.servlet.jsp.tagext.SimpleTagSupport`) обеспечивает поддержку новых тегов JSP.

Пользовательский объект `mytag`, задаваемый как `prefix=" mytag "`, вызывает тег `helloWorld` из библиотеки jsp2-example-taglib.tld .

В этой библиотеке класс `HelloWorldSimpleTag` (полное имя `jsp2.examples.simpletag.HelloWorldSimpleTag`) для задания тега `helloWorld` определяется с помощью XML-фрагмента, приведенного в листинге 11.24.

Листинг 11.24. XML-фрагмент задания тега `helloWorld`

```
<tag>
<description>Outputs Hello, World</description>
    <name>helloWorld</name>
<tag-class>jsp2.examples.simpletag.HelloWorldSimpleTag</tag-class>
<body-content>empty</body-content>
</tag>
```

Как видим, задать свои теги JSP можно, но не очень просто.

Прочтя данный раздел, не следует пугаться — собственные теги JSP вам, вероятнее всего, создавать не придется, поскольку имеются готовые библиотеки тегов, которые постоянно усовершенствуются.

11.5. Технология JSTL — JSP Standard Tag Library

Для облегчения работы с JSP сообщество программистов разработало библиотеку JSTL — JSP Standard Tag Library (библиотека стандартных тегов JSP).

Проект, иллюстрирующий конструкции JSTL, можно создать на основе примера JSTL Example. Для этого следует выбрать **File/New Project.../Samples/Java Web/JSTL Example**.

Мы уже знаем, что в JSP возможны теги вида `<имяОбъекта:оператор>`.

Так делается и в примере использования оператора `out` в JSTL Example (листинг 11.25).

Листинг 11.25. Пример использования JSTL-тега `out`

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
  <title>JSTL: Expression Language Support -- Expr Example</title>
</head>
<body bgcolor="#FFFFFF">

<h3><c:out></h3>

<table border="1">
  <c:forEach var="customer" items="${customers}">
    <tr>
      <td><c:out value="${customer.lastName}"/></td>
      <td><c:out value="${customer.phoneHome}" default="no home phone specified"/></td>
      <td>
        <c:out value="${customer.phoneCell}" escapeXml="false">
          <font color="red">no cell phone specified</font>
        </c:out>
      </td>
```

```
    </tr>
  </c:forEach>
</table>
```

В примере есть пользовательский объект `c` (сокращение от customer — покупатель), задаваемый как `prefix="c"`, и оператор `out`, заданный в библиотеке taglib по интернет-адресу **http://java.sun.com/jsp/jstl/core**. В примере этот объект задается тегом `taglib` использования библиотеки ядра JSTL:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Отметим, что все необходимые библиотеки JSTL (архивы с пакетами, содержащими нужные классы) уже добавлены в проект. Для примера с объектом `out` это пакет org.apache.taglibs.standard.tag.el.core. Его можно увидеть в окне **Projects** в разделе **Libraries**, архив JSTL 1.1 — standard.jar.

После запуска данного приложения в появившемся документе в списке **Examples** это будет первый пример — **General-Purpose Tags**. При переходе по данной гиперссылке получаем пример, представленный на рис. 11.24.

После нажатия на "шестеренки" получаем результат работы программы (рис. 11.25).

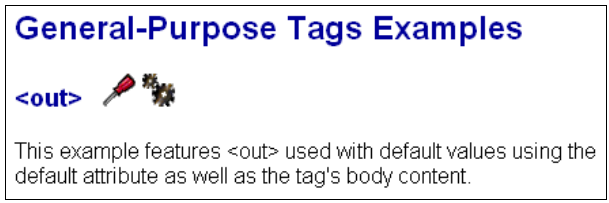


Рис. 11.24. Первый пример — **General-Purpose Tags**

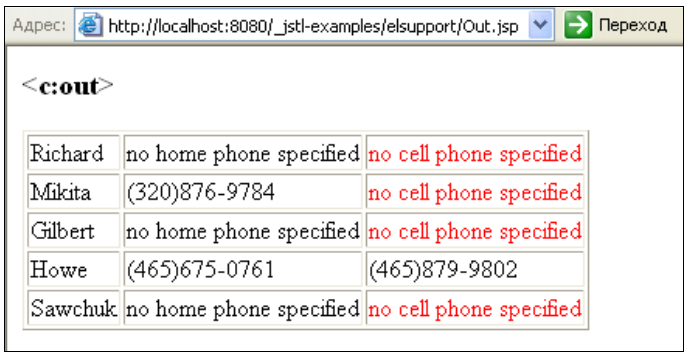


Рис. 11.25. Пример использования оператора `c:out`

В примере (см. листинг 11.25) используется объект `customer` (покупатель), заданный в файле Customer.java, расположенном в пакете org.apache.taglibs.standard.examples.beans, т. е. в самом первом в **Source Packages** пакете.

В данном примере есть объект `customers` (покупатели), заданный в файле Customers.java, расположенном в том же пакете.

В классе `customer` заданы поля `lastName`, `phoneHome`, `phoneCell` и др., а также ряд методов, которые тоже можно вызывать в сценарии. С помощью оператора `forEach` (заданного аналогично оператору `out`) осуществляется перебор всех объектов `customer`, агрегированных в объект `customers` (список покупателей). А тег `<c:out>` выводит необходимую информацию в документ.

Подведем некоторые итоги.

В JSP и JSTL предусмотрено много программных средств, поддерживающих обработку со стороны сервера клиентских запросов и генерацию сервером HTML-документов, — но это тема для отдельной книги. В данном учебном пособии имеющиеся технологии затронуты совсем немного — просто дана информация об их существовании и некоторых возможностях.

Точно так же для программирования в локальных и глобальных компьютерных сетях в пакете java.net имеется немалое количество средств разного уровня, описание которых требует отдельной книги.

Это и Web-адресация (классы `URL`, `HttpURLConnection`, `URI`, `JarURLConnection`, `URLClassLoader`), и IP-адресация (классы `InetAddress`, `InetAddress4`, `InetAddress6`, `NetworkInterface`), и управление соединениями через сокеты (классы `Socket`, `SocketAddress`, `InetSocketAddress`, `ServerSocket`, `SocketPermission`). Классы `NetPermission`, `Authentificator` и `PasswordAuthentification` обеспечивают поддержку авторизации (запрос и обработку имени и пароля).

Не менее важная и объемная тема — сетевое программирование баз данных. В Java есть очень много уже готовых решений в данной области, а NetBeans предоставляет заготовки и примеры, а также различные инструменты управления и средства отладки.

В данной книге совсем не затронута тема создания модулей EJB (Enterprise Java Beans) и разработки приложений Java EE (Enterprise Applications) — а ведь именно это основная сфера применения Java-технологий.

Подведем итог: программирование на Java в компьютерных сетях — это широкая область деятельности, имеющая много наработок и технологий, которая нуждается в значительном количестве разработчиков для освоения и использования этих технологий, а в дальнейшем и для их усовершенствования.

Так что желающих освоить даже основы сетевых возможностей Java ждет весьма длительная и интересная работа.

Краткие итоги по главе

- HTML — язык разметки гипертекста. Гипертекст (сверхтекст) — это текст, содержащий дополнительные возможности, в частности гиперссылки. Документ, написанный на языке HTML, представляет собой текстовый файл, содержащий собственно текст, несущий информацию пользователю, и теги разметки (murkup tags).
- Теги — это определенные последовательности символов, заключенные между символами < и >. Они предназначены для форматирования и разметки документа. Теги

бывают парные (контейнеры) и непарные. Действие парного тега начинается с открывающего тега и заканчивается при встрече соответствующего ему закрывающего, признаком которого является символ `/`.

- При ссылке на документ, находящийся на другом сервере, необходимо указывать адрес (URL — Uniform Recourses Location) этого документа: сетевой адрес сервера и путь к этому документу на сервере. Если документ находится на том же сервере, но в другой папке, достаточно указать только путь к этой папке. Гипертекстовая ссылка имеет вид:

  `<a href="имя_файла">текст_ссылки</a>`.

- Для создания обычного апплета требуется задать класс-наследник класса `java.applet.Applet`, а для апплетов, использующих компоненты `Swing` и/или библиотеку Sun JFC (Java Foundation Classes), необходимо задать класс-наследник класса `javax.swing.JApplet`.

- В классе апплета требуется переопределить ряд методов:

  `init`, `start`, `update`, `paint`, `getAppletInfo`, `stop`, `destroy`.

- Директива `@Override` — необязательная аннотация, поясняющая человеку, читающему текст программы, что данный метод переопределяет родительский. Одновременно является метаданным для компилятора. Если метод аннотирован символом `@Override`, но на самом деле не переопределяет родительский метод (например, имеет другую сигнатуру), то компилятор на основании этой аннотации выдает сообщение об ошибке.

- Сервлеты — это приложения Java, запускаемые со стороны сервера. Они имеют возможность доступа к файловой системе и другим ресурсам сервера через набор управляющих конструкций, предопределенных в рамках пакета javax.servlet и технологии JSP.

- Технология JSP заключается в наличии дополнительных конструкций в HTML- или XML-документах, которые позволяют осуществлять вызовы сценариев (скриптов), написанных на языке Java. При этом удается очень просто и удобно обрабатывать данные или элементы документа и внедрять в него результаты обработки.

Типичные ошибки

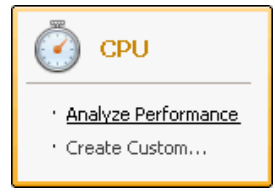
- Пытаются запустить HTML-документ на просмотр из среды NetBeans с помощью кнопки или пункта меню **Run Main Project**.

- Не учитывают, что работоспособность апплета зависит не только от его кода, но и от того, с помощью каких тегов и их атрибутов он встроен в HTML-документ, а также от того, с помощью каких средств осуществляется просмотр (AppletViewer или браузер; вид браузера; вызов браузера из NetBeans или вызов средствами операционной системы).

- При работе с GlassFish v2 забывают закрыть окно работающего сервлета при новом запуске сервлета (при работе с GlassFish v3 или Tomcat Apache эта проблема отсутствует).

Задания

- Написать апплет, рисующий эллипс, и HTML-документ, в который включен данный апплет. Проверить показ этого HTML-документа в AppletViewer и браузере.
- Самостоятельно создать сервлет на основе примера **HelloWorldExample**. Изучить работу и исходный код этого примера. Видоизменить файл HelloWorldExample.java так, чтобы обеспечить вывод документа с данными: фамилией, именем, отчеством на русском и английском языках (в зависимости от установленного на компьютере языкового формата). В конце вывести дату генерации документа. Для этого использовать объект типа `Date` или `GregorianCalendar` (самостоятельно придумайте, как преобразовать его в строку).



ГЛАВА 12

Вложенные классы

12.1. Виды вложенных классов

Начиная с JDK 1.2, в язык Java были введены новые средства для работы с классами, позволяющие реализовать дополнительные возможности инкапсуляции и композиции — так называемые вложенные (nested) классы.

Они делятся на две категории:

- статические (static) вложенные классы и интерфейсы — используются для задания совершенно самостоятельных классов и интерфейсов внутри классов. Статические вложенные классы должны при задании иметь модификатор `static`. Вложенные интерфейсы всегда считаются имеющими модификатор `static`. Имя класса верхнего уровня используется в качестве квалификатора в пространстве имен, во всем остальном они ведут себя как обычные классы;
- нестатические (non-static), или, что то же, внутренние (inner) классы — служат для создания экземпляров, принадлежащих экземплярам класса верхнего уровня (т. е. их экземпляры не могут существовать вне объектов верхнего уровня). Внутренний класс можно задавать только для создания его экземпляров, не допускается создания методов класса или переменных внутреннего класса. Экземпляр внутреннего класса мы будем называть *внутренним объектом* (inner object), а экземпляр внешнего класса — *внешним объектом* (outer object). Внутренние классы делятся на три разновидности:
  - внутренние классы общего вида (будем называть их обычными внутренними классами), заданные внутри определения класса (на уровне задания полей данных и методов). Как правило, такие классы реализуют агрегацию, т. е. вариант композиции, в котором встроенные объекты очень тесно связаны с внешним объектом и не могут без него существовать;
  - локальные (local) внутренние классы — задаются внутри блоков программного кода в блоках инициализации или методах. Они носят вспомогательный характер, область видимости и жизни экземпляров этих классов ограничивается соответствующим блоком программного кода. Классы такого рода следует использовать с большой осторожностью, т. к. они делают программный код менее читаемым, увеличивают количество создаваемого в памяти "мусора" и потребляют дополнительные ресурсы процессора. С другой стороны, локальные классы по-

зволяют скрыть особенности реализации методов, не вынося на всеобщее обозрение код этих классов;

- анонимные (anonimous) (другое название — безымянные) внутренние классы — совмещают декларацию, реализацию и вызов. Не имеют ни имени, ни собственного конструктора (вместо него вызывается прародительский конструктор со специальным синтаксисом вызова). Анонимные классы обычно применяют в обработчиках событий.

12.2. Статические (static) вложенные классы и интерфейсы

Статические вложенные классы обычно используются для объединения во внешнем классе нескольких связанных друг с другом классов — чтобы обойтись без создания дополнительного пакета. Листинг 12.1 иллюстрирует задание статического вложенного класса во внешнем классе.

Листинг 12.1. Задание статического вложенного класса

```
class ИмяВнешнегоКласса{
     тело внешнего класса

     static class ИмяВложенногоКласса{
       тело вложенного класса
     }

     продолжение тела внешнего класса
}
```

Экземпляры статического вложенного класса, а также методы класса и поля класса получают в имени квалификатор — имя класса верхнего уровня.

Например, доступ к полю идет так:

```
ИмяВнешнегоКласса.ИмяВложенногоКласса.имяПоля,
```

а обращение к методу класса так:

```
ИмяВнешнегоКласса.ИмяВложенногоКласса.имяМетода(список параметров).
```

Пусть у нас имя внешнего класса `C1`, а вложенного — `C_nested`.

Тогда создание экземпляра вложенного класса может идти, например, так:

```
C1.C_nested obj=new C1.C_nested();
```

Особенностью использования статических вложенных классов является то, что во внешнем классе могут быть поля, имеющие тип вложенного класса. При этом для данного случая квалификация именем внешнего класса не нужна. Отметим, что в этом случае применяется то же правило, что и при доступе к обычным полям или методам, заданным в классе.

Пример приведен в листинге 12.2.

Листинг 12.2. Пример использования статического вложенного класса

```
class C1{
     private C_nested obj1;

     static class C_nested {
       тело вложенного класса
     }

     C_nested getNested(){
      return obj1;
     }
}
```

При компиляции для статических вложенных классов создаются самостоятельные классы `.class`, имеющие имя:

```
имяВнешнегоКласса$имяВложенногоКласса.class
```

Точно такое же имя возвращается методом

```
объектВложенногоКласса.getClass().getName()
```

или входит в строку, возвращаемую методом

```
объектВложенногоКласса.toString().
```

А вот

```
объектВложенногоКласса.getClass().getCanonicalName()
```

возвращает имя вложенного класса через точку.

Задание вложенного интерфейса аналогично заданию вложенного класса (листинг 12.3).

Листинг 12.3. Задание вложенного интерфейса (считается статическим)

```
class ИмяВнешнегоКласса{
     тело внешнего класса

     interface ИмяВложенногоИнтерфейса{
       объявление констант и заголовков методов
     }

     продолжение тела внешнего класса
}
```

Вложенные интерфейсы всегда считаются имеющими модификатор `static`.

Реализовать вложенный интерфейс можно в постороннем классе, при этом имя интерфейса квалифицируется именем внешнего класса. Если же реализация идет в самом внешнем классе, то квалификации именем этого класса не требуется.

12.3. Внутренние (inner) классы

Внутренний вложенный класс общего вида (обычный внутренний класс) задается так же, как статический вложенный, но только без модификатора `static` перед именем этого класса (листинг 12.4).

Листинг 12.4. Задание внутреннего вложенного класса общего вида

```
class ИмяВнешнегоКласса{
     тело внешнего класса

     class ИмяВнутреннегоКласса{
       тело внутреннего класса
     }

     продолжение тела внешнего класса
}
```

Для обычных внутренних классов экземпляры создаются через *имя внешнего объекта*, что принципиально отличает их от обычных не вложенных и статических вложенных классов.

Синтаксис таков:

сначала идет создание внешнего объекта:

```
ИмяВнешнегоКласса имяОбъекта = new ИмяВнешнегоКласса(параметры);
```

затем создается нужное число экземпляров внутреннего класса:

```
ИмяВнешнегоКласса.ИмяВнутреннегоКласса имя1 =
                 имяОбъекта.new ИмяВнутреннегоКласса(параметры);

ИмяВнешнегоКласса.ИмяВнутреннегоКласса имя2 =
                 имяОбъекта.new ИмяВнутреннегоКласса(параметры);
```

и т. д.

Достаточно часто из внутреннего объекта необходимо обратиться к внешнему. Такое обращение идет через имя внешнего класса и ссылку `this` на текущий объект:

```
ИмяВнешнегоКласса.this
```

Это ссылка на внешний объект (его естественно назвать родительским объектом).

А доступ к полю или методу внешнего объекта в этом случае, естественно, идет так:

```
ИмяВнешнегоКласса.this.имяПоля
ИмяВнешнегоКласса.this.имяМетода(список параметров).
```

К сожалению, в Java, в отличие от языка JavaScript, нет зарезервированного слова `parent` для обращения к родительскому объекту. Будем надеяться, что в дальнейшем в Java появится этот гораздо более наглядный и удобный способ обращения к родителю.

Пример работы с внутренними классами приведен в листинге 12.5.

Листинг 12.5. Пример работы с внутренними классами

```
package java_gui_example;
public class OuterClass {
    int a=5;
    public OuterClass() {
    }

    public class InnerClass{
        int x=1,y=1;

        public class InnerClass2 {
            int z=0;
            InnerClass2(){
               System.out.println("InnerClass2 object created");
            }

           void printParentClassNames(){
               System.out.println("InnerClass.this.x="+
                                                    InnerClass.this.x);
               System.out.println("OuterClass.this.a="+
                                                    OuterClass.this.a);
           }
        }

    }

    InnerClass inner1;
    InnerClass.InnerClass2 inner2;

    public void createInner() {
        inner1=this.new InnerClass();
        inner2=inner1.new InnerClass2();
        System.out.println("inner1 name="+inner1.getClass().getName());
        System.out.println("inner1 canonical name="+
                           inner1.getClass().getCanonicalName());
    }
}
```

Листинг 12.5 иллюстрирует пример, в котором задан класс `OuterClass` (внешний класс). В него вложен класс `InnerClass` (внутренний класс).

Если в приложении задать переменную типа `OuterClass` и создать соответствующий объект:

```
OuterClass outer1=new OuterClass();
```

то после этого можно создать внутренние объекты:

```
outer1.createInner();
```

Доступ к внешним объектам иллюстрируется при вызове метода:

```
outer1.inner2.printParentClassNames();
```

Заметим, что при создании внутреннего объекта в произвольном классе, а не в реализации класса `OuterClass`, вместо вызовов

```
InnerClass  inner1=this.new InnerClass();
```

и

```
InnerClass.InnerClass2 inner2= inner1.new InnerClass2();
```

придется написать:

```
OuterClass.InnerClass  inner3=outer1.new InnerClass();
OuterClass.InnerClass.InnerClass2 inner4=inner3.new InnerClass2();
```

Конструктор `InnerClass()` вызывается при этом без квалификации именем внешнего класса — попытка использования конструктора в виде

```
OuterClass.InnerClass.InnerClass2 inner4=
               inner3.new OuterClass.InnerClass2();
```

вызовет ошибку компиляции!

Вариант применения внутренних классов, в котором внутренний объект создается и используется вне кода внешнего класса, вообще говоря, не может считаться нормальным, поскольку он нарушает принцип инкапсуляции программного кода. Как уже говорилось, необходимость во внутренних классах обычно возникает в случаях, когда внешний класс описывает сложную систему, состоящую из частей, каждая из которых, в свою очередь, является системой, очень тесно связанной с внешней. Для такого варианта агрегации идеология внутренних классов подходит очень хорошо. В нашем же случае внутренний объект по идеологии агрегирован во внешний (порождается им и не может без него существовать), но фактически после рождения никак с ним не связан. И даже рождение обеспечивает чисто формальную связь. Поэтому таких конструкций следует избегать.

Совсем другое дело, когда мы создаем в каком-либо классе переменную, ссылающуюся на внутренний объект, существующий во внешнем объекте. Этим мы не нарушаем принципа инкапсуляции даже в тех случаях, когда с помощью такой переменной управляем внутренним объектом или ссылаемся на вновь созданный объект, с которым некоторое время работаем, а затем назначаем ссылку на него переменной, заданной во внешнем классе.

Можно заметить, что при работе с внутренними объектами правильное понимание терминологии имеет большое значение. Следует очень четко понимать различие между классами, объектами и переменными, ссылающимися на объекты.

Причем если внутренние классы во всех смыслах являются "внутренними", то для их экземпляров и ссылающихся на них переменных это не так. Ссылочные переменные соответствующего типа можно создавать в любом классе. Точно так же, как мы уже знаем, в любом классе можно создавать внутренние объекты и работать с ними. Поэтому термин "внутренний объект" неточен с точки зрения потенциальных возможностей синтаксиса Java. В этом плане более правильно говорить об *объекте внутреннего класса* (inner class object). В то же время используемая автором терминология проще, понятней и задает мотивацию применения таких объектов. Не всякую потенциально доступную возможность нужно реализовывать!

12.4. Локальные (local) классы

Никаких особенностей в применении локальных классов нет, за исключением того, что область существования их, а также их экземпляров ограничена тем блоком, в котором они заданы. В листинге 12.6 приведен пример использования локального класса.

Листинг 12.6. Пример использования локального класса

```
class LocalClass1 {
    public LocalClass1(){
        System.out.println("LocalClass1 object created");
    }
}
LocalClass1 local1=new LocalClass1();
```

Этот код можно вставить в любой метод (например, в обработчик события нажатия на кнопку). Конечно, данный пример чисто иллюстративный.

12.5. Анонимные (anonimous) классы и обработчики событий

Анонимный (безымянный) класс объявляется без задания имени класса и переменных данного безымянного типа — задается только конструктор класса вместе с его реализацией. У анонимного класса может быть только один экземпляр, причем он создается сразу при объявлении класса. Поэтому перед объявлением анонимного класса следует ставить оператор `new`. Анонимный класс должен быть наследником какого-либо класса или интерфейса, и соответствующий тип должен быть указан перед списком параметров конструктора.

Синтаксис задания анонимного класса таков:

```
new ИмяПрародителя(список параметров конструктора) {
  тело  класса
}
```

Как уже говорилось, анонимные классы обычно используют в обработчиках событий, причем сама необходимость в таких классах, по мнению автора, вызвана неудачной организацией в Java работы с обработчиками событий.

Пример использования анонимного класса в "слушателе" события (о них речь пойдет в следующем разделе) иллюстрирует листинг 12.7.

Листинг 12.7. Пример использования анонимного класса в "слушателе" события

```
        addMouseMotionListener(
           new java.awt.event.MouseMotionAdapter(){
             public void mouseDragged(java.awt.event.MouseEvent e){
```

```
            System.out.println("Mouse dragged at: x="+
                               e.getX()+" y="+e.getY()
                              );
        }
    }
);
```

12.6. Анонимные (anonimous) классы и слушатели событий (listeners)

Событие в Java (будем называть его программным событием, или сокращенно — просто событием) — это объект, возникающий при наступлении какого-либо события в реальном мире при взаимодействии с ним компьютера (будем называть его физическим событием). Например, физическим событием может быть нажатие на клавишу. При наступлении некоторых физических событий возникают программные события — создаются объекты, имеющие тип, зависящий от того, какое событие наступило.

Обработчики событий — это подпрограммы, которые выполняют некоторый код при наступлении программного события (например, код, который будет выполнен при нажатии пользователем на кнопку `jButton1` во время работы приложения).

В Java к каждому объекту, поддерживающему работу с неким событием, могут добавляться слушатели (listeners) событий этого типа — объекты-обработчики событий. Они являются экземплярами специальных классов `Listeners`, в которых заданы методы, реагирующие на соответствующие типы событий.

Классы и интерфейсы для работы с событиями графической системы заданы в пакетах: java.awt, java.awt.event и javax.swing.event.

Важнейший тип событий в пакете java.awt:

- `java.awt.AWTEvent` — абстрактный класс, прародительский для всех классов событий.

Важнейшие типы событий в пакете java.awt.event:

- `ActionEvent` — событие действия (как правило, нажатие);
- `AdjustmentEvent` — изменение значения в линии прокрутки (для компонентов с линией прокрутки);
- `ComponentEvent` — компонент переместился, изменил размер или видимость (visibility) (показался или был скрыт);
- `ContainerEvent` — содержимое компонента-контейнера изменилось (какой-либо компонент был в него добавлен или изъят);
- `FocusEvent` — компонент получил или потерял фокус;
- `HierarchyEvent` — изменение положения компонента в физической иерархии (иерархии агрегации). Например, удаление родительского компонента, смена компонентом родителя (перетаскивание с одного компонента на другой) и т. п.;
- `InputEvent` — произошло событие ввода. Базовый класс для классов событий ввода (`KeyEvent`, `MouseEvent`);

- `InputMethodEvent` — произошло событие ввода. Содержит информацию об обрабатываемом тексте;
- `ItemEvent` — событие, возникающее в случае, если пункт (item) был отмечен (selected) или с него была снята отметка (deselected);
- `KeyEvent` — событие нажатия на клавишу;
- `MouseEvent` — событие мыши;
- `PaintEvent` — событие отрисовки. Служит для управления очередью событий и не может быть использовано для управления отрисовкой вместо методов `paint` или `update`;
- `TextEvent` — событие, возникающее в случае, если текст в текстовом компоненте изменился;
- `WindowEvent` — окно изменило статус (открылось, закрылось, максимизировалось, минимизировалось, получило фокус, потеряло фокус).

Также много классов событий есть в пакете javax.swing.event.

Для того чтобы программа могла обработать событие какого-то типа, в приложение требуется добавить объект `event listener` (слушатель события) соответствующего типа.

Этот тип — класс, который должен реализовать интерфейс слушателя, являющийся наследником интерфейса `java.util.EventListener`. Имя интерфейса слушателя обычно складывается из имени события и слова `Listener`.

Чтобы упростить реализацию интерфейсов, в Java для многих интерфейсов событий существуют так называемые адаптеры (adapters) — классы, в которых все необходимые методы интерфейсов слушателей уже реализованы в виде ничего не делающих заглушек. Так что в наследнике адаптера требуется только переопределение необходимых методов, не заботясь о реализации всех остальных. Перечислим важнейшие интерфейсы и адаптеры слушателей:

- `ActionEvent` — `ActionListener`;
- `AdjustmentEvent` — `AdjustmentListener`;
- `ComponentEvent` — `ComponentListener` — `ComponentAdapter`;
- `ContainerEvent` — `ContainerListener` — `ContainerAdapter`;
- `FocusEvent` — `FocusListener` – `FocusAdapter`;
- `HierarchyEvent` — `HierarchyBoundsListener` — `HierarchyBoundsAdapter`;
- `InputEvent` — нет интерфейсов и адаптеров;
- `InputMethodEvent` — `InputMethodListener`;
- `ItemEvent` — `ItemListener`;
- `KeyEvent` — `KeyListener` — `KeyAdapter`;
- `MouseEvent` — `MouseListener` — `MouseAdapter`:
  - `MouseMotionListener` — `MouseMotionAdapter` (motion — движение). Событие возникает при движении мыши;
  - `MouseWheelListener` — `MouseWheelAdapter` (wheel — колесо). Событие возникает при прокручивании колесика мыши;

- PaintEvent — нет интерфейсов и адаптеров;
- TextEvent — TextListener;
- WindowEvent — WindowListener — WindowAdapter:
  - WindowFocusListener. Событие возникает при получении или потере окном фокуса;
  - WindowStateListener. Событие возникает при изменении состояния окна.

Все компоненты Swing — потомки javax.swing.JComponent.

А в этом классе заданы методы добавления к компоненту многих из упомянутых слушателей:

addComponentListener, addFocusListener и т. д.

В классах компонентов, обладающих специфическими событиями, заданы методы добавления слушателей этих событий.

В листинге 12.8 повторен код листинга 12.7 с разъяснениями.

Листинг 12.8. Переопределение метода MouseDragged в анонимном классе

```
addMouseMotionListener(
   new java.awt.event.MouseMotionAdapter(){
     public void mouseDragged(java.awt.event.MouseEvent e){
        System.out.println("Mouse dragged at: x="+
                           e.getX()+" y="+e.getY()
                          );
     }
   }
);
```

В качестве параметра метода addMouseMotionListener выступает анонимный класс типа java.awt.event.MouseMotionAdapter, переопределяющий метод mouseDragged.

В интерфейсе MouseMotionListener имеются два метода:

- mouseDragged(MouseEvent e);
- mouseMoved(MouseEvent e).

Поскольку мы не переопределили заглушку mouseMoved, то наш объект-обработчик событий типа MouseEvent (движение мыши порождает события именно такого типа) не будет ничего делать для обычного движения.

А вот метод mouseDragged в нашем объекте-обработчике переопределен, поэтому при перетаскиваниях (когда идет движение мыши с нажатой кнопкой) будет выводиться текст в консольное окно.

Допустим, мы поместили код с добавлением слушателя в обработчик события нажатия на какую-либо кнопку (например, jButton1). После срабатывания обработчика наш компонент (главная форма приложения, если мы добавили слушателя ей) станет обрабатывать события типа MouseMotion. При этом на каждое такое событие в консольное окно будет осуществляться вывод позиции мыши, но только в том случае, если идет перетаскивание — когда в пределах формы мы нажали кнопку мыши и не отпуская ее пере-

таскиваем. Причем неважно, какая это кнопка (их можно при необходимости программно различить).

Если мы расположим на форме панель **jPanel1** и заменим вызов `addMouseMotionListener` (реально это вызов `this.addMouseMotionListener`) на `jPanel1.addMouseMotionListener`, то слушатель расположится не в форме (объект `this`), а на панели. Тогда события перетаскивания будут перехватываться только панелью. При этом важно, в какой области началось перетаскивание, но безразлично, в какой области оно продолжается — события от перетаскивания, начавшегося внутри панели, будут возникать даже при перемещении курсора мыши за пределы окна приложения.

Если нажать кнопку добавления слушателя два раза, то в списке слушателей станет два объекта-обработчика событий `MouseEvent`. Объекты из списка слушателей обрабатывают события по очереди, в порядке их добавления. Поэтому каждое событие будет обрабатываться дважды. Если добавить еще объекты слушателей, то события будут обрабатываться соответствующее число раз. Конечно, в реальных ситуациях если добавляют более одного объекта-слушателя для события, то они не повторяют одни и те же действия, а по-разному реагируют на это событие. Либо они являются экземплярами одного класса, но с разными значениями полей данных, либо (что еще чаще) — экземплярами разных классов. Например, для события `MouseEvent` существует интерфейс `MouseListener` и реализующий его адаптер `MouseAdapter`, имеющий четыре метода:

- `mouseClicked(MouseEvent e);`
- `mouseEntered(MouseEvent e);`
- `mouseExited(MouseEvent e);`
- `mouseReleased(MouseEvent e).`

Экземпляры классов-наследников `MouseAdapter` будут совсем по-другому обрабатывать события `MouseEvent`. Аналогично можно создать экземпляр наследника `MouseMotionListener`, который будет реагировать не на перетаскивание, а на простое движение мыши.

Обычно классы слушателей, наследующие от адаптеров, делают анонимными, совмещая декларацию, реализацию и вызов экземпляра класса.

Краткие итоги по главе

- В Java имеются встроенные классы, которые позволяют реализовать дополнительные возможности инкапсуляции и композиции. Они делятся на несколько категорий:
  - вложенные (nested) классы и интерфейсы — используются для задания совершенно самостоятельных классов и интерфейсов внутри классов. Должны при задании иметь модификатор `static`. Имя класса верхнего уровня является квалификатором в пространстве имен, во всем остальном они ведут себя как обычные классы;
  - внутренние (inner) классы — служат для создания экземпляров, принадлежащих экземплярам класса верхнего уровня (т. е. их экземпляры не могут существовать вне объектов верхнего уровня). Не допускается создание методов класса или переменных внутреннего класса. Внутренний класс задается так же, как вложен-

ный, но только без модификатора `static` перед именем этого класса. Внутренние классы позволяют реализовать в Java большинство возможностей модулей из процедурных языков программирования (в этом случае в качестве модуля выступает внешний класс);

- локальные (local) классы — задаются внутри блоков программного кода в методах или блоках инициализации. Они носят вспомогательный характер, область видимости и жизни экземпляров этих классов ограничивается соответствующим блоком программного кода. Как и в случае внутренних классов, это позволяет реализовать в Java возможности модулей из процедурных языков программирования (и в этом случае в качестве модуля также выступает внешний класс);
- анонимные (безымянные, anonimous) классы — совмещают декларацию, реализацию и вызов. Не имеют ни имени, ни конструктора. Их обычно используют в обработчиках событий.

♦ Анонимный класс объявляется без задания имени класса и переменных данного безымянного типа — задается только конструктор класса вместе с его реализацией. У анонимного класса может быть только один экземпляр, причем он создается сразу при объявлении класса. Поэтому перед объявлением анонимного класса следует ставить оператор `new`. Безымянный класс должен быть наследником какого-либо класса или интерфейса, и соответствующий тип должен быть указан перед списком параметров конструктора.

♦ Синтаксис задания безымянного класса таков:

```
new ИмяПрародителя(список параметров конструктора) {
   тело конструктора
}
```

♦ Программные события в Java — объекты, имеющие тип, зависящий от того, какое физическое событие наступило. Обработчики событий — подпрограммы, которые выполняют некоторый код при наступлении программного события.

♦ В Java обработчики событий являются объектами — экземплярами специальных классов `Listeners` (слушателей). В этих классах заданы методы, реагирующие на соответствующий тип событий. Объекты обработчиков можно добавлять к компонентам, способным перехватывать соответствующие типы событий.

♦ Классы и интерфейсы для работы с событиями графической системы заданы в пакетах: java.awt, java.awt.event и javax.swing.event.

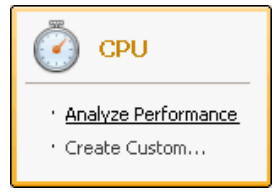
♦ Классы, отвечающие за прослушивание событий, реализуют соответствующие интерфейсы (наследников интерфейса `java.util.EventListener`). Для того чтобы упростить реализацию интерфейсов, в Java для многих интерфейсов событий существуют так называемые адаптеры (adapters) — классы, в которых все необходимые методы интерфейсов слушателей уже реализованы в виде ничего не делающих заглушек. Так что в наследнике адаптера требуется осуществить только переопределение необходимых методов, не заботясь о реализации всех остальных. Обычно эти классы делают анонимными, совмещая декларацию, реализацию и вызов.

Типичные ошибки

- Путают назначение статических и нестатических вложенных классов.
- Пытаются создать экземпляр статического вложенного класса без указания перед именем конструктора вложенного класса имени внешнего класса.
- Пытаются создать экземпляр обычного вложенного класса с указанием перед именем конструктора вложенного класса имени внешнего класса.
- Пытаются создать экземпляр обычного вложенного класса без указания перед оператором `new` имени внешнего объекта.

Задания

- Написать приложение, иллюстрирующее работу вложенных, внутренних и локальных классов.
- Написать приложение, иллюстрирующее работу анонимных классов в обработчиках событий.



ГЛАВА 13

Приложение с графическим интерфейсом — *DesktopApplication*

В качестве примера использования почти всех средств, о которых рассказывалось в данном учебном курсе, рассмотрим заготовку (skeleton) приложения **DesktopApplication** с графическим пользовательским интерфейсом, предлагающуюся разработчикам в среде NetBeans 6.5. Создание этого примера было описано в *разд. 2.16*. На том этапе изучения Java пример был слишком сложен, но теперь он нам по силам.

13.1. Структура простой заготовки *DesktopApplication*

На рис. 13.1 показан внешний вид проекта **DesktopApplication1**. Напомним, что следующая заготовка приложения **DesktopApplication** будет называться **DesktopApplication2** и т. д.

Проект содержит много пакетов.

Первым идет пакет **META-INF.services**, включающий файл org.jdesktop.application.Application (внутри этого файла находится очень простое содержимое — строка **desktopapplication1.DesktopApplication1**, указывающая имя класса сервиса для интерфейса Application).

Начиная с JDK 3, в Java появилась концепция *поставщиков сервисов* (service providers — провайдеры сервисов). Этот вариант регистрации сервисов (служебных программ) основан на задании пути к провайдеру в `classPath`, что заметно упрощает настройку системы.

Переменная окружения `classPath` задает пути к файлам, необходимым для работы Java-машины. Ее значение может быть настроено для каждого приложения.

В качестве сервиса, обеспечивающего работу некоторого интерфейса, может выступать любой класс, реализующий этот интерфейс. Для использования данного класса в качестве сервиса достаточно включить в файл манифеста (пакет **META-INF.services**, соответствующий папке META-INF\services) файл с именем интерфейса (с полным путем), а в этом файле указать имя класса, реализующего интерфейс. Вызов из интерфейса метода `newInstance` приводит к поиску необходимого класса и его загрузке. Для поддержки работы поставщиков сервисов в JDK 6 был добавлен класс `java.util.ServiceLoader`.

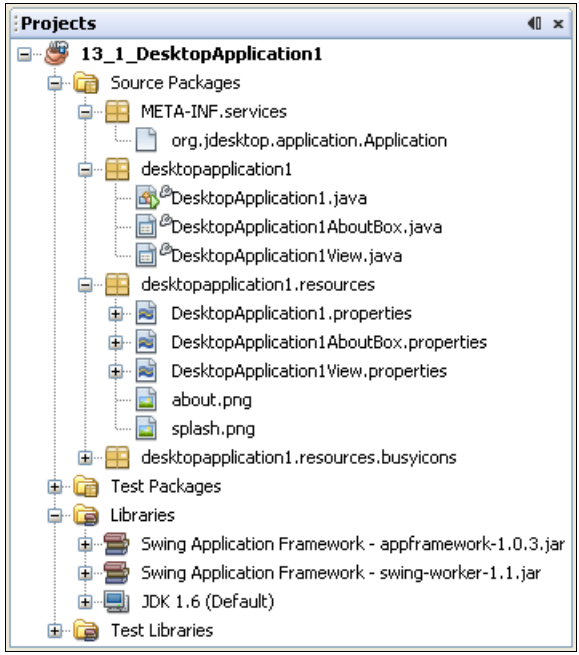


Рис. 13.1. Состав проекта **DesktopApplication1**

В пакете **desktopapplication1** содержатся файлы исходных кодов для трех классов:

- **DesktopApplication1.java** — файл, в котором задан класс `DesktopApplication1`. Содержит главный метод приложения, где реализован алгоритм запуска приложения;
- **DesktopApplication1AboutBox.java** — файл, в котором задан класс экранной формы, вызываемой по нажатию пункта меню **Help/About**. Отвечает за вывод формы справки (**About Box**);
- **DesktopApplication1View.java** — класс, отвечающий за показ главной экранной формы приложения.

Пакет **desktopapplication1.resources** нам уже знаком по *главе 11*. Он содержит файлы свойств для локализации классов пакета **desktopapplication1**, а также файлы изображений, которые нужны этим классам.

Пакет **desktopapplication1.resources.busyicons** содержит файлы изображений-иконок, которые нужны для панели задач.

Тестовые пакеты предназначены для создания тестов и тестирования с их помощью правильности работы приложения. Но создание тестов — отдельная тема, и в данной книге она освещаться не будет.

Наше приложение нуждается в нескольких библиотеках (точнее, в отдельных пакетах и классах из этих библиотек) — их архивы (jar-файлы) содержатся в папке Libraries (библиотеки). Наиболее важная библиотека — Swing Application Framework, из которой используются пакеты **org.desktop.application** и **org.jdesktop.swingworker**, и библиотека **JDK 1.6**, являющаяся на компьютере автора библиотекой Java по умолчанию (default).

13.2. Исходный код класса *DesktopApplication*

Класс `DesktopApplication1` (листинг 13.1), как уже говорилось, запускает наше приложение. Он является потомком класса `SingleFrameApplication`, прародителя классов экранных форм с заголовком и меню. В заголовке параграфа указано имя `DesktopApplication`, а не `DesktopApplication1`, т. к. для следующего приложения это будет `DesktopApplication2`, затем `DesktopApplication3` и т. п.

Листинг 13.1. Исходный код класса `DesktopApplication1`

```
/*
 * DesktopApplication1.java
 */

package desktopapplication1;

import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;

/**
 * The main class of the application.
 */
public class DesktopApplication1 extends SingleFrameApplication {

    /**
     * At startup create and show the main frame of the application.
     */
    @Override protected void startup() {
        show(new DesktopApplication1View(this));
    }

    /**
     * This method is to initialize the specified window by injecting
     * resources. Windows shown in our application come fully initialized
     * from the GUI builder, so this additional configuration
     * is not needed.
     */
    @Override protected void configureWindow(java.awt.Window root) {
    }

    /**
     * A convenient static getter for the application instance.
     * @return the instance of DesktopApplication1
     */
    public static DesktopApplication1 getApplication() {
        return Application.getInstance(DesktopApplication1.class);
    }

    /**
     * Main method launching the application.
     */
```

```
    public static void main(String[] args) {
        launch(DesktopApplication1.class, args);
    }
}
```

Рассмотрим отдельные методы, заданные в классе `DesktopApplication1`.

- Метод `startup()`.

```
@Override protected void startup() {
    show(new DesktopApplication1View(this));
}
```

первоначально заданный в классе `Application` и обычно переопределяемый в потомках (как в апплетах!), выполняется после запуска класса на выполнение — обычно методом `launch`, также заданным в классе `Application`. Отвечает за создание и инициализацию графического пользовательского интерфейса приложения.

Напомним, что аннотация `@Override` поясняет человеку, читающему текст программы, что данный метод переопределяет родительский. Одновременно аннотация является метаданными для компилятора. Если метод аннотирован символом `@Override`, но на самом деле не переопределяет родительский метод (например, имеет другую сигнатуру), то компилятор на основании этой аннотации выдает сообщение об ошибке.

- Метод `configureWindow(Window root)`.

```
@Override protected void configureWindow(java.awt.Window root) {
}
```

предназначен для обеспечения окна необходимыми ресурсами. В данном приложении он имеет пустую реализацию, т. к. все необходимые ресурсы предоставляются окну создателем пользовательского интерфейса класса `DesktopApplication1View`. Так что класс `Window`, нужный только для задания параметра в этом методе, реально в данном приложении не используется.

- Метод `getApplication()`.

```
public static DesktopApplication1 getApplication() {
    return Application.getInstance(DesktopApplication1.class);
}
```

Функция `getApplication()`, реализация которой приведена в этом фрагменте, предназначена для возврата ссылки на объект-приложение заданного типа. Возврат производится с помощью вызова метода класса `Application.getInstance(DesktopApplication1.class)`, создающего объект типа `DesktopApplication1`. В качестве параметра выступает ссылка на класс приложения.

Особенность объекта-приложения в том, что это *объект-синглетон*, всегда существующий в единственном экземпляре. Повторные попытки создания таких объектов приводят к возврату ссылки на уже существующий синглетон. Таким образом, сколько бы объектов мы ни пытались создать вызовами `Application.getInstance (DesktopApplication1.class)`, эти вызовы всегда будут ссылаться на один и тот же объект-синглетон, создаваемый при первом вызове метода `getInstance`.

При запуске приложения создается еще один синглетон, контекст приложения, доступ к которому можно получить из класса или объекта приложения с помощью метода `getContext()`. Этот синглетон — "незримый спутник" приложения и позволяет получать о нем различные сведения. Например, его метод `getApplication()` дает ссылку на объект-приложение, `getClipboard()` — ссылку на буфер обмена приложения, `getFocusOwner()` — ссылку на компонент приложения, владеющий фокусом, и т. д. О данном синглетоне подробнее говорится далее в *разд. 13.4*.

А почему бы не воспользоваться для доступа к объекту-приложению простой ссылкой `this`? Зачем нужен метод `getApplication`?

Если из другого класса (например, из `DesktopApplication1View`) требуется доступ к объекту-приложению, ссылку необходимо возвращать с помощью функции. Для класса `Application` ее естественно назвать `getApplication`. Поскольку объект-приложение может существовать только в единственном экземпляре, то используется метод класса (`static`), который возвращает ссылку на приложение-синглетон (если такой объект уже существует) или создает этот объект (если он отсутствует). А ссылка `this` вернула бы ссылку на экземпляр `DesktopApplication1View`, а вовсе не на экземпляр `DesktopApplication1`.

- Метод `main`.

```
public static void main(String[] args) {
    launch(DesktopApplication1.class, args);
}
```

Метод `main`, реализация которого приведена в этом фрагменте, запускается в главном классе приложения сразу после создания класса. Заметим, что на этот момент объекта-приложения не существует — `main` является методом класса.

Объект-приложение, имеющий тип `DesktopApplication1`, создается в методе `main` с помощью вызова метода `launch(DesktopApplication1.class, args)`. В качестве первого параметра задается тип создаваемого объекта (ссылка на класс — *см. разд. 6.12*), в качестве второго — строка `args` с переданными снаружи (из операционной системы) параметрами приложения, т. е. командная строка.

13.3. Исходный код класса *DesktopApplicationView*

Приведем только те элементы класса `DesktopApplication1View` (листинг 13.2), которые для нас существенны. Автоматически сгенерированный код инициализации элементов экранной формы заменен многоточием и рассматриваться не будет.

Листинг 13.2. Исходный код класса `DesktopApplication1View`

```
/*
 * DesktopApplication1View.java
 */

package desktopapplication1;
```

```
import org.jdesktop.application.Action;
import org.jdesktop.application.ResourceMap;
import org.jdesktop.application.SingleFrameApplication;
import org.jdesktop.application.FrameView;
import org.jdesktop.application.TaskMonitor;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import javax.swing.Icon;
import javax.swing.JDialog;
import javax.swing.JFrame;

/**
 * The application's main frame.
 */
public class DesktopApplication1View extends FrameView {

   public DesktopApplication1View(SingleFrameApplication app) {
       super(app);

       initComponents();

       // status bar initialization — message timeout, idle icon and
       // busy animation, etc
       ResourceMap resourceMap = getResourceMap();
       int messageTimeout =
                    resourceMap.getInteger("StatusBar.messageTimeout");

       messageTimer = new Timer(messageTimeout, new ActionListener() {

           public void actionPerformed(ActionEvent e) {
               statusMessageLabel.setText("");
           }

       });

       messageTimer.setRepeats(false);
       int busyAnimationRate =
                   resourceMap.getInteger("StatusBar.busyAnimationRate");
       for (int i = 0; i < busyIcons.length; i++) {
           busyIcons[i] =
                   resourceMap.getIcon("StatusBar.busyIcons["+i+"]");
       }

       busyIconTimer= new Timer(busyAnimationRate, new ActionListener() {
           public void actionPerformed(ActionEvent e) {
               busyIconIndex = (busyIconIndex + 1) % busyIcons.length;
               statusAnimationLabel.setIcon(busyIcons[busyIconIndex]);
           }
       });

       idleIcon = resourceMap.getIcon("StatusBar.idleIcon");
       statusAnimationLabel.setIcon(idleIcon);
       progressBar.setVisible(false);
```

```
        // connecting action tasks to status bar via TaskMonitor
        TaskMonitor taskMonitor = new TaskMonitor(getApplication().
                                                        getContext());
        taskMonitor.addPropertyChangeListener(new
                                    java.beans.PropertyChangeListener() {
          public void propertyChange(java.beans.PropertyChangeEvent evt) {
              String propertyName = evt.getPropertyName();
              if ("started".equals(propertyName)) {
                  if (!busyIconTimer.isRunning()) {
                      statusAnimationLabel.setIcon(busyIcons[0]);
                      busyIconIndex = 0;
                      busyIconTimer.start();
                  }
                  progressBar.setVisible(true);
                  progressBar.setIndeterminate(true);
              } else if ("done".equals(propertyName)) {
                  busyIconTimer.stop();
                  statusAnimationLabel.setIcon(idleIcon);
                  progressBar.setVisible(false);
                  progressBar.setValue(0);
              } else if ("message".equals(propertyName)) {
                  String text = (String)(evt.getNewValue());
                  statusMessageLabel.setText((text == null) ? "" : text);
                  messageTimer.restart();
              } else if ("progress".equals(propertyName)) {
                  int value = (Integer)(evt.getNewValue());
                  progressBar.setVisible(true);
                  progressBar.setIndeterminate(false);
                  progressBar.setValue(value);
              }
          }
        });
    }


    @Action
    public void showAboutBox() {
        if (aboutBox == null) {
            JFrame mainFrame =
                       DesktopApplication1.getApplication().getMainFrame();
            aboutBox = new DesktopApplication1AboutBox(mainFrame);
            aboutBox.setLocationRelativeTo(mainFrame);
        }
        DesktopApplication1.getApplication().show(aboutBox);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
```

```
 @SuppressWarnings("unchecked")
 // <editor-fold defaultstate="collapsed" desc="Generated Code">
 private void initComponents() {
...
 }// </editor-fold>

 // Variables declaration - do not modify
 private javax.swing.JPanel mainPanel;
 private javax.swing.JMenuBar menuBar;
 private javax.swing.JProgressBar progressBar;
 private javax.swing.JLabel statusAnimationLabel;
 private javax.swing.JLabel statusMessageLabel;
 private javax.swing.JPanel statusPanel;
 // End of variables declaration

 private final Timer messageTimer;
 private final Timer busyIconTimer;
 private final Icon idleIcon;
 private final Icon[] busyIcons = new Icon[15];
 private int busyIconIndex = 0;

 private JDialog aboutBox;
}
```

Класс `DesktopApplication1View` устроен сложнее, чем класс приложения (`DesktopApplication1`), поэтому мы будем его обсуждать не только в данном разделе, но и в нескольких последующих *(разд. 13.4—13.6)*.

В листинге 13.2 сначала идет импорт различных пакетов.

Заметим, что вместо первых пяти операторов импорта можно было бы написать:

```
import org.jdesktop.application.*;
```

Но непосредственное перечисление импортируемых классов правильнее, поскольку позволяет программисту, читающему исходный код, лучше понять, с какими внешними классами связан данный класс. Более того, если после модификации кода какой-то класс перестал быть необходимым, среда разработки выдаст предупреждение о неиспользуемом импорте и предложит его убрать.

Класс `DesktopApplication1View` — наследник класса `View`, базового для всех элементов Java с графическим интерфейсом. Кроме того, он и сам является базовым для всех классов экранных форм, имеющих заголовок окна.

Методы, заданные в классе `DesktopApplication1View`:

- `public DesktopApplication1View(SingleFrameApplication app)` — конструктор;
- `public void showAboutBox()` — акция (об акциях будет рассказано отдельно);
- `private void initComponents()` — автоматически генерируемый код инициализации компонентов.

Поля (переменные), заданные в классе `DesktopApplication1View`:

- `private javax.swing.JPanel mainPanel` — панель. Основная область приложения, где можно размещать пользовательские компоненты;

- `private javax.swing.JMenuBar menuBar` — область главного меню приложения;
- `private javax.swing.JPanel statusPanel` — область *строки состояния* приложения. Это небольшая прямоугольная область в самой нижней части окна приложения, в которой обычно показывается вспомогательная информация о текущем состоянии приложения;
- `private javax.swing.JProgressBar progressBar` — индикатор выполнения. Это узкий прямоугольник, в котором показывается "бегущий столбик", отмечающий процент выполнения задания;
- `private javax.swing.JLabel statusAnimationLabel` — метка, в которой показываются циклически сменяемые иконки, имитирующие вращение первоначальной иконки;
- `private javax.swing.JLabel statusMessageLabel` — метка, в которой показывается сообщение о процессе выполнения задания, дублирующее работу индикатора выполнения;
- `private final Icon idleIcon` — иконка бездействия, показывающая состояние приложения (idle — никакой работы не выполняется). Иконкой (icon) называют небольшую картинку. В обычном состоянии показывается иконка бездействия, а когда идет работа индикатора выполнения, то циклически сменяются изображения, вызывающие впечатление вращающейся иконки;
- `private final Icon[] busyIcons = new Icon[15]` — массив, в котором хранится набор из 15 иконок, последовательный показ которых приводит к видимости вращения иконки (идет работа);
- `private int busyIconIndex = 0` — индекс текущей показываемой иконки;
- `private final Timer messageTimer` — таймер метки `statusMessageLabel`;
- `private final Timer busyIconTimer` — таймер иконки `busyIcon` (идет работа). Служит для циклического изменения этой иконки и состояния индикатора выполнения при действиях, вызывающих его работу;
- `private JDialog aboutBox` — ссылается на экранную форму справки.

Аннотация `@SuppressWarnings("unchecked")`, которая стоит перед заданием этих полей, применяется для подавления предупреждений, которые могут возникнуть при инициализации компонентов формы. По-видимому, это либо остатки отладочного фрагмента кода, либо средство по удалению мешающих предупреждений при работе программы под какой-либо операционной системой или с каким-либо вариантом Look and Feel.

13.4. Конструктор главной формы приложения. Ресурсы приложения и концепция управляющих синглетонов

В листинге 13.3 приведен конструктор главной формы приложения. Обсудим его работу.

Листинг 13.3. Исходный код конструктора `DesktopApplication1View`

```
public DesktopApplication1View(SingleFrameApplication app) {
    super(app);
```

```
    initComponents();

    // status bar initialization - message timeout, idle icon and busy
    //animation, etc
    ResourceMap resourceMap = getResourceMap();
    int messageTimeout =
                    resourceMap.getInteger("StatusBar.messageTimeout");

    messageTimer = new Timer( messageTimeout, new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            statusMessageLabel.setText("");
        }

    });

    messageTimer.setRepeats(false);
    int busyAnimationRate =
                resourceMap.getInteger("StatusBar.busyAnimationRate");
    for (int i = 0; i < busyIcons.length; i++) {
        busyIcons[i] =resourceMap.getIcon("StatusBar.busyIcons["+i+"]");
    }

    busyIconTimer = new Timer( busyAnimationRate, new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            busyIconIndex = (busyIconIndex + 1) % busyIcons.length;
            statusAnimationLabel.setIcon(busyIcons[busyIconIndex]);
        }
    });
}
```

В качестве параметра конструктору `DesktopApplication1View(...)` главной формы приложения передается наш объект-приложение `app`, имеющий тип, совместимый с `SingleFrameApplication`. Напомним, что главная форма приложения создается с помощью этого конструктора в методе `startup()` при вызове оператора:

```
show(new DesktopApplication1View(this));
```

В теле конструктора, как обычно, сначала вызывается прародительский конструктор. Затем идет вызов метода `initComponents()` добавления на форму компонентов и их инициализации (автоматически сгенерированный редактором визуального проектирования код). Затем переменной `resourceMap` присваивается ссылка на таблицу ресурсов приложения, хранящихся в папке src\desktopapplication1\DesktopApplication1.resources. В дереве проекта она показана как пакет **DesktopApplication1.resources**. Доступ к ресурсам осуществляется с помощью метода `getResourceMap()`.

На этом следует остановиться подробнее, поскольку использование ресурсов опирается на одну важную концепцию Java, часто применяемую при работе с компонентами — создание *синглетонов-спутников*, являющихся *управляющими объектами*. Такие объекты очень часто используются при построении сложных программных систем по схеме *(паттерну)* "модель-визуализация-управляющий объект" — Model-View-Controller (MVC).

Метод `getResourceMap()` обеспечивает сокращенную запись вызова:

```
org.jdesktop.application.Application.getInstance().
                                          getContext().getResourceMap()
```

В данном случае вызов может быть короче, т. к. ссылка на объект-приложение передается через список параметров:

```
app.getContext().getResourceMap()
```

Метод `getContext()` возвращает ссылку на синглетон-спутник, сопровождающий объект-приложение. Этот объект управляет так называемыми *разделяемыми объектами* приложения (shared objects), т. е. такими объектами, которые доступны разным потокам приложения. Например, акциями (actions), ресурсами (resources) и задачами (tasks).

Синглетоны-спутники создаются для того, чтобы расширить возможности работы с объектами некоторого класса без добавления методов и полей в сам класс. Это позволяет обеспечить независимое от оригинального класса изменение классов спутников и их иерархии с возможностью (при необходимости) замены одних типов управляющих объектов на другие. Например, можно создать управляющие объекты, обеспечивающие проверку прав доступа (менеджеры безопасности) при работе в компьютерных сетях, либо (как в данном случае) использовать менеджер разделяемых объектов.

Если мы запишем в исходном коде вместо сокращенного варианта вызова полный, то столкнемся с проблемой переноса очень длинных выражений, в которых присутствует оператор точка (.), означающий доступ к полю или методу.

Синтаксис Java разрешает оба варианта переноса:

- с переносом точки на новую строку:

```
ResourceMap resourceMap = org.jdesktop.application.Application
                  .getInstance().getContext().getResourceMap();
```

- с оставлением точки на предыдущей строке:

```
ResourceMap resourceMap = org.jdesktop.application.Application.
                   getInstance().getContext().getResourceMap();
```

В качестве параметра метода `getResourceMap` в полном варианте вызова можно использовать ссылку на класс, т. к. в одном классе (как в нашем случае) могут храниться ресурсы для разных классов:

```
ResourceMap resourceMap = org.jdesktop.application.Application.
         getInstance().getContext().
             getResourceMap(DesktopApplication1View.class);
```

При сокращенном вызове `getResourceMap()` мы получаем доступ к ресурсам только для самого класса `DesktopApplication1`, идентично вызову `getResourceMap (DesktopApplication1.class)`.

Вообще говоря, в Java таблица `map` — это список пар (имя-значение).

Таблица ресурсов устроена несколько сложнее. В ней, кроме обычных списков пар ("имя-значение"), хранятся и другие данные, в нашем случае — изображения в формате PNG (усовершенствованный аналог хорошо известного формата GIF).

А пары ("имя-значение") хранятся в файлах свойств (properties), для каждого класса — свой файл:

- для класса `DesktopApplication1` — файл DesktopApplication1.properties;
- для `DesktopApplication1View` — DesktopApplication1View.properties;
- для `DesktopApplication1AboutBox` — DesktopApplication1AboutBox.properties.

Увидеть содержимое пакета (папки) desktopapplication1.resources и файлов свойств можно, развернув соответствующие узлы дерева проекта. Так что ссылка на класс используется для того, чтобы выбрать соответствующий классу файл свойств. Эти файлы можно редактировать, открыв их двойным щелчком в дереве проекта. А вот изображения можно просматривать, но средств их редактирования в NetBeans пока нет.

Файлы свойств можно добавлять в любой пакет Java. Для этого в дереве проектов следует вызвать контекстное меню щелчком правой кнопки мыши по имени пакета, после чего выбрать **New/Other/**, затем в категории **Other/** выбрать **Property File**, ввести имя файла и нажать кнопку **Finish**. После того как один раз было проведено создание файла свойств, в контекстном меню появится пункт **Property File**, и создание очередного файла свойств можно осуществлять вызовом **New/Property File**. Вообще, контекстное (всплывающее) меню в NetBeans автоматически настраивается по последним из сделанных вызовов, что заметно упрощает работу опытным пользователям, но может иногда затруднить жизнь начинающим, если они не знают об этой особенности.

Задание начального значения переменной `messageTimeout` (тайм-аут сообщения) является хорошим примером того, каким образом извлекать данные из файла ресурсов:

```
int messageTimeout = resourceMap.getInteger("StatusBar.messageTimeout");
```

В объекте `resourceMap` типа `ResourceMap` имеется большое количество методов для извлечения данных: `getBoolean`, `getByte`, `getColor`, `getDouble`, `getFloat`, `getFont`, `getIcon`, `getImageIcon`, `getInteger` и т. д. Все эти методы имеют в качестве параметра строку, задающую имя ключа.

При этом имя ключа (т. е. тех символов, по которым ищется значение) может быть составным — состоящим из частей, разделенных точками. Например, файл свойств у класса `DesktopApplication1View` для языка по умолчанию (Default Language) выглядит так, как показано в листинге 13.4.

Листинг 13.4. Фрагмент содержимого файла свойств класса `DesktopApplication1View` для языка по умолчанию

```
# Resources for the ShellFrame class

# top-level menus

fileMenu.text = File
helpMenu.text = Help

# @Action resources

showAboutBox.Action.text = &About...
showAboutBox.Action.shortDescription = Show the application's information dialog
```

```
# status bar resources

StatusBar.messageTimeout = 5000
StatusBar.busyAnimationRate = 30
StatusBar.idleIcon = busyicons/idle-icon.png
StatusBar.busyIcons[0]  = busyicons/busy-icon0.png
StatusBar.busyIcons[1]  = busyicons/busy-icon1.png
StatusBar.busyIcons[2]  = busyicons/busy-icon2.png
StatusBar.busyIcons[3]  = busyicons/busy-icon3.png
StatusBar.busyIcons[4]  = busyicons/busy-icon4.png
StatusBar.busyIcons[5]  = busyicons/busy-icon5.png
StatusBar.busyIcons[6]  = busyicons/busy-icon6.png
StatusBar.busyIcons[7]  = busyicons/busy-icon7.png
StatusBar.busyIcons[8]  = busyicons/busy-icon8.png
StatusBar.busyIcons[9]  = busyicons/busy-icon9.png
StatusBar.busyIcons[10] = busyicons/busy-icon10.png
StatusBar.busyIcons[11] = busyicons/busy-icon11.png
StatusBar.busyIcons[12] = busyicons/busy-icon12.png
StatusBar.busyIcons[13] = busyicons/busy-icon13.png
StatusBar.busyIcons[14] = busyicons/busy-icon14.png
```

При получении с помощью переменной `resourceMap` ресурса-иконки по ее имени методом `getIcon("...")`

(например: `resourceMap.getIcon("StatusBar.idleIcon")`)

мы имеем ссылку на объект-иконку, имя которой указано в качестве параметра. Из приведенного в листинге 13.4 примера также видно, каким образом можно задавать ресурсы для массива.

Значение `messageTimeout`, которое мы извлекли из таблицы ресурсов, используется при создании объекта-таймера `messageTimer` конструктором:

```
Timer(messageTimeout, new ActionListener(){... } )
```

Первый параметр конструктора (`messageTimeout`) — время в миллисекундах, через которое срабатывает таймер (порождает событие срабатывания таймера).

Второй параметр (`new ActionListener()`) — ссылка на объект слушателя этого события. В данной программе такой объект создается с помощью одной из наиболее неструктурных конструкций Java — анонимного класса. В нем совмещены декларация, реализация и вызов. В методе `actionPerformed`, заданном в анонимном классе, описан обработчик события срабатывания таймера. В нем очищается текст в метке `statusMessageLabel`.

Оператор `messageTimer.setRepeats(false)` (установить повторения) устанавливает режим работы таймера — таймер переводится в режим однократного действия (генерации события таймера), без повторений.

Значение переменной `busyAnimationRate` задает время обновления вращающейся иконки "идет работа" строки состояния, определяемое по таймеру. По умолчанию в проекте стоит значение 30 (задержка в миллисекундах). Если его заменить на 90, то вращение будет проходить в три раза медленнее.

Далее в теле конструктора идет цикл заполнения массива иконок, имитирующих вращение иконки "идет работа".

Переменная `busyIconTimer` служит для хранения ссылки на объект-таймер, создаваемый в правой части оператора с помощью конструктора:

```
Timer(busyAnimationRate, new ActionListener() {...} )
```

Объект-слушатель типа `ActionListener`, создаваемый с помощью безымянного класса, отвечает за обработку события таймера `busyIconTimer`. По этому событию происходит изменение индекса крутящихся иконок (циклическое, от 0 до 14 с шагом 1), а затем показ иконки с этим индексом в метке `statusAnimationLabel`, имеющей обычный тип `JLabel`. Отображение осуществляет метод `statusAnimationLabel.setIcon`.

Напомним, что по холсту (Canvas) любого компонента `Swing` можно рисовать, а метка может служить для вывода не только текста, но и изображений. Изображение, назначаемое метке, авторы NetBeans называют иконкой (небольшим изображением), хотя реально на размер этого изображения не ставится ограничений.

На этом код конструктора главной формы приложения заканчивается.

13.5. Аннотация *@Action* и показ справки

В оставшейся не рассмотренной части кода класса `DesktopApplication1View` представляют интерес метод `showAboutBox` и фрагмент кода, относящийся к объекту `taskMonitor` (листинг 13.5).

Листинг 13.5. Пометка метода аннотацией `@Action`

```
@Action public void showAboutBox() {
    if (aboutBox == null) {
        JFrame mainFrame =
                DesktopApplication1.getApplication().getMainFrame();
        aboutBox = new DesktopApplication1AboutBox(mainFrame);
        aboutBox.setLocationRelativeTo(mainFrame);
    }
    DesktopApplication1.getApplication().show(aboutBox);
}
```

В задании метода `showAboutBox` присутствует еще один символ метаданных — аннотация `@Action`. Этот символ означает, что помеченный им метод представляет собой реализацию акции для интерфейса `Action`, т. е. обработчик `actionPerformed` объекта-события типа `action` (акция). Данный интерфейс не содержит методов, в отличие от предыдущих рассмотренных нами случаев использования интерфейсов. Имя интерфейса в данном случае используется как метка.

Такой метод назначения обработчиков событий появился в JDK 5, является альтернативой создания обработчика `actionPerformed` и обладает большей гибкостью.

Правда, у задания обработчика событий с помощью акций имеется существенный недостаток по сравнению с обычным вариантом — проверка на правильность сигнатуры обработчика происходит не на этапе компиляции программы, как обычно, а на этапе вызова обработчика. Более подробную информацию о таком методе назначения обработчиков событий можно найти в документации по интерфейсам `application.Application` и `application.ProxyActions`, а также классам `application.ApplicationActionMap` и `application.ApplicationAction`.

Любой метод, помеченный символом `@Action`, можно назначить в окне **Properties** любому компоненту (например, кнопке, пункту меню или кнопке панели инструментов — имена заданных акций появляются в выпадающем списке в свойстве `action` этих компонентов). Если требуется отменить обработчик события, достаточно выбрать в этом списке значение `null`.

Можно комбинировать оба способа задания обработчиков событий одновременно. При этом сначала исполняются обработчики `actionPerformed`, и только затем — акции.

В нашем случае акция заключается в показе формы `aboutBox`. При этом применяется уже известный нам паттерн "Отложенное создание объекта" — при первом вызове, когда `aboutBox == null`, объект сначала создается, а затем показывается. Если же объект уже существует, он сразу показывается.

Класс `DesktopApplication1AboutBox` формы справки достаточно простой (листинг 13.6).

Листинг 13.6. Исходный код класса `DesktopApplication1AboutBox`

```
/*
 * DesktopApplication1AboutBox.java
 */

package desktopapplication1;

import org.jdesktop.application.Action;

public class DesktopApplication1AboutBox extends javax.swing.JDialog {

    public DesktopApplication1AboutBox(java.awt.Frame parent) {
        super(parent);
        initComponents();
        getRootPane().setDefaultButton(closeButton);
    }

    @Action public void closeAboutBox() {
        setVisible(false);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */

    Generated Code

    // Variables declaration - do not modify
    private javax.swing.JButton closeButton;
    // End of variables declaration

}
```

Класс справки `DesktopApplication1AboutBox` является наследником класса `JDialog`. Диалоги, в отличие от окон (**Frame**, **View**), не самостоятельны — при закрытии родительско-

го (parent) окна они автоматически закрываются, и если родительское окно становится невидимым, то и дочерние также скрываются, а если становится видимым, то эти диалоги также показываются (если были открыты).

В конструкторе формы справки сначала традиционно вызывается родительский конструктор, затем инициализируются компоненты формы (вызывается код, автоматически сгенерированный редактором визуального проектирования).

После этого кнопка **closeButton**, используемая для закрытия формы, назначается кнопкой по умолчанию (Default Button). Это означает, что когда форма в фокусе, то нажатие клавиши <Enter> приводит к нажатию этой кнопки независимо от того, на каком элементе формы находится фокус. Назначение осуществляется с помощью вызова метода `setDefaultButton` из объекта типа `RootPane` (Основная панель), возвращаемого вызовом `getRootPane()`. Основная панель — это внутренняя область формы, на которой размещены все элементы формы, кроме заголовка и меню.

Метод `closeAboutBox()` — акция, обрабатывающая событие закрытия формы. При этом форма просто становится невидимой.

Теперь мы можем закончить анализ акции `showAboutBox` класса `DesktopApplication1View`.

Строка

```
JFrame mainFrame = DesktopApplication1.getApplication().getMainFrame();
```

предназначена для создания временной переменной `mainFrame`, используемой в следующем вызове. Она хранит ссылку на главное окно приложения (**Main Frame**).

Отметим, что создатели NetBeans начали последовательно проводить политику введения промежуточных переменных вместо встраивания вызовов функций в качестве параметров (что полностью совпадает с точкой зрения автора на этот вопрос). При этом во многих случаях (когда данная рекомендация нарушается) в редакторе исходного кода возникает предупреждение и предлагается решить эту проблему введением локальной переменной (variable) или введением глобальной переменной — нового поля (field).

13.6. Концепция управляющего объекта-модели. Модель *TaskMonitor*, класс *Task* и обработчик *propertyChange*

Нам осталось проанализировать последние фрагменты кода класса `DesktopApplication1View`.

Сначала задается промежуточная переменная `taskMonitor`, которой назначается ссылка на создаваемый объект типа `TaskMonitor`:

```
// connecting action tasks to status bar via TaskMonitor
TaskMonitor taskMonitor = new TaskMonitor(getApplication().getContext());
```

Этот объект — так называемая *модель* для компонентов с графическим пользовательским интерфейсом (GUI).

В Java для работы с визуальными компонентами применяется концепция управляющих объектов. Сложным компонентам (например, таблицам) сопутствуют управляющие объекты, по одному на каждый экземпляр компонента. Доступ к данным отделен от доступа к другим свойствам компонента, не имеющим отношения к данным. Модель —

управляющий объект для этих данных. Например, чтобы получить доступ к данным таблицы, следует обращаться к модели таблицы, а не напрямую к объекту-таблице. Объект модели агрегирован в управляемый объект (например, таблицу) и может быть получен методом `getModel`. Новый управляющий объект может быть назначен методом `setModel`.

Данная концепция является выражением принципа отделения данных и их внешнего представления, а также отделения бизнес-логики от логики визуализации.

`TaskMonitor` — это класс моделей другого вида, предназначенных для визуализации компонентов GUI в многопоточных приложениях, когда содержимое компонента должно обновляться из фонового потока (background thread). Объект данной модели наблюдает за основным потоком выполнения и всеми фоновыми потоками, зарегистрированными управляющим объектом приложения (типа `ApplicationContex`), и производит визуализацию (работу с GUI), необходимую для этих потоков.

Абстрактный параметризованный класс `Task` (задание) предназначен для создания фоновых заданий (потоков выполнения) приложения. Параметризованный класс — это такой класс, в котором используемый тип задан шаблоном (template). Шаблон задается в треугольных скобках и содержит имена типов, которые в классе не конкретизированы, а должны быть указаны в наследниках. Концепция шаблонов требует отдельного подробного разъяснения и в данной книге не описывается.

Объекты типа `Task` при выполнении фоновой задачи всегда вызывают следующие важнейшие методы:

- `doInBackground()` — выполнять в фоновом режиме;
- `done()` — сделано;
- `finished()` — завершено.

Когда объект типа `Task` создается, то в фоновом режиме начинается выполнение метода `doInBackground`. После окончания выполнения задания автоматически запускается метод `done` (он помечен как `final` и поэтому не может быть переопределен).

В зависимости от результатов выполнения задания метод `done` вызывает один из следующих вспомогательных методов, выполняющих необходимые завершающие действия:

- `succeeded` (все сделано удачно) — когда все хорошо;
- `cancelled` (отказ от выполнения) — когда произошел отказ от выполнения с помощью вызова метода `cancel(true)`;
- `interrupted` (прервано) — когда выполнение задания было прервано;
- `failed` (неудача) — когда выполнение задания завершилось неудачно.

После окончания метода `done`, т. е. выполнения завершающих действий с помощью вспомогательных методов или возникновения исключения при их выполнении, автоматически вызывается метод `finished`.

Во время работы фонового потока (метода `doInBackground`) обычно применяются следующие вспомогательные методы:

- `setMessage(String message)` — задать сообщение;
- `setProgress(float percentage)` — задать значение полю `progress` (процент выполнения задания);

- `setProgress(float value, float min, float max)` — то же, но с автоматическим вычислением процента по параметрам: `value`, `min` и `max`;
- `setProgress(int value, int min, int max)` — то же, но параметры имеют тип `int`.

Также в классе `Task` имеется большое количество других вспомогательных методов, которые мы обсуждать не будем.

Пусть мы хотим, чтобы при выборе пункта меню **Save** (Сохранить) запускался фоновый поток и показывалась индикация процента выполнения задания. В формирующейся заготовке возникает класс, заданный внутри класса `DesktopApplication1View` и имеющий пустую реализацию методов. О том, как создать задачу, излагается в следующем разделе, а пока рассмотрим получающуюся при этом заготовку класса задачи `SaveTask` (листинг 13.7).

Листинг 13.7. Заготовка класса задачи `SaveTask`

```
private class SaveTask extends org.jdesktop.application.Task<Object, Void> {
    SaveTask(org.jdesktop.application.Application app) {
        // Runs on the EDT.  Copy GUI state that
        // doInBackground() depends on from parameters
        // to SaveTask fields, here.
        super(app);
    }
    @Override protected Object doInBackground() {
        // Your Task's code here.  This method runs
        // on a background thread, so don't reference
        // the Swing GUI from here.
        return null;  // return your result
    }
    @Override protected void succeeded(Object result) {
        // Runs on the EDT.  Update the GUI based on
        // the result computed by doInBackground().
    }
}
```

Для того чтобы стало ясно, как наполнить заготовку содержимым, приведем пример класса `SaveTask` (задание сохранения), выполняющего необходимые действия.

Предположим, что у нас включен импорт операторами:

```
import org.jdesktop.application.Task;
import org.jdesktop.application.Application;
```

Теперь посмотрим, как устроен класс `SaveTask` и что он делает (листинг 13.8).

Листинг 13.8. Исходный код работоспособного класса задачи `SaveTask`

```
    private class SaveTask extends Task<Object, Void> {

        SaveTask(Application app) {
            super(app);
        }
```

```
        @Override protected Object doInBackground() {
           progressBar.setVisible(true);
            try {
                for(int i = 0; i < 10; i++) {
                     setMessage("Working... [" + i + "]");
                    Thread.sleep(150L);
                    setProgress(i, 0, 9);
                }
                Thread.sleep(150L);
            } catch(InterruptedException ignore) { }
            return null;
        }

        @Override protected void succeeded(Object result) {
            // Runs on the EDT.  Update the GUI based on
            // the result computed by doInBackground().
        }
    }
```

`SaveTask` — это обычный внутренний класс, т. к. он задан без спецификатора `static` (т. е. экземпляры `SaveTask` могут существовать только внутри объекта главной формы приложения). Класс является наследником параметризованного класса `Task` и конкретизирует его параметризованные типы конкретными типами `Object` и `Void`. Следует обратить внимание, что слово `Void` в параметризации

```
Task< Object, Void>
```

написано с большой буквы.

Это имя особого класса, не имеющего экземпляров и служащего для создания ссылки на оболочечный класс для примитивного типа `void`. В нашем случае метод `doInBackground()` не имеет параметров, и в качестве списка параметров было бы логично задать `void`. Но из-за особенности задания класса `Task` мы можем задавать только объектный тип, и нам требуется тип `Void` как объектный синоним типу `void`.

Замечание

Если бы в функции `doInBackground()` в качестве возвращаемого значения также использовался тип `Void`, то в этом методе потребовался бы возврат результата данной функции в виде `return null`. В отличие от методов типа `void`, для методов типа `Void` обязателен оператор `return`, а значение `null` совместимо с типом `Void`.

В классе `SaveTask` переопределяется метод `doInBackground()`, запускающий в фоновом режиме выполнение задания. Аннотация `@Override`, как мы уже знаем, поясняет то, что родительский метод с таким же именем и такой же сигнатурой переопределен, и дает информацию об этом компилятору.

Цикл `for` обеспечивает с помощью метода `setMessage`, унаследованного от класса `Task`, вывод в строку состояния сообщения: `Working...` (Работает...) и цифру с номером шага, заключенную в квадратные скобки. Этот вывод происходит благодаря тому, что изменение свойства `message` методом `setMessage` порождает для задания событие `propertyChange`, которое и обрабатывается в соответствующем обработчике (он обсуждается далее). После каждого такого вывода поток выполнения фонового задания приос-

танавливается на 150 мс (0,15 с) методом класса `Thread.sleep(150L)`. Именно этот метод вынуждает использовать блок `try-catch`, т. к. он может порождать исключительную ситуацию `InterruptedException`.

Оператор `setProgress(i, 0, 9)` задает значение свойства `progress` нашего задания (значение `i`) и границы его изменения (от 0 до 9). Конечно, достаточно было бы назначить границы всего один раз. Но поскольку задания по показу прогресса всегда выполняются в фоновом режиме, то особая скорость не нужна. Поэтому проще каждый раз задавать эти границы и не плодить перегруженные варианты метода.

Метод `finished()` изменяет значение свойства `message` с помощью метода `setMessage`. Как и в предыдущем случае, обработчик `propertyChange` обеспечивает вывод соответствующего текста в строке состояния.

Рассмотрим теперь код обработчика события `propertyChange`, вызываемого по событию изменения свойств нашего объекта-задания из управляющего объекта `taskMonitor`. Объект-обработчик создается с помощью безымянного класса и, как обычно, добавляется в список слушателей событий (листинг 13.9).

Листинг 13.9. Добавление обработчика события `propertyChange` в список слушателей событий

```
taskMonitor.addPropertyChangeListener(new
                            java.beans.PropertyChangeListener() {
    public void propertyChange(java.beans.PropertyChangeEvent evt) {
        String propertyName = evt.getPropertyName();
        if ("started".equals(propertyName)) {
            if (!busyIconTimer.isRunning()) {
                statusAnimationLabel.setIcon(busyIcons[0]);
                busyIconIndex = 0;
                busyIconTimer.start();
            }
            progressBar.setVisible(true);
            //progressBar.setIndeterminate(true);
        } else if ("done".equals(propertyName)) {
            busyIconTimer.stop();
            statusAnimationLabel.setIcon(idleIcon);
            progressBar.setVisible(false);
            progressBar.setValue(0);
        } else if ("message".equals(propertyName)) {
            String text = (String)(evt.getNewValue());
            statusMessageLabel.setText((text == null) ? "" : text);
            messageTimer.restart();
        } else if ("progress".equals(propertyName)) {
            int value = (Integer)(evt.getNewValue());
            progressBar.setVisible(true);
            progressBar.setIndeterminate(false);
            progressBar.setValue(value);
        }
    }
});
```

Строковая переменная `propertyName` хранит имя свойства, изменение которого породило событие. Далее содержимое переменной сравнивается со значениями строк `"started"`, `"done"`, `"message"` и `"progress"`. Метод `equals` позволяет сравнивать не ссылки на объекты-строки, а именно значения:

```
if ("started".equals(propertyName))
...
if ("done".equals(propertyName))
...
```

и т. д.

Почему сравнение ведется таким странным образом? Почему не использовать более естественную форму записи:

```
if (propertyName.equals("started"))
...
if (propertyName.equals("done"))
...
```

В Java данный паттерн оператора `equals` необходим по следующей причине:

- если сравниваются значения двух объектных переменных `obj1` и `obj2`, то сравнения `obj1.equals(obj2)` и `obj2.equals(obj1)` будут эквивалентны только в том случае, когда обе эти переменные имеют значения, отличные от `null`;
- если `obj1` может быть `null`, а `obj2` является неизменяемым (константным) объектом, то следует делать проверку `obj2.equals(obj1)`, поскольку попытка вызова метода `equals` с помощью переменной `obj1` вызовет `NullPointerException` в случае, когда `obj1==null`.

Имя первого свойства компонента, которое обрабатывается, — `"started"` (стартовали). Соответствующее событие, как уже было сказано, возникает сразу после старта задания. Сначала проверяется, запущен ли таймер `busyIconTimer`, и если он не запущен (`!busyIconTimer.isRunning()`), то устанавливается первая иконка из массива вращающихся иконок (индекс 0) и запускается таймер. Затем показывается индикатор выполнения.

После этого идет оператор `progressBar.setIndeterminate(true)` установки индикатора выполнения для показа недетерминированных величин. По умолчанию включен режим `progressBar.setIndeterminate(false)`, а показ идет для величин, границы изменения которых известны. Перевод в индетерминированный режим приводит к тому, что индикатор выполнения при каждой новой установке значения автоматически циклически меняет положение столбика, отмечающего ход выполнения задания (прогресс). В нашем случае границы изменения детерминированы, и перевод в индетерминированный режим нарушает правильный показ хода выполнения. Поэтому строку с данным оператором следует закомментировать, как это сделано в листинге 13.9.

Следующая проверка относится к изменению свойства `done` (сделано), т. е. к событию завершения работы задания. В этом случае таймер `busyIconTimer` останавливается, в метке `statusAnimationLabel` показывается иконка "бездействие системы", индикатор выполнения скрывается, и свойству `progress` присваивается значение 0 (чтобы в следующем цикле показ начинался с нулевой длины столбика).

Далее проверяется изменение свойства `message` (сообщение). Вообще говоря, можно было бы узнать новое значение у объекта-задания с помощью метода `getMessage()`, но создатели заготовки приложения пошли другим путем. Они не хранят ссылку на объект-задание, а добывают значение непосредственно из объекта-события с помощью вызова `(String)( evt.getNewValue() )`. В этом выражении функция `evt.getNewValue()` возвращает ссылку на новое значение свойства. Но ссылка имеет тип `Object`, поэтому требуется преобразование к типу `String`. Скобки, окружающие вызов функции `evt.getNewValue()`, можно не ставить, но тогда не очень ясно, к какому объекту применяется преобразование типа — к возвращаемому результату или к объекту `evt`.

Если бы нам было необходимо узнать прежнее значение свойства, то следовало вызвать функцию `evt.getOldValue()`.

Замечание

Такой способ выяснения в обработчике `propertyChanged` нового и старого значения свойств относится ко всем вообще свойствам, а не только к свойству `message` нашего объекта-задания.

Полученный текст сообщения выводится в метку `statusMessageLabel` оператором:

```
statusMessageLabel.setText((text == null) ? "" : text);
```

В нем предусмотрены два случая:

- если ссылка на строку сообщения имеет значение `null`, то выводится пустая строка;
- если ссылка указывает на строку (пусть даже пустую — это все равно существующий объект!), то выводится эта строка.

Далее оператор `messageTimer.restart()` заново запускает таймер сообщения (напомним, что он работает не в циклическом, а в однократном режиме, в отличие от циклически работающего `busyIconTimer`).

Наконец, последняя проверка относится к событию `progress`, которое циклически порождается в классе `DoNothing` оператором `setProgress(i, 0, 9)`.

Оператор

```
int value = (Integer)(evt.getNewValue())
```

уже известным нам способом получает новое значение свойства из объекта-события `evt`.

Поскольку возвращаемое значение обязано быть объектом, а свойство целочисленное, то используется объект оболочечного класса `Integer`. При присваивании целочисленной переменной `value` оболочечный объект автоматически распаковывается (unboxing).

Затем индикатор выполнения оператором `progressBar.setVisible(true)` делается видимой, переводится в детерминированный режим оператором `progressBar.setIndeterminate(false)` (зачем, спрашивается, было из него выходить?) и устанавливается длина столбика индикатора выполнения оператором `progressBar.setValue(value)`.

Выбрав пункт меню **File/Save** в примере, приведенном на DVD (13_06_DesktopApplication1), можно наблюдать работу фонового потока (рис. 13.2 и 13.3).

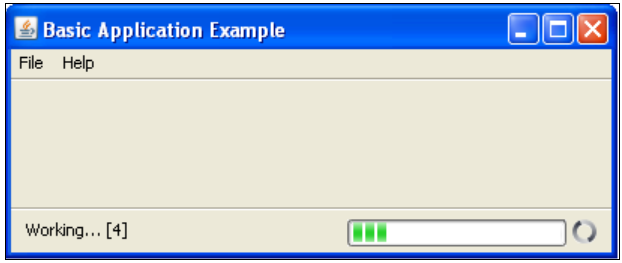


Рис. 13.2. Индикация процента выполнения задачи, i=4

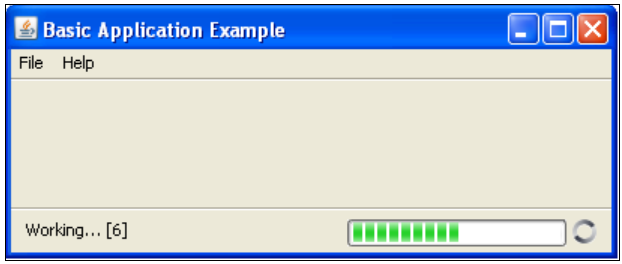


Рис. 13.3. Индикация процента выполнения задачи, i=6

13.7. Создание задачи типа *Task*. Назначение иконок и добавление изображений

До сих пор мы говорили о том, как устроено приложение с фоновым потоком выполнения. Теперь научимся создавать на основе заготовки **DesktopApplication** такие приложения.

Создадим заготовку **DesktopApplication**. Добавим в меню **File** новый пункт: выберем в контекстном меню (после щелчка правой кнопкой мыши по этому пункту) **Add From Palette/Menu Item**; назначим этому пункту надпись **Save** (Сохранить) и переместим его перед пунктом **Exit** (Выход); вызовем для этого пункта контекстное меню и выберем в нем пункт **Set Action...** (рис. 13.4).

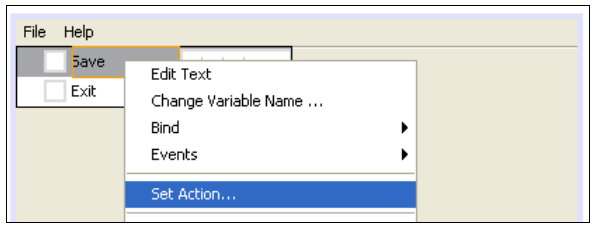


Рис. 13.4. Формирование фоновой задачи. Шаг 1 — вызов формы задания акции

Появится диалоговая форма, в которой в поле **Action** стоит **<none>** (отсутствует).

Следует вызвать выпадающий список и выбрать в нем последний пункт — **Create New Action...** (Создать новую акцию) — как это показано на рис. 13.5.

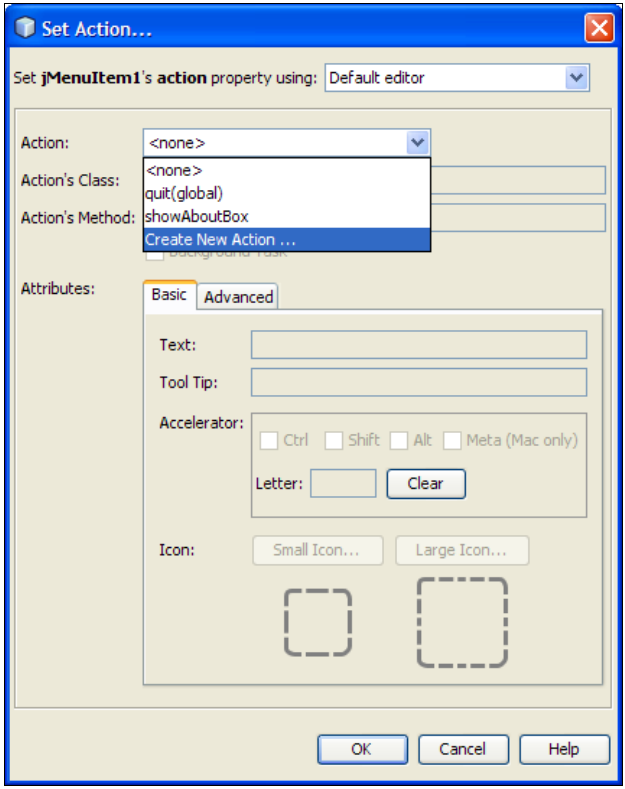


Рис. 13.5. Формирование фоновой задачи. Шаг 2 — создание новой акции

В результате в поле **Action's Class** будет установлено значение **desktopapplication1.DesktopApplication1View** (но в выпадающем списке его можно заменить на **desktopapplication1.DesktopApplication1**), а поля **Action's Method** (Метод, сопоставленный акции), **Background Task** (Фоновая задача) и **Attributes** (Атрибуты) станут доступными для изменения (рис. 13.6).

В поле **Action's Method** введем имя **save**, которое хотим дать нашей акции. Установим флажок **Background Task** (иначе будет создана заготовка обычного метода, без создания фоновой задачи) (рис. 13.6).

Элементы вкладки **Attributes** определяют внешний вид меню.

Поле **Text** отвечает за надпись для компонента, которому назначена акция (в нашем случае — для пункта меню).

Tool Tip — всплывающая подсказка для пункта.

Accelerator — горячие клавиши *(см. разд. 2.11)*.

Также акции можно назначить кнопки: **Small Icon...** (малую) и **Large Icon...** (большую).

Иконка — это картинка размером 16×16 пикселов (маленькая иконка) или 32×32 пикселов (большая), хотя реально среда NetBeans поддерживает произвольный размер иконок.

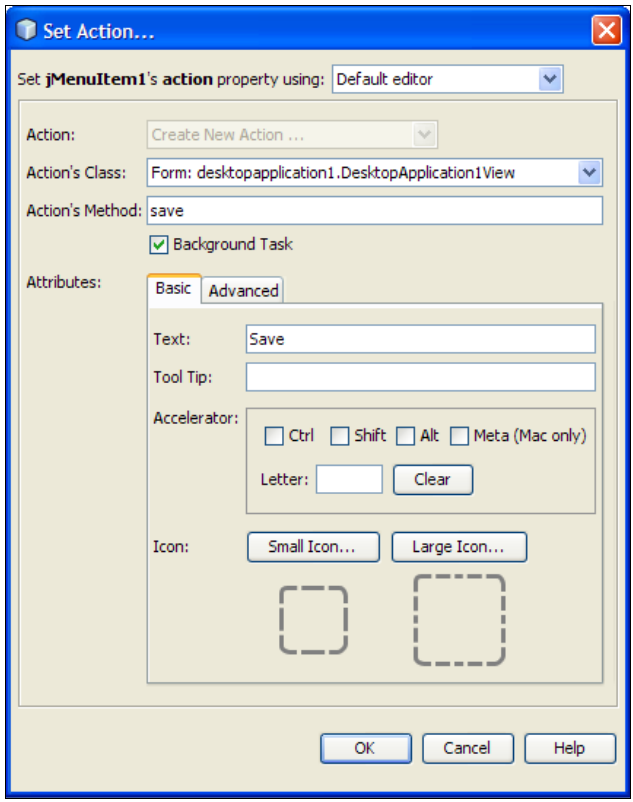


Рис. 13.6. Формирование фоновой задачи. Шаг 3 — окончание создания заготовки задачи

После нажатия кнопки **OK** в исходном коде появится акция `Save()` типа `Task`:

```
@Action
public Task save() {
    return new SaveTask(getApplication());
}
```

Кроме того, появится заготовка встроенного класса `SaveTask`, исходный код которого приводился в предыдущем разделе. Акция выполняет единственное действие — создает и возвращает объект типа `SaveTask`.

Мы уже знаем, как можно заполнить данную заготовку, чтобы на ее основе создать работающее приложение.

Немного усовершенствуем наш проект. Из палитры компонентов **Swing Containers** добавим на форму панель инструментов **JToolBar**. Затем разместим на панели инструментов кнопку **JButton** и назначим этой кнопке уже существующую акцию **Save** — она появится в выпадающем списке акций в диалоге **Set Action...**, и нужно будет выбрать ее вместо **Create New Action...**.

Назначим в качестве иконок изображения: **save.png** и **save24.png** с помощью кнопок **Small Icon...** и **Large Icon...**. Нажатие такой кнопки приводит к появлению диалога, в котором можно выбрать пункт **Image Within Project** (изображение, уже существую-

щее в проекте) — имена файлов появляются в выпадающем списке **File**, в котором по умолчанию показывается надпись **Select...** (Выбрать) — рис. 13.7.

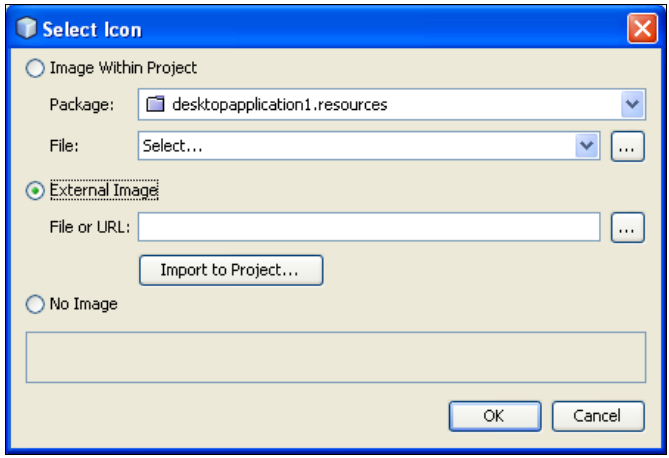


Рис. 13.7. Диалог назначения иконки

Но мы добавляем внешние файлы в проект, импортировав изображение в проект с помощью кнопки **Import to Project...**. В этом случае не нужно будет заботиться о том, чтобы данное изображение распространялось (deploy) вместе с приложением. Если изображение не добавить в проект, то на другом компьютере данное изображение не найдется.

При запуске проекта 13_07_DesktopApplication2 мы обнаружим назначение больших и маленьких иконок: большие показываются на кнопках панели инструментов, а маленькие — в пунктах меню.

Теперь добавим таким же способом кнопки **Copy** (Копировать) и **Paste** (Вставить) с созданием соответствующих акций.

И наконец, чтобы проект приобрел вид, нужный нам для дальнейшей работы, добавим в основную область экранной формы изображение. Для этого поместим на форму метку и растянем ее так, чтобы она занимала всю рабочую область. Напомним, что по канве любого компонента `Swing` можно рисовать, и метка может служить для вывода не только текста, но и изображений.

Щелкнем в окне **Properties** по кнопке ..., стоящей справа от поля **icon** (рис. 13.8). Откроется форма (рис. 13.9), которая очень похожа на форму выбора иконки, с которой мы уже знакомы (см. рис. 13.7), но имеет несколько дополнительных полей.

Следует установить флажок **Define as a Resource** (Задать как ресурс), что позволяет добавить выбранный файл в список ресурсов.

Установка флажка **Localized** дает возможность показывать различные изображения для разных языковых форматов, что заметно упрощает создание локализованных приложений.

С помощью выпадающего списка **Key** (Ключ) можно заменить содержимое, показываемое меткой, на поля (ключи) других компонентов проекта — изображения или текст. В нашем случае этого лучше не делать.

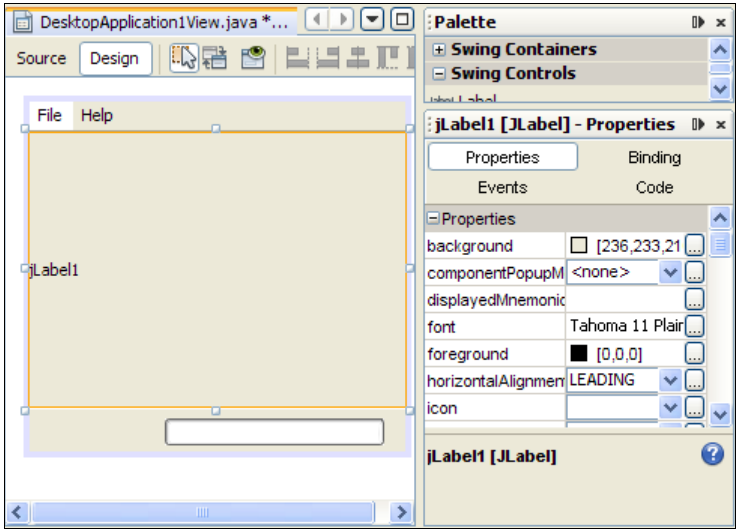


Рис. 13.8. Назначение метке изображения

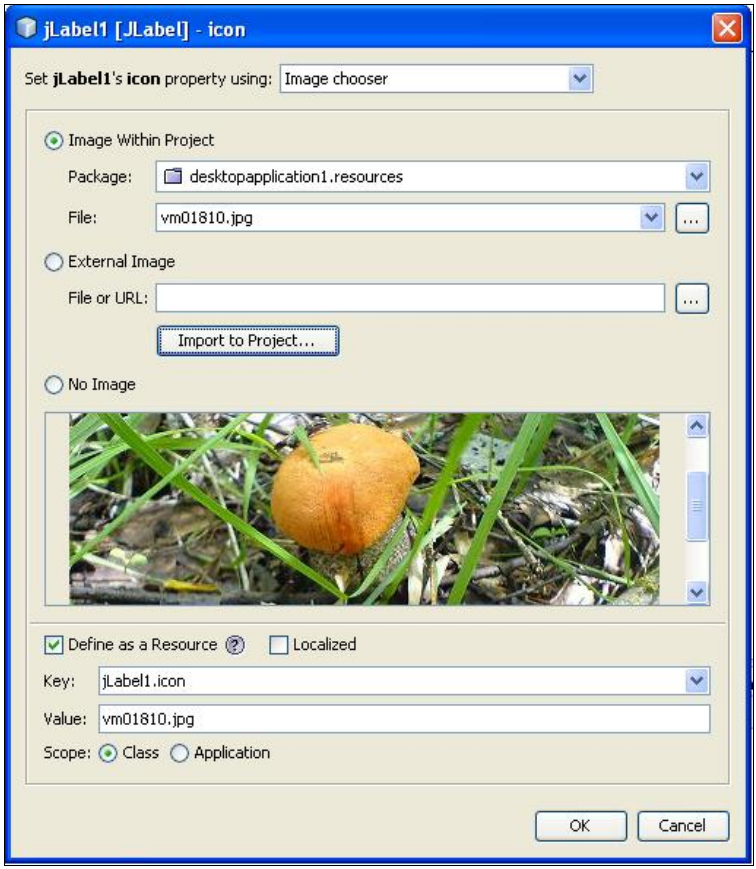


Рис. 13.9. Приложение с изображением, назначенным метке

Поле **Value** (Значение) позволяет назначить изображению в проекте новое имя (по умолчанию сохраняется старое).

Радиогруппа **Scope** (Область видимости) (в нижней части формы) позволяет задать правило видимости изображения в проекте — на уровне класса или всего приложения.

Добавить в проект изображение также можно его копированием в папку соответствующего пакета, в результате чего оно появится в дереве проекта в пакете ресурсов. В этой же папке можно найти файлы изображений, имеющиеся в проекте. Например, вы можете найти сделанную автором фотографию гриба (файл vm01810.jpg) в папке 13_07_DesktopApplication3\src\desktopapplication1\resources.

После назначения метке изображения наше приложение при сохранении данных (нажатии на кнопку панели инструментов с изображением дискеты) будет выглядеть так, как на рис. 13.9 (при нажатии на эту кнопку идет имитация работы, отображаемая в строке состояния, но реально никакие другие действия не выполняются).

13.8. Локализация приложения

Мы уже знаем, каким образом в Java создаются локализованные версии серверных приложений. Приложения, предназначенные для работы на обычных компьютерах (рабочих станциях), локализуются аналогично. В качестве языкового формата по умолчанию используется английский язык. Нам достаточно добавить русский языковый формат. Но сначала отредактируем файл свойств DesktopApplication1.properties для языка по умолчанию (default language). Содержимое этого файла приведено в листинге 13.10.

Листинг 13.10. Содержимое автоматически сгенерированного файла свойств `DesktopApplication1.properties` для языка по умолчанию

```
# Application global resources

Application.name = DesktopApplication1
Application.title = Basic Application Example
Application.version = 1.0
Application.vendor = Sun Microsystems, Inc.
Application.homepage = http\://appframework.dev.java.net
Application.description = A simple java desktop application based on Swing
Application Framework
Application.vendorId = Sun
Application.id = ${Application.name}
Application.lookAndFeel = system
```

Заменим информацию о производителе программы (vendor), о домашней странице приложения (homepage), идентификаторе производителя (vendorID). Идентификатор приложения получается автоматически — это имя приложения, заданное в первой строчке. Знак доллара с фигурными скобками означает, что внутри фигурных скобок стоит выражение, значение которого нужно вычислить (как в технологии JSP). Обратите внимание, что внешний вид приложения **lookAndFeel** можно задавать непосредственно в файле свойств.

У автора получился файл, приведенный в листинге 13.11.

Листинг 13.11. Содержимое файла свойств класса `DesktopApplication1` для языка по умолчанию после внесения информации пользователя

```
# Application global resources

Application.name = DesktopApplication1
Application.title =  Example of Localized Application
Application.version = 1.0
Application.vendor = Vadim V. Monakhov
Application.homepage = http\://barsic.spbu.ru/
Application.description = A simple localized java desktop application based on Swing
Application Framework
Application.vendorId = Monakhov
Application.id = ${Application.name}
Application.lookAndFeel = system
```

Естественно, читатели, выполняющие данное упражнение, должны подставить свои собственные данные.

Теперь найдем в дереве проекта в пакете **desktopapplication1.resources** имя файла свойств **DesktopApplication1.properties** и щелкнем по нему правой кнопкой мыши. Выберем пункт контекстного меню **Add Locale...** (добавить языковый формат), в появившейся форме в списке предопределенных форматов выберем **ru_RU — русский / Россия** и нажмем кнопку **OK**.

В файле свойств для русского языкового формата исправим текст на русский (листинг 13.12).

Листинг 13.12. Содержимое файла свойств класса `DesktopApplication1` для русского языкового формата после внесения информации пользователя

```
# Application global resources

Application.name = DesktopApplication1
Application.title =  Пример локализованного приложения
Application.version = 1.0
Application.vendor = Вадим Валериевич Монахов
Application.homepage = http\://barsic.spbu.ru/
Application.description = Простой пример локализованного приложения, основанного на
Swing Application Framework
Application.vendorId = Monakhov
Application.id = ${Application.name}
Application.lookAndFeel = system
```

Идентификаторы приложения и производителя лучше оставлять одинаковыми в разных языковых форматах, за исключением случаев, когда ваша фирма имеет в разных странах разные названия.

При запуске приложения мы обнаружим, что работа по локализации приложения не доведена до конца. Заголовок стал русским, но пункты меню и всплывающие подсказки остались английскими. Это связано с тем, что главная экранная форма приложения пока не локализована. Для ее локализации требуется добавить русский языковый формат в файл свойств этой формы DesktopApplication1Frame.properties и отредактировать его.

Вариант для языка по умолчанию приведен в листинге 13.13.

Листинг 13.13. Содержимое файла свойств класса `DesktopApplication1Frame` для языка по умолчанию

```
# Resources for the DesktopApplication1View class

# top-level menus

fileMenu.text = File
helpMenu.text = Help

# @Action resources

showAboutBox.Action.text = &About...
showAboutBox.Action.shortDescription = Show the application's information dialog

# status bar resources

StatusBar.messageTimeout = 5000
StatusBar.busyAnimationRate = 30
StatusBar.idleIcon = busyicons/idle-icon.png
StatusBar.busyIcons[0]  = busyicons/busy-icon0.png
...[1]

jMenuItem1.text=Save

save.Action.text=Save

save.Action.shortDescription=Save file

jButton1.text=

save.Action.accelerator=ctrl pressed s

save.Action.largeIcon=save24.png

save.Action.smallIcon=save.png

save.Action.icon=save.png

...
```

[1] Файлы свойств довольно длинные, поэтому приведены только их фрагменты. В листингах 13.13 и 13.14 пропущенные места заменены многоточием.

```
jLabel1.text=

#NOI18N
jLabel1.icon=vm01810.jpg
```

Вариант для русского языкового формата будет выглядеть так, как показано в листинге 13.14.

Листинг 13.14. Содержимое файла свойств класса `DesktopApplication1Frame` для русского языкового формата

```
# Resources for the DesktopApplication1View class

# top-level menus

fileMenu.text = Файл
helpMenu.text = Справка

# @Action resources

showAboutBox.Action.text = &О программе...
showAboutBox.Action.shortDescription = Показывает форму с информацией о программе

# status bar resources

StatusBar.messageTimeout = 5000
StatusBar.busyAnimationRate = 30
StatusBar.idleIcon = busyicons/idle-icon.png
StatusBar.busyIcons[0]  = busyicons/busy-icon0.png
...

jMenuItem1.text=Сохранить

save.Action.text=Сохранить

save.Action.shortDescription=Сохранить файл

jButton1.text=

save.Action.accelerator=ctrl pressed s

save.Action.largeIcon=save24.png

save.Action.smallIcon=save.png

save.Action.icon=save.png

...

jLabel1.text=

#NOI18N
jLabel1.icon=vm01810.jpg
```

Следует обратить внимание на символ `&` (амперсанд), встречающийся в тексте для пунктов меню, например, перед пунктом `About` (О программе). Символ, перед которым поставлен знак амперсанда, становится акселератором.

Мы не будем останавливаться на достаточно простой локализации формы справки, она проделывается аналогично. При запуске приложения кажется, что все замечательно и локализация завершена. Но, как обычно бывает в программировании, это заблуждение. Попробуйте нажать кнопку **Save**, и вы обнаружите появление нелокализованной надписи **Working...** в строке состояния. Как ее локализовать?

Эта надпись выводится в методе `doInBackground()` класса `DoNothing`, который расположен в классе `DesktopApplication1View`, поэтому мы будем менять ресурсы класса `DesktopApplication1View`. В файл свойств этого класса для языка по умолчанию после строки:

```
StatusBar.busyIcons[14] = busyicons/busy-icon14.png
```

добавим строки:

```
StatusBar.Working = Working
StatusBar.Done = Done.
```

Аналогично, в файл для русского языкового формата добавим:

```
StatusBar.Working = Идет работа
StatusBar.Done = Выполнено.
```

Теперь нам нужно в методе `doInBackground()` вместо фиксированных строк вывести локализованные значения, полученные из таблицы, на которую ссылается переменная `resourceMap`. Но в оригинальной заготовке она существует только в конструкторе. Поэтому мы вынесем задание этой переменной из конструктора в сам класс `DesktopApplication1View`, а инициализацию переменной проведем в конструкторе, т. е. вместо фрагмента кода из листинга 13.15 напишем код, приведенный в листинге 13.16.

Листинг 13.15. Задание переменной `resourceMap` внутри конструктора

```
public DesktopApplication1View () {
    super(app);
    initComponents();
    // status bar initialization
    ResourceMap resourceMap = getResourceMap();
```

Листинг 13.16. Задание переменной `resourceMap` в качестве глобальной

```
ResourceMap resourceMap;

public DesktopApplication1View () {
    super(app);
    initComponents();
    // status bar initialization
    resourceMap = getResourceMap();
```

А в классе `SaveTask` заменим

```
setMessage("Working... [" + i + "]");
```

на

```
setMessage(resourceMap.getString("StatusBar.Working")+"... [" + i + "]");
```

И вместо

```
setMessage("Done.");
```

напишем:

```
setMessage(resourceMap.getString("StatusBar.Done"));
```

В примере 13_08 _DesktopApplication2 реализован описанный пример локализации приложения, но локализация не доведена до конца — эту работу мы предоставляем доделать читателю.

Имеется один момент, на который следует обратить внимание. Если мы проведем рефакторинг, затрагивающий имена, задействованные в файлах свойств, то возникнут проблемы с соответствием имен. Имена в файлах свойств придется менять вручную. Почему? Ведь мы провели рефакторинг! Но преобразования при рефакторинге — это не волшебство, а использование возможностей жесткой типизации языка. Такого рода действия из распространенных языков программирования возможны только для Java и Pascal. Другие языки, такие как C, C++, Basic и JavaScript, имеют слишком слабую типизацию, для них невозможно обеспечить однозначность преобразований. А файлы свойств — это просто текст, и ни о какой типизации не может идти речи.

13.9. Измерение времени и досрочное прекращение выполнения задания

Достаточно часто при длительном выполнении заданий требуется узнать, сколько времени прошло с его начала.

Для этого необходимо задать переменную, ссылающуюся на наш объект-задачу:

```
Task<Object, Void> task;
```

Затем в конструкторе задачи следует настроить ссылку на создаваемый объект задачи, добавив оператор после вызова прародительского конструктора:

```
SaveTask(org.jdesktop.application.Application app) {
    super(app);
    task=this;
}
```

И, наконец, в том месте программы, где мы хотим получить ответ в секундах, нужно написать вызов:

```
long t=task.getExecutionDuration(java.util.concurrent.TimeUnit.SECONDS);
```

Если мы используем нажатие на кнопку для вывода в окно консоли полученного значения, то код обработчика события будет выглядеть так, как в листинге 13.17.

Листинг 13.17. Вывод в консоль информации о времени выполнения

```
private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    if (task==null){
        System.out.println("t=0 сек, задание не было запущено");
        return;
    }

    System.out.println("t=" + (float)task.getExecutionDuration(
              java.util.concurrent.TimeUnit.MILLISECONDS)/1000 +" сек");
}
```

Обратите внимание на обязательность проверки на `null` ссылки `task`. Если бы мы не сделали этой проверки, то нажатие на кнопку во время бездействия системы привело бы к возбуждению исключительной ситуации `NullPointerException` методом `task.getExecutionDuration`.

Для досрочного прекращения задания следует вызвать метод `cancel`, унаследованный от класса `org.jdesktop.swingworker.SwingWorker`:

```
private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
    if (task!=null){
        task.cancel(true);
    }
}
```

Пример 13_09_DesktopApplication1 приложения со сделанными добавлениями можно найти на прилагаемом к книге DVD.

Реакцию задачи на досрочное прекращение (реализацию метода `cancelled`) читателю рекомендуется написать самостоятельно.

13.10. Работа с межпрограммным буфером обмена — Clipboard

Копирование строк и изображений из буфера обмена в программу

Пакет **java.awt.datatransfer** и, в частности, заданный в нем класс `Clipboard` обеспечивают поддержку работы с буфером обмена. Мы будем рассматривать работу с двумя форматами данных: текстом и изображениями.

Для начала рассмотрим операцию получения текстовой или графической информации из буфера обмена в приложение (пример 13_10_DesktopApplication1).

Разместим на форме компонент `JTextArea` (текстовая область, компонент многострочного ввода и редактирования текста). Установим размер шрифта 12, чтобы текст не был слишком мелким.

Затем напишем метод с именем `doPaste()` (выполнить вставку). Это нужно сделать, чтобы организовать правильную работу с индикатором занятости. Соответствующий код приведен в листинге 13.18.

Листинг 13.18. Копирование текстовой и графической информации из буфера обмена в приложение

```
import java.awt.Image; //vm
import javax.swing.ImageIcon;//vm
import java.awt.datatransfer.*; //vm
...

org.jdesktop.application.ApplicationContext appContext =
        Application.getInstance(DesktopApplication1.class).getContext();
Clipboard clipboard = appContext.getClipboard();
Image img = null;
ImageIcon icon = null;

public void  doPaste () {
   System.out.println("do paste");
   try{
      Transferable clipData = clipboard.getContents(null);
      if (clipData.isDataFlavorSupported(DataFlavor.stringFlavor)){
          String s=(String)clipData.
                              getTransferData(DataFlavor.stringFlavor);
          jTextArea1.replaceSelection(s);
      }else
          if (clipData.isDataFlavorSupported(DataFlavor.imageFlavor)){
              img = (Image)clipData.
                              getTransferData(DataFlavor.imageFlavor);
              if (img!=null){
                  icon = (ImageIcon)jLabel1.getIcon();
                  icon.setImage(img);
              }
          }
   } catch(Exception e){
   }
}
```

Сначала, как обычно, мы импортируем необходимые интерфейсы. Затем вводим и инициализируем переменную `clipboard`, которая дает доступ к буферу обмена. Кроме того, мы вводим переменную `img` для временного (промежуточного) сохранения вставляемого изображения в формате `Image`. Разработчики Sun сделали два различных формата работы с изображениями — с использованием абстрактного класса `Image` (базового для классов, занимающихся обработкой изображений), а также с помощью интерфейса `Icon` и класса `ImageIcon` (для показа иконок в графическом пользовательском интерфейсе).

Метод `doPaste()` мы начинаем с отладочного оператора `System.out.println("do paste")`. Такого рода вывод в консольное окно будем делать для того, чтобы лучше видеть последовательность выполнения методов в многопоточной среде.

Блок `try-catch` обязателен только для операторов `getTransferData`. Но мы заключили в него все рабочие операторы метода `paste()`, чтобы не писать двух разных блоков `try-catch`, загромождающих программу.

Оператор `Transferable clipData = clipboard.getContents(null)` выдает нам содержимое буфера обмена в некотором внутреннем формате Java. В методе `clipboard.getContents` должен быть параметр типа `Object`. Но, согласно документации JDK, он пока не используется, поэтому мы поставили вместо него `null`.

Далее мы проверяем, имеется ли в буфере обмена текстовое содержимое либо содержимое, совместимое с типом `Image`. Эту информацию возвращает булева функция `isDataFlavorSupported(DataFlavor flavor)` (поддерживается ли вид данных).

Здесь:

- `Data` — данные;
- `Supported` — поддерживается;
- `Flavor` — в технической литературе обычно означает "вид", "разновидность".

Значение, равное `DataFlavor.stringFlavor`, означает, что в буфере находится строковый вид данных. А если оно равно `DataFlavor.imageFlavor`, то там находится изображение.

Строка:

```
String s=(String)clipData.getTransferData(DataFlavor.stringFlavor)
```

обеспечивает преобразование данных из промежуточного формата, в котором они находятся в объекте `clipData`, в строковый формат.

Тип, к которому идет преобразование, задается параметром функции `getTransferData`, преобразующей данные (`Transfer` означает "передача", "перевод", "перенос").

Но для того чтобы для каждого типа данных не вводить свой оператор, возвращаемое значение имеет тип `Object`, а для этой функции обязательно приведение типа. В данном случае оно приводится к типу `String`, а для изображений — к типу `Image`.

Оператор `jTextArea1.replaceSelection(s)` заменяет текст, выделенный в компоненте `jTextArea1`, содержимым, извлеченным из буфера. Если выделения нет, то текст вставляется на то место, где стоит курсор.

Фрагмент, относящийся к вставке изображений, при работе с буфером обмена отличается только именами константы и типом данных, к которым идет преобразование. Но вот назначение изображения метке имеет свою специфику, связанную с различием типов `Image` и `ImageIcon`. Изображение типа `Image`, полученное из буфера обмена, мы должны назначить метке `jLabel1`.

Это делается так:

сначала мы получаем ссылку на объект-иконку, назначенную метке:

```
icon = (ImageIcon)jLabel1.getIcon();
```

Затем этой иконке назначается изображение типа `Image`:

```
icon.setImage(img);
```

На этом работа с копированием из буфера межпрограммного обмена заканчивается.

Теперь мы можем вызывать метод `doPaste()` в любом месте, где это необходимо (например, из акции `paste` в методе `doInBackground()`):

```
@Override
protected Object doInBackground() {
```

```
            doPaste();
            return null;
        }
```

К сожалению, пока в JDK поддерживаются только два формата, читаемые из буфера обмена, — строки и изображения. Но с записью дело обстоит еще хуже — стандартными средствами копирование в буфер поддерживается только для текста.

Копирование строк из программы в буфер обмена. Класс *StringSelection*

Рассмотрим пример такого копирования. Напишем метод `copyStringToClipboard()` (листинг 13.19) и расположим его перед акцией `copy`.

Листинг 13.19. Копирование строки в буфер обмена

```
public void  copyStringToClipboard() {
    System.out.println("copyStringToClipboard ");
    String s= jTextArea1.getSelectedText();
    if (s==null)
        return;
    StringSelection clipData=new StringSelection(s);
    clipboard.setContents(clipData,clipData);
}
```

В методе сначала идет отладочный оператор вывода в консольное окно (в рабочей версии его, конечно, нужно убрать).

Следующий оператор считывает в строковую переменную `s` текст, выделенный в компоненте `jTextArea1`. Если выделения нет, то выходим из подпрограммы. Это сделано для того, чтобы избежать проблем использования пустой строки в качестве объекта.

Далее мы создаем из строки `s` объект типа `StringSelection`, который необходим оператору `setContents` для передачи данных в буфер обмена. Без объектов этого формата вполне можно было бы обойтись при разумном построении модели обмена с буфером.

И наконец, оператор `clipboard.setContents(clipData,clipData)` заносит в буфер содержимое строки. При этом первый параметр — это передаваемые данные, а второй — управляющий объект, связанный с содержимым буфера.

При добавлении вызова `copyStringToClipboard()` в метод `doInBackground()` акции `copy` или, например, в обработчик события нажатия на кнопку выделенный текст скопируется из компонента `jTextArea1` в буфер обмена. То, что это произошло, легко проверить в том же приложении, нажав кнопку **Paste**.

Контрольный вопрос

> Заметил ли читатель, что метод `copyStringToClipboard()` написан идеологически неправильно? Что в нем нужно исправить и почему? Еще раз просмотрите его исходный код и выскажите свои соображения.

Ответ достаточно прост: общий по идеологии метод ориентирован на конкретный способ реализации пользовательского интерфейса. Правильное решение — сделать метод

с параметром типа `String`, через который в метод передается копируемая в буфер обмена строка. И уже в одном конкретном случае будет делаться вызов:

```
copyStringToClipboard(jTextArea1.getSelectedText());
```

Реализацию данной идеи оставляем читателю.

Рассмотрим устройство класса `StringSelection`, чтобы лучше понять, как создавать аналогичные классы для поддержки обмена другими видами данных через буфер обмена.

Класс реализует интерфейсы `Transferable` и `ClipboardOwner`.

Метод `public DataFlavor[] getTransferDataFlavors()` возвращает массив видов данных (flavors), поддерживаемых данным классом. Это два типа: `DataFlavor.stringFlavor` и устаревший, не рекомендуемый к использованию (deprecated) вид `DataFlavor.plainTextFlavor`.

Метод `public boolean isDataFlavorSupported(DataFlavor flavor)` осуществляет проверку, поддерживает ли данный класс вид данных `flavor`. Если это один из указанных ранее форматов, то возвращается `true`, иначе — `false`. Если `flavor==null`, то возбуждается исключение `NullPointerException`.

Метод `public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException` преобразует строку в формат типа `Object` и возбуждает исключение `UnsupportedFlavorException` в том случае, когда в метод передается переменная `flavor` для вида данных, не поддерживаемых классом `StringSelection`. Исключение `IOException` классом `StringSelection` никогда не возбуждается и зарезервировано лишь для того, чтобы оно могло возбуждаться каким-нибудь потомком этого класса.

Метод `public void lostOwnership(Clipboard clipboard, Transferable contents)` (потерять владение) предназначен для поддержки интерфейса `ClipboardOwner` (владелец буфера обмена). Такое владение необходимо только для разделяемых между разными программами (shared) ресурсов. Например, при вырезании файлов в буфер обмена (операция `cut`). Для строк такое владение не нужно, поэтому для класса `StringSelection` метод имеет пустую реализацию.

Копирование изображений из программы в буфер обмена. Создание класса *ImageSelection*

Для того чтобы обеспечить поддержку копирования изображений в буфер обмена (пример 13_10_DesktopApplication2), сначала создадим пакет `clipboardsupport`, а в нем класс `ImageSelection`, по функциональности аналогичный классу `StringSelection`, но работающий не со строками, а с изображениями (листинг 13.20).

Листинг 13.20. Класс `ImageSelection`, обеспечивающий поддержку копирования изображений из приложений в буфер обмена

```
/*
 * ImageSelection.java
 *
 * Created on 01.09.2007, 14:33:32
 */
```

```
package clipboardsupport;
/**
 * @author Vadim Monakhov
 */

import java.awt.Image;
import java.awt.datatransfer.*;

/**
 *
 * A <code>Transferable</code> which implements the capability required
 * to transfer an <code>Image</code>.
 *
 * This <code>Transferable</code> supports
 * <code>DataFlavor.imageFlavor</code>
 *
 * @see java.awt.datatransfer.DataFlavor#imageFlavor
 * @see java.awt.datatransfer.Transferable
 */
public class ImageSelection implements Transferable, ClipboardOwner {
    private Image image;

    /**
     * Creates a <code>Transferable</code> capable of transferring
     * the specified <code>Image</code>.
     */
    public ImageSelection(Image image) {
        this.image = image;
    }

    private static final DataFlavor[] flavors = {
        DataFlavor.imageFlavor
    }

    /**
     * Returns an array of flavors in which this
     * <code>Transferable</code>
     * can provide the data. <code>DataFlavor.imageFlavor</code>
     * is the single properly supported flavor.
     *
     * @return an array of length one, whose element is
     *         <code>DataFlavor.imageFlavor</code>
     */
    public DataFlavor[] getTransferDataFlavors() {
        return flavors.clone();
    }

    /**
     * Returns whether the requested flavor is supported
     * @return <b>true</b> if <code>flavor</code> is equal to
     *   <code>DataFlavor.imageFlavor</code> otherwise <b>false</b>
     * @throws NullPointerException if flavor is <code>null</code>
     */
```

```
    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return flavor.equals(DataFlavor.imageFlavor);
    }

    /**
     * Transfers image to the <code>Transferable</code>'s data format
     * if the requested <code>flavor</code> is
     * <code>DataFlavor.Image</code>.
     * @param flavor the requested flavor for the data
     * @return the data in the requested <code>DataFlavor.Image</code>
     * flavor
     * @throws UnsupportedFlavorException if the requested data flavor is
     *         not equivalent to <code>DataFlavor.Image</code>
     * @throws NullPointerException if flavor is <code>null</code>
     * @see java.awt.datatransfer.DataFlavor
     */
    public Object getTransferData(DataFlavor flavor)
            throws UnsupportedFlavorException {
        if (!flavor.equals(DataFlavor.imageFlavor)) {
            throw new UnsupportedFlavorException(flavor);
        }
        return image;
    }

    /**
     *Implements the only method of
     * the <code>ClipboardOwner</code> interface.
     */
    public void lostOwnership(Clipboard clipboard,Transferable contents){
    }

}
```

Поскольку разрабатывался библиотечный класс, то автор создал для него отдельный пакет и старался следовать стилю оформления документации из JDK 6. Устройство класса очень похоже на обсуждавшееся в предыдущем разделе устройство класса `StringSelection`, только относится не к строкам, а к изображениям.

В пояснении нуждаются методы `getTransferDataFlavors()` и `isDataFlavorSupported (DataFlavor flavor)`.

В методе `getTransferDataFlavors()` используется переменная класса (`static`) `flavors`, представляющая собой массив, инициализирующийся при создании класса:

```
    private static final DataFlavor[] flavors = {
        DataFlavor.imageFlavor
    }
```

Переменная `flavors` является закрытой (private) и окончательной (final).

Метод

```
    public DataFlavor[] getTransferDataFlavors() {
        return flavors.clone();
    }
```

возвращает ссылку на эту переменную.

Но т. к. она закрытая, то приходится вызывать метод `clone()` и создавать копию ссылки. Назовем этот вариант метода первым.

Конечно, можно было бы написать более простой для понимания вариант метода (назовем его вторым):

```
public DataFlavor[] getTransferDataFlavors() {
    return new DataFlavor[]{DataFlavor.imageFlavor};
}
```

В этом варианте метод возвращает новый экземпляр массива типа `DataFlavor[]`.

Чем второй вариант хуже первого? В нем при каждом вызове метода создается новый объект-массив, который после окончания вызова должен удаляться сборщиком мусора, что, конечно, нехорошо.

В первом варианте появления нового объекта удается избежать. При вызове создается только новая ссылка, а массив имеется в единственном экземпляре и хранится в классе.

Метод:

```
public boolean isDataFlavorSupported(DataFlavor flavor) {
    return flavor.equals(DataFlavor.imageFlavor);
}
```

кажется очень простым, но в нем имеется вызов: `flavor.equals(DataFlavor.imageFlavor)`, нуждающийся в пояснении.

Почему не использовано сравнение `flavor==DataFlavor.imageFlavor`, а написано `flavor.equals(DataFlavor.imageFlavor)`?

Это так называемые объекты-значения. Поэтому нам необходимо сравнивать не ссылки на объекты `flavor` и `DataFlavor.imageFlavor`, а их содержание. В подавляющем большинстве случаев объекты-значения сравнивают по содержанию хранящихся в них данных с помощью метода `equals`.

Зададим теперь вопрос, имеет ли значение в вызове метода `equals` порядок записи? Что будет, если мы запишем не `flavor.equals(DataFlavor.imageFlavor)`, а `DataFlavor.imageFlavor.equals(flavor)`? Такой вызов вполне допустим и почти всегда будет давать точно такой же результат, кроме одного-единственного случая — когда в переменной `flavor` хранится значение `null`. В этом случае первый вариант метода вызовет `NullPointerException`, а второй вариант возвратит `false`.

Первый вариант, безусловно, предпочтительнее, поскольку именно таким образом документирована работа метода `isDataFlavorSupported` в классе `StringSelection`, аналог которого мы строим. Точно такой же вызов имеется и в методе `getTransferData`.

Как мы видим, в данном случае намеренно выбран вариант, противоположный описанному в *разд. 13.6* паттерну сравнения, чтобы вызвать диагностику ошибки в тех ситуациях, когда происходит попытка использования значения `null`.

После написания класса `FileSelection` копирование изображений в буфер обмена становится столь же простым, как и копирование строк. Создадим в классе `DesktopApplication1View` метод `copyImageToClipboard`, аналогичный рассмотренному ранее методу `copyStringToClipboard` (листинг 13.21).

Листинг 13.21. Копирование изображения в буфер обмена

```
    public void copyImageToClipboard(Image img) {
        System.out.println("copyImageToClipboard");
        if (img == null) {
            return;
        }
        clipboardsupport.ImageSelection clipData =
                new clipboardsupport.ImageSelection(img);
        clipboard.setContents(clipData, clipData);
    }
```

В проекте создадим кнопку, при нажатии на которую изображение из метки `jLabel1` будет копироваться в буфер обмена (листинг 13.22).

Листинг 13.22. Пример копирования изображения в буфер обмена

```
private void jButton6ActionPerformed(java.awt.event.ActionEvent evt) {
        icon = (ImageIcon)jLabel1.getIcon();
        img = icon.getImage();
        copyImageToClipboard(img);
}
```

13.11. Многопоточная система вычислений и индикации

Приведенный в заготовке **DesktopApplication** код кажется очень подходящим для того, чтобы во время длительной обработки осуществлять индикацию работы и останавливать ее по окончании этой работы. Однако это не так. При попытке включить длительно исполняющийся код мы обнаружим проблему.

Если включим код вычислений в акцию, то нам придется смешивать вычисления и индикацию, получится сложный и абсолютно неструктурный код.

Если же мы включим вычисления в обычный обработчик события, то придется ждать окончания вычислений и только после этого запустится индикация (поскольку такие обработчики событий выполняются перед акциями, а не после них). А отказываться от механизма создания фонового потока и обмена с ним информацией и самостоятельно обслуживать индикацию не хочется.

Для решения этой проблемы нам придется заняться многопоточным программированием.

Для начала создадим класс `ThreadedCalc`, в котором будут проводиться вычисления (листинг 13.23).

Листинг 13.23. Класс `ThreadedCalc`, обеспечивающий многопоточность вычислений

```
package desktopapplication1;

public class ThreadedCalc extends Thread{
```

```
    double r=0;
    Thread indicationThread =null;

    /** Creates a new instance of ThreadedCalc */
    public ThreadedCalc() {
        super();
    }

    public void setIndicationThread(Thread indicationThread){
        this.indicationThread=indicationThread;
        System.out.println("indicationThread: "+indicationThread);
    }

    @Override
    public void run(){
        System.out.println("calcThread.run();");
        for(int i=1;i<=10000000;i++)
            r=Math.sin(r+1);
        System.out.println("calcThread finished");
        if (indicationThread!=null)
            indicationThread.interrupt();
    }
}
```

В классе `ThreadedCalc` имеются два поля:

- `double r` — предназначено для хранения результатов вычислений;
- `Thread indicationThread` — служит для хранения ссылки на фоновый поток выполнения, в котором осуществляется вывод индикации в строке состояния (ссылка необходима для того, чтобы после окончания вычислений калькулятор мог прервать работу потока выполнения индикации).

Метод `setIndicationThread(Thread indicationThread)` служит для того, чтобы можно было установить снаружи (из другого потока) ссылку на поток индикации (отладочное сообщение в рабочем варианте программы нужно будет закомментировать). Поток калькулятора должен быть совершенно независим от потока индикации!

Замечание

Одно из главнейших правил структурного программирования автор формулирует так: "*Независимые вещи должны быть независимы*". Это вовсе не тавтология! К сожалению, программный код имеет тенденцию "пускать метастазы", т. е. устанавливать взаимозависимость между методами и классами за счет использования глобальных переменных и глобальных типов данных. В результате совершенно независимые фрагменты программного кода оказываются переплетенными. Мы уже видели подобные примеры в предыдущих разделах. Казалось бы, для копирования данных из буфера обмена, кроме типов `Clipboard` и `Image`, нам ничего не было нужно. Но для типа `Clipboard` появилась необходимость обращения к классу `Transferable`. Для класса `Transferable` оказался необходим тип `DataFlavor` и т. д.

Метод `run` в нашем классе очень прост. После отладочного сообщения идет длительный цикл. Автор поставил 20 млн итераций (на медленных компьютерах можно в несколько

раз уменьшить это число, а на очень быстрых увеличить). Время должно быть в 2—4 раза меньше, чем время, отведенное на цикл индикации (чтобы мы могли увидеть эффект прерывания потока выполнения индикации после окончания вычислений).

Затем опять идет отладочный вывод в окно консоли. Последний оператор `indicationThread.interrupt()` как раз и прерывает поток выполнения индикации. Именно для него необходимо устанавливать ссылку `indicationThread`. Если она по каким-то причинам не установлена, полагалось бы возбуждать исключение, но для упрощения примера автор этого не сделал.

Перейдем теперь к классу `DesktopApplication1View`. Вычисления будем запускать в обычном обработчике событий `ActionPerformed` для кнопки вставки (**Paste**) панели инструментов. Напомним, что такой обработчик выполняется до выполнения аналогичной акции, что легко проверить по отладочному выводу. Создается обработчик обычным способом (листинг 13.24). То, что кнопке или пункту ввода уже назначена акция, на создание обработчика `ActionPerformed` никак не влияет. Но прежде в классе `DesktopApplication1View` нам необходимо задать переменную `calcThread` (она нужна, чтобы передать в поток вычислений ссылку на фоновый поток, создающийся для работы метода `doInBackground`).

Листинг 13.24. Обработчик событий `ActionPerformed` для кнопки вставки *(Paste)* панели инструментов

```
ThreadedCalc calcThread;

private void jButton3ActionPerformed (java.awt.event.ActionEvent evt) {
    System.out.println("pasteToolBarButtonActionPerformed");
    calcThread=new ThreadedCalc();
    calcThread.start();
    System.out.println("after calcThread.start()");
    doPaste();
}
```

В обработчике сначала создается поток выполнения вычислений, а ссылка на него сохраняется в переменной `calcThread`. Пример имитирует длительную обработку, происходящую при вставке. Затем мы даем потоку команду стартовать, после чего выполняем отладочный вывод в окно консоли.

И в самом конце стоит оператор реальной вставки из буфера обмена. Оператор `doPaste();` выполняется практически мгновенно, так что все равно, где в данном методе его поставить — с таким же успехом его можно написать в самом начале (все равно его работа завершается гораздо раньше, чем закончит работу поток `calcThread`). А вот если бы процесс выполнения оператора `paste` был длительный, его потребовалось бы переместить в класс `ThreadedCalc` и выполнять в его потоке выполнения.

Класс `PasteTask` (листинг 13.25) очень похож на класс `SaveTask` (см. листинг 13.8), который мы рассматривали в предыдущих разделах этой главы. Но имеются три отличия, отмеченные в листинге подчеркиванием.

Листинг 13.25. Класс `PasteTask`, обеспечивающий в фоновом потоке вставку из буфера обмена

```
private class PasteTask extends Task<Object, Void> {

    PasteTask(org.jdesktop.application.Application app) {
        // Runs on the EDT.  Copy GUI state that
        // doInBackground() depends on from parameters
        // to PasteTask fields, here.
        super(app);
    }

    @Override
    protected Object doInBackground() {
        System.out.println("paste doInBackground");
        calcThread.setIndicationThread(Thread.currentThread());
        try {
            for(int i = 0; i < 10; i++) {
              setMessage(resourceMap.getString("StatusBar.Working") +
                           "... [" + i + "]");
              Thread.sleep(350L);
              setProgress(i, 0, 9);
            }
            Thread.sleep(150L);
        } catch(InterruptedException e) {
            System.out.println("catch InterruptedException");
        }
        return null;
    }
    protected void finished() {
        setMessage(resourceMap.getString("StatusBar.Done"));
    }


    @Override
    protected void succeeded(Object result) {
        // Runs on the EDT.  Update the GUI based on
        // the result computed by doInBackground().
    }
}
```

Первое изменение — вызов `calcThread.setIndicationThread`, который устанавливает в потоке выполнения вычислений ссылку на поток индикации. Его параметр — ссылка, возвращаемая методом класса `Thread.currentThread()`. Как мы знаем, данный метод возвращает ссылку на текущий поток, т. е. в нашем случае на фоновый поток индикации, в котором выполняется метод `doInBackground`.

Второе изменение — увеличение в операторе `Thread.sleep(350L)` времени сна потока индикации между итерациями со 150 до 350 мс. Так сделано, чтобы индикация могла осуществляться в несколько раз дольше, чем идут вычисления.

И, наконец, третье изменение — перехват исключения `InterruptedException`, которое возникает в случае, когда другой поток прерывает выполнение данного потока методом `interrupt()`. Как мы знаем, обычно такое прерывание требуется для того, чтобы прекратить индикацию по окончании вычислений. Но мы это делаем только для того, чтобы при наступлении прерывания осуществить отладочный вывод в окно консоли. Исключение `InterruptedException` приводит к выходу из цикла `for`, в котором происходит вывод индикации, а выполнение задания заканчивается возвратом значения `null`.

Работающий вариант программы и его исходные коды можно посмотреть в примере 13_11_DesktopApplication1 на прилагаемом к книге DVD. Автор не претендует на то, что его вариант оптимален и не нуждается в дальнейших доработках, наоборот — очевидно, что следует провести ряд усовершенствований, но это все оставляется в качестве задания читателю.

Краткие итоги по главе

- В NetBeans 6.5 имеется возможность создать **DesktopApplication** — заготовку приложения с графическим пользовательским интерфейсом.
- *Строка состояния* (statusBar) — это небольшая прямоугольная область в самой нижней части окна приложения, в которой обычно показывается информация о текущем состоянии приложения.
- *Индикатор выполнения* (progressBar) — узкий прямоугольник, в котором показывается "бегущий" столбик, отмечающий процент выполнения задания.
- *Объекты-синглетоны* — такие объекты, которые существуют всего в одном экземпляре.
- Класс `ApplicationContext` служит для создания объектов-синглетонов методом класса `getInstance()`. Этот синглетон является "незримым спутником" приложения и позволяет получать о нем различные сведения. Например, метод `getApplication()` дает ссылку на сам объект-приложение, `getClipboard()` — ссылку на буфер обмена приложения, `getFocusOwner()` — ссылку на компонент приложения, владеющий фокусом.
- *Разделяемыми объектами* приложения (shared objects) называются такие объекты, которые доступны разным потокам приложения.
- В Java перенос очень длинных выражений, в которых присутствует оператор точка (.), означающий доступ к полю или методу, может осуществляться как с переносом точки на новую строку, так и с оставлением точки на предыдущей строке.
- Пары "имя-значение" хранятся в файлах свойств (properties), для каждого класса свой файл. Эти файлы могут иметь версии для каждого языкового формата (locale).
- Добавить в проект изображения проще всего их копированием в папку соответствующего пакета.
- Аннотация `@Action` означает, что помеченные таким образом методы являются реализацией акций для интерфейса `Action`, т. е. представляют собой обработчики `actionPerformed` объектов-событий типа `action` (акция). Данный интерфейс не содержит методов — имя интерфейса в данном случае используется как метка.
- Сначала исполняются обработчики `actionPerformed`, и только затем — акции.

- Класс `Task` предназначен для поддержки отображения в фоновом режиме информации в приложениях, основанных на компонентах NetBeans.
- Когда объект типа `Task` (задание) создается, в фоновом режиме начинается выполнение метода `doInBackground`. После окончания выполнения задания автоматически запускается метод `done`. В зависимости от результатов выполнения задания метод `done` вызывает один из следующих методов, выполняющих необходимые завершающие действия:
  - `succeeded` (все сделано удачно) — когда все хорошо;
  - `cancelled` (отказ от выполнения) — когда произошел отказ от выполнения с помощью вызова метода `cancel(true)`;
  - `interrupted` (прервано) — когда выполнение задания было прервано;
  - `failed` (неудача) — когда выполнение задания завершилось неудачно;
  - `finished` (завершено) — самый последний метод, вызываемый после выполнения завершающих действий или при возникновении исключения.
- Пакет `java.awt.datatransfer` и, в частности, заданные в нем классы `Clipboard` и `DataFlavor`, а также интерфейс `Transferable` обеспечивают поддержку работы с буфером обмена.
- Для копирования строк из программы в буфер обмена дополнительно используется класс `StringSelection`, реализующий интерфейсы `Transferable` и `ClipboardOwner`.
- Пока в JDK поддерживаются только два формата, читаемые из буфера обмена, — строки и изображения. Но с записью дело обстоит еще хуже — стандартными средствами копирование в буфер поддерживается только для текста.
- Для того чтобы обеспечить поддержку копирования изображений в буфер обмена, можно создать класс `ImageSelection`, по функциональности аналогичный классу `StringSelection`, но работающий не со строками, а с изображениями.
- Многопоточное программирование позволяет решить проблему индикации вычислений, управляемой из потока, в котором выполняются эти вычисления.

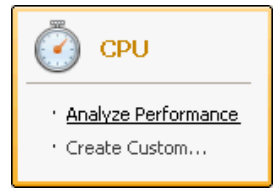
Типичные ошибки

- Путают синглетон объекта-приложения, ссылка на который получается методом `Application.getInstance(DesktopApplication1.class)` или `getApplication()`, с синглетоном контекста, ссылка на который получается методом `getContext()`.
- Забывают, что в `DesktopApplication` у метода `getResourceMap` имеется не только вызов без параметров, позволяющий получать ресурсы, относящиеся к классу самого приложения, но и вариант вызова с указанием имени класса в качестве параметра, позволяющий получать ресурсы, относящиеся к указанному в качестве параметра классу.
- При локализации приложения редактируют значения свойств для одного языкового формата, забывая соответствующим образом изменить другие. Лучше иметь всего два языковых формата (русский и default, по умолчанию совпадающий с английским), но соответствующих друг другу, чем три и более, но не соответствующих.

- Считают сравнения объектов `obj1.equals(obj2)` и `obj2.equals(obj1)` эквивалентными.

Задания

- Завершить локализацию приложения 13_08 _DesktopApplication1.
- Написать реакцию задачи на досрочное прекращение (реализацию метода `cancelled`) приложения 13_09_DesktopApplication1.
- Переписать реализацию метода `copyStringToClipboard` в соответствии с рекомендациями, данными в *разд. 13.10*.
- Доработать приложение так, чтобы можно было копировать в буфер обмена участок изображения заданной ширины и высоты из метки `jLabel1`.



ГЛАВА 14

Программирование мобильных телефонов

Одно из основных преимуществ NetBeans 6.5 — наличие развитых средств разработки приложений Java ME (Java Micro Edition) для мобильных устройств, и в первую очередь для мобильных телефонов.

14.1. Спецификация Java Micro Edition. Конфигурации и профили. Мидлеты

Для версии Java Micro Edition не существует JDK в том же смысле, как и для Java Standard Edition или Java Enterprise Edition. Имеется набор спецификаций Java Specification Request (JSR), в которых заданы конфигурации (Configuration) и профили (Profile).

Конфигурация — это такая спецификация, в которой задаются:

- тип и объем оперативной памяти мобильного устройства;
- тип и скорость процессора;
- параметры работы с сетью, к которой это устройство подсоединяется.

Профиль конкретизирует тип используемого устройства и параметры пользовательского интерфейса в рамках заданной конфигурации.

Существуют две важнейшие конфигурации, определяющие параметры приложений и устройств, для которых они создаются.

- Connected Limited Device Configuration (CLDC) — приложения для мобильных телефонов и наладонных компьютеров (PDA) с очень небольшим (512 Кбайт или менее) объемом памяти и, как правило, 16-битовыми процессорами. Использует специальную версию Java-машины — Kilobyte Virtual Machine (KVM), которая не поддерживает работу с числами с плавающей точкой, Java Native Interface (JNI), интерфейс Reflection, JIT-компиляцию. Данные приложения могут работать и на устройствах с большими ресурсами.

  Поддерживаемые профили:

  - Mobile Information Device Profile (MIDP) — профиль мобильных информационных устройств. Память не менее 160 Кбайт, дисплей не менее 96×54 пикселов, поддержка работы в WWW по протоколу HTTP 1.1;

- PDA Profile (PDAP) — профиль PDA (наладонных компьютеров);
- Information Module Profile 2.0 (IMP) — профиль информационного модуля.

♦ Connected Device Configuration (CDC) — приложения для продвинутых мобильных телефонов и смартфонов, веб-телефонии и других устройств, имеющих 2 Мбайт памяти (и более) и 32-битовый процессор. Использует специальную версию Java-машины — Compact Virtual Machine (CVM), для программиста ее основное отличие от JVM — ввод/вывод с помощью пакета javax.microedition.io.

Поддерживаемые профили:

- Foundation Profile — основной профиль;
- Personal Basis Profile — базовый персональный профиль;
- Personal Profile — персональный профиль;
- RMI Profile (Remote Methods Invocation Profile) — профиль удаленного вызова методов;
- Game Profile — игровой профиль.

В рамках профиля MIDP имеется особый вид приложений для мобильных устройств, называемый *мидлетами* (MIDlets — объединение аббревиатуры MID и второй половины от слова applet). Мидлет — это приложение, класс которого унаследован от `javax.microedition.midlet.MIDlet`. Оно распространяется в виде jar-архива (как обычные приложения) и сопровождающего его файла дескрипторов jad (Java application descriptor).

Мидлеты — самые простые, легкие и наиболее переносимые приложения для мобильных устройств.

14.2. Создание приложений для мобильных устройств

Чтобы создать приложение для мобильного телефона, выберем в главном меню **File/New Project...**, а в открывшейся форме — категорию **Java ME** (рис. 14.1).

Нам предлагается на выбор несколько типов проектов:

♦ **Mobile Application** — мобильное приложение для профиля MIDP;

♦ **Mobile Class Library** — библиотека классов для создания мобильных приложений;

♦ **Mobile Project with Existing MIDP Sources** — мобильное приложение для профиля MIDP на основе импорта существующих исходных кодов Java;

♦ **Import Wireless Toolkit Project** — импорт в NetBeans проекта, созданного с использованием пакета Wireless Toolkit;

♦ **CDC Application** — приложение CDC для продвинутых мобильных устройств;

♦ **CDC Class Library** — библиотека классов для создания приложений CDC;

♦ **Import CDC Pack 5.5 Project** — импорт в NetBeans приложения, созданного с использованием модуля CDC Pack для NetBeans 5.5;

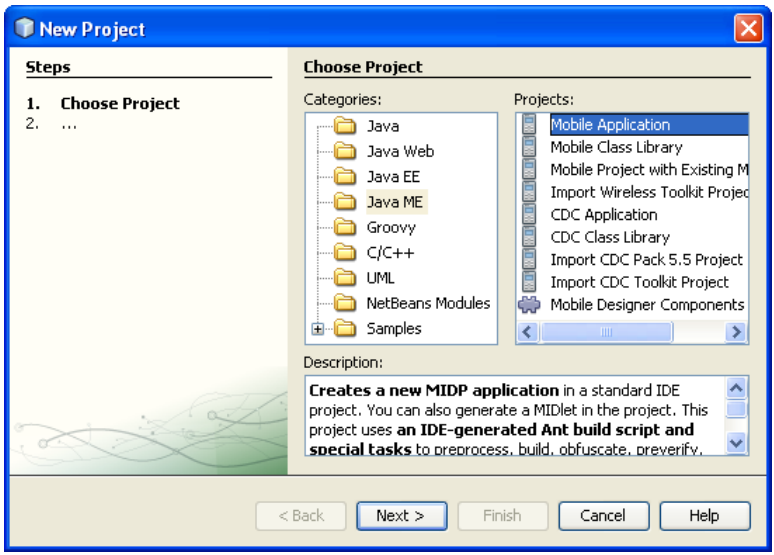


Рис. 14.1. Создание приложения для мобильных устройств (шаг 1)

- **Import CDC Toolkit Project** — импорт приложения, созданного с использованием пакета CDC Toolkit;
- **Mobile Designer Components** — мастер для создания визуальных компонентов Java ME в модуле Netbeans.

Выберем самый первый вариант (Mobile Application), нажмем кнопку **Next >**. Появится форма, в которой можно изменить имя и место расположения проекта (рис. 14.2). Кроме того, по умолчанию установлен флажок **Create Hello MIDlet** — нам предлагается создать традиционный простейший вариант программы, выдающей приветствие: **Hello, World!** (Здравствуй, мир!).

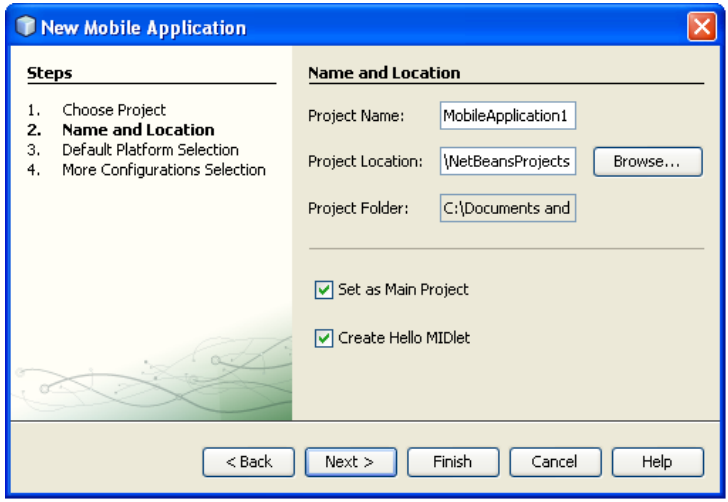


Рис. 14.2. Создание приложения для мобильных устройств (шаг 2)

Нажмем кнопку **Finish**, получим пример простейшего приложения для телефона, которое можно запустить. Если все прошло нормально, после запуска увидим на экране эмулятор телефона. Автору пришлось после создания первого приложения Java ME сначала выполнить **Clean and Build**, и только после этого приложение запустилось.

Если же нажать кнопку **Next >**, то можно выбрать платформу (рис. 14.3):

- инструментарий, с помощью которого должна идти разработка;
- тип устройства, для которого разрабатывается приложение;
- конфигурацию и профиль.

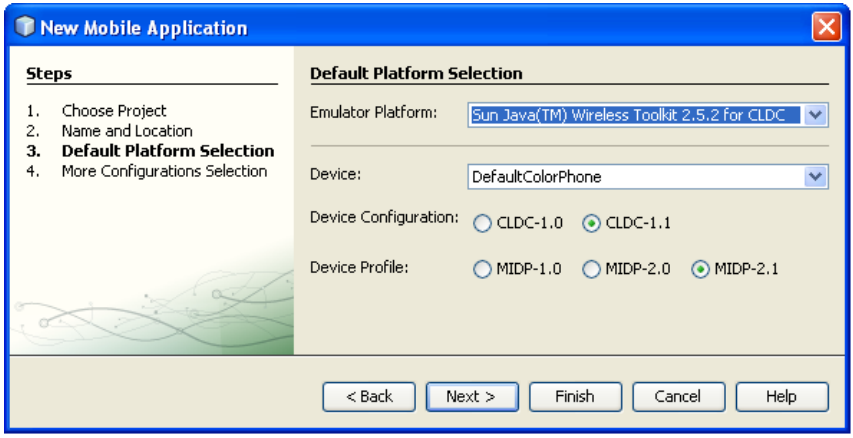


Рис. 14.3. Дополнительная возможность — выбор платформы

В настоящее время со средой NetBeans поставляется только один инструментарий (Toolkit) — **Sun Java(TM) Wireless Toolkit 2.5.2** (Инструментарий для устройств беспроводной связи от Sun для платформы Java, версия 2.5.2).

Тип устройства (**Device**) можно выбрать из списка:

- **DefaultColorPhone** — стандартный телефон с цветным экраном;
- **DefaultGrayPhone** — стандартный телефон с черно-белым экраном;
- **MediaControlSkin** — устройство с внешним видом, определяемым программным окружением;
- **QwertyDevice** — устройство с клавиатурным вводом/выводом (определяет, какой эмулятор будет выбран для запуска приложения).

По умолчанию используется телефон с цветным экраном, конфигурация CLDC-1.1 и профиль MIDP-2.1.

В NetBeans 5.5 требовалось, чтобы путь к папке, выбранной для создания приложения, не содержал русских букв, иначе при попытке запуска приложения только на одно мгновение появлялось окно эмулятора мобильного телефона и сразу закрывалось. Начиная с NetBeans 6.0, эта проблема была решена.

После нажатия кнопки **Finish** будет создан проект. По умолчанию происходит переход в экран Flow (поток), отражающий функциональную схему выполнения команд устрой-

ства. Но мы не будем пользоваться этим режимом и переключимся в режим редактирования исходного кода (вкладка **Source**, рис. 14.4, пример 14_2_MobileApplication1 на прилагаемом к книге DVD). Переходить в дизайнер пользовательского интерфейса будем посредством вкладки **Screen** (Экран) (рис. 14.5).

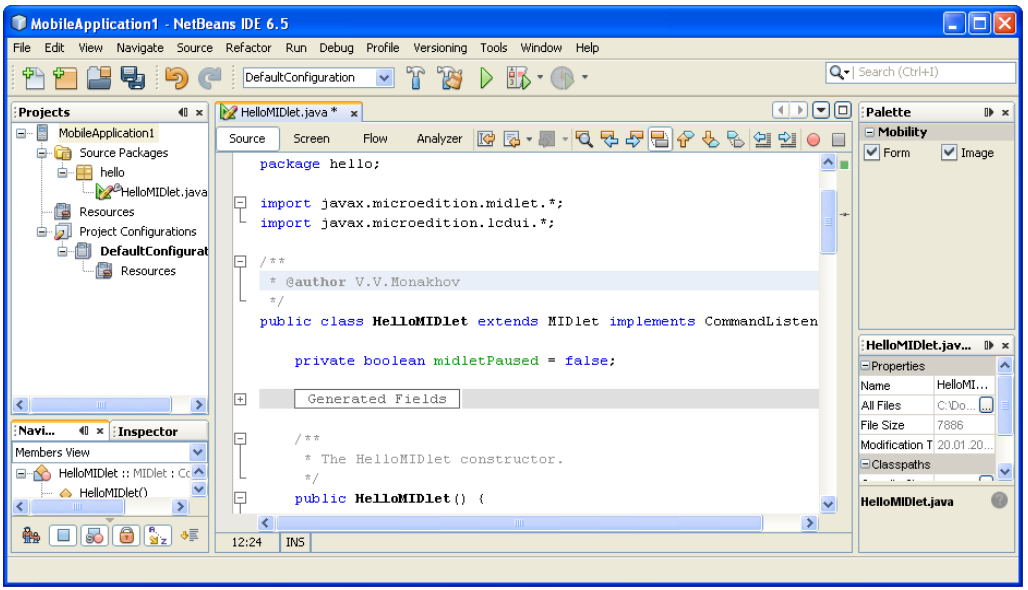


Рис. 14.4. Внешний вид проекта мидлета, вкладка **Source**

После запуска приложения мы увидим окно эмулятора (рис. 14.6), где на экране телефона в заголовке появится надпись:

Select one to launch: (Выберите один из следующих пунктов для запуска).

Под надписью будет единственный пункт списка, который сразу окажется выделенным: **HelloMIDlet**.

После нажатия мышью на кнопку **Select** (рис. 14.7) или на правую управляющую кнопку, над которой высвечивается надпись **Launch** (Запустить), наш мидлет запускается, а на экране показывается надпись:

Hello Hello, World!

Показываемый полужирным шрифтом текст **Hello** — это заголовок, а текст "Hello, World!" — содержимое этого пункта (рис. 14.8).

Нажатие на левую управляющую кнопку, расположенную под пунктом экрана **Exit** (Выход), приводит к выходу из приложения.

Закрыть эмулятор можно либо нажатием на круглую кнопку в правом верхнем углу телефона (имитирующую кнопку выключения телефона), либо нажав на кнопку в правом верхнем углу окна программы.

В папке dist, как обычно, создаются те файлы, которые нужны для распространения нашего приложения. В данном случае это jad- и jar-файлы. Аббревиатура jad означает Java Application Descriptor (Описание приложения Java).

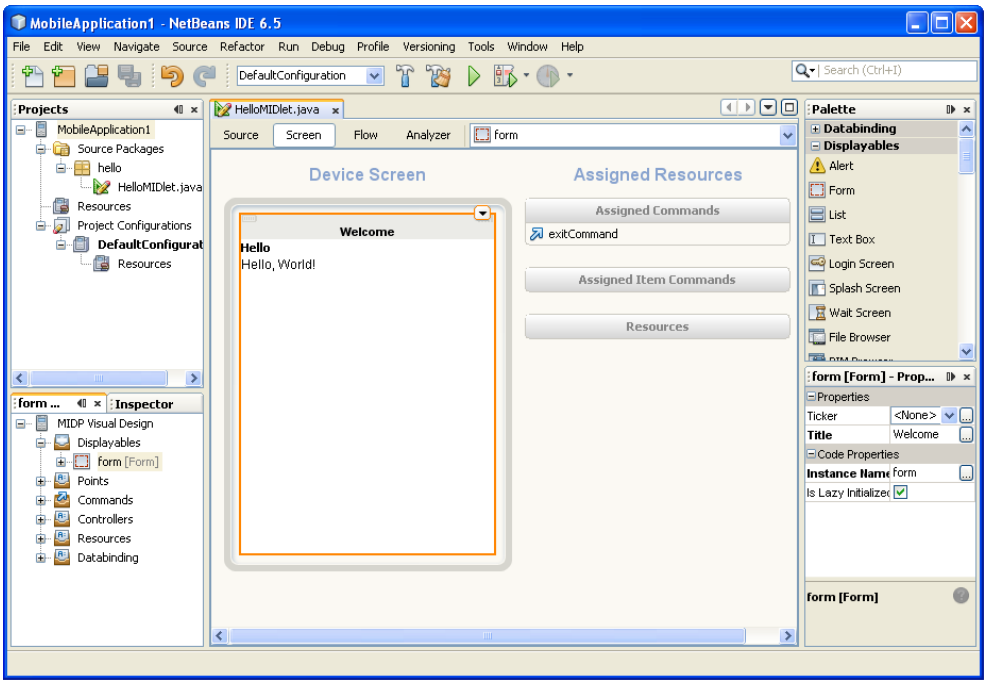


Рис. 14.5. Внешний вид проекта мидлета, вкладка **Screen**

Рис. 14.6. Мидлет, запущенный на эмуляторе мобильного телефона

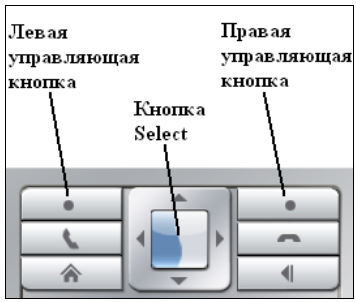


Рис. 14.7. Кнопки эмулятора

Рис. 14.8. Мидлет после нажатия кнопки **Select**

Содержимое jad-файла, автоматически создаваемого средой NetBeans для нашего приложения, приведено в листинге 14.1.

Листинг 14.1. Содержимое jad-файла

```
MIDlet-1: HelloMIDlet, ,hello.HelloMIDlet
MIDlet-Jar-Size: 2066
MIDlet-Jar-URL: MobileApplication1.jar
MIDlet-Name: MobileApplication1
MIDlet-Vendor: Vendor
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.1
```

Отметим, что слово Vendor означает производитель, и в паре

```
MIDlet-Vendor: Vendor
```

необходимо будет установить имя разработчика или компании:

```
MIDlet-Vendor: V.V.Monakhov
```

Это делается в свойствах проекта в разделе **Application Descriptor** (Файл описания приложения — так официально называется jad-файл).

Большинству современных мобильных телефонов необходим jad-файл для того, чтобы узнать параметры приложения. Только телефоны Sony Ericsson не нуждаются в jad-файлах, им достаточно jar-файла.

14.3. Дизайнер пользовательского интерфейса мидлета

Часто считают, что программирование устройств — это очень сложная работа, доступная только профессионалам высшей квалификации. Во многих случаях это действительно так, но не всегда. Покажем, как написать приложение для мобильного телефона практически не программируя, а занимаясь только проектированием пользовательского интерфейса. Подобная задача вполне доступна по уровню сложности даже школьнику, не говоря уже о студентах или профессиональных программистах.

Кроме вкладок **Source** (Редактор исходного кода) и **Screen** (Дизайнер экрана) имеются вкладки **Flow** (Поток) — дизайнер последовательности выполнения акций и **Analyzer** (Анализатор) — окно, предназначенное для отыскания неиспользуемых компонентов и тех, которые не поддерживаются старым профилем MIDP 1.0. Работу с вкладками **Flow** и **Analyzer** мы в данной книге обсуждать не будем.

В редактор исходного кода мидлета можно переключиться нажатием на вкладку **Source**. Но элементы исходного кода, сформированные автоматически, нельзя изменить, а можно только добавить дополнительный код в некоторых, заранее помеченных, местах.

Но мы в этом будем разбираться позже, т. к. сначала важнее научиться формировать логическую структуру взаимодействия приложения с пользователем, а это делается в режиме **Screen** (Дизайнер экрана) (рис. 14.9).

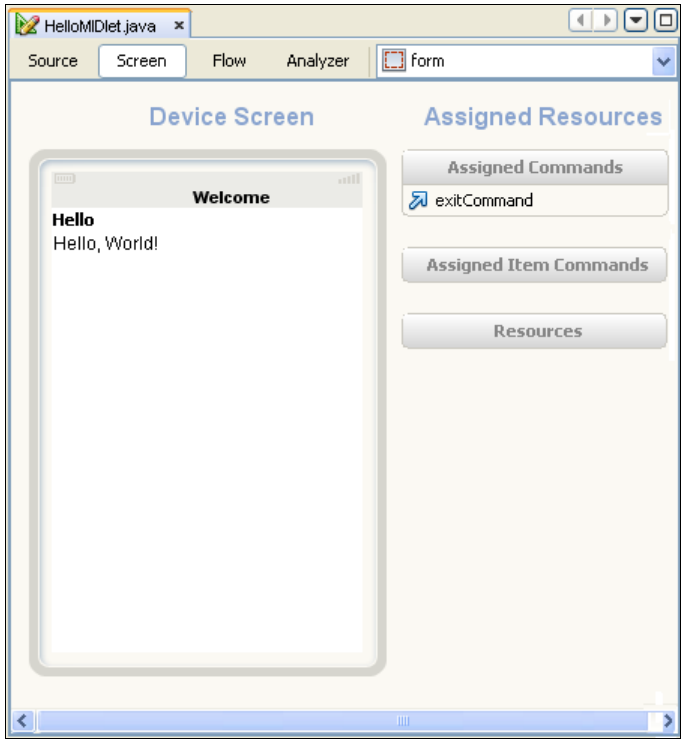


Рис. 14.9. Дизайнер экрана мидлета

Если мы захотим изменить фразу **Hello, World!** на **Привет!**, то в исходном коде мидлета нам это не удастся. Повторяем, что исходный код, сгенерированный автоматически, менять нельзя.

А вот в дизайнере экрана изменить данный текст очень просто. Двойным щелчком мыши в области этой фразы мы входим в режим ее редактирования и вводим необходимый текст. После запуска приложения можно наблюдать сообщение **Привет!**, что приятно, поскольку с русским языком проблем не возникает.

Аналогично имя пункта **Hello** меняем на **Приветствие**.

При щелчке в области экрана устройства вне какого-либо его элемента в окне свойств появляются свойства экранной формы **helloForm** нашего приложения.

Их всего четыре (см. рис. 14.5):

- **Ticker** — бегущая строка. По умолчанию не подключена. Может быть добавлена следующим образом: в окне свойств нажать на кнопку ... в поле **Ticker**, затем в появившейся форме нажать кнопку **Add** и в поле **Ticker String** ввести текст бегущей строки;
- **Title** — заголовок. По умолчанию стоит Welcome. Меняем его на **Пример мидлета**;
- **Instance name** — имя экземпляра класса нашей формы. По умолчанию стоит form. Оставляем неизменным;
- **Is Lazy Initialized** (с отложенной инициализацией). Булево свойство по умолчанию имеет значение `true` (установлен флажок). Это означает, что пока элемент не потребуется в первый раз, он не инициализируется, а значит, не потребляет ресурсов и, что важнее, не занимает времени на инициализацию в момент запуска приложения. Данный режим полезен для не очень мощных в вычислительном плане устройств, т. к. если бы инициализация всех элементов происходила сразу при старте приложения, то пользователю пришлось бы долго ждать, пока она не закончится. А разнесение по времени заметно уменьшает величину задержек в каждый конкретный момент времени.

Попробуем установить размер окна мобильного устройства. Для этого требуется щелчком правой кнопки мыши по имени проекта в окне **Projects** вызвать контекстное меню, выбрать в нем команду **Properties** (в самом низу меню), а затем в открывшейся форме — категорию **Abilities** (Способности) (рис. 14.10). Далее нужно в графах **ScreenHeight** (Высота экрана) и **ScreenWidth** (Ширина экрана) установить новые значения. Правда, подавляющее большинство современных телефонов достаточно "умные", и эти значения можно не менять. По умолчанию стоит значение 240×320 пикселов (в информационных изданиях по мобильным телефонам сначала указывают ширину экрана, а затем его высоту). Это значение характерно для более совершенных моделей.

Изменение размера экрана важно при работе с реальным телефоном — размер окна эмулятора при этом, к сожалению, не меняется.

Форма **Properties** весьма полезна — в ней можно настраивать много различных параметров. В частности, в категории **Application Descriptor** можно задать параметры, указываемые в jad-файле, в **Build/Creating JAR** — имена для создаваемых jar- и jad-файлов и т. д.

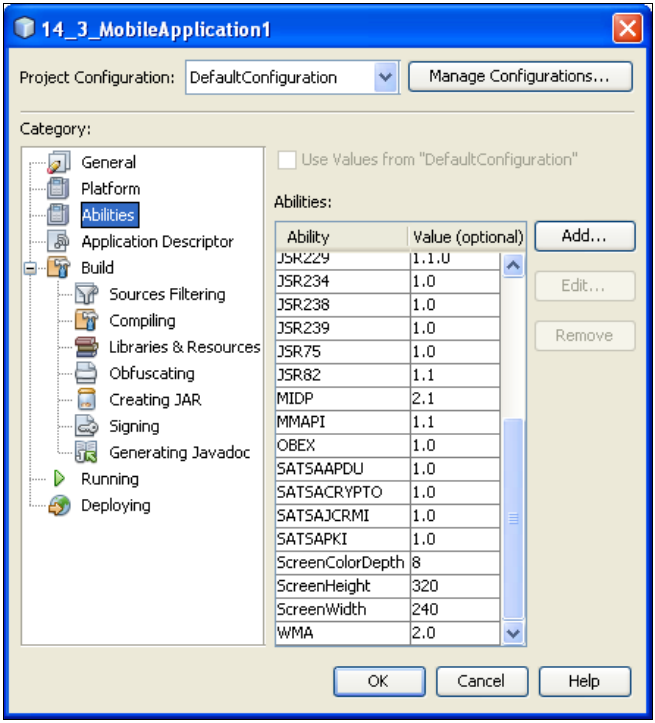


Рис. 14.10. Установка допустимого размера экрана для приложения

В табл. 14.1 приведены размеры экранов некоторых распространенных моделей мобильных телефонов.

Таблица 14.1. *Размеры экранов некоторых моделей мобильных телефонов*

Название модели мобильного телефона	**Экран (Ширина × Высота)**
Alcatel OT E220, OT E225	80×102
Sony Ericsson J100i	96×64
Nokia 2310	96×68
Sagem My100x	101×64
Sagem MyC2-3, My201x	101×80
Voxtel RX11	120×64
Alcatel OT E801 Motorola W220 Nec E1101 Nokia 2610, 2626, 6020 Philips Xenium 9@9a, Sagem My200C Samsung GT-E1117, GT-E1410 Sony Ericsson J230i	128×128

Таблица 14.1 *(окончание)*

Название модели мобильного телефона	**Экран (Ширина × Высота)**
Alcatel OT C560 Nokia 2330, 2605, 2608, 5200, 6060i, 6070, 6080, 6085, 6111, 6125, 7360 Philips Xenium 9@9d, Xenium 9@9f, 588 Sagem My401C Samsung C240, C300, E250, E2510, X160, X210, X300, X510, X520, X530, X630, X640, X680 Sony Ericsson K310i, K320i, K510i, W300i, Z310i, Z530i Voxtel V350, 1iD	128×160
BenQ-Siemens C81 Siemens C75	132×176
Nokia N70, N72, N91, 3250	176×208
BenQ-Siemens S88 Motorola MOTOKRZR K1, MOTORAZR V3i, MOTORIZR Z3, MOTOSLVR L7e Nec E949 Sagem My700x Samsung E390, E500, E570, E780, U100, X820 Sony Ericsson F305, K550i, K610i, K750i, S302, W700i, W710i, W810i, Z550i, Z610i, Z710i Voxtel BD50, V700, VS600, VS800, RX600, 2iD, 3iD, W210	176×220
Nokia 5500, 6230i, 8800	208×208
BenQ-Siemens E71, EL71, EF81 Nokia E50, E61i, E63, E65, N71, N73, N85, N93, N93i, 5130, 5300, 6131, 6208c, 6233, 6280, 6288, 6290, 7100, 7373, 7390, 8800 Samsung D600, D830, D840, D900, E900, P310, U810 Sony Ericsson C510, G705, K790i, M600i, P990i, TM506, T700, W508, W705, W715, W850i, W880i, W902, W950i Voxtel W420, W740	240×320
Samsung A867	240×400
Samsung i770	320×320
Nokia 6260	320×480
Nokia N80	352×416
Nokia N97, 5800	360×640

В правом верхнем углу экрана дизайна, как обычно, находится палитра компонентов. Она состоит из девяти категорий (табл. 14.2).

Таблица 14.2. *Категории палитры компонентов*

№	Категория	Состав
1	**DataBinding** (Связывание данных при работе с базами данных)	DataSet — источник данных; IndexableDataSet — индексируемый источник данных; ContactsDataSet — источник данных с контактами
2	**Dispayables** (Дисплеи, Экраны)	Alert — предупреждение; Form — форма; List — список; TextBox — область ввода/вывода текста; Login Screen — экран ввода учетной записи; Splash Screen — экран-заставка; Wait Screen — экран ожидания; File Browser — диалог выбора файлов; PIM Browser — диалог просмотра электронной записной книжки (PIM — Personal Information Manager); SMS Composer — экран создания SMS-сообщения
3	**Commands** (Команды)	Back Command — команда "Обратно"; Cancel Command — команда "Отменить"; Exit Command — команда "Выход"; Help Command — команда "Справка"; Item Command — команда "Пункт" (например, для таблицы); OK Command — команда "OK"; Screen Command — экранная команда; Stop Command — команда "Стоп"
4	**Elements** (Элементы)	Choice Element — элемент выбора (CheckBox); List Element — элемент "Список"
5	**Items** (Элементы формы)	Choice Group — группа выбора; Date Field — поле данных; Gauge — измеритель (индикатор выполнения); ImageItem — элемент показа изображения; Spacer — разделительная черта; String Item — пункт вывода строки (метка); Text Field — поле ввода и редактирования текста; Table Item — элемент "Таблица"
6	**Flow** (Элементы режима потока выполнения)	Entry Point — точка входа; If — условие; Call Point — точка вызова; List Action — акция для списка; Previous Screen Action — акция для предыдущего экрана; Switch Case — выбор варианта; Switch — выполнение метода при совпадении значения

Таблица 14.2 *(окончание)*

№	Категория	Состав
7	**Resources** (Ресурсы)	Font — шрифт; Image — изображение; Ticker — часы, таймер; Simple Table Model — простая модель таблицы; Simple Cancellable Task — простое задание, которое можно прервать
8	**SVG components** (Компоненты масштабируемой векторной графики)	SVG Menu Element — элемент меню в формате векторной графики; SVG Menu Action — акция SVG-меню; SVG Image — изображение в формате векторной графики; SVG Form — форма в формате векторной графики; SVG Menu — меню в формате векторной графики (SVG-меню); SVG Player — проигрыватель в формате векторной графики; SVG Splash Screen — экран-заставка в формате векторной графики; SVG Wait Screen — экран ожидания в формате векторной графики
9	**Custom** (Компоненты, заданные пользователем)	Категория для компонентов, устанавливаемых пользователем (программистом, разрабатывающим приложение)

14.4. Исходный код мидлета *Hello, World!*

Простейший пример требовал от нас только работы с дизайнером пользовательского интерфейса. Но при создании реального приложения необходимо менять исходный код приложения. Поэтому рассмотрим исходный код мидлета **Hello, World!**, приведенный в листинге 14.2 (пример 14_4_MobileApplication1 на прилагаемом к книге диске). Подчеркиванием выделены методы, задаваемые в мидлете, их мы будем анализировать отдельно. Серой заливкой помечены фрагменты кода, свернутые по умолчанию (collapsed) в среде разработки, а при разворачивании также имеющие серый фон.

Данный мидлет, как и все прочие, является наследником класса `javax.microedition.midlet.MIDlet`. Для работы с пользовательским интерфейсом (User's Interface — UI) импортируется пакет `javax.microedition.lcdui`. Аббревиатура lcdui означает сокращение от Liquid Cristal Device User's Interface (пользовательский интерфейс устройства с жидкокристаллическим экраном).

Листинг 14.2. Исходный код мидлета *Hello, World!*

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package hello;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 * @author V.V.Monakhov
 */
public class HelloMIDlet extends MIDlet implements CommandListener {

    private boolean midletPaused = false;

    //<editor-fold defaultstate="collapsed" desc=" Generated Fields ">
    private Command exitCommand;
    private Form form;
    private StringItem stringItem;
    //</editor-fold>

    /**
     * The HelloMIDlet constructor.
     */
    public HelloMIDlet() {
    }

    //<editor-fold defaultstate="collapsed" desc=" Generated Method: initialize ">
    /**
     * Initilizes the application.
     * It is called only once when the MIDlet is started. The method
     * is called before the <code>startMIDlet</code> method.
     */
    private void initialize() {
        // write pre-initialize user code here

        // write post-initialize user code here
    }
    //</editor-fold>

    //<editor-fold defaultstate="collapsed" desc=" Generated Method: startMIDlet ">
    /**
     * Performs an action assigned to the Mobile Device — MIDlet Started
     * point.
     */
    public void startMIDlet() {
        // write pre-action user code here
        switchDisplayable(null, getForm());
        // write post-action user code here
    }
    //</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated Method: resumeMIDlet ">
    /**
     * Performs an action assigned to the Mobile Device
     */
    public void resumeMIDlet() {
        // write pre-action user code here

        // write post-action user code here
    }
    //</editor-fold>
```

```
//<editor-fold defaultstate="collapsed" desc=" Generated Method: switchDisplayable ">
    /**
     * Switches a current displayable in a display. The <code>display</code> instance
is taken from <code>getDisplay</code> method. This method is used by all actions in the
design for switching displayable.
     * @param alert the Alert which is temporarily set to the display; if
<code>null</code>, then <code>nextDisplayable</code> is set immediately
     * @param nextDisplayable the Displayable to be set
     */
    public void switchDisplayable(Alert alert,
                                  Displayable nextDisplayable) {
        // write pre-switch user code here
        Display display = getDisplay();
        if (alert == null) {
            display.setCurrent(nextDisplayable);
        } else {
            display.setCurrent(alert, nextDisplayable);
        }
        // write post-switch user code here
    }
//</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed"
      desc=" Generated Method: commandAction for Displayables ">
    /**
     * Called by a system to indicated that a command has been invoked on a particular
displayable.
     * @param command the Command that was invoked
     * @param displayable the Displayable where the command was invoked
     */
    public void commandAction(Command command, Displayable displayable) {
        // write pre-action user code here
        if (displayable == form) {
            if (command == exitCommand) {
                // write pre-action user code here
                exitMIDlet();
                // write post-action user code here
            }
        }
```

```
        // write post-action user code here
    }
    //</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated Getter: exitCommand ">
    /**
     * Returns an initiliazed instance of exitCommand component.
     * @return the initialized component instance
     */
    public Command getExitCommand() {
        if (exitCommand == null) {
            // write pre-init user code here
            exitCommand = new Command("Exit", Command.EXIT, 0);
            // write post-init user code here
        }
        return exitCommand;
    }
    //</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated Getter: form ">
    /**
     * Returns an initiliazed instance of form component.
     * @return the initialized component instance
     */
    public Form getForm() {
        if (form == null) {
            // write pre-init user code here
            form = new Form("Welcome", new Item[] { getStringItem() });
            form.addCommand(getExitCommand());
            form.setCommandListener(this);
            // write post-init user code here
        }
        return form;
    }
    //</editor-fold>
```

```
    //<editor-fold defaultstate="collapsed" desc=" Generated Getter: stringItem ">
    /**
     * Returns an initiliazed instance of stringItem component.
     * @return the initialized component instance
     */
    public StringItem getStringItem() {
        if (stringItem == null) {
            // write pre-init user code here
            stringItem = new StringItem("Hello", "Hello, World!");
            // write post-init user code here
        }
        return stringItem;
    }
    //</editor-fold>
```

```
    /**
     * Returns a display instance.
     * @return the display instance.
     */
    public Display getDisplay () {
        return Display.getDisplay(this);
    }

    /**
     * Exits MIDlet.
     */
    public void exitMIDlet() {
        switchDisplayable (null, null);
        destroyApp(true);
        notifyDestroyed();
    }

    /**
     * Called when MIDlet is started.
     * Checks whether the MIDlet have been already started and
     * initialize/starts or resumes the MIDlet.
     */
    public void startApp() {
        if (midletPaused) {
            resumeMIDlet ();
        } else {
            initialize ();
            startMIDlet ();
        }
        midletPaused = false;
    }

    /**
     * Called when MIDlet is paused.
     */
    public void pauseApp() {
        midletPaused = true;
    }

    /**
     * Called to signal the MIDlet to terminate.
     * @param unconditional if true, then the MIDlet has to be unconditionally terminated
and all resources has to be released.
     */
    public void destroyApp(boolean unconditional) {
    }
}
```

Проанализируем методы, заданные в данном классе, а также автоматически сгенерированные методы.

Конструктор:

```
public HelloMIDlet() {
}
```

пуст, но при необходимости в него можно написать свой код. Очевидно, что мидлет создается и запускается специальным образом, а не с помощью этого конструктора (хотя он внутри этого вызова и используется).

Метод `initialize()` инициализирует пользовательский экран приложения. Он выполняется единственный раз при запуске мидлета (листинг 14.3).

Листинг 14.3. Метод `initialize()`

```
private void initialize() {
    //write pre-initialize user code here

    //write post-initialize user code here
}
```

Комментариями

`//write ... user code here` (написать здесь пользовательский код)

помечены те участки кода, которые разрешено менять (вместо комментариев можно вставлять свой собственный исходный код). Вместо многоточия в соответствующих местах стоят названия областей кода:

- `pre-initialize code` — код, выполняющийся до инициализации;
- `post-initialize code` — код, выполняющийся после инициализации;
- `pre-action code` — код, исполняющийся до выполнения акции, и т. д.

Метод `startMIDlet()` запускается при начале работы мидлета после его инициализации (листинг 14.4).

Листинг 14.4. Метод `startMIDlet()`

```
public void startMIDlet() {
    // write pre-action user code here
    switchDisplayable(null, getForm());
    // write post-action user code here
}
```

Функция `getForm()` возвращает ссылку на инициализированный объект формы. Реализация этой функции дана далее в теле мидлета.

Метод `switchDisplayable(Alert alert, Displayable displayable)` переключает экран мобильника на новый вид, задаваемый переменной `displayable`.

`Alert` (Предупреждение) — экран со строковым сообщением или с изображением, которое показывается на короткое время (может быть задано методом `setTimeout`) для выдачи предупреждения или другого информационного сообщения. Если `alert` равно `null`,

то сразу показывается новый экран. Реализация метода witchDisplayable задается далее в самом мидлете.

Метод resumeMIDlet() (продолжить выполнение мидлета) предназначен для возобновления работы мидлета после паузы, когда мидлет приостановил работу (это обычно бывает важно для игр). Этот метод по умолчанию имеет пустую реализацию (листинг 14.5).

Листинг 14.5. Метод resumeMIDlet()

```
    public void resumeMIDlet() {
        // write pre-action user code here

        // write post-action user code here
    }
```

Реализация метода switchDisplayable (о котором уже говорилось) очень проста и фактически повторяет метод display.setCurrent. Но зато дает возможность программисту добавить код, выполняемый до переключения (pre-switch code), и код, выполняемый после переключения (post-switch code) (листинг 14.6).

Листинг 14.6. Метод switchDisplayable

```
public void switchDisplayable(Alert alert, Displayable nextDisplayable) {
    // write pre-switch user code here
    Display display = getDisplay();
    if (alert == null) {
        display.setCurrent(nextDisplayable);
    } else {
        display.setCurrent(alert, nextDisplayable);
    }
    // write post-switch user code here
}
```

Метод commandAction осуществляет обработку команд. В качестве параметров этого метода передается текущая команда и текущий экран и выполняется код, соответствующий конкретной команде. В нашем случае имеется только один экран — **helloForm**, и одна команда — **exitCommand**, так что содержимое метода очень простое (листинг 14.7).

Листинг 14.7. Метод commandAction

```
    public void commandAction(Command command, Displayable displayable) {
        // write global pre-action user code here
        if (displayable == form) {
            if (command == exitCommand) {
                // write pre-action user code here
                exitMIDlet();
```

```
                // write post-action user code here
            }
        }
        // write global post-action user code here
    }
```

Как уже говорилось, комментариями, начинающимися с `// write` (написать), помечены те места кода, которые разрешено менять (вместо комментариев можно вставлять свой собственный исходный код):

- код, выполняющийся до начала любой акции (`global pre-action code`), одинаковый для всех акций;
- код, выполняющийся до начала одной конкретной акции (`pre-action code`);
- код, выполняющийся после окончания данной акции (`post-action code`);
- код, исполняющийся после выполнения любой акции (`global post-action code`), одинаковый для всех акций.

Если вызывается акция `exitCommand`, то вызов `exitMIDlet()` (выйти из мидлета) обеспечивает выход из мидлета. Если в нашем приложении появятся новые акции, то появятся другие сравнения, имеющие вид, аналогичный приведенному в листинге 14.8.

Листинг 14.8. Обработка акций, назначенных элементам приложения

```
if (command == exitCommand) {
    // write pre-action user code here
    exitMIDlet();
    // write post-action user code here
} else
if (command == команда2) {
    // write pre-action user code here
    оператор2;
    // write post-action user code here
} else
if (command == команда3) {
    // write pre-action user code here
    Оператор3;
    // write post-action user code here
}
```

Методы `getExitCommand()` и `getStringItem()` устроены совершенно идентично. Это *фабричные методы*, создающие экземпляры соответствующих классов.

Конструкторы Java, "унаследованные" jn C++, обладают множеством недостатков, самый большой из которых — статическое связывание имени конструктора с именем класса (ведь эти имена всегда должны совпадать!). В результате написать полиморфный код невозможно. Если же мы "спрячем" вызов конструктора в функцию, которая будет создавать объект с помощью конструктора и возвращать ссылку на этот объект, то получим фабричный метод, который обладает динамическим связыванием. Заметим, что при вызове фабричных методов для создания нового объекта не нужно писать оператор `new`.

Функция getForm также относится к числу фабричных методов, но устроена чуть сложнее (листинг 14.9).

Листинг 14.9. Фабричная функция getForm

```
public Form getForm() {
    if (form == null) {
        // write pre-init user code here
        form = new Form("Welcome", new Item[] { getStringItem() });
        form.addCommand(getExitCommand());
        form.setCommandListener(this);
        // write post-init user code here
    }
    return form;
}
```

При первом вызове (когда form == null) она создает объект-форму, добавляет в нее компоненты и слушателей событий, а при последующих вызовах возвращает ссылку на этот экземпляр. В методе возможна вставка кода инициализации, предшествующего созданию формы. Также возможна вставка кода инициализации, выполняемого после создания формы и добавления компонентов и слушателей (обычно этот код используется для задания нужного состояния компонентов).

Метод getDisplay() является сокращенным вариантом вызова ссылки на текущий экран Display.getDisplay(this).

В методе exitMIDlet имеются три оператора (листинг 14.10).

Листинг 14.10. Метод exitMIDlet

```
public void exitMIDlet() {
    switchDisplayable (null, null);
    destroyApp(true);
    notifyDestroyed();
}
```

Первый из них (switchDisplayable(null, null);) устанавливает в качестве текущего дисплея ссылку null, чем обеспечивается освобождение ссылки на объект дисплея, а сборщик мусора может утилизировать данный объект.

Оператор destroyApp(true); — это вызов деструктора, обеспечивающего высвобождение из памяти ресурсов, связанных с приложением. Как уже говорилось ранее, вызов деструктора обязателен в тех случаях, когда приложение использует внешние ресурсы, являющиеся критическими (т. е. высвобождение которых жизненно важно). В нашем случае данный деструктор ничего не делает, т. к. никакие внешние ресурсы мы специально не занимали.

И наконец, последний оператор notifyDestroyed(); выдает уведомление о том, что приложение высвободило все ресурсы и разрушено.

Завершают исходный код три метода (они подчеркнуты), содержимое которых разрешено редактировать (листинг 14.11). Метод startApp вызывается при запуске приложе-

ния (в том числе при его перезапуске), `pauseApp` — при его приостановке, `destroyApp` — деструктор, о котором уже говорилось ранее.

Листинг 14.11. Методы мидлета, содержимое которых разрешено редактировать

```
 public void startApp() {
    if (midletPaused) {
        resumeMIDlet ();
    } else {
        initialize ();
        startMIDlet ();
    }
    midletPaused = false;
}

public void pauseApp() {
    midletPaused = true;
}

public void destroyApp(boolean unconditional) {
}
```

14.5. Заготовка мидлета "Крестики-нолики"

Создадим теперь полноценное приложение, выполняющее более сложные действия. Для этого повторим те же действия, что и при создании приложения **Hello, World!**, но в диалоге задания параметров установим в качестве имени приложения **CrossesZeros**. Затем можно применить рефакторинг и изменить имя класса с `HelloMidlet` на `CrossesZeros`.

Перейдем в дизайнер экрана и удалим метку **stringItem**, с помощью которой выводилась надпись **Hello, World!**. Для этого воспользуемся командой контекстного меню **Delete**, появляющегося при щелчке правой кнопкой мыши в области этой метки.

В окне **Properties** в поле для свойства **Title** (Заголовок) нашей формы введем текст `Пример мидлета` и нажмем клавишу <Enter> (можно было сделать двойной щелчок по надписи и отредактировать текст прямо в области формы). Окончание ввода обязательно заканчивать нажатием клавиши <Enter>. В заголовке формы появится введенный текст.

В палитре компонентов зайдем в категорию **Items** и перетащим на наш экран (в форму с именем **form**) компонент `TableItem` (таблицу) — он последний в этой категории.

Дважды щелкнем по заголовку таблицы и изменим текст **tableItem1** на **Крестики-нолики**.

Двойной щелчок по основной области таблицы, где пока написано **<To table model specified>** (модель таблицы не задана), приведет к открытию диалога установки модели (основных свойств) таблицы.

В появившейся форме (рис. 14.11) первоначально в списке моделей имеется только **<None>**, т. е. модели, которые могут быть связаны с таблицей, отсутствуют. Все элементы формы, кроме кнопок **Add**, **OK** и **Cancel**, недоступны, а значения полей **Rows**

(Ряды — число рядов) и **Columns** (Столбцы — число столбцов) равны нулю. Следует нажать кнопку **Add** (Добавить) для добавления модели таблицы. При этом становятся доступными кнопки добавления столбцов (**Add Row**) и строк (**Add Column**).

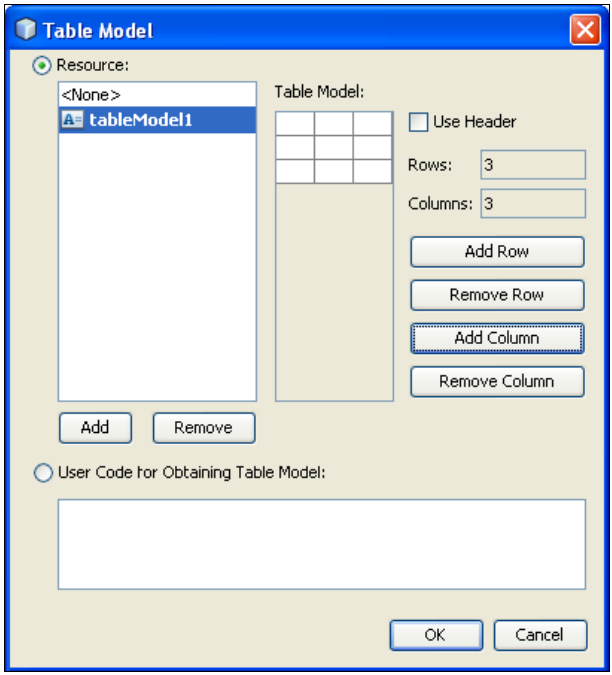


Рис. 14.11. Задание модели таблицы

В таблице нам не нужны заголовки, поэтому оставим пункт **Use Header** (Использовать заголовок) неотмеченным.

В каждую ячейку таблицы вставим по 10 пробелов (чтобы не ошибиться, лучше это делать с помощью вставки из буфера обмена). Ввод заканчивается нажатием на клавишу <Enter> или входом в режим редактирования другой ячейки. После вставки в последнюю ячейку не забудьте нажать клавишу <Enter>. Конечно, можно было бы сделать инициализацию таблицы программным путем, оставляем это в качестве упражнения читателю.

Щелкнем по имени **exitCommand** (Команда выхода) либо в области **Assigned Commands** (Присвоенные команды) окна дизайнера (см. рис. 14.9), либо в соответствующем подузле дерева **Commands** в окне **Navigator** (нижнее левое окно на рис. 14.5). После этого в окне свойств появятся свойства команды **exitCommand**, где мы изменим значение свойства **Label** (Метка) с **Exit** на **Выход** (рис. 14.12).

Теперь запустим приложение. После запуска мидлета в окне эмулятора увидим нашу таблицу (рис. 14.13), по которой можно осуществлять навигацию посредством стрелок, расположенных по бокам кнопки **Select**.

Теперь пора переходить к анализу и последующему редактированию исходного кода. В листинге 14.12 приведено чуть отредактированное содержимое этого кода — из него

убраны ненужные сейчас комментарии, с которыми мы уже знакомы по предыдущему мидлету. Подчеркиванием выделены те элементы кода, которые отличаются от предыдущего мидлета.

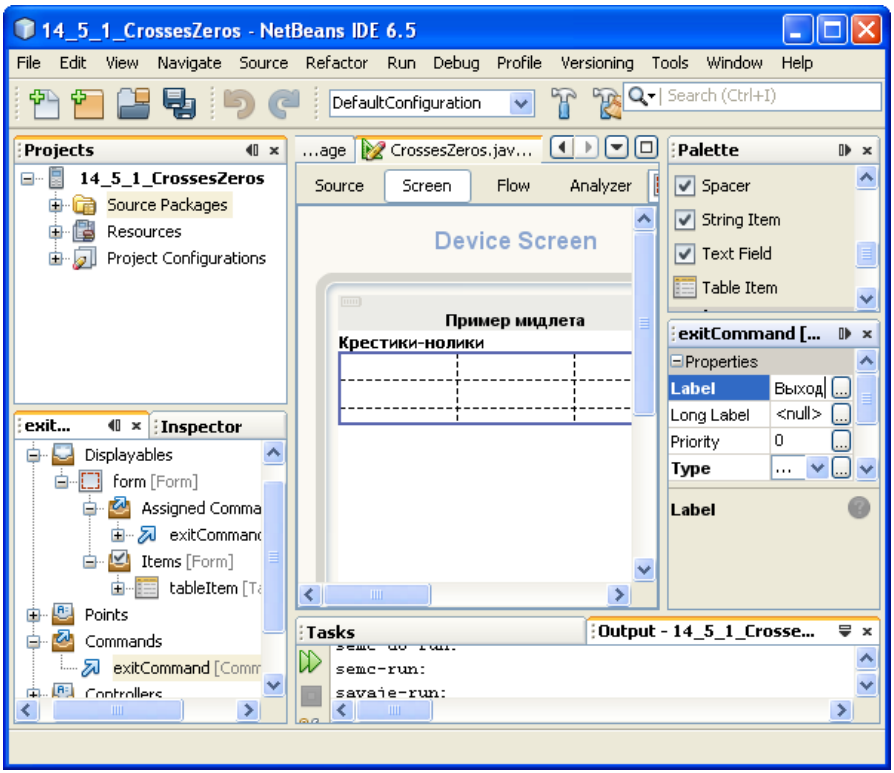


Рис. 14.12. Просмотр свойств команды **exitCommand**

Листинг 14.12. Исходный код мидлета без комментариев

```
package hello;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.SimpleTableModel;
import org.netbeans.microedition.lcdui.TableItem;

public class CrossesZeros extends MIDlet implements CommandListener {

    private boolean midletPaused = false;

    private Command exitCommand;
    private Form form;
    private TableItem tableItem;
    private SimpleTableModel tableModel1;
```

```
public CrossesZeros() {
}

private void initialize() {
    // write pre-initialize user code here

    // write post-initialize user code here
}

public void startMIDlet() {
    // write pre-action user code here
    switchDisplayable(null, getForm());
    // write post-action user code here
}

public void resumeMIDlet() {
    // write pre-action user code here

    // write post-action user code here
}

public void switchDisplayable(Alert alert,
                                    Displayable nextDisplayable) {
    // write pre-switch user code here
    Display display = getDisplay();
    if (alert == null) {
        display.setCurrent(nextDisplayable);
    } else {
        display.setCurrent(alert, nextDisplayable);
    }
    // write post-switch user code here
}

public void commandAction(Command command, Displayable displayable) {
    // write pre-action user code here
    if (displayable == form) {
        if (command == exitCommand) {
            // write pre-action user code here
            exitMIDlet();
            // write post-action user code here
        }
    }
    // write post-action user code here
}

public Command getExitCommand() {
    if (exitCommand == null) {
        // write pre-init user code here
        exitCommand = new
          Command("\u0412\u044B\u0445\u043E\u0434", Command.EXIT, 0);
```

```
            // write post-init user code here
        }
        return exitCommand;
    }

    public Form getForm() {
        if (form == null) {
            // write pre-init user code here
            form = new Form("\u041F\u0440\u0438\u043C\u0435\u0440
\u043C\u0438\u0434\u043B\u0435\u0442\u0430",
                            new Item[] { getTableItem() });
            form.addCommand(getExitCommand());
            form.setCommandListener(this);
            // write post-init user code here
        }
        return form;
    }

    public TableItem getTableItem() {
        if (tableItem == null) {
            // write pre-init user code here
            tableItem = new TableItem(getDisplay(),
"\u041A\u0440\u0435\u0441\u0442\u0438\u043A\u0438-\u043D\u043E\u043B\u0438\u043A\u0438");
            tableItem.setModel(getTableModel1());
            // write post-init user code here
        }
        return tableItem;
    }

    public SimpleTableModel getTableModel1() {
        if (tableModel1 == null) {
            // write pre-init user code here
            tableModel1 = new SimpleTableModel(new java.lang.String[][] {
                new java.lang.String[] { "          ", "          ",
                                         "          " },
                new java.lang.String[] { "          ", "          ",
                                         "          " },
                new java.lang.String[] { "          ", "          ",
                                         "          " }}, null);
            // write post-init user code here
        }
        return tableModel1;
    }

    public Display getDisplay() {
        return Display.getDisplay(this);
    }

    public void exitMIDlet() {
        switchDisplayable (null, null);
```

```
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp() {
        if (midletPaused) {
            resumeMIDlet ();
        } else {
            initialize ();
            startMIDlet ();
        }
        midletPaused = false;
    }

    public void pauseApp() {
        midletPaused = true;
    }

    public void destroyApp(boolean unconditional) {
    }

}
```

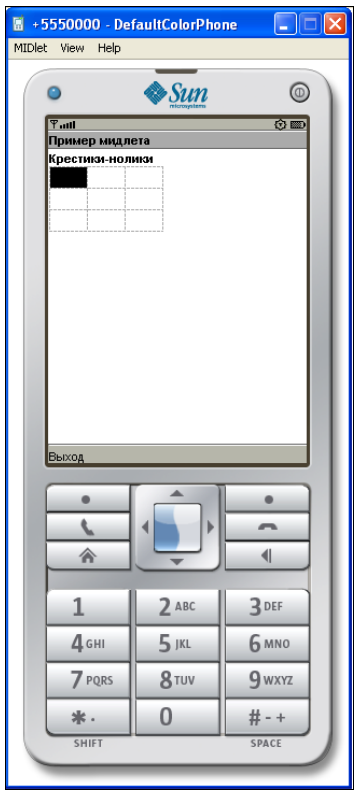


Рис. 14.13. Внешний вид заготовки в окне эмулятора

В данном коде новыми для нас являются только те части программы, которые относятся к работе с таблицей `TableItem`, а также кодирование русских букв с помощью управляющих последовательностей Unicode при инициализации текста для элементов `form` и `tableItem`.

Функция `getTableItem()` очень похожа на рассматривавшиеся ранее функции `getForm()` и `getStringItem()`. Но в вызываемом в этой функции конструкторе имеются не два, а три параметра. В качестве третьего параметра должна подставляться ссылка на модель таблицы (эта ссылка получается с помощью вызова функции `getTableModel1()`). А сам объект модели таблицы создается автоматически при помещении таблицы на форму.

Напомним, что при работе с таблицами разделяют работу с самой таблицей и работу с ее данными. За работу с данными отвечает модель — управляющий объект для этих данных.

Функция `getTableModel1()` также очень похожа на рассматривавшиеся ранее фабричные методы `getForm()` и `getStringItem()`. Во время первого вызова ссылка на объект равна `null`, поэтому вызывается конструктор `SimpleTableModel()`, а ссылка настраивается на возвращаемый им объект модели. В качестве первого параметра этому методу передается двумерный массив строк, которые должны быть видны в ячейках таблицы. Массив создается с помощью вызова оператора `new`, после которого идет статический инициализатор двумерного массива, состоящий из инициализаторов трех одномерных строковых подмассивов (рядов таблицы) (листинг 14.13).

Листинг 14.13. Создание и инициализация двумерного массива строк

```
new String[][] {
       new java.lang.String[] { "          ","          ","          "},
       new java.lang.String[] { "          ","          ","          "},
       new java.lang.String[] { "          ","          ","          "}
                }
```

Если бы мы не задали начальные значения в ячейках таблицы, то строки в инициализации были бы пустыми. Напомним, что пустая строка является полноценным объектом, и ссылка на нее не равна `null`. А вот если строковая переменная не указывает на объект-строку, то ссылка равна `null`, а попытка работы с такой строкой в большинстве случаев вызывает исключение `NullPointerException`.

Теперь необходимо научиться добавлять в мидлет свой собственный код. Для начала в редакторе исходного кода напишем заготовку метода, который будет выполняться при нажатии на правую управляющую кнопку (мы ее назовем **Ходить**) или на центральную часть кнопки **Select** (см. рис. 14.7). Такая заготовка, имеющая пустую реализацию, называется *заглушкой*. При усовершенствовании проекта мы ее заполним кодом. Дадим имя `setCrossAndZero` (поставить крестик и нолик) нашему методу-заглушке с пустой реализацией. В дальнейшем после заполнения заглушки при вызове этого метода сначала будет ставиться крестик в той ячейке, которую выбрал игрок, а затем компьютер (мобильный телефон) поставит нолик:

```
    public void setCrossAndZero() {

    }
```

Метод-заглушку `setCrossAndZero` можно вставить в любое место кода класса мидлета, доступное для редактирования, например в самый конец, после реализации метода `destroyApp` (но, конечно, перед закрывающей скобкой!). У нас получится проект, представленный на сопроводительном DVD как 14_5_CrossesZeros1.

Нам необходимо, чтобы вызов данного метода был добавлен в код акции, выполняющейся при нажатии на правую управляющую кнопку, но до этого требуется выполнить ряд предварительных действий.

Сначала нужно задать акцию, соответствующую правой управляющей кнопке. Для этого необходимо перейти в режим дизайна **Screen**, а затем перетащить из палитры компонентов из категории **Commands** элемент **Item Command** в область таблицы. Другой способ — воспользоваться командой контекстного меню **New/Add**, показывающегося при щелчке правой кнопкой мыши в окне **Navigator** по узлу **Assigned commands** для этого элемента. В результате в дереве элементов в окне **Navigator** (рис. 14.14) появится новый узел **itemCommand**.

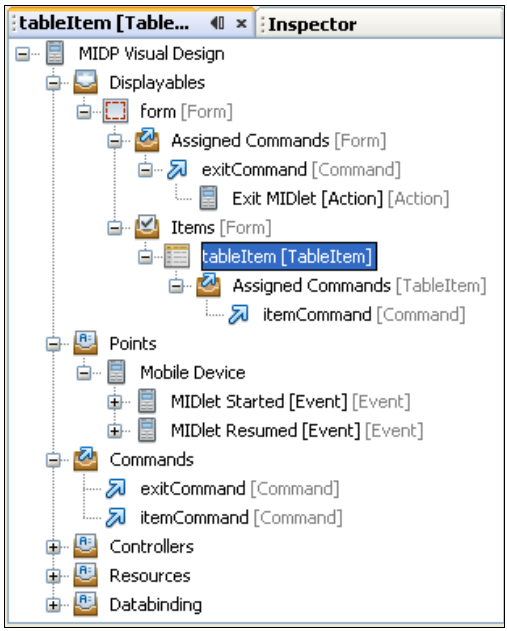


Рис. 14.14. Узел **itemCommand** в окне **Navigator**

Если вы при перетаскивании промахнулись, и команда оказалась назначена другому элементу (в нашем приложении он только один — форма **form**), не стоит расстраиваться — ее можно в любой момент времени перетащить в узел **Assigned Commands** (Назначенные команды) другого элемента.

Одну и ту же команду можно назначить нескольким элементам. Добавить элементу уже заданную команду можно с помощью пункта контекстного меню **New/Add** для этого элемента.

На экране дизайнера после добавления команды в правой части окна дизайна (**Assigned Resources**) появится область с заголовком **Assigned Item Commands** (Назначенные

пункту команды), а в ней надпись **itemCommand[tableItem]** (команда **itemCommand** для пункта **tableItem**) (рис. 14.15).

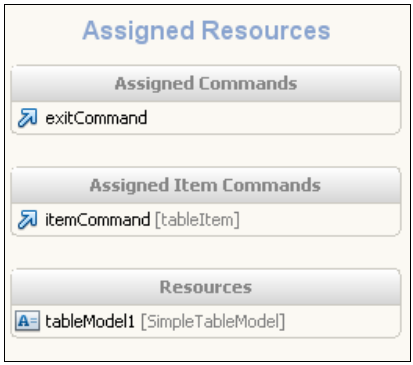


Рис. 14.15. Область **Assigned Resources** окна дизайнера после добавления команды

Щелкнем по области **itemCommand** в окне инспектора или окне дизайна, а в окне свойств зададим в поле **Label** вместо предопределенного значения **Item** строку **Ходить**. Затем щелчком правой кнопки мыши по области **itemCommand** вызовем пункт контекстного меню **Go to Source** (Перейти к исходному коду), после чего окажемся в режиме редактирования кода в появившемся после этого методе `getItemCommand()` (листинг 14.14).

Листинг 14.14. Исходный код метода `getItemCommand()`

```
public Command getItemCommand() {
    if (itemCommand == null) {
        // write pre-init user code here
        itemCommand = new Command("\u0425\u043E\u0434\u0438\u0442\u044C",
                                  Command.ITEM, 0);
        // write post-init user code here
    }
    return itemCommand;
}
```

Команда в Java ME представляется объектом, который служит меткой (т. е. объектом, с которым производится сравнение) для того, чтобы в акции выполнился код, соответствующий данной команде.

Метод `getItemCommand()` возвращает объект-команду, назначенную компоненту `TableItem`, а в случае первого вызова сначала создает этот объект.

Теперь при запуске приложения в правой нижней части экрана появится надпись **Ходить**.

Мы также хотим, чтобы нажатие центральной части кнопки **Select** (см. рис. 14.7) дублировало нажатие кнопки **Ходить**. Для этого перейдем в режим дизайна (**Screen**), выделим таблицу щелчком мыши и в окне редактора свойств в пункте **Default Command** (команда по умолчанию) в выпадающем списке выберем **itemCommand1**.

Теперь необходимые команды назначены, но при нажатии на соответствующие кнопки, естественно, ничего не происходит, т. к. наш метод `setCrossAndZero()` пока не назначен акции (реакции на вызов команды) и, кроме того, является заглушкой и ничего не делает. Данный проект на прилагаемом DVD называется 14_5_CrossesZeros2.

14.6. Код бизнес-логики мидлета

Теперь можно позаботиться о назначении акции и о наполнении заглушки реально работающим содержимым.

Обеспечим вызов метода `setCrossAndZero()` при нажатии кнопки **Ходить** или центральной управляющей клавиши (листинг 14.15). Наша модификация автоматически сгенерированного кода выделена подчеркиванием.

Листинг 14.15. Исходный код акции `commandAction`

```
public void commandAction(Command command, Item item) {
     // write pre-action user code here
     if (item == tableItem) {
         if (command == itemCommand) {
             // write pre-action user code here

             // write post-action user code here
             setCrossAndZero();
         }
     }
     // write post-action user code here
 }
```

После данного изменения при запуске программы заглушка уже будет вызываться, но пока ничего не будет делать.

Теперь наполним ее содержимым. Для начала зададим константы `CROSS` (крестик), `ZERO` (нолик) и `EMPTY` (пустая), которые будут нужны для проверки содержимого ячеек таблицы и установки в них крестиков и ноликов:

```
public String CROSS="    X     ";
public String ZERO ="    0     ";
public String EMPTY="          ";
public int count=3;
```

Переменная `count` (счетное количество чего-либо) будет служить для задания числа ячеек в рядах и колонках таблицы. Это название традиционно используется в программировании для обозначения числа элементов. Как уже говорилось, писать в каждом месте программы число 3 (или 2 вместо `count-1`) — очень плохой стиль программирования.

Во-первых, исходный код с осмысленным названием переменной гораздо лучше понимается.

Во-вторых, если вы захотите изменить алгоритм и написать такую же программу для таблицы 4×4 или 5×5, то придется вносить исправления в заранее не известное число

мест программы, да к тому же имеется вероятность ошибки (можно забыть внести где-нибудь исправление или, наоборот, заменить значение там, где оно должно остаться прежним).

Все константы и методы мы будем описывать с пакетной областью видимости, поскольку наш проект очень простой и опасаться ошибочного доступа к объектам из-за слишком свободной видимости элементов программы не приходится. В принципе, все их можно было бы описать как `private`.

Напишем булеву функцию `cellIsEmpty(int col, int row)`, которая проверяет, является ли ячейка таблицы с номером столбца `col` и номером ряда `row` пустой (листинг 14.16).

Листинг 14.16. Исходный код функции `cellIsEmpty`

```
public boolean cellIsEmpty(int col, int row) {
    String s = (String) tableModel1.getValue(col, row);
    if (s == EMPTY) {
        return true;
    } else {
        return false;
    }
}
```

Эта функция нужна для проверки того, что та ячейка, куда мы хотим поставить крестик или нолик, не содержит поставленного ранее крестика или нолика. Значение ячейки таблицы получается функцией `getValue(col, row)`, вызываемой управляющим объектом (объектом модели `simpleTableModel1`). Функция возвращает значение типа `Object`, поэтому необходимо приведение типа.

Рассмотренный код вполне работоспособен, но в нем имеется потенциальная ошибка. Мы бы ее могли не заметить. Кто из читателей заметил ошибку при просмотре приведенного фрагмента? Но среда NetBeans выдает предупреждение в столбце слева от исходного кода в виде оранжевого треугольника с восклицательным знаком внутри и подчеркиванием потенциально опасного фрагмента кода волнистой оранжевой линией (рис. 14.16).

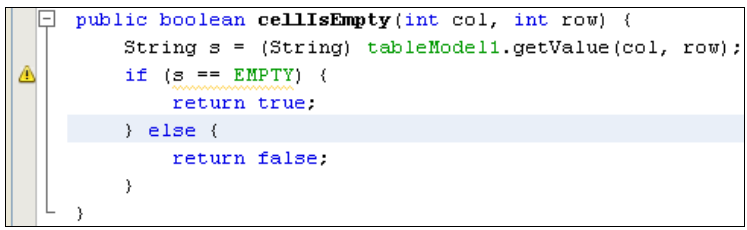


Рис. 14.16. Предупреждение среды NetBeans

При наведении на этот фрагмент курсора мыши появляется всплывающая подсказка, поясняющая возможную ошибку: **Comparing Strings using == or !=** (Сравнение строк с использованием == или !=). Мы знаем, что обычно строки сравнивают по

содержимому, а не по равенству ссылок на объекты-строки. Так что правильнее было бы написать не `if(s==EMPTY)`, а `if(s.equals(EMPTY))`. В данном конкретном случае это не имеет значения, т. к. Java хранит строковые константы в единственном экземпляре, поэтому сравнение ссылок будет эквивалентно сравнению содержимого строк. Но при работе со строками общего вида так бывает не всегда, поэтому лучше все-таки последовать совету NetBeans и исправить оператор.

Булева функция `tableIsFilled()` проверяет, не заполнена ли таблица полностью, когда уже невозможно больше сделать ни одного хода (листинг 14.17).

Листинг 14.17. Исходный код функции `tableIsFilled()`

```
public boolean tableIsFilled() {
    String s;
    boolean result = true;
    check:
    for (int col = 0; col < count; col++) {
        for (int row = 0; row < count; row++) {
            if (cellIsEmpty(col, row)) {
                result = false;
                break check;
            }
        }
    }
    return result;
}
```

Идет перебор всех ячеек таблицы (циклы по столбцам и строкам) и проверяется, не пуста ли какая-нибудь из них. Если такая ячейка находится, то внешний цикл, помеченный меткой `check`, прекращается, а функция возвращает `false`. Если же ни одной незаполненной ячейки не найдено, то возвращается значение `true`.

Процедура `setZero()` (листинг 14.18) устанавливает в случайно выбранную ячейку таблицы нолик. Процедурой называется функция, не возвращающая никакого значения. Напомним, что в Java и других C-образных языках процедуры считаются функциями, возвращающими значение `void` — пустой, пустота.

Листинг 14.18. Исходный код процедуры `setZero()`

```
public void setZero() {
    if (tableIsFilled()) {
        return;
    }
    java.util.Random r = new java.util.Random();
    int col = Math.abs(r.nextInt()) % 3;
    int row = Math.abs(r.nextInt()) % 3;
    if (cellIsEmpty(col, row)) {
        tableModel1.setValue(col, row, ZERO);
        return;
```

```
        } else {
            setZero();
        }
    }
```

Если выбранная ячейка занята, осуществляется рекурсивный вызов — снова вызывается процедура `setZero()`. Данный код вполне работоспособен, но он содержит методологическую ошибку, относящуюся к написанию программного кода. Попробуйте понять, как работает процедура, и найти, что в ней не очень хорошо.

Конечно, плох сам алгоритм установки нолика компьютером. Довольно трудно проиграть такому партнеру, который случайно ставит нолики в первую попавшуюся пустую клетку. Но методологическая ошибка не в этом, а в использовании числа 3 вместо значения `count`. При изменении числа ячеек в таблице алгоритм станет работать неправильно! А без подробного анализа исходного кода понять, что в данном случае необходимо число 3 заменить на новое значение, окажется невозможно.

Процедура `setCross(int col, int row)` (листинг 14.19) устроена схожим образом, но ставит не нолик, а крестик, и не в случайно выбранную ячейку, а в ту, номера столбца и строки которой переданы в списке параметров.

Листинг 14.19. Исходный код процедуры `setCross()`

```
public void setCross(int col, int row) {
    if (tableIsFilled()) {
        return;
    }
    if (cellIsEmpty(col, row)) {
        tableModel1.setValue(col, row, CROSS);
        return;
    }
}
```

Булевы функции `rowIsFilled` (строка заполнена) и `columnIsFilled` (столбец заполнен) (листинг 14.20), а также `mainDiagonalIsFilled` (главная диагональ заполнена) и `auxiliaryDiagonalIsFilled` (вспомогательная диагональ заполнена) (листинг 14.21) очень похожи друг на друга. В них организуется цикл, в котором проверяется, что соответствующая строка, столбец или диагональ заполнены символом, передаваемым в функцию в качестве параметра.

Листинг 14.20. Исходный код функций `rowIsFilled` и `columnIsFilled`

```
public boolean rowIsFilled(int row, String symbol) {
    String sCell;
    for (int col = 0; col < count; col++) {
        sCell = (String) tableModel1.getValue(col, row);
        if (!sCell.equals(symbol)) {
            return false;
        }
    }
```

```
        return true;
    }

    public boolean columnIsFilled(int col, String symbol) {
        String sCell;
        for (int row = 0; row < count; row++) {
            sCell = (String) tableModel1.getValue(col, row);
            if (!sCell.equals(symbol)) {
                return false;
            }
        }
        return true;
    }
```

Булевы функции `anyRowIsFilled` и `anyColumnIsFilled` (листинг 14.21) проверяют в цикле все столбцы и, соответственно, строки таблицы, — не найдется ли среди них полностью заполненного (заполненной) заданным символом. Если находится, то функция возвращает `true`, иначе — `false`.

А функции `mainDiagonalIsFilled` и `auxiliaryDiagonalIsFilled` (тот же листинг 14.21) проверяют, не заполнена ли главная диагональ таблицы (идущая из левого верхнего угла в правый нижний) и вспомогательная (идущая из правого верхнего угла в левый нижний).

Данные проверки необходимы для того, чтобы выяснить, не кончилась ли игра, нет ли столбца, строки или диагонали, полностью заполненной крестиками или ноликами.

Листинг 14.21. Исходный код функций `anyRowIsFilled`, `anyColumnIsFilled`, `mainDiagonalIsFilled` и `auxiliaryDiagonalIsFilled`

```
public boolean anyRowIsFilled(String symbol) {
    for (int row = 0; row < count; row++) {
        if (rowIsFilled(row, symbol)) {
            return true;
        }
    }
    return false;
}

public boolean anyColumnIsFilled(String symbol) {
    for (int column = 0; column < count; column++) {
        if (columnIsFilled(column, symbol)) {
            return true;
        }
    }
    return false;
}

public boolean mainDiagonalIsFilled(String symbol) {
    String sCell;
    for (int i = 0; i < count; i++) {
        sCell = (String) tableModel1.getValue(i, i);
```

```
            if (!sCell.equals(symbol)) {
                return false;
            }
        }
        return true;
    }

    public boolean auxiliaryDiagonalIsFilled(String symbol) {
        String sCell;
        for (int i = 0; i < count; i++) {
            sCell = (String) tableModel1.getValue(i,count-1-i);
            if (!sCell.equals(symbol)) {
                return false;
            }
        }
        return true;
    }
```

Функция `winning` (листинг 14.22) проверяет, не выигрывает ли участник, ставящий символ (это крестик `CROSS` или нолик `ZERO`) в таблицу.

Листинг 14.22. Исходный код функции `winning`

```
    public boolean winning(String symbol) {
        if ( anyRowIsFilled(symbol) ||
             anyColumnIsFilled(symbol) ||
             mainDiagonalIsFilled(symbol) ||
             auxiliaryDiagonalIsFilled(symbol)) {
            return true;
        }
        return false;
    }
```

Теперь мы можем заполнить "заглушку" `setCrossAndZero` функциональным содержанием. Процедура `setCrossAndZero` (листинг 14.23) с помощью процедуры `setCross` ставит крестик в ячейке, выбранной игроком (но только в том случае, если игра не окончена и эта ячейка пуста), после чего с помощью процедуры `setZero` ставит нолик в случайно выбранную пустую ячейку (но только в том случае, если крестики не выиграли). При победе одной из сторон в заголовке таблицы выводится соответствующее сообщение.

Листинг 14.23. Исходный код процедуры `setCrossAndZero()`

```
    public void setCrossAndZero() {
        if (winning(CROSS) || winning(ZERO)) {
            return;
        }
        int row = tableItem.getSelectedCellRow();
        int col = tableItem.getSelectedCellColumn();
        String s = (String) tableModel1.getValue(col, row);
```

```
        if (cellIsEmpty(col, row)) {
            setCross(col, row);
        } else {
            return;
        }
        if (winning(CROSS)) {
            tableItem.setLabel("Победили крестики!");
            return;
        }
        setZero();
        if (winning(ZERO)) {
            tableItem.setLabel("Победили нолики!");
        }
    }
```

После написания приведенных методов мы получим код, представленный в примере 14_6_CrossesZeros1 на DVD, прилагаемом к книге. Если мы запустим этот пример и посмотрим, как он работает, то обнаружим, что после сделанного хода новые крестики и нолики появляются только при передвижении курсора на новую ячейку. Причина проста — содержимое экрана не обновляется после изменения содержимого таблицы. И разумным способом данную проблему не решить. Причем это ошибка разработчиков компонента, а не наша. Дело в том, что в компоненте `TableItem` метод `repaint()`, позволяющий обновить изображение компонента, сделан разработчиками защищенным (protected).

Существует не очень удачный способ решения этой проблемы. После изучения исходного кода компонента становится ясно, что метод `repaint` вызывается, например, в методе `setBorders` класса `TableItem` (листинг 14.24).

Листинг 14.24. Устройство метода `setBorders`

```
    public void setBorders(boolean showBorders) {
        if (this.borders != showBorders) {
            this.borders = showBorders;
            repaint();
        }
    }
```

Метод `setBorders` устанавливает режим отрисовки таблицы: если его параметр имеет значение `true`, то в таблице рисуются границы ячеек, а если он равен `false`, то не рисуются. После каждого изменения значения таблица перерисовывается благодаря вызову метода `repaint()`. Таким образом, можно изменить исходный код метода так, как указано в листинге 14.25. Добавленные две строки выделены подчеркиванием.

Листинг 14.25. Усовершенствованный вариант процедуры `setZero()`

```
    public void setZero() {
        if (tableIsFilled()) {
            return;
        }
```

```
        java.util.Random r = new java.util.Random();
        int col = Math.abs(r.nextInt()) % 3;
        int row = Math.abs(r.nextInt()) % 3;
        if (cellIsEmpty(col, row)) {
            tableModel1.setValue(col, row, ZERO);
            tableItem.setBorders(false);
            tableItem.setBorders(true);

            return;
        } else {
            setZero();
        }
    }
```

Аналогичные действия выполняются и при изменении цветовой модели таблицы, при изменении ее размера и т. д. Но проще всего воспользоваться изменением булевого параметра в методе `setBorders`. Как будет работать мидлет с добавлением двух данных строк, можно посмотреть, запустив пример 14_6_CrossesZeros2.

Конечно, данный способ решения проблемы — это "закапывание ошибки", а не ее исправление.

Написанные нами методы бизнес-логики работают, и работают правильно. Но они не столь хороши, как может показаться — при изменении кода, которое мы сделаем в следующем разделе, это станет заметно.

14.7. Создание собственного компонента, являющегося наследником *TableItem*, и мидлета на его основе. Создание библиотеки

Для того чтобы разработать полноценное приложение без "закапывания ошибки", напишем свой собственный компонент, являющийся наследником `TableItem`. Помимо всего прочего, это будет хорошей практикой в освоении методики создания собственных компонентов NetBeans для Java Micro Edition.

Как мы знаем, следует отделять код, предназначенный для повторного использования, от однократно задействованного кода. Точно так же следует отделять многократный код общего назначения от специализированного (не настолько общего) многократно используемого. В нашем случае мы собираемся усовершенствовать компонент `TableItem`. Естественно, что этот компонент не имеет непосредственного отношения к нашей задаче, поскольку он заменяет `TableItem` в любых программах Java Micro Edition. Это означает, что необходимо создать библиотечный компонент, т. е. такой, который пригоден для самого широкого набора задач. Таким образом, необходим не просто отдельный класс, а отдельная библиотека.

Создадим новый проект библиотеки компонентов для мобильных устройств: **File/ New Project.../**, затем выберем категорию **Java ME**, тип проекта **Mobile Class Library**, нажмем кнопку **Next >** и в открывшейся форме введем в качестве имени библиотеки **MobileItemsLibrary**.

После нажатия кнопки **Finish** появится проект, в котором будем формировать новый компонент.

Создадим в нашем проекте пакет `itemspackage`: щелчок правой кнопкой мыши по имени пакета **Default Package**, выбор пункта **New/Java Package...** в контекстном меню, ввод в качестве имени пакета `itemspackage`.

Затем аналогично создадим в этом пакете заготовку класса `TableItemVM` (щелчок правой кнопкой мыши по имени пакета `itemspackage`, выбор пункта **New/Java Class...** в контекстном меню, ввод в качестве имени класса `TableItemVM`, нажатие кнопки **Finish**). Аббревиатура VM — сокращение от Vadim Monakhov.

Далее заменим код класса на приведенный в листинге 14.26.

Листинг 14.26. Исходный код усовершенствованного компонента таблицы

```
/*
 * vmTableItem.java
 *
 */

package itemspackage;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import org.netbeans.microedition.lcdui.*;

/**
 *
 * @author Vadim Monakhov
 */
public class TableItemVM extends TableItem {

    public TableItemVM (Display display, String title, TableModel model) {
        super(display, title, model);
    }

    public void repaintTable(){
        super.repaint();
    }
}
```

Данный код крайне прост.

Во-первых, мы импортируем необходимые нам классы из пакетов:

```
javax.microedition.midlet,
javax.microedition.lcdui
org.netbeans.microedition.lcdui,
```

после чего объявляем наш класс наследником `TableItem` и задаем конструктор классического вида (в нем имеется только вызов прародительского конструктора с теми же параметрами).

И наконец, мы задаем метод `repaintTable()`, который будет заменять нам метод `repaint()`. Довольно странная особенность задания метода `repaint()` в классе `javax.microedition.lcdui.CustomItem` заключается в том, что он объявлен как `final protected`, т. е. предназначенный для использования в наследниках, но недоступный для переопределения. Из-за наличия модификатора `protected` данный метод нельзя вызывать из объекта — он может быть использован только при задании класса-наследника. Но в наследнике мы не можем задать метод с тем же именем и сигнатурой (т. е. без параметров) из-за присутствия модификатора `final`. Поэтому в наследнике нам приходится давать методу, осуществляющему перерисовку таблицы, другое имя — `repaintTable()`.

В нашем классе нет ошибок, но если мы посмотрим на редактор исходного кода, то он подчеркнет красной волнистой линией несколько мест в коде класса (рис. 14.17).

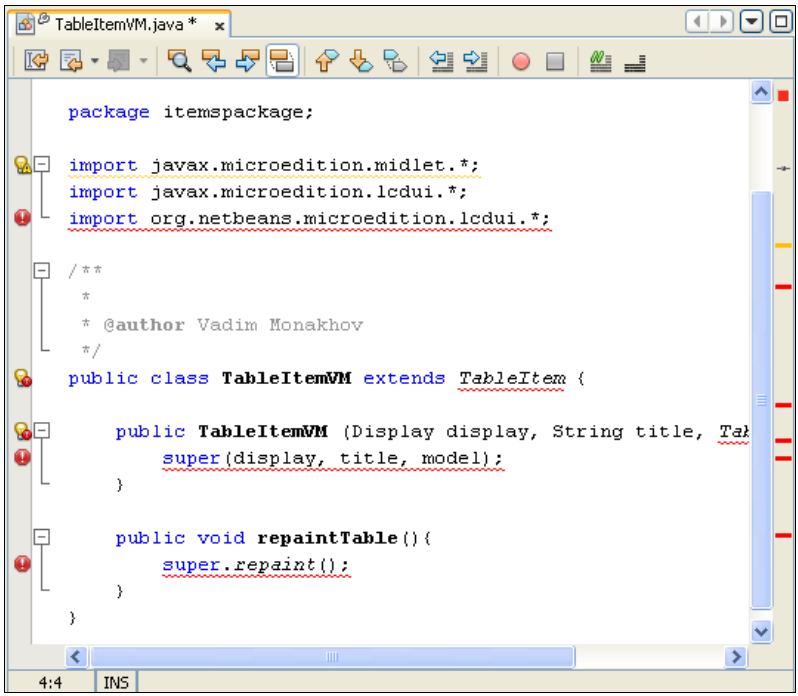


Рис. 14.17. Диагностика ошибок из-за отсутствия библиотеки в ресурсах проекта

Дело в том, что среда разработки не может импортировать пакет org.netbeans.microedition.lcdui — он не входит в состав библиотек NetBeans, путь к которым находится автоматически. Вообще, если по каким-то причинам идет сообщение, что импортируемый пакет не найден, это означает, что jar-файл, содержащий данный пакет, необходимо добавить к ресурсам проекта.

Щелкнем правой кнопкой мыши в дереве проектов по имени папки **Resources** (Ресурсы), тем самым вызовем контекстное меню (рис. 14.18), выберем в нем пункт **Add Library...** (Добавить библиотеку). В появившейся форме выберем библиотеку **NetBeans MIDP Components** и нажмем кнопку **Add Library** (рис. 14.19).

Диагностика ошибок исчезнет, и мы можем скомпилировать наш компонент.

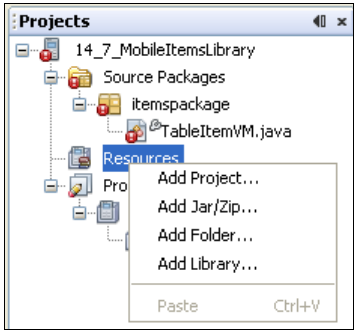


Рис. 14.18. Добавление библиотеки компонентов в ресурсы — шаг 1

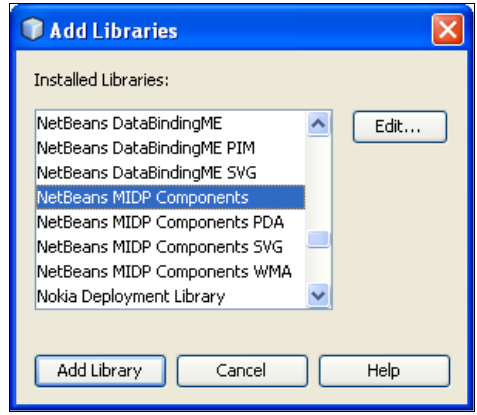


Рис. 14.19. Добавление библиотеки компонентов в ресурсы — шаг 2

Если мы захотим добавить библиотеку, не установленную в среде NetBeans, а находящуюся в jar-файле, следует выбрать пункт **Add Jar/Zip...** и указать соответствующий путь.

Если мы хотим узнать, где размещается на диске та или иная библиотека, можно вызвать в главном меню среды NetBeans пункт **Tools/Libraries** и выбрать имя нужной библиотеки. После этого путь к jar-файлу с двоичными кодами скомпилированной библиотеки будет показан на вкладке **Classpath**, а путь к zip-архиву с исходными кодами библиотеки (если он присутствует) — на вкладке **Sources**.

После компиляции проекта в папке NetBeansProjects\MobileItemsLibrary\dist мы обнаружим файл MobileItemsLibrary.jar, в котором содержится скомпилированный код нашего компонента.

Теперь необходимо добавить наш компонент в палитру компонентов. Это делается в два этапа: сначала наша библиотека добавляется в список установленных в среде NetBeans библиотек, после чего она добавляется в палитру компонентов.

Для выполнения первого этапа (добавления нашей библиотеки в список установленных в среде NetBeans библиотек) сначала необходимо вызвать в главном меню среды NetBeans пункт **Tools/Libraries**, нажать кнопку **New library...** и в открывшейся форме (рис. 14.20) ввести имя нужной библиотеки (**MobileItemsLibrary**). В качестве типа библиотеки оставить тип, предлагаемый по умолчанию, — **Class Libraries**, после чего нажать кнопку **OK**. В списке библиотек в диалоговой форме **Library Manager** (рис. 14.21) появится имя **MobileItemsLibrary**, но сам jar-файл библиотеки пока не добавлен.

Далее на вкладке **Classpath** после нажатия кнопки **Add JAR/Folder** (Добавить JAR/папку) нужно выбрать путь к нашему jar-файлу скомпилированной библиотеки (например, C:\ Documents and Settings \User \Мои документы \NetBeansProjects \MobileItemsLibrary \dist \MobileItemsLibrary.jar). Этот путь появится в окне вкладки **Classpath**, после чего можно нажать кнопку **OK**, закрывающую форму **Library Manager**.

Второй этап (добавление установленной библиотеки в палитру компонентов) можно проводить разными способами. С точки зрения автора наиболее удобный из них следующий.

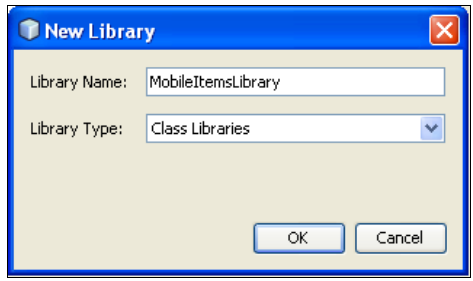


Рис. 14.20. Добавление в NetBeans собственной библиотеки — шаг 1

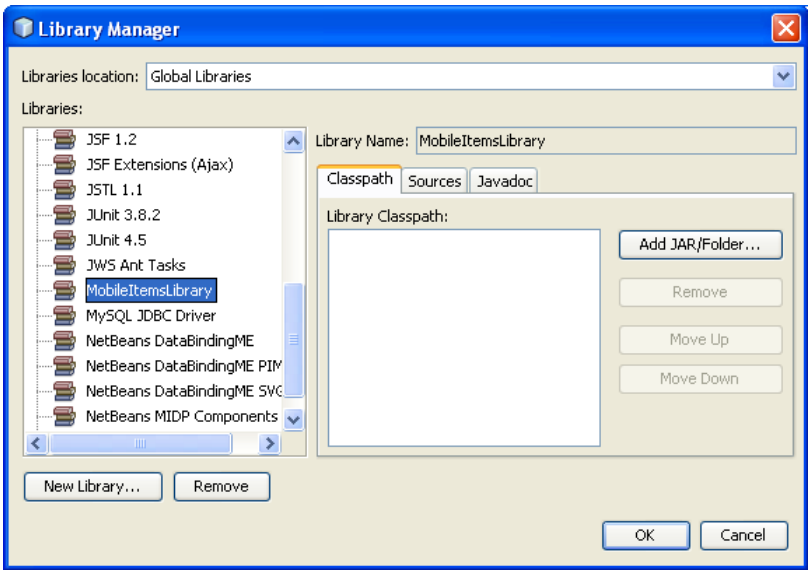


Рис. 14.21. Добавление в NetBeans собственной библиотеки — шаг 2

В редакторе перейдем в редактирование исходного кода нашего компонента. После чего вызовем в главном меню NetBeans пункт **Tools/Palette/MIDP Visual Mobile Designer**. В открывшейся форме (рис. 14.22) в качестве страницы палитры, к которой будем добавлять компонент, выберем имя **Custom** (Компоненты по заказу). Не следует добавлять свои компоненты на стандартные страницы палитры — это крайне дурной тон!

Нажмем кнопку **Add to Palette...** (Добавить в палитру), в появившейся форме **Add to Palette wizard** (рис. 14.23) в выпадающем списке открытых проектов (раздел **Select Project**) выберем наш проект **MobileItemsLibrary** и нажмем кнопку **Next >**. Появится форма с таким же заголовком **Add to Palette wizard** и с именем компонента **itempackage.TableItemVM** (рис. 14.24).

Выделим это имя и нажмем кнопку **Finish** — установка компонента закончена.

Замечание

Можно разрабатывать компонент в пакете `itemspackage` непосредственно в проекте CrossesZeros и устанавливать компонент в палитру из этого проекта (так, например, было сделано в проекте 14_7_CrossesZeros1).

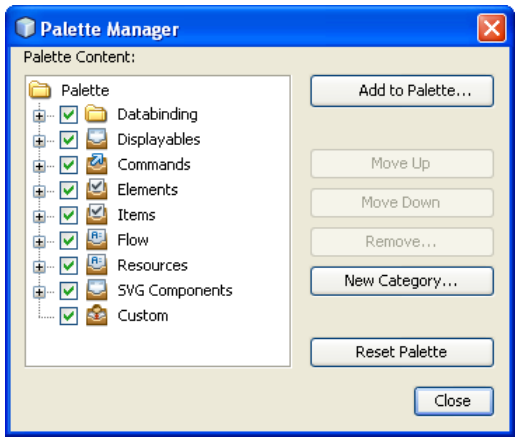


Рис. 14.22. Добавление компонента MIDP из библиотеки в палитру компонентов — шаг 1

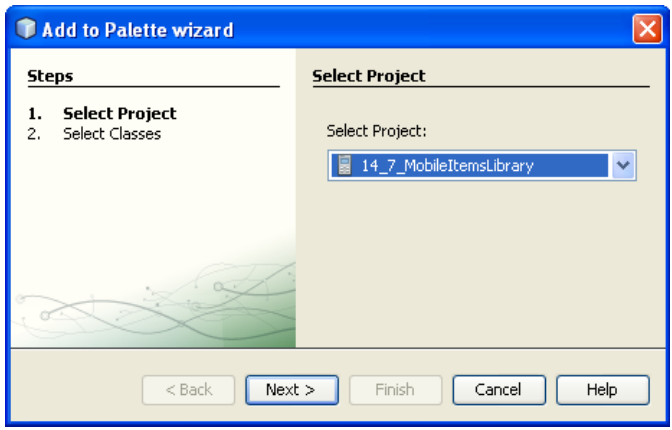


Рис. 14.23. Добавление компонента MIDP из библиотеки в палитру компонентов — шаг 2

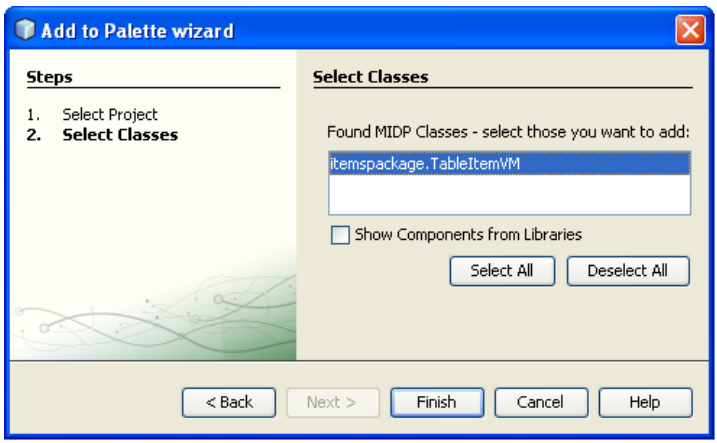


Рис. 14.24. Добавление компонента MIDP из библиотеки в палитру компонентов — шаг 3

Чтобы установленный компонент появился в палитре, нужно в опять появившейся на экране форме **Palette Manager** нажать кнопку **Reset Palette** (Переустановить палитру). Отметим, что для добавления и удаления компонентов форму **Palette Manager** можно вызвать в любое время с помощью пункта меню **Tools/Palette/ Mobility Components**. Но в ряде случаев (например, для удаления только что установленного компонента) требуется открыть проект приложения Java ME (например, MobileApplication1 или 14_7_CrossesZeros1) — вызов данного менеджера из проекта с библиотекой классов оставляет кнопку **Remove** недоступной (рис. 14.25), и компонент удалить невозможно.

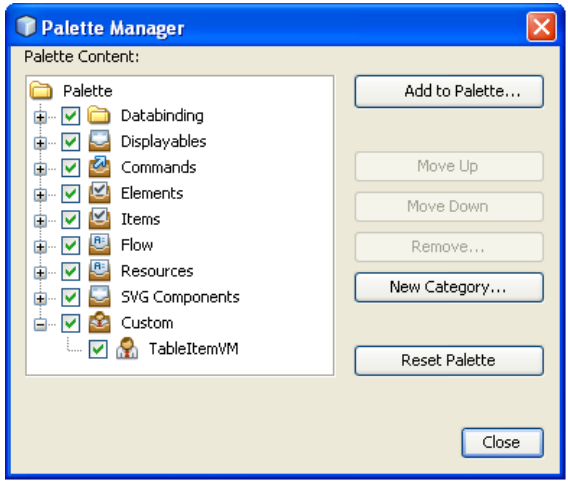


Рис. 14.25. Окончание установки компонента MIDP в палитру

После установки компонента в дереве формы **Palette Manager** (Менеджер палитры) в узле **Custom** появится новый узел **TableItemVM** (рис. 14.25). Кроме того, в режиме дизайна мы увидим наш компонент в палитре компонентов на странице **Custom**. Теперь им можно будет пользоваться в любом проекте с мидлетами почти наравне со стандартными компонентами (только каждый раз в ресурсы проекта придется добавлять библиотеку **MobileItemsLibrary**). Сейчас можно воспроизвести все наши действия по созданию игры "Крестики-нолики", но уже с данным компонентом.

Когда мы будем создавать приложение на основе нового компонента, лучше сформировать новый мидлет `Hello` и воспроизвести весь процесс разработки приложения "Крестики-нолики" с новым компонентом. При этом в нашем коде бизнес-логики потребуется заменить имя `tableItem` на `tableItemVM` и вставить вызов `repaintTable()` в метод `setZero()` (листинг 14.27).

Листинг 14.27. Усовершенствованный вариант метода `setZero()`

```
public void setZero() {
    if (tableIsFilled()) {
        return;
    }
    java.util.Random r = new java.util.Random();
```

```
        int col = Math.abs(r.nextInt()) % 3;
        int row = Math.abs(r.nextInt()) % 3;
        if (cellIsEmpty(col, row)) {
            tableModel1.setValue(col, row, ZERO);
            tableItemVM.repaintTable();
            return;
        } else {
            setZero();
        }
    }
```

Отметим, что все фрагменты кода бизнес-логики, где нам пришлось переименовать `tableItem` на `tableItemVM`, были написаны идеологически неправильно. Параметр `tableItem` должен передаваться в соответствующие методы через список параметров — в этом случае сам исходный код методов бизнес-логики не будет меняться при замене `tableItem` на `tableItemVM`. А замена должна проводиться в той части кода, которая относится к пользовательскому интерфейсу, и только в вызове методов.

Теперь скомпилируем и запустим приложение, при этом обнаружим, что возникает исключительная ситуация с сообщением **display parameter cannot be null**. Дело в том, что для нашего компонента в автоматически сгенерированном коде для метода `getTableItemVM()` в вызове `new TableItemVM` в качестве первого параметра вместо вызова `getDisplay()` стоит `null`. А при таком значении параметра `display` возбуждается исключительная ситуация. Причем заменить сгенерированный код нельзя. Но никто не мешает нам закомментировать этот код (неизменяемый код помечен подчеркиванием), а после комментария написать правильный код (листинг 14.28).

Листинг 14.28. Исправление ошибки в автоматически сгенерированном коде конструктора таблицы

```
public TableItemVM getTableItemVM() {
    if (tableItemVM == null) {
        // write pre-init user code here
    /*
        tableItemVM = new TableItemVM(null, null, null);
        tableItemVM.addCommand(getItemCommand());
        tableItemVM.setItemCommandListener(this);
        tableItemVM.setDefaultCommand(getItemCommand());
        tableItemVM.setLabel( "\u041A\u0440\u0435\u0441\u0442\u0438\u043A\u0438-
\u043D\u043E\u043B\u0438\u043A\u0438");
        tableItemVM.setModel(getTableModel1());
        // write post-init user code here
     */
        tableItemVM = new TableItemVM(getDisplay(),
"\u041A\u0440\u0435\u0441\u0442\u0438\u043A\u0438-
\u043D\u043E\u043B\u0438\u043A\u0438",null);
        tableItemVM.addCommand(getItemCommand());
        tableItemVM.setItemCommandListener(this);
        tableItemVM.setDefaultCommand(getItemCommand());
```

```
        //оператор tableItemVM.setLabel ненужен
        tableItemVM.setModel(getTableModel1());
    }
    return tableItemVM;
}
```

При этом пришлось закомментировать, а потом повторить еще и четыре строки с совершенно правильными операторами, т. к. у нас нет возможности редактировать автоматически сгенерированный код. А в вызове конструктора `TableItemVM` обязательно указать третий параметр, равный `null`, иначе наше приложение не будет компилироваться.

Теперь приложение сможет нормально работать, перерисовывая экран благодаря вызову метода `repaintTable()` (пример на DVD 14_7_CrossesZeros2).

Следует отметить, что это приложение — не более чем заготовка. К сожалению, команда по умолчанию не действует для джойстика, и необходимо обрабатывать события нажатия клавиш устройства (но это уже выходит за рамки книги, в которой дается только краткое описание работы с мобильными устройствами).

14.8. Запуск мидлета "Крестики-нолики" на реальном телефоне

Проще всего запускать мидлеты на современных телефонах Sony Ericsson. В них предусмотрена возможность вызова приложений Java непосредственно из jar-архивов, скопированных в память телефона (jad-файл не нужен).

При копировании через Bluetooth или инфракрасный порт приложение автоматически опознается, нам предлагается сохранить его в папке приложений или в папке игр (рис. 14.26, *а*). После сохранения предлагается сразу запустить приложение (рис. 14.26, *б*). В случае согласия можно сразу увидеть наши "Крестики-нолики" в действии (рис. 14.26, *в*).

В случае копирования через кабель с компьютера в память телефона (в том числе на карту внешней памяти) скопированный файл можно запустить на телефоне с помощью менеджера файлов (рис. 14.27). При установке курсора на имя файла приложения в меню команд телефона возникает пункт **Установ.** (сокращение от Установить). Нажатие на соответствующую клавишу приводит к появлению тех же диалогов, что и при передаче через Bluetooth или инфракрасный порт — предлагается сохранить его в папке приложений или папке игр и т. д.

На рис. 14.26 и 14.27 приведены фотографии установки приложения Java на телефон Sony Ericsson K710i.

Для телефонов Samsung процедура установки иная — необходимо скопировать jad- и jar-файлы в память телефона, выбрать в менеджере файлов jad-файл и воспользоваться опцией установки. Без jad-файла установка не производится. Обратите внимание, что устанавливать приложение нужно из памяти телефона, а не с flash-карты. Кроме того, в jad-файле обязательно должны быть указаны два параметра: `MIDlet-Jar-Size` и `MIDlet-Jar-URL`, например:

```
MIDlet-Jar-Size: 47592
MIDlet-Jar-URL: Applic1.jar
```

а *б* *в*

Рис. 14.26. Запуск мидлета на телефоне Sony Ericsson при копировании через Bluetooth или ИК-порт

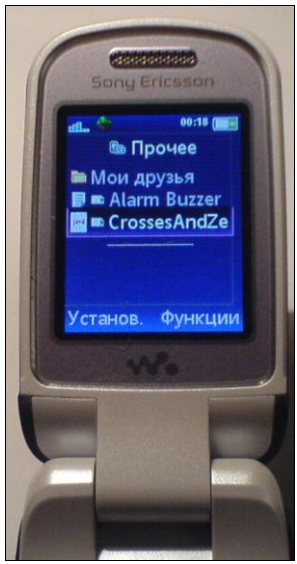


Рис. 14.27. Запуск мидлета на телефоне Sony Ericsson из файла

Не следует редактировать сам jad-файл. Эту информацию необходимо добавлять в свойства проекта в разделе **Application Descriptor** (Файл описания приложения — так официально называется jad-файл).

Для телефонов других фирм, поддерживающих Java, обычно скачивают файл приложения с WAP-сайта Интернета, указывая для этого в качестве ссылки имя jad-файла, находящегося на этом сайте.

14.9. Пример мидлета с переключением экрана и списком выбора

Создадим мидлет **Hello, World!**. Переключимся в режим **Screen**, в окне **Navigator** щелчком мыши по пункту **Displayables** вызовем контекстное меню, выберем в нем пункт **New/Add/List** (рис. 14.28).

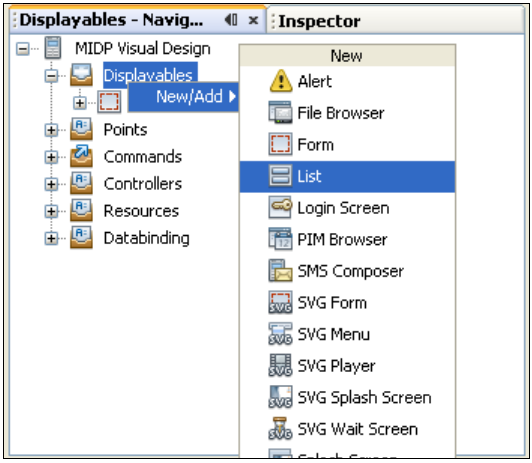


Рис. 14.28. Добавление в мидлет экрана со списком

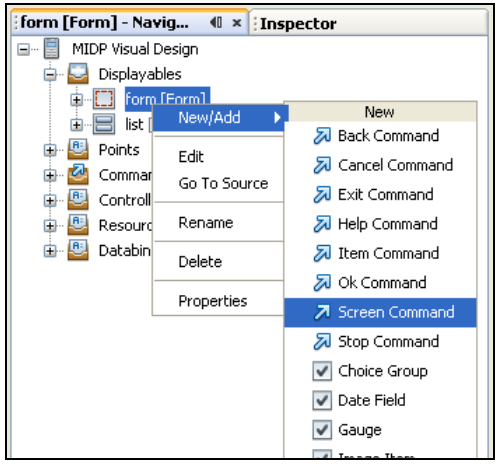


Рис. 14.29. Добавление команды переключения экрана

Точно так же после разворачивания появившегося узла **list[List]** следует по очереди добавлять в его подузел **Elements** несколько элементов **List Element**.

Аналогично добавим команду **Screen Command** (Команда Экран) главной форме (рис. 14.29).

В узле **Assigned Commands** экрана **form[Form]** появится подузел **Screen Command[Command]**. Выделим его и в окне свойств (**Properties**) в самом первом поле **Action** нажмем кнопку [...].

Появится диалог, позволяющий задать параметры акции соответствующей команде. По умолчанию в нем выбран вариант **Do Nothing** (Ничего не делать). Необходимо выбрать вариант **Switch to Displayable** (Переключить на новый вид экрана), а в выпадающем списке выбрать экран **list** и нажать кнопку **OK** (рис. 14.30).

Узлу **Assigned Commands** экрана **list[List]** добавим команду **Back Command** (Команда Обратно), а в окне установки акции для этой команды с целью переключения обратно к форме **form** сменим вариант **Do Nothing** на **Switch to Displayable** (Переключить экран).

Далее можно перевести на русский язык надписи мидлета. Например, надпись **Screen** для команды **Screen Command[Command]** можно заменить на **Экран**, после чего запустить получившийся мидлет (пример на DVD 14_9_MobileApplication).

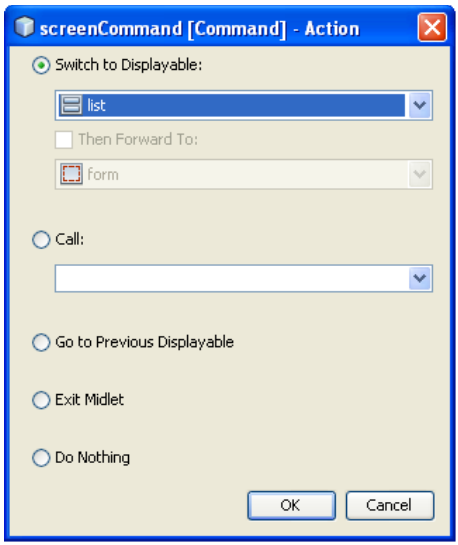


Рис. 14.30. Задание параметров акции соответствующей команде

14.10. Примеры приложений для мобильных телефонов, поставляемые со средой NetBeans

В поставке NetBeans 6.5 имеется очень много примеров приложений для мобильных устройств как для конфигурации CLDC для профиля MIDP, так и (хотя и в значительно меньшем количестве) для конфигурации CDC.

В NetBeans 6.5 количество категорий с примерами Java ME возросло с одной до трех. Выберем пункт меню **File/New Project.../** и в открывшейся форме (рис. 14.31) для начала выберем категорию **Samples/Java ME**.

Проанализировать все приложения, конечно, невозможно — это материал для отдельной книги. Поэтому ограничимся кратким перечислением некоторых из них.

Самым первым идет **Visual Designer Custom Components** (компоненты по заказу для визуального дизайнера), содержащий четыре примера (рис. 14.32):

- **Login Screen Example** — показывает работу экрана авторизации, когда предлагается ввести имя и пароль;
- **SMS Composer Example** — редактор SMS-сообщений с имитацией их отсылки;
- **File Browser Example** — иллюстрирует работу диалога выбора файла в памяти телефона;
- **PIM Browser Example** — иллюстрирует работу с электронной записной книжкой (адресной книгой, контактами) телефона. Напомним, что PIM — Personal Information Manager.

Рассмотрим еще некоторые приложения.

- **Simple Game created with Game Builder** — пример игры, созданной с помощью Редактора игр (рис. 14.33).

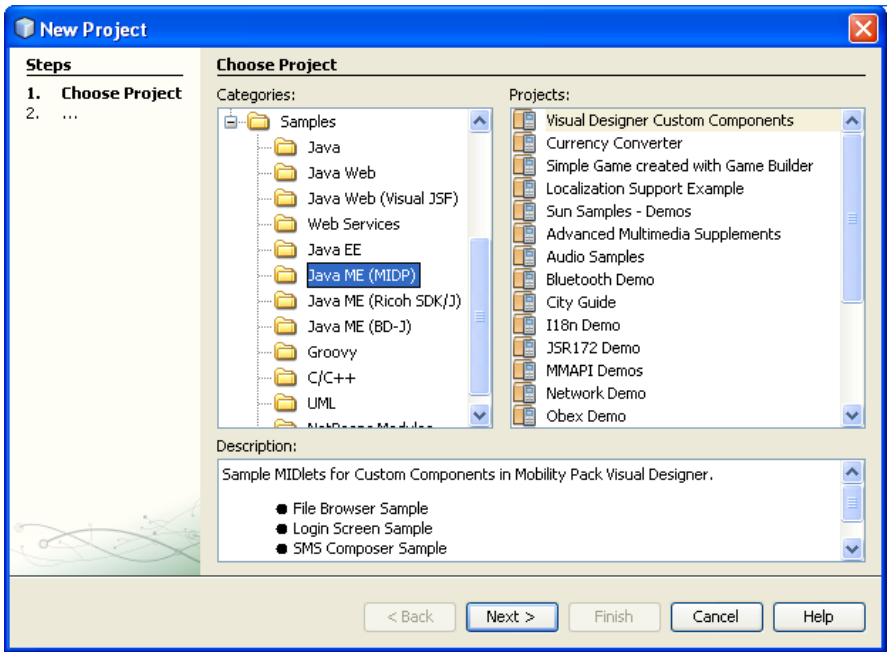


Рис. 14.31. Проекты примеров приложений для мобильных устройств

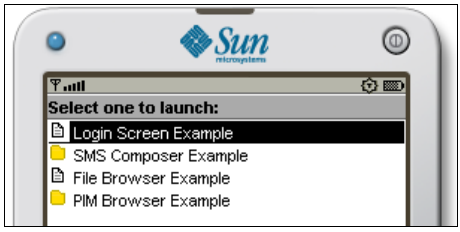


Рис. 14.32. Четыре примера **Visual Designer Custom Components**

Рис. 14.33. Экран примера **Simple Game created with Game Builder**

Вместе с заготовкой игры поставляется и сам редактор (файл GameDesign.java). При открытии этого файла в редакторе можно перейти на вкладку **Game Builder** и редактировать сцены, слои игры и спрайты (растровые изображения, свободно перемещающиеся по экрану).

- **Sun Samples — Demos** — двенадцать различных примеров, подготовленных фирмой Sun (рис. 14.34). Сначала идет иллюстрация работы с цветными областями, затем идет пример получения системной информации (рис. 14.35).
- **Audio Samples** — позволяет программисту использовать разные варианты проигрывания музыкальных файлов (у вас есть шанс написать свой собственный проигрыватель для мобильников!).
- **Advanced Multimedia Supplements** — развитые музыкальные возможности. Запуск данных приложений происходит не очень быстро, так что после выбора соответствующего приложения придется немного подождать. Рисунок 14.36 иллюстрирует обработку изображений, работу радиотюнера, управление видеокамерой, дает пример простейшей игрушки с озвучиванием "Движущийся вертолет", и, наконец, представляет работающую заготовку музыкального центра с эквалайзером и ревербератором (рис. 14.37).

Рис. 14.34. Экран примера **Sun Samples — Demos**

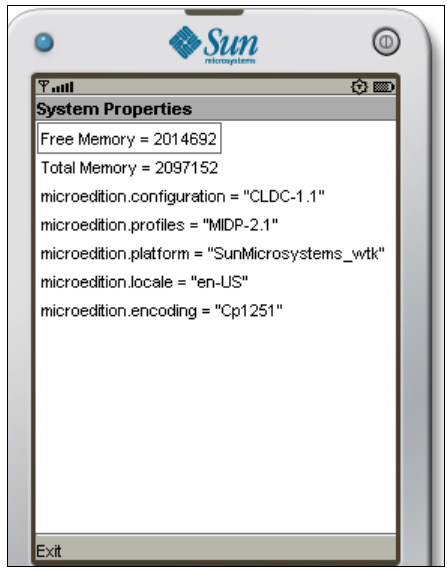


Рис. 14.35. Вариант **Properties** примера **Sun Samples — Demos**

Две оставшихся категории примеров, относящиеся к Java ME (см. рис. 14.31), **Samples/Java ME (Rico SDK/J)** и **Samples/Java ME (BD-J)**, требуют установки на компьютере CDC-совместимого эмулятора (SDK или платформы). В связи с тем, что такой комплект не входит в состав NetBeans, мы не будем рассматривать примеры из данной категории.

На этом мы остановимся. Как уже говорилось, даже краткое описание поставляемых с NetBeans 6.5 примеров занимает много места, не говоря уже об анализе их исходных кодов.

Рис. 14.36. Пример **Advanced Multimedia Supplements**

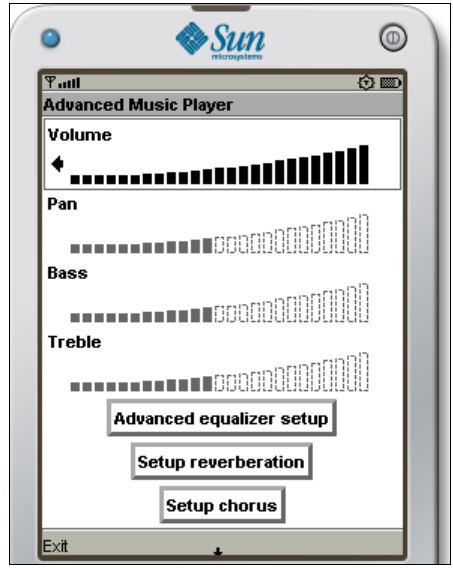


Рис. 14.37. Music Effects — эквалайзер и ревербератор

Запускайте NetBeans и создавайте свои программные продукты на основе этих примеров. Творите и развлекайтесь. Порадуйте и удивите своих друзей и знакомых, написав пусть простую, но свою программу для мобильника! С NetBeans это становится очень просто. Причем такая деятельность гораздо интереснее и полезнее, чем бессмысленная игра в компьютерные игры. А вот книжки читать надо! Причем в бумажном, а не в электронном виде.

Краткие итоги по главе

- Для версии Java Micro Edition не существует JDK — здесь имеется набор спецификаций Java Specification Request (JSR), в которых заданы конфигурации (Configuration) и профили (Profile).
- *Конфигурация* — это такая спецификация, в которой задаются:
  - тип и объем оперативной памяти мобильного устройства;
  - тип и скорость процессора;
  - параметры работы с сетью, к которой это устройство подсоединяется.
- *Профиль* конкретизирует тип используемого устройства и параметры пользовательского интерфейса в рамках заданной конфигурации.
- Имеются две важнейшие конфигурации, определяющие параметры приложений и устройств, для которых они создаются:
  - Connected Limited Device Configuration (CLDC) — приложения для мобильных телефонов и наладонных компьютеров (PDA) с очень небольшим (512 Кбайт или менее) объемом памяти. Наиболее важный профиль — Mobile Information Device Profile (MIDP) — это профиль мобильных информационных устройств;

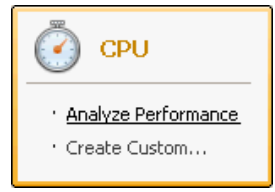
  - Connected Device Configuration (CDC) — приложения для "продвинутых" мобильных телефонов и смартфонов, веб-телефонии и других устройств, имеющих 2 Мбайт памяти (и более) и 32-битовый процессор.
- В рамках профиля MIDP существует особый вид приложений для мобильных устройств, называемый *мидлетами*. Мидлет — это приложение, класс которого унаследован от `javax.microedition.midlet.MIDlet`. Оно распространяется в виде jar-архива (как обычные приложения) и сопровождающего его файла дескрипторов jad (Java application descriptor).
- Мидлеты — самые простые, легкие и наиболее переносимые приложения для мобильных устройств.
- Мидлеты распространяются через jad- и jar-файлы. Аббревиатура jad означает Java Application Descriptor (Описание приложения Java).
- Метод `commandAction` осуществляет обработку команд. В нем выполняется код, соответствующий каждой конкретной команде при вызове этой команды.
- Комментариями `//write ... user code here` (написать здесь пользовательский код) помечены те фрагменты кода мидлета, которые разрешено менять (вместо комментариев можно вставлять свой собственный исходный код).
- Если по каким-то причинам идет сообщение, что импортируемый пакет не найден, это означает, что jar-файл, содержащий данный пакет, необходимо добавить к ресурсам проекта.
- На современных телефонах Sony Ericsson предусмотрена возможность запуска мидлетов непосредственно из jar-архивов, скопированных в память телефона (jad-файл не нужен).
- Для запуска мидлетов из файла на современных телефонах Samsung необходимо скопировать jad- и jar-файлы в память телефона, выбрать в менеджере файлов jad-файл и воспользоваться опцией установки. При этом в jad-файле обязательно должны быть указаны два параметра: `MIDlet-Jar-Size` и `MIDlet-Jar-URL`.
- В поставку NetBeans 6.5 входит очень много интересных примеров, раскрывающих практически все возможности работы с мобильными устройствами.

Типичные ошибки

- Считают, что мидлеты могут работать только на устаревших моделях мобильных устройств, а на современных должны запускаться приложения CDC.
- Считают, что приложения Java на мобильных телефонах медленно стартуют и потребляют много ресурсов (это заблуждение связано с тем, что многие приложения Java для мобильных устройств не используют "ленивую" инициализацию и другие современные технологии).
- При написании собственного компонента забывают документировать добавленные поля и переопределенные или добавленные методы.

Задания

- Доработать пример 14_9_MobileApplication: обеспечить различные акции при выборе того или иного пункта меню (в том числе с выдачей экрана `Alert` для одного из этих пунктов).



ГЛАВА 15

Компонентное программирование

Компонент — это автономный элемент программного обеспечения, предназначенный для многократного применения (может распространяться для встраивания в другие программы в виде скомпилированного кода класса).

Компонент подключается к другим программам с помощью интерфейсов и взаимодействует с программной средой через события, причем в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать данный компонент.

Технология JavaBeans предоставляет возможность написания компонентного программного обеспечения на языке Java. Напоминаем, что Beans по-английски означает "зерна" (обыгрывается происхождение названия Java от любимого создателями языка сорта кофе). Компоненты JavaBeans в литературе часто упоминаются просто как Beans.

15.1. Компонентная архитектура JavaBeans

Компонент JavaBeans является классом Java и имеет три типа атрибутов.

- *Методы* компонента JavaBeans не отличаются от других методов объектов в Java — они описывают поведение компонента. Общедоступные методы компонента могут вызываться из других компонентов или из обработчиков событий.
- *Свойства* (Properties) компонента характеризуют его внешний вид и поведение, их можно изменить в процессе визуального проектирования с помощью редактора свойств (Property Editor), а некоторые из свойств (положение компонента, его размер, текст) — вручную. Свойство задается комбинацией геттера и сеттера (метода по чтению и метода по записи).
- *События* (Events) используются для связи между компонентами. При помещении компонента на экранную форму среда разработки исследует компоненты и определяет, какие программные события данный компонент может порождать (рассылать), а какие получать (обрабатывать).

По окончании работы со средой разработки состояние компонентов сохраняется в файле с помощью механизма *сериализации* (serialization) — представления объектов Java в

виде потока байтов. При последующей загрузке проекта сохраненное состояние компонентов считывается из файла.

Компонент JavaBeans можно включить в состав более сложных (составных) компонентов, приложений, сервлетов, пакетов, модулей. Причем обычно это делается с помощью сред визуального проектирования.

Компоненты JavaBeans предоставляют свои общедоступные методы и события для режима визуального проектирования. Доступ к ним возможен в том случае, когда их названия соответствуют особым шаблонам проектирования (bean design patterns). Для задания свойства требуется, чтобы существовали геттер и сеттер для этого свойства (см. *разд. 15.2*).

Компонент может быть установлен в среду разработки, в этом случае кнопки доступа к компонентам выносятся на палитру (palette) или панель инструментов (toolbox). Вы можете создать экземпляр компонента на проектируемой экранной форме в режиме **Design** (Дизайн) путем выбора его кнопки на панели и перетаскивания на форму, после чего можно изменять его свойства, писать обработчики событий, включать в состав других компонентов и т. д.

В NetBeans существует несколько способов создания компонента JavaBeans.

Наиболее простой — с помощью мастера создания компонента, о котором будет сказано в следующем разделе.

Другой способ — создание для компонента класса `BeanInfo`, обеспечивающего поставку необходимой информации о данном компоненте. Этот класс должен реализовывать интерфейс `BeanInfo`, а его имя должно состоять из имени компонента и названия интерфейса. Например, для компонента `MyComponent` это будет `MyComponentBeanInfo`. Существует класс-адаптер `SimpleBeanInfo` для интерфейса `BeanInfo`. В его наследнике достаточно переписать методы, подлежащие модификации. Среда NetBeans позволяет с помощью мастера создать заготовку такого класса.

Для редактирования некоторых свойств в среде визуального проектирования требуется специальный объект — редактор свойств (Property Editor). Среда NetBeans позволяет с помощью мастера создать заготовку и для такого класса.

15.2. Создание компонента в NetBeans 6.5

В NetBeans 6.5 для создания собственного компонента необходимо зайти в проект с графическим интерфейсом (например, Desktop Application), после чего выбрать в меню **File/New File.../JavaBeans Objects/JavaBeans Component** и нажать кнопку **Next >** (рис. 15.1).

Затем в появившейся форме (рис. 15.2) в поле **Class Name** необходимо ввести имя компонента. В качестве примера мы введем **MyBean**. Обязательно следует выбрать пакет (**Package**), в котором нужно создавать компонент — мы выберем пакет нашего приложения (**desktopapplication**). После этого следует нажать на кнопку **Finish**.

Код получившейся заготовки приведен в листинге 15.1 (пример 15_2_DesktopApplication1 на прилагаемом к книге DVD).

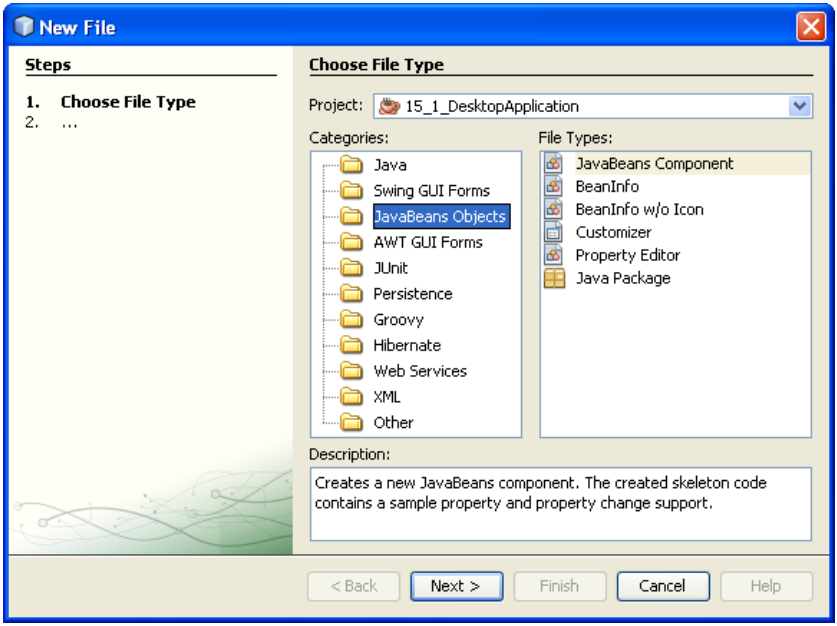


Рис. 15.1. Создание компонента JavaBeans (шаг 1)

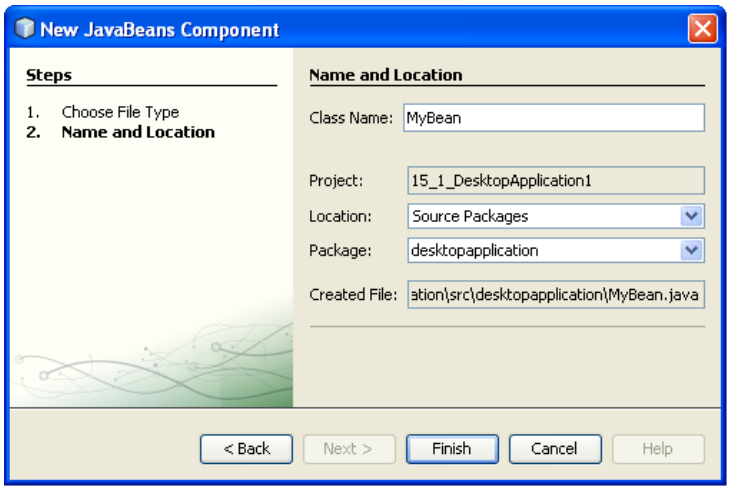


Рис. 15.2. Создание компонента JavaBeans (шаг 2)

Листинг 15.1. Сгенерированный средой NetBeans исходный код заготовки компонента

```
package desktopapplication;

import java.beans.*;
import java.io.Serializable;
```

```
public class MyBean extends Object implements Serializable {

public static final String PROP_SAMPLE_PROPERTY = "sampleProperty";
    private String sampleProperty;
    private PropertyChangeSupport propertySupport;

    public MyBean() {
        propertySupport = new PropertyChangeSupport(this);
    }

    public String getSampleProperty() {
        return sampleProperty;
    }

    public void setSampleProperty(String value) {
        String oldValue = sampleProperty;
        sampleProperty = value;
        propertySupport.firePropertyChange(PROP_SAMPLE_PROPERTY,
                                           oldValue, sampleProperty);
    }
    public void addPropertyChangeListener(
                                PropertyChangeListener listener) {
        propertySupport.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(
                                PropertyChangeListener listener) {
        propertySupport.removePropertyChangeListener(listener);
    }
}
```

Заметим, что строка

```
public class MyBean implements Serializable {
```

эквивалентна следующей:

```
public class MyBean extends Object implements Serializable {
```

Таким образом, в созданной заготовке считается, что класс нашего компонента унаследован от класса `Object`. В большинстве случаев, конечно, приходится заменять `Object` на тот класс, от которого мы хотим наследовать компонент, например, на `JPanel`.

В созданном компоненте присутствует заготовка для строкового свойства `sampleProperty`. Геттер `public String getSampleProperty()` обеспечивает чтение значения свойства, а сеттер `public void setSampleProperty(String value)` — установку нового значения.

Служебный объект `private PropertyChangeSupport propertySupport` обеспечивает поддержку работы с обработчиком события `PropertyChange`. Отметим, что "property change" означает "изменение свойства". Это событие должно возникать при любом изменении свойств нашего компонента.

Как уже говорилось, в каждом объекте, поддерживающем работу с неким событием (у нас это событие `PropertyChange`), имеется список объектов-слушателей событий

(listeners). Иногда их называют зарегистрированными слушателями. Методы с названием `fireИмяСобытия` (fire — стрелять, в данном случае — "выстрелить" событием) осуществляют поочередный вызов зарегистрированных слушателей из списка для данного события, передавая им событие на обработку. Здесь это метод `propertySupport.firePropertyChange`. Сначала он обеспечивает создание объекта-события (если значение свойства действительно изменилось), а потом поочередно вызывает слушателей этого события для его обработки.

Методы:

```
public void addPropertyChangeListener(PropertyChangeListener listener)
```

и

```
public void removePropertyChangeListener(PropertyChangeListener listener)
```

обеспечивают для компонента возможность добавления и удаления объекта-слушателя — обработчика события `Property Change`.

Если потребуется создать другие свойства или добавить (удалить) обработчики других событий, придется делать это вручную. В NetBeans 5.5 можно было воспользоваться соответствующим мастером, в NetBeans 6.0 поддержка создания компонентов вообще отсутствовала, а в NetBeans 6.5 она реализована в ограниченном варианте: можно просматривать имеющиеся свойства и обработчики, но создавать их нужно вручную.

Для просмотра имеющихся свойств и обработчиков выделим в дереве проектов узел **MyBean.java**, а в окне **Navigator** в выпадающем списке переключимся со стоящего по умолчанию просмотра **Members View** (Просмотр членов класса) на вариант просмотра **Bean Patterns** (Pattern означает образец) — рис. 15.3.

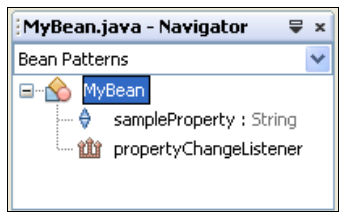


Рис. 15.3. Просмотр свойств и событий компонента в окне **Navigator**

Также можно создать для нашего компонента сопроводительный класс `BeanInfo`, который предоставляет дополнительную информацию о компоненте в виде объектов `FeatureDescriptor` (описание свойств) для каждого из имеющихся свойств, событий и методов. В NetBeans 6.5 создание компонента `BeanInfo` происходит очень просто — достаточно в окне **Projects** вызвать контекстное меню для узла **MyBean.java** и выбрать в нем пункт **BeanInfo Editor**. В появившейся диалоговой форме (рис. 15.4) следует подтвердить, что мы хотим создать новый компонент `BeanInfo`.

После этого будет автоматически создан файл с исходными кодами класса `MyBeanBeanInfo`. При переключении в режим **Designer** для этого класса мы увидим информацию о свойствах и событиях компонента (рис. 15.5) — пример 15_1_DesktopApplication2 на прилагаемом к книге DVD.

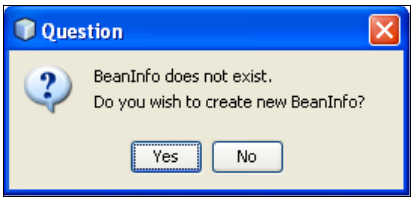


Рис. 15.4. Создание компонента `BeanInfo`

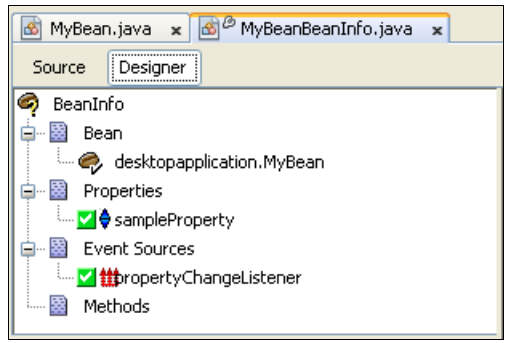


Рис. 15.5. Просмотр свойств и событий компонента с помощью класса `BeanInfo`

Если компонент имеет собственный тип какого-либо свойства, либо на значения свойства должны быть наложены какие-либо ограничения, необходимо создать класс `PropertyEditor` (Редактор свойств), позволяющий пользователю устанавливать свои значения для этого свойства. Это делается с помощью вызова **File/New File.../JavaBeans Objects/Property Editor**.

Если мы хотим, чтобы наш компонент можно было визуально проектировать, т. е. чтобы в редакторе кроме вкладки **Sourse** появлялась вкладка **Design**, для нашего компонента необходимо задать также класс `Customizer`. Это можно сделать с помощью вызова **File/New File.../JavaBeans Objects/ Customizer**.

15.3. Пример создания компонента в NetBeans 6.5 — панель с заголовком

В качестве простейшего примера визуального компонента создадим панель, у которой имеется заголовок (title). Унаследуем наш компонент от класса `javax.swing.JPanel`.

Для этого в импорте запишем:

```
import javax.swing.*;
```

а в качестве родительского класса напишем `JPanel`.

С помощью рефакторинга заменим имя `myBean` на `JTitledPanel`, имя поля `sampleProperty` на `title`, а константу `PROP_SAMPLE_PROPERTY` уберем, написав в явном виде имя свойства `title` в методе `firePropertyChange`. Методы `getSampleProperty` и `setSampleProperty` переименуем в `getTitle` и `setTitle`.

Отреагируем на предупреждения, выдаваемые редактором, и добавим аннотацию `@Override` перед методами `addPropertyChangeListener` и `removePropertyChangeListener`.

Обязательно следует присвоить начальное значение полю `title`, но в созданной заготовке это не делается. Код задания переменной `title` выглядит следующим образом:

```
private String title;
```

Поэтому в исходном коде в месте, где задается поле `title`, необходимо заменить данную строку на следующую:

```
private String title = "Заголовок";
```

Для показа заголовка необходимо импортировать классы `java.awt.Graphics`, `java.awt.geom.Rectangle2D` (см. чуть ниже) и переопределить в нашем классе `JTitledPanel` метод `paint` (листинг 15.2).

Листинг 15.2. Переопределение метода `paint`

```
@Override
public void paint(Graphics g){
    super.paint(g);
    FontMetrics fontMetrics=g.getFontMetrics();
    Rectangle2D rect = fontMetrics.getStringBounds(title, g);
    g.drawString(title,
                 (int)Math.round((this.getWidth()-rect.getWidth())/2),
                 10);
}
```

Чтобы можно было пользоваться классами `Graphics`, `FontMetrics` и `Rectangle2D`, нам следует добавить импорт:

```
import java.awt.*;
import java.awt.geom.Rectangle2D;
```

Отметим, что можно было бы не вводить переменные `fontMetrics` и `rect`, а сразу писать в методе `drawString` соответствующие функции в следующем виде:

```
g.drawString(title,
            (int)Math.round( ( this.getWidth() -
            g.getFontMetrics().getStringBounds(title,g).getWidth() )/2 ),
            10);
```

Но от этого текст программы стал бы еще менее читаемым, даже несмотря на попытки отформатировать текст подобающим образом.

Еще одно необходимое изменение — добавление вызова функции `repaint()` в последнюю строку метода `setTitle`. Если этого не сделать, то после изменения свойства компонент не перерисуется с вновь установленным заголовком.

В результате получим код компонента, приведенный в листинге 15.3.

Листинг 15.3. Исходный код разработанного компонента `JTitledPanel`

```
/*
 * JTitledPanel.java8

*/

package desktopapplication;

import java.beans.*;
import java.io.Serializable;
import javax.swing.*;              //добавлено вручную
import java.awt.*;                 //добавлено вручную
import java.awt.geom.Rectangle2D; //добавлено вручную
```

```
/**
 * @author В.Монахов
 */
public class JTitledPanel extends JPanel implements Serializable {

   private String title="Заголовок"; //изменено вручную
   private PropertyChangeSupport propertySupport;

   public JTitledPanel() {
       super();//добавлено вручную
       propertySupport = new PropertyChangeSupport(this);
   }

   public String getTitle() {
       return title;
   }

   public void setTitle(String value) {
       String oldValue = title;
       title = value;
       propertySupport.firePropertyChange("title", oldValue, title);
       repaint();           //добавлено вручную
   }

   @Override
   public void addPropertyChangeListener(
                                      PropertyChangeListener listener) {
       propertySupport.addPropertyChangeListener(listener);
   }

   @Override
   public void removePropertyChangeListener(
                                       PropertyChangeListener listener){
       propertySupport.removePropertyChangeListener(listener);
   }

   @Override
   public void paint(Graphics g){ //метод добавлен вручную
     super.paint(g);
     FontMetrics fontMetrics=g.getFontMetrics();
     Rectangle2D rect = fontMetrics.getStringBounds(title, g);
     g.drawString(title,(int)Math.round((this.getWidth() -
                                          rect.getWidth())/2), 10);
   }
}
```

Компонент необходимо скомпилировать, после чего его можно добавить в палитру. Для этого, если мы редактируем файл JTitledPanel.java в окне редактора исходного кода, в меню **Tools** следует выбрать пункт **Add to Palette**. Если мы редактируем другой файл, можно в окне **Projects** вызвать контекстное меню для узла **JTitledPanel** и в нем вызвать пункт меню **Tools/Add to Palette...**

Далее в появившемся диалоге необходимо указать палитру, на которую будет добавлен компонент (желательно выбрать **Beans**, чтобы не путать наши компоненты со стандартными) (рис. 15.6). После нажатия кнопки **OK** этот компонент можно использовать наравне с другими. Теперь мы сможем менять текст заголовка как в редакторе свойств на этапе визуального проектирования (рис. 15.7), так и программно во время работы приложения — пример 15_3_DesktopApplication на прилагаемом к книге DVD.

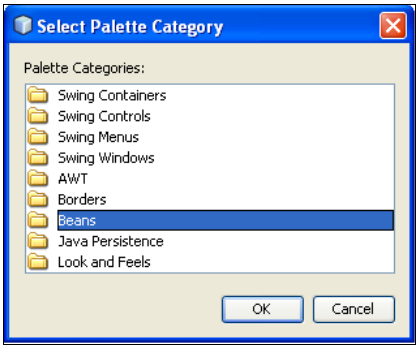


Рис. 15.6. Выбор палитры, на которую будет добавлен компонент

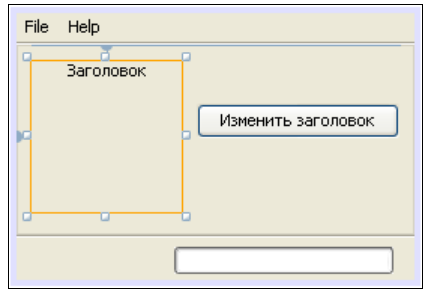


Рис. 15.7. Использование созданного компонента

Мы также можем рисовать по нашей панели, и заголовок при этом будет виден, как и отрисовываемые примитивы. Например, мы можем вывести по нажатию какой-нибудь кнопки строку "`Тест`" (но до этого, конечно, нужно добавить импорт `java.awt.*` и `java.awt.geom.Rectangle2D`) (листинг 15.4).

Листинг 15.4. Отрисовка текста на панели `JTitledPanel`

```
Graphics g=jTitledPanel1.getGraphics();
FontMetrics fontMetrics=g.getFontMetrics();
Rectangle2D rect = fontMetrics.getStringBounds("Тест", g);
g.drawString("Тест",10,30 );
```

Отметим, что между текстом "Тест", нарисованным на панели, и текстом "Заголовок", также находящимся на ней, имеется принципиальное отличие — при перекрытии нашего компонента другим окном или при изменении размеров формы текст "Тест" исчезает, а "Заголовок" сохраняется. Это происходит потому, что заголовок отрисовывает метод `paint`, который всегда вызывается при перерисовке компонента.

При усовершенствовании кода нашего компонента нет необходимости каждый раз удалять его из палитры компонентов и заново устанавливать — достаточно после внесения изменений заново скомпилировать проект (команда **Build main project** в главном меню или нажатие клавиши <F11>), после чего закрыть страницу редактирования класса, использующего компонент (в нашем случае **DesktopView**), и заново ее открыть (например, двойным щелчком по имени класса в окне **Projects**).

В документации по NetBeans написано, что для этого достаточно скомпилировать проект и затем обновить форму с помощью комбинации клавиш <Shift>+<Ctrl>+<R>, но данный способ не работает.

Следует обратить внимание на то, что при копировании или переименовании проекта, использующего разработанный нами компонент, при переходе к дизайну могут возникать проблемы. Будет сообщаться, что компонент не найден, хотя файл с его исходными кодами присутствует в проекте. В таком случае необходимо скомпилировать файл JTitledPanel.java (вызвать в дереве проектов контекстное меню для этого файла и выбрать пункт **Compile File**), и проблема будет решена.

15.4. Добавление в компонент новых свойств

В технологии JavaBeans существует две основных разновидности свойств: **Bound** (связанное свойство) и **Constrained** (свойство с ограничениями).

Свойства вида **Bound** — обычные свойства, при изменении любого из них порождается событие `propertyChangeEvent`.

Свойства вида `Constrained` требуют проверки задаваемого значения свойства на принадлежность к области допустимых значений. Если это условие не выполняется, то возбуждается исключительная ситуация. При изменении таких свойств порождается событие `VetoableChangeEvent`. Слово Vetoable происходит от Veto able — способный на введение ограничения, накладывание вето.

Свойства могут являться массивами, их называют *индексированными* (иногда векторными).

Добавим в компонент новое свойство. Пусть мы хотим задать свойство `titleShift` типа `int`, которое будет задавать сдвиг по высоте нашего заголовка относительно верха панели. Добавим в исходный код компонента текст, приведенный в листинге 15.5.

Листинг 15.5. Код, добавленный для свойства `titleShift`

```
/**
 * Holds value of property titleShift.
 */
private int titleShift;

/**
 * Getter for property titleShift.
 * @return Value of property titleShift.
 */
public int getTitleShift() {
    return this.titleShift;
}

/**
 * Setter for property titleShift.
 * @param titleShift New value of property titleShift.
 */
public void setTitleShift(int titleShift) {
    int oldTitleShift = this.titleShift;
    this.titleShift = titleShift;
```

```
        propertySupport.firePropertyChange ("titleShift",
                        new Integer (oldTitleShift),
                        new Integer (titleShift));
        repaint();//добавлено вручную
    }
```

Правда, как и в предыдущем случае, в автоматически сгенерированный код сеттера пришлось добавить оператор `repaint()`.

Для того чтобы в режиме дизайна добавленное свойство отобразилось в компоненте и появилось в окне **Properties**, следует скомпилировать весь проект, закрыть страницу редактирования класса, использующего компонент (в нашем случае **DesktopView**), и заново ее открыть (например, двойным щелчком по имени класса в окне **Projects**).

Проект 15_4_DesktopApplication на прилагаемом к книге DVD иллюстрирует результат добавления в `JTitledPanel` свойства `titleShift`.

Добавление индексированного свойства иллюстрирует код, приведенный в листинге 15.6.

Листинг 15.6. Код, добавленный для индексированного свойства

```
    /**
     * Holds value of property arr.
     */
    private double[] arr;

    /**
     * Indexed getter for property arr.
     * @param index Index of the property.
     * @return Value of the property at <CODE>index</CODE>.
     */
    public double getArr(int index) {
        return this.arr[index];
    }

    /**
     * Getter for property arr.
     * @return Value of property arr.
     */
    public double[] getArr() {
        return this.arr;
    }

    /**
     * Indexed setter for property arr.
     * @param index Index of the property.
     * @param arr New value of the property at <CODE>index</CODE>.
     */
    public void setArr(int index, double arr) {
        this.arr[index] = arr;
```

```
        propertySupport.firePropertyChange ("arr", null, null );
    }

    /**
     * Setter for property arr.
     * @param arr New value of property arr.
     */
    public void setArr(double[] arr) {
        double[] oldArr = this.arr;
        this.arr = arr;
        propertySupport.firePropertyChange ("arr", oldArr, arr);
    }
```

15.5. Добавление в компонент новых событий

Поскольку мы наследуем компонент от класса `JPanel`, то большинство необходимых событий он уже умеет генерировать. Иногда может потребоваться задать новый тип событий, но чаще всего удается использовать готовые интерфейсы слушателей. Например, мы хотим, чтобы возникло событие `java.awt.event.TextEvent`, связанное с изменением текста заголовка. В обычной панели `JPanel` свойства, связанные с текстом, отсутствовали, и это событие в ней не поддерживалось. Интерфейс `java.awt.event.TextListener` имеет всего один метод `textValueChanged(TextEvent e)`, так что в адаптере нет необходимости.

Для создания такого события для нашего компонента потребуется добавить поддержку события.

В Java два типа источников событий:

- Unicast Event Source — источник порождает целевые объекты событий, которые передаются одному слушателю-приемнику. Cast — список исполнителей, Unit — единичный, Unicast — от Unit и Cast — один обработчик, Source — источник. В этом случае список слушателей не создается, а резервируется место только для одного из них.
- Multicast Event Source — источник порождает целевые объекты событий, которые передаются нескольким слушателям-приемникам. Multi — много, Multicast — много обработчиков. В этом случае для событий данного типа создается список слушателей.

Очевидно, что для некоторых типов событий обязательно создавать список слушателей, хотя для Unicast Event Source реализация оказывается проще. Нам список не нужен, поэтому выберем первый вариант.

Нам требуется интерфейс, поддерживающий событие `java.awt.event.TextEvent`. Для этого необходима поддержка интерфейса `java.awt.event.TextListener` для Unicast-источника. Соответствующий код приведен в листинге 15.7.

Листинг 15.7. Код для поддержки интерфейса `TextListener` в случае Unicast-источника

```
/**
 * Utility field holding the TextListener.
 */
private transient java.awt.event.TextListener textListener =  null;

/**
 * Registers TextListener to receive events.
 * @param listener The listener to register.
 */
public synchronized void addTextListener(
                            java.awt.event.TextListener listener) throws
                            java.util.TooManyListenersException {
    if (textListener != null) {
        throw new java.util.TooManyListenersException ();
    }
    textListener = listener;
}

/**
 * Removes TextListener from the list of listeners.
 * @param listener The listener to remove.
 */
public synchronized void removeTextListener(
                                  java.awt.event.TextListener listener) {
    textListener = null;
}
```

Для поддержки fire-метода `fireTextListenerTextValueChanged` к предыдущему варианту, приведенному в листинге 15.7, необходимо добавить новый код (листинг 15.8).

Листинг 15.8. Код, обеспечивающий поддержку fire-метода

```
/**
 * Notifies the registered listener about the event.
 *
 * @param object Parameter #1 of the <CODE>TextEvent<CODE> constructor.
 * @param i Parameter #2 of the <CODE>TextEvent<CODE> constructor.
 */
private void fireTextListenerTextValueChanged(java.lang.Object object,int i){
    if (textListener == null) return;
    java.awt.event.TextEvent e = new java.awt.event.TextEvent (object, i);
    textListener.textValueChanged (e);
}
```

Этот код еще не обеспечивает генерации события в нашем компоненте, но предоставляет возможность сделать это путем добавления одной строчки в код метода `setTitle`.

Перед вызовом метода `repaint()` мы напишем:

```
fireTextListenerTextValueChanged(this, java.awt.event.TextEvent.TEXT_VALUE_CHANGED);
```

В качестве первого параметра fire-метода идет ссылка на объект-источник события, в качестве второго — идентификатор типа события. Найти, где задается идентификатор, просто — достаточно перейти мышью по гиперссылке `java.awt.event.TextEvent`, появляющейся в среде разработки при нажатии клавиши <Ctrl>, и посмотреть исходный код конструктора.

Данную гиперссылку можно получить в строке:

```
java.awt.event.TextEvent e = new java.awt.event.TextEvent (object, i);
```

в теле метода `fireTextListenerTextValueChanged`, где, собственно, и используется этот не очень-то понятный на первый взгляд параметр.

Для того чтобы событие `textValueChanged` появилось в списке событий компонента, в окне **jTitledPanel1[JTitledPanel]-Properties/Events** требуется (как мы ранее делали при добавлении для свойств) скомпилировать проект, закрыть в окне редактирования вкладку **DesktopView.java** и снова ее открыть.

Проект 15_5_DesktopApplication1 иллюстрирует получившийся в результате код.

Теперь для нашего компонента можно назначать и удалять обработчик события `textValueChanged` как непосредственно на этапе визуального проектирования, так и программным путем.

Покажем, каким образом это делается на этапе визуального проектирования. Выделим компонент `jTitledPanel1` и выберем в окне **jTitledPanel1[JTitledPanel]-Properties** вкладку **Events**, а в ней событие **textValueChanged** (рис. 15.8). Нажмем кнопку ..., находящуюся рядом с полем, — вызовется диалог добавления и удаления обработчиков событий. Вначале он пустой, т. к. обработчиков не назначено. Займемся добавлением обработчика, нажав кнопку **Add...** (Добавить...) — рис. 15.9. Вообще, существует правило: если у какого-либо пункта меню имеется подобная кнопка, то при нажатии на нее появится диалог для задания значений в этом пункте.

Введем в качестве имени обработчика события (event handler) **myHandler** и нажмем кнопку **OK** (рис. 15.9). В списке обработчиков Handlers-формы **Handlers for textValueChanged** появится имя **myHandler**. После закрытия этой формы по нажатию кнопки **OK** в исходном коде приложения (а не компонента!) появится код:

```
    private void myHandler(java.awt.event.TextEvent evt) {
// TODO add your handling code here:
    }
```

Вместо комментария `// TODO add your handling code here:`, как обычно, следует написать свой код обработчика, например, такой:

```
javax.swing.JOptionPane.showMessageDialog( null,"Text="+
                                           jTitledPanel1.getTitle() );
```

Проект 15_5_DesktopApplication2 иллюстрирует получившийся в результате код.

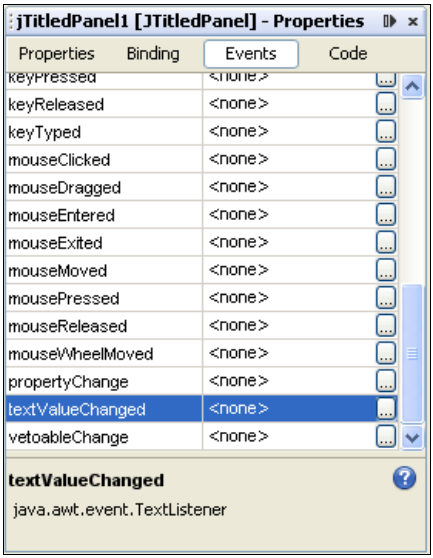


Рис. 15.8. Создание обработчика события на этапе визуального проектирования (шаг 1)

Рис. 15.9. Создание обработчика события на этапе визуального проектирования (шаг 2)

Краткие итоги по главе

- Компонент — это автономный элемент программного обеспечения, предназначенный для многократного применения (может распространяться для встраивания в другие программы в виде скомпилированного кода класса).
- Компонент подключается к другим программам с помощью интерфейсов.
- Взаимодействие с программной средой осуществляется по событиям, причем в программе, использующей компонент, можно назначать обработчики событий, на которые умеет реагировать данный компонент.
- Компонент JavaBeans является классом Java и имеет три типа атрибутов:
  - *Методы* компонента JavaBeans не отличаются от других методов объектов в Java — они описывают поведение компонента. Общедоступные методы компонента можно вызывать из других компонентов или из обработчиков событий.
  - *Свойства* (Properties) компонента JavaBeans характеризуют его внешний вид и поведение, их значения можно менять в процессе визуального проектирования с помощью редактора свойств (Property Editor), а для некоторых свойств (положение компонента, его размер, текст) — вручную. Свойство задается комбинацией геттера и сеттера (метода по чтению и метода по записи).
  - *События* (Events) служат для связи между компонентами. При помещении компонента на экранную форму среда разработки исследует компоненты и определяет, какие программные события данный компонент может порождать (рассылать), а какие получать (обрабатывать).

- Наиболее простой способ создания компонента — использование мастера среды NetBeans.
- Методы с названием `fireИмяСобытия` (fire — стрелять, в данном случае — "выстрелить" событием) поочередно вызывают зарегистрированные слушатели из списка для данного события, передавая им событие на обработку.
- Методы `addИмяСобытияListener` и `removeИмяСобытияListener` обеспечивают для компонента возможность добавления и удаления объекта-слушателя — обработчика события.
- В NetBeans 6.5 задавать в компоненте новые или удалять ненужные свойства или события приходится вручную, в отличие от NetBeans 5.5, где эти операции поддерживались с помощью специальных мастеров.
- Чтобы добавить компонент в палитру, следует открыть файл компонента в окне редактора исходного кода, а в меню **Tools** выбрать пункт **Add to Palette...**, либо в дереве проектов в контекстном меню для данного компонента выбрать пункт **Tools/Add to Palette...** После чего в появившемся диалоге указать палитру, на которую будет добавлен компонент (желательно выбирать **Beans**, чтобы не путать наши компоненты со стандартными).
- Для того чтобы в режиме дизайна добавленное свойство отобразилось в компоненте и появилось в окне **Properties**, следует скомпилировать весь проект, закрыть страницу редактирования класса, использующего компонент (в нашем случае **DesktopApplivationView1**), и заново ее открыть (например, двойным щелчком по имени класса в окне **Projects**).
- Свойства вида **Bound** — обычные свойства. При изменении таких свойств порождается событие `PropertyChange`. Свойства вида **Constrained** требуют проверки задаваемого значения свойства на принадлежность к области допустимых значений. Если значение не удовлетворяет этому условию, то возбуждается исключительная ситуация. При изменении таких свойств порождается событие `VetoableChangeEvent`.
- Свойства могут быть массивами. Такие свойства называют *индексированными* (или векторными).
- В Java два типа источников событий:
  - Unicast Event Source — источник порождает целевые объекты событий, которые передаются одному слушателю-приемнику. При этом список слушателей не создается, а резервируется место только для одного обработчика.
  - Multicast Event Source — источник порождает целевые объекты событий, которые передаются нескольким слушателям-приемникам. В этом случае для событий данного типа создается список слушателей.
- Генерация события в компоненте обеспечивается вручную вызовом fire-метода или другим способом.

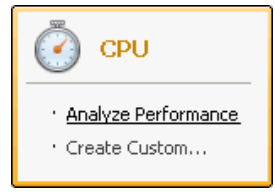
Типичные ошибки

- При отрисовке надписей по графическому контексту забывают импортировать пакеты `java.awt.Graphics`, `java.awt.FontMetrics` и `java.awt.geom.Rectangle2D`.

- При задании в компоненте сеттера для свойства забывают добавить в конце сеттера оператор `repaint()`, из-за чего после изменения свойства в режиме проектирования с компонентом не происходит видимых изменений.
- При задании в компоненте нового события забывают добавить в конце сеттера, изменяющего связанное с событием поле, соответствующий fire-метод, порождающий событие.

Задания

- Создать собственный компонент `JTitledPane`, описанный в данной главе.
- Усовершенствовать компонент, обеспечив добавление в него свойства `titleColor`. Подсказка: установка красного цвета рисования в качестве текущего цвета вывода графических примитивов для объекта `Graphics g` осуществляется вызовом метода `g.setColor(Color.red)`.
- Усовершенствовать компонент, обеспечив генерацию в нем событий типа `TitleShiftEvent`, предварительно создав соответствующий интерфейс. По желанию можно добавить и событие изменения цвета заголовка.
- Усовершенствовать компонент, предусмотрев добавление в него свойства `titleFont` и методов, обеспечивающих установку нужного размера и типа шрифта.



ГЛАВА 16

Платформо-зависимые методы и модуль C/C++ Pack

Для прикладного программирования средств Java в подавляющем большинстве случаев хватает. Но иногда возникает необходимость подключить к программе ряд системных вызовов либо обеспечить доступ к библиотекам, написанным на других языках программирования. Для таких целей в Java используются методы, объявленные с модификатором `native` (родной). Это слово означает, что при выполнении метода производится вызов ориентированного на конкретную платформу (т. е. платформо-зависимого) двоичного кода, а не платформо-независимого байт-кода, как во всех других случаях.

Технология для объявления и реализации таких методов носит название JNI — Java Native Interface.

Заголовок платформо-зависимого метода описывается в классе Java, а его реализация осуществляется на языках программирования C или C++, позволяющих создавать динамически подключаемые библиотеки (DLL — Dynamic Link Library под Windows, Shared Objects под UNIX-образными операционными системами).

Для разработки программ, написанных на C или C++, в NetBeans служит модуль C/C++ Pack.

16.1. Использование динамических библиотек. Объявление платформо-зависимых (native) методов

Объявление платформо-зависимого метода в Java имеет вид:

```
Модификаторы native ВозвращаемыйТип имяМетода(список параметров);
```

Тело платформо-зависимого метода не задается — оно является внешним и загружается в память компьютера с помощью загрузки той библиотеки, из которой этот метод должен вызываться:

```
System.loadLibrary(ИмяБиблиотеки);
```

При этом имя библиотеки задается без пути и расширения, а для UNIX-образных операционных систем из имени библиотеки убирается префикс lib. Например, если под Windows библиотека имеет имя MyLib.dll, то нужно указывать:

```
System.loadLibrary("MyLib");
```

А под UNIX или Linux вызов `System.loadLibrary("MyLib")` требуется использовать для библиотеки с именем `libMyLib.so`.

Если файл не найден, возбуждается непроверяемая исключительная ситуация `UnsatisfiedLinkError`.

Если требуется указать имя библиотеки с путем, то применяется вызов

```
System.load (ИмяБиблиотекиСПутем);
```

который во всем остальном аналогичен вызову метода `loadLibrary`.

После того как библиотека загружена, с точки зрения использования в программе вызов платформо-зависимого метода ничем не отличается от вызова любого другого метода.

Для создания библиотеки с методами, предназначенными для работы в качестве родных, обычно используется язык C++. В JDK существует утилита javah.exe, предназначенная для создания заголовков C++ из скомпилированных классов Java. Покажем, как ею пользоваться на примере класса `ClassWithNativeMethod`. Создадим этот класс в пакете нашего приложения. Реализация данного класса очень проста (листинг 16.1).

Листинг 16.1. Класс с native-методом

```
package java_example_pkg;
public class ClassWithNativeMethod {

    /** Creates a new instance of ClassWithNativeMethod */
    public ClassWithNativeMethod() {
    }

   public native void myNativeMethod();
}
```

16.2. Интерфейс JNI (Java Native Interface) взаимодействия Java с C/C++. Соответствие типов Java и C++

Программный интерфейс JNI позволяет:

- создавать и изменять объекты Java (включая массивы и строки);
- вызывать методы Java;
- перехватывать и создавать исключительные ситуации;
- подгружать классы и получать о них информацию;
- проверять типы во время выполнения.

Для понимания дальнейшего текста желательно знакомство читателя с основами языка C++. В то же время знания Java и изложенного в нескольких следующих абзацах материала достаточно для понимания большей части приведенных далее конструкций.

Синтаксис управляющих конструкций языка C++ такой же, как у Java, поскольку в обоих этих языках они повторяют управляющие конструкции языка C.

Одно из наиболее бросающихся в глаза различий — использование в C/C++ так называемых макроопределений (команд препроцессора). Эти команды начинаются с символа `#` в начале строки и заканчиваются концом строки или началом символа многострочного комментария (`/*`).

Примеры:

`#include` *имяФайла* (include означает включить) — указывает, что содержимое файла *имяФайла* с точки зрения компилятора включено в текст на месте данной декларации (как будто оно находится не в другом файле, а в этом).

`#define` *имяМакроопределения* (define означает задать) — задает специальную переменную с именем *имяМакроопределения* и назначает ей некоторую последовательность для замещения. Если в тексте встречается имя макроопределения, препроцессор заменяет это имя на последовательность замещения (фрагмент программы).

`#ifdef`, `#ifndef`, `#else`, `#endif` (означают: если задано, если не задано, в противном случае, окончание оператора `ifdef`) — составляющие оператора условной компиляции. Позволяют вставлять в текст программы те или иные фрагменты кода в зависимости от выполнения или невыполнения условия. Например, в макроопределении

```
#ifdef __cplusplus
extern "C" {
#endif
```

в текст программы при компиляции будет вставлен фрагмент

```
extern "C" {
```

в случае, когда в программе задано макроопределение `__cplusplus`. А если оно не задано, данный фрагмент в программу не добавится и компилироваться не будет. Использование include-файлов с различным содержимым позволяет подключать те или иные макроопределения и, в конечном итоге, те или иные библиотеки. При этом программа будет работать с разными библиотеками в зависимости от того, каковы настройки компилятора — какие макроопределения заданы ему в качестве предваряющих текст программы.

Огромная гибкость механизма макроопределений зачастую приводит к сложностям при работе с проектами C++. Например, последовательность задания в проекте путей для подключения библиотек, указанных в директивах `#include`, не является частью программы и часто не документируется, но может привести к тому, что проект с совершенно корректным кодом и абсолютно правильными путями, но указанными не в той последовательности, не будет компилироваться.

Еще одно наиболее существенное различие между C/C++ и Java заключается в повсеместном присутствии в C и C++ указателей. Переменные ссылочного типа задаются в C/C++ с помощью оператора `*` (того же самого, что и для умножения!).

Например, декларация

```
JNIEnv* env;
```

задает переменную-указатель `env`, хранящую адрес ячейки, имеющей тип `JNIEnv`. Сам указатель имеет тип `JNIEnv*`. Для доступа к ячейке, на которую ссылается указатель,

требуется перед именем указателя написать символ `*`, например `*env`. Такую операцию по-английски называют dereferencing (снятие указателя), а в русскоязычной традиции для нее существует абсолютно неадекватный оригиналу термин "разыменование".

Возможна совершенно эквивалентная с точки зрения языков C++ и C , но менее понятная декларация

```
JNIEnv *env;
```

Она означает, что переменная `*env` имеет тип `JNIEnv`. Но ведь мы задаем переменную-указатель, а не динамическую переменную, на которую этот указатель ссылается! Отметим, что хотя оба варианта декларации разрешены как в C++, так и в C, первый вариант предпочитают программисты на C++, а второй — на C.

Динамическое создание объектов в C++ происходит, как и в Java, с помощью оператора `new`, после которого указывается конструктор. Но переменная, в которой хранится адрес объекта, является не ссылкой, а указателем. Поэтому для доступа к самому объекту требуется поставить оператор `*` перед именем данной переменной.

Если мы имеем переменную-указатель, ссылающуюся на объект, доступ к полям и методам этого объекта осуществляется не через точку, как в Java, а через комбинацию из двух символов `->`, символизирующую стрелку. Например, `p->GetArrayLength()`. То же правило действует, когда в объекте имеется поле данных ссылочного типа (например, указатель, ссылающийся на другой объект).

Еще одна черта, отличающая программирование на C и C++ от программирования на Java, — крайне слабый контроль типов. Например, булев тип рассматривается как числовой; адреса можно не только складывать, но и умножать и т. п. Это дает большую гибкость и эффективность при написании системного программного обеспечения, но становится причиной многочисленных недиагностируемых ошибок в прикладных программах.

Наконец, в языке C++ присутствуют заголовочные файлы (header files, h-файлы). В них обычно выносятся заголовки классов C++, где декларируются поля и методы этих классов (указывается только заголовок метода). А реализация методов находится в главном файле с расширением cc или cpp.

После приведенного очень краткого экскурса в язык C++ (отражающий ничтожно малую часть его возможностей) перейдем к созданию заголовков C++ из скомпилированных классов Java.

Для того чтобы воспользоваться утилитой javah.exe, скомпилируем проект и перейдем в папку build\classes, в которой расположена папка с пакетом нашего приложения java_example_pkg и папка META-INF.

Далее потребуется работа в режиме командной строки. Для этого либо можно использовать какой-либо менеджер командной строки (Far, Total Commander и т. п.), либо запустить приложение операционной системы Windows, предназначенное для работы в режиме командной строки — выбрать в меню **Пуск** пункт **Программы/Стандартные/Командная строка**. Для просмотра в Far результата вывода (диагностики ошибок) следует нажать комбинацию клавиш <Ctrl>+<O>, для возврата в обычный режим — опять нажать эту комбинацию.

В режиме командной строки выполним команду:

```
javah.exe java_example_pkg.ClassWithNativeMethod
```

Задавать имя класса необходимо с полной квалификацией, т. е. с указанием перед ним имени пакета. Если путь к утилите javah.exe в системе не задан, то необходимо указывать имя этой утилиты вместе с путем.

В результате в папке появится файл java_example_pkg_ClassWithNativeMethod.h со следующим содержимым (листинг 16.2).

Листинг 16.2. Сгенерированный h-файл

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class java_example_pkg_ClassWithNativeMethod */

#ifndef _Included_java_example_pkg_ClassWithNativeMethod
#define _Included_java_example_pkg_ClassWithNativeMethod
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     java_example_pkg_ClassWithNativeMethod
 * Method:    myNativeMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Функция `Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod(JNIEnv *, jobject)`, написанная на языке C++, должна обеспечивать реализацию метода `myNativeMethod()` в классе Java.

Имя функции C++ состоит из:

- префикса `Java`;
- разделителя _;
- модифицированного имени пакета (знаки подчеркивания _ заменяются на _1), в нашем случае `java_1example_1pkg`;
- разделителя _;
- имени класса, у нас это `ClassWithNativeMethod`;
- разделителя _;
- имени платформо-зависимого метода, в нашем случае `myNativeMethod`.

Первый параметр `JNIEnv *` в рассматриваемой функции обеспечивает доступ платформо-зависимого кода к параметрам и объектам, передающимся из функции C++ в Java (в частности, для доступа к стеку).

Для методов объекта, как в нашем примере, второй параметр имеет имя `jobject` — это ссылка на объект, из которого вызывается платформо-зависимый метод. Если же задается метод класса (мы такой пример не рассматриваем), то второй параметр в автоматически сгенерированном коде будет иметь имя `jclass` — это ссылка на класс, из которого вызывается платформо-зависимый метод.

В языке C++ нет ссылок, но в Java все переменные объектного типа являются ссылками, поэтому второй параметр функции C++ отождествляется с соответствующей ссылочной переменной Java.

Если в платформо-зависимом методе имеются параметры, то список параметров функции C++ расширяется. Например, если мы зададим метод

```
public native int myNativeMethod(int i);
```

то список параметров функции C++ станет таким:

```
(JNIEnv *, jobject, jint)
```

А тип функции станет `jint` вместо `void`.

В табл. 16.1 приведено соответствие типов Java и C/C++. Это соответствие на уровне исходного кода C/C++ задается в файле jdk1.6.0_11\include\jni.h.

Таблица 16.1. *Соответствие типов Java и C++*

Тип Java	Тип JNI (C++)	Характеристика типа JNI
`boolean`	`jboolean`	1 байт, беззнаковый
`byte`	`jbyte`	1 байт
`char`	`jchar`	2 байта, беззнаковый
`short`	`jshort`	2 байта
`int`	`jint`	4 байта
`long`	`jlong`	8 байтов
`float`	`jfloat`	4 байта
`double`	`jdouble`	8 байтов
`void`	`void`	—
`Object`	`jobject`	Базовый для остальных классов
`Class`	`jclass`	Ссылка на класс Java
`String`	`jstring`	Строки Java
`массив`	`jarray`	Базовый для классов массивов
`Object[]`	`jobjectArray`	Массив объектов
`boolean[]`	`jbooleanArray`	Массив булевых значений
`byte[]`	`jbyteArray`	Массив байтов (знаковых значений длиной в 1 байт)

Таблица 16.1 *(окончание)*

Тип Java	Тип JNI (C++)	Характеристика типа JNI
`char[]`	`jcharArray`	Массив кодов символов
`short[]`	`jshortArray`	Массив коротких целых
`int[]`	`jintArray`	Массив целых
`long[]`	`jlongArray`	Массив длинных целых
`float[]`	`jfloatArray`	Массив значений `float`
`double[]`	`jdoubleArray`	Массив значений `double`
`Throwable`	`jthrowable`	Обработчик исключительных ситуаций

В реализации метода требуется объявить переменные. Листинг 16.3 иллюстрирует реализацию функции на C++, вычисляющую квадрат переданного в метод значения и возвращающую его в качестве результата (пример чисто учебный).

Листинг 16.3. Функция C++, вычисляющая квадрат переданного в метод значения

```
#include "java_example_pkg_ClassWithNativeMethod.h"
JNIEXPORT jint JNICALL Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *env, jobject obj, jint i ){
    return i*i
};
```

Отметим, что при работе со строками и массивами для получения и передачи параметров требуется ссылочная переменная `env` (от англ. *environment* — окружение). Она имеет тип `JNIEnv` (окружение JNI) и позволяет получать доступ к объекту C++, отвечающему за доступ к памяти, выделенной JNI под массивы и объекты. Например, получение длины целого массива, переданного в переменную `jintArray intArr`, будет выглядеть так:

```
jsize length=(*env)->GetArrayLength(env, intArr);
```

Выделение памяти под переданный массив:

```
jint *intArrRef=(*env)->GetIntArrayElements(env, intArr,0);
```

Далее с массивом `intArr` можно работать как с обычным массивом C++.

Высвобождение памяти из-под массива:

```
(*env)->ReleaseIntArrayElements(env, intArr, intArrRef ,0);
```

Есть аналогичные функции для доступа к элементам массивов всех примитивных типов:

`GetBooleanArrayElements`, `GetByteArrayElements`, `GetDoubleArrayElements` и др.

Эти функции копируют содержимое массивов Java в новую область памяти, с которой и идет работа в C++. Для массивов-объектов имеется не только функция

`GetObjectArrayElement`, но и `SetObjectArrayElement` — для получения и изменения отдельных элементов таких массивов.

Строка Java `jstring s` преобразуется в массив символов C++ так:

```
const char *sRef=(*env)->GetStringUTFChars(env,s,0);
```

Ее длина находится так:

```
int s_len=strlen(sRef);
```

Высвобождается из памяти так:

```
(*env)->ReleaseStringUTFChars(env,s,sRef);
```

Сделаем некоторые уточнения.

При вызове native-метода первым аргументом передается специальный интерфейсный указатель. Затем идет ссылка на класс или на объект — в зависимости от того, является ли метод методом класса или методом объекта. Идущие далее аргументы соответствуют обычным аргументам метода Java.

Примитивные типы передаются в native-методы по значению, объектные — по ссылке. Виртуальная машина не удаляет объект, переданный в native-код, пока не получит оттуда уведомление о том, что этот объект уже не нужен.

Ссылки на объекты в рамках JNI подразделяются на глобальные и локальные. Локальные существуют только во время выполнения метода, при этом объекты, на которые они ссылаются, автоматически удаляются по окончании работы метода. Глобальные ссылки необходимо удалять явно. Объекты, на которые ссылаются локальные ссылки, также могут быть удалены явно (это требуется, например, для экономии памяти при работе с большими массивами данных).

16.3. Пакет C/C++ Pack. Подключение компиляторов C++, C и библиотек

В состав NetBeans 6.5 входит пакет C/C++ Pack, позволяющий разрабатывать приложения C/C++. Нас он интересует только в качестве инструмента создания динамических библиотек, подключаемых к Java.

Имеется возможность создавать проекты C/C++:

- на основе файлов с уже существующим исходным кодом;
- новые приложения;
- динамические библиотеки;
- статические библиотеки.

Новый проект C/C++ создается обычным образом: через меню **File/New Project/** (рис. 16.1).

Необходимо разочаровать поклонников C/C++: в поставку NetBeans компилятор C/C++ не входит, поэтому потребуется компилятор от стороннего производителя. По умолчанию NetBeans работает с компилятором g++ из свободно распространяемого комплекта GCC (GNU Compilers Collection).

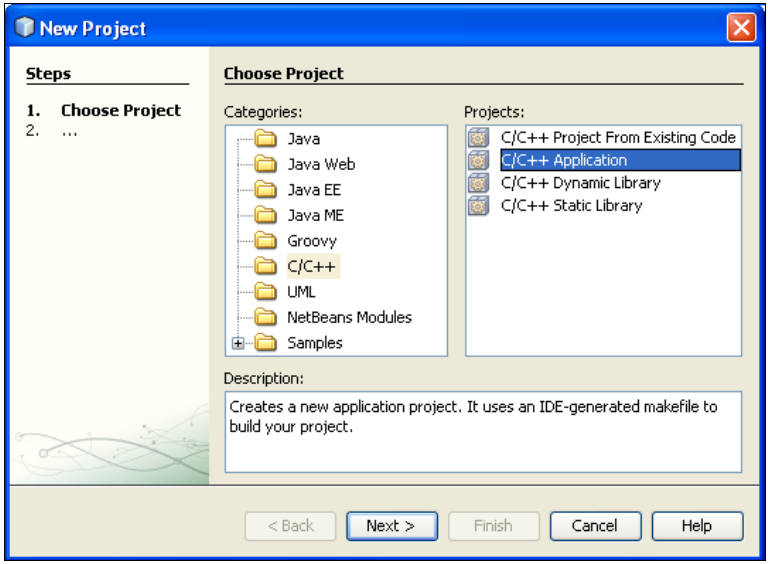


Рис. 16.1. Создание нового проекта C/C++

Среда NetBeans анализирует переменные окружения, имеющиеся в системе, и по доступным путям самостоятельно пытается подключить необходимые программы с известными ей именами (make.exe, gdb.exe, gcc.exe, g++.exe).

Оригинальный компилятор GCC под Windows работать не может, т. к. он рассчитан на Linux. Но существуют проекты Cygwin (**http://cygwin.com/**) и MinGW (**http://mingw.org/**), также развиваемые GNU, в которых компиляторы GCC адаптированы под Windows и распространяются свободно в составе соответствующих комплектов, называемых коллекциями.

Для установки коллекции Cygwin следует с сайта **http://cygwin.com/** загрузить файл setup.exe, запустить его и выбрать опцию **Download Without Installing** (Загружать без установки), после чего определить папку, в которой будут сохраняться файлы дистрибутива (по умолчанию это та папка, откуда запускают файл setup.exe, ее следует оставить без изменения). Потом нужно выбрать способ подключения к Интернету. Для большинства пользователей подойдет вариант **Use IE5 Settings** (Использовать установки Internet Explorer версии 5 или более поздней). Затем из предлагаемого списка нужно выбрать сайт, с которого следует проводить загрузку (вообще говоря, любой из них). Далее в появившейся форме необходимо указать набор загружаемых программ. Выберите в категории **Devel** (от Development — разработка) следующие файлы:

- **gcc-g++:C++ Compiler**;
- **gdb: The GNU Debugger**;
- **make:The GNU version of the 'make' utility**.

Остальные необходимые файлы при этом отметятся автоматически (в том числе **gcc-core:C Compiler**) — и эти отметки не следует снимать!

В указанном комплекте загружается около 25 Мбайт, считая сам инсталлятор.

Те, кому в работе необходим компилятор Fortran, могут также отметить файл

gcc-g77:Fortran Compiler;

NetBeans 6.5 обеспечивает поддержку работы с этим языком программирования, до сих пор важным для физиков.

После загрузки вы можете переносить файлы дистрибутива и устанавливать их на компьютеры без использования Интернета — нужно запустить файл setup.exe и выбрать опцию **Install from Local Directory** (Загружать из локальной папки), после чего повторить уже описанную процедуру выбора папки инсталляции и инсталлируемых файлов.

По умолчанию инсталлятор предлагает установить Cygwin в папку C:\Cygwin, именно туда и нужно проводить установку, не следует устанавливать Cygwin в папку C:\Program files. Это связано с тем, что некоторые программы Cygwin неправильно работают с файлами и папками, в именах которых встречаются пробелы или неанглоязычные (например, русские) символы. Это же относится к комплекту Mingw.

После инсталляции комплекта требуется вручную скопировать библиотеку cygwin1.dll из папки C:\cygwin\bin в папку WINDOWS\system32 (по крайней мере, в Windows XP).

Кроме того, необходимо в операционной системе задать путь к папке C:\cygwin\bin (Панель управления/Система, закладка **Дополнительно**, кнопка **Переменные среды**, изменить значение системной переменной `Path` — добавить в конец текст `;C:\cygwin\bin`).

Согласно официальной информации, опубликованной на сайте **http://www.netbeans.org/community/releases/65/cpp-setup-instructions.html**, NetBeans 6.5 испытан в работе с комплектом компиляторов:

Cygwin 1.5.21

Cygwin gcc-core (C compiler) 3.4.4

Cygwin gcc-c++ (C++ compiler) 3.4.4

Cygwin gdb (GNU Debugger) 6.8

Cygwin make 3.81

Однако предоставляемая для загрузки с официального сайта **http://cygwin.com/** версия Cygwin на момент написания второго издания книги уже была более поздняя (1.5.25-14), и с ней среда NetBeans 6.5 нормально работала. Версии остальных программ соответствовали рекомендуемым.

Проверить версии программ комплекта Cygwin (а также то, правильно ли установлен путь) можно в командной строке (например, в файловом менеджере Far) с помощью команд

```
gcc --version
g++ --version
make --version
gdb --version
```

Напомним, что просмотреть в Far результаты вывода можно, нажав комбинацию клавиш <Ctrl>+<O>, а повторное ее нажатие возвращает в обычный режим.

Для комплекта MinGW процедура аналогична, но инсталляция происходит проще и удобнее. Сначала с сайта **http://sourceforge.net/project/showfiles.php?group_id=2435**, на котором выкладываются официальные дистрибутивы MinGW, следует скачать файл инсталлятора (Automated MinGW Installer). На момент написания книги это был файл MinGW-5.1.4.exe.

После запуска инсталлятора, кроме отмеченных по умолчанию базовых средств компиляции (MinGW Base Tools), следует отметить флажком **g++ compiler** и утилиту MinGW Make. Те, кому в работе необходим компилятор Fortran, могут также отметить файл **g77 Compiler**.

Объем загружаемых файлов около 18 Мбайт. Отметим, что в комплект MinGW отладчик gdb не входит, но можно скачать с сайта **http://sourceforge.net/project/showfiles.php?group_id=2435** архив и скопировать соответствующие папки из архива в папки установленного комплекта MinGW.

Пока среда NetBeans не всегда устойчиво работает с пакетами Cygwin и MinGW. При этом иногда приходится использовать утилиту make из Cygwin, а компилятор g++ из MinGW. Поэтому имеет смысл держать на компьютере оба комплекта: как Cygwin, так и MinGW.

После установки комплектов Cygwin и/или MinGW следует проверить и изменить настройки NetBeans для работы с компиляторами и утилитами C/C++. Это делается в форме, вызываемой с помощью пункта меню **Tools/Options**, кнопка **C/C++** (рис. 16.2).

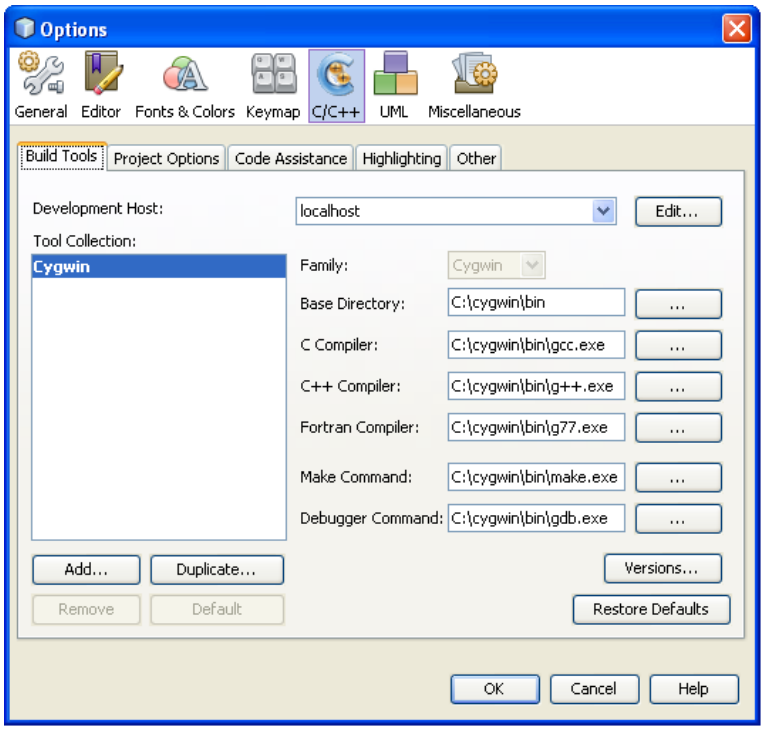


Рис. 16.2. Настройки работы с компиляторами и утилитами C/C++

На вкладке **Build Tools** в области **Tool Collection** (Коллекция инструментов) указаны найденные средой коллекции. При необходимости можно добавить коллекцию с помощью кнопки **Add...** (Добавить).

Ненужные коллекции можно удалить, выделив соответствующую строку и нажав кнопку **Remove** (Удалить), — они удаляются только из настроек NetBeans, а не с диска, где

установлены. Поскольку NetBeans 6.5 иногда путает настройки для разных коллекций, следует оставить в окне **Tool Collection** только одну коллекцию. Далее мы будем работать с Cygwin.

В правой части окна указаны пути к соответствующим компиляторам. Если какая-то из необходимых программ не найдена, вместо имени файла показывается красным цветом надпись **<Not Found>** (Не найдено). В этом случае можно добавить путь к ней с помощью кнопки ..., расположенной справа от соответствующего пункта ввода.

Возможна отладка работы приложений на удаленном компьютере — соответствующий путь можно добавить с помощью кнопки **Edit...** По умолчанию работа идет на локальном компьютере (local host).

Если на вашем компьютере установлено более одной коллекции компиляторов, то можно выбрать нужную коллекцию. Но, как уже говорилось, это может приводить к проблемам, и на начальном этапе работы следует подключать к NetBeans только Cygwin. К тому же необходимо помнить, что иногда NetBeans записывает скомпилированные файлы в папки, относящиеся не к тому компилятору: у автора возникла ситуация, когда при компиляции в коллекции Cygwin (и всех соответствующим образом проведенных настройках), результаты попадали в папку dist\Release\MinGW-Windows, а не в папку dist\Release\ Cygwin-Windows. Но это не вызывало путаницы, т. к. в папке dist\Release имелась только одна папка.

Учтите, что компиляторы из любой коллекции могут пользоваться сторонними утилитами make.exe и gdb.exe. Поэтому, например, вы можете в коллекции Cygwin подключить компилятор g++ из MinGW. Или в коллекции MinGW использовать утилиту make.exe из Cygwin (ее родная утилита несовместима с NetBeans) или отладчик gdb из CygWin. Точно так же можно сменить и сами компиляторы.

Но даже после настройки путей к утилитам и компиляторам проекты все еще не будут компилироваться. На вкладке **Code Assistance** нужно добавить пути к необходимым библиотекам для компиляторов C и C++. Мы будем пользоваться преимущественно компилятором C++, но компилятор C пригодится для осваивания простейших примеров. Обратите внимание, что настройки для компилятора C никак не связаны с настройками для компилятора C++, и пути к include-папкам для них необходимо настраивать независимо.

Например, если потребуется работать с native-методами, то обязательно подключение папок include и include\win32 из JDK (рис. 16.3).

При использовании операторов:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

для компиляторов Cygwin следует добавить путь: Cygwin\usr\include (у автора это C:\Cygwin\usr\include). Для компиляторов MinGW это MinGW\include.

Для подключения библиотеки

```
#include <iostream>
```

следует добавить путь:

Cygwin\lib\gcc\i686-pc-cygwin\3.4.4\include\c++ (в MinGW это MinGW\include\c++\3.4.5).

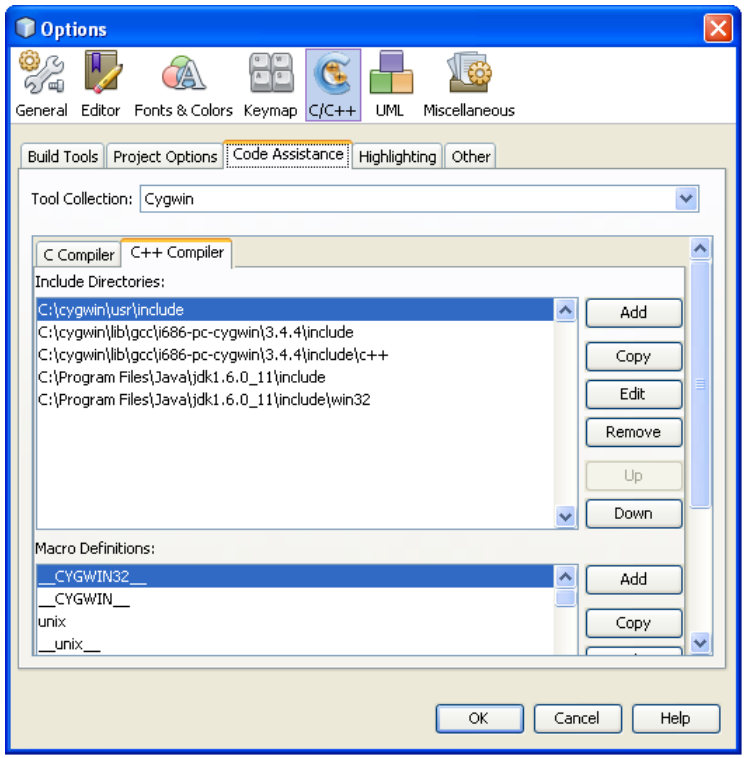


Рис. 16.3. Настройка путей к библиотекам C/C++

Заголовочный файл `iostream` не имеет расширения. Устаревшая версия той же библиотеки находится в файле `iostream.h` (в другой папке), но ее использовать не рекомендуется. Это вообще распространенная практика — давать заголовочным файлам исправленных версий библиотек то же имя, но без расширения h.

И наконец, для подключения библиотеки

```
#include <stdarg.h>
```

следует добавить путь Cygwin\lib\gcc\i686-pc-cygwin\3.4.4\include.

Настройка путей вручную очень утомительна, и в NetBeans было добавлено средство автоматической установки этих путей. Для такой установки следует прокрутить вниз правую полосу прокрутки (рис. 16.4). После этого необходимо нажать кнопку **Reset Settings** (Сбросить установки) и подтвердить в появившемся диалоге, что мы действительно готовы потерять все настройки в данной форме. После чего в форме появится большое число путей по умолчанию для данной коллекции.

Правда, пути

C:\cygwin\usr\include

и

C:\cygwin\usr\include\w32api

перепутаны местами: их порядок необходимо поменять с помощью кнопки **Up** (Вверх) или **Down** (Вниз).

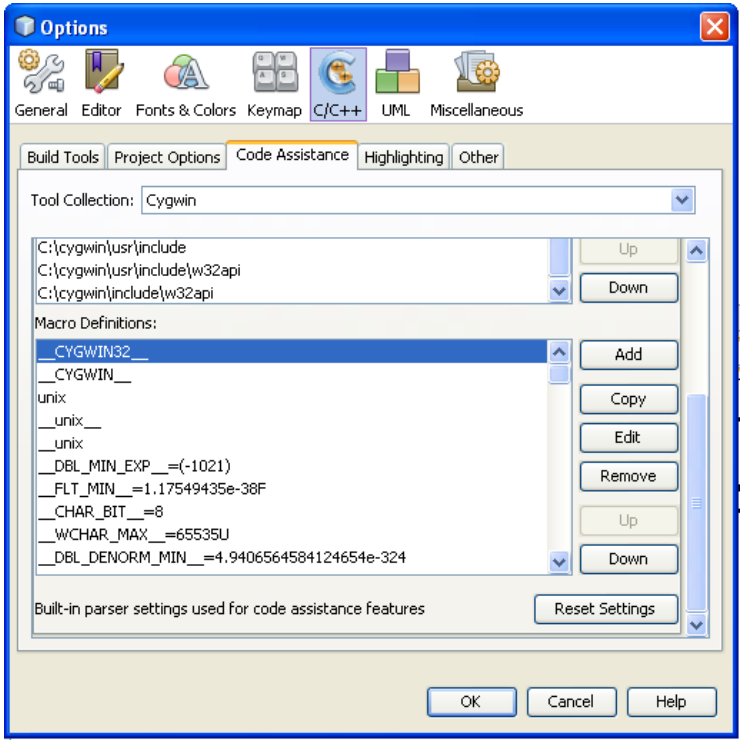


Рис. 16.4. Настройки путей к библиотекам C/C++

Хочется отметить еще одну неприятную ошибку в C/C++ Pack: довольно часто операции **Build** и **Clean** перестают работать, выдавая диагностику

```
*** multiple target patterns.Stop.
```

В этом случае следует вручную удалить в проекте папки build и dist с помощью какого-либо файлового менеджера.

Кроме общих опций можно задавать параметры компиляции для каждого отдельного проекта. Это особенно важно из-за того, что у NetBeans часто возникают проблемы с нахождением include-папок, пути к которым заданы в общих настройках. В частности, такая проблема регулярно возникала у автора с путями к include-папкам JDK, необходимым для подключения JNI. Поэтому приходится задавать пути к этим папкам непосредственно для проекта — этот вариант всегда работает.

Чтобы задать параметры компиляции для отдельного проекта, следует щелкнуть правой кнопкой мыши по имени проекта C++ и в появившемся контекстном меню выбрать самый нижний пункт — **Properties**. Появится форма, в которой можно задать общие свойства компиляции проекта (рис. 16.5): компьютер, на котором будет запускаться приложением (Development Host), используемую коллекцию компиляторов (Tool Collection), подключение/отключение компиляторов C, C++ и Fortran (**C Compiler Requied** — требуется ли компилятор) и т. д.

Важнейшие из свойств в категории **C/C++**:

- **General** (Общие) — можно задавать пути к исходным кодам проекта из различных папок. Также можно задавать кодировку (по умолчанию стоит UTF-8).

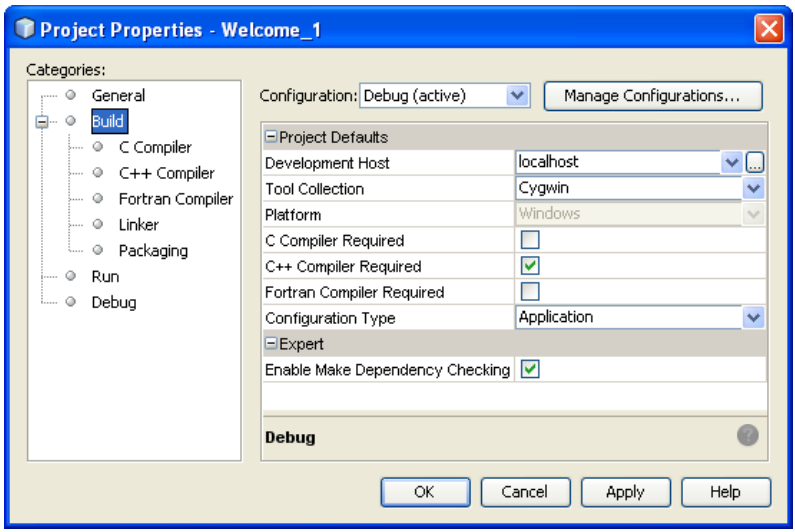


Рис. 16.5. Настройка общих свойств компиляции отдельного проекта C/C++

- **Build**, узел **C++ Compiler** (рис. 16.6), пункты позволяют задавать:
  - **Include Directories** — пути к папкам include, необходимым для проекта;
  - **Preprocessor Definitions** — опции препроцессора;
  - **Development Mode** (режим разработки) — режим компиляции, т. е. уровни оптимизации и диагностики компилятора;
  - **Warning Level** (уровень предупреждений) — степень детализации предупреждений компиляции;

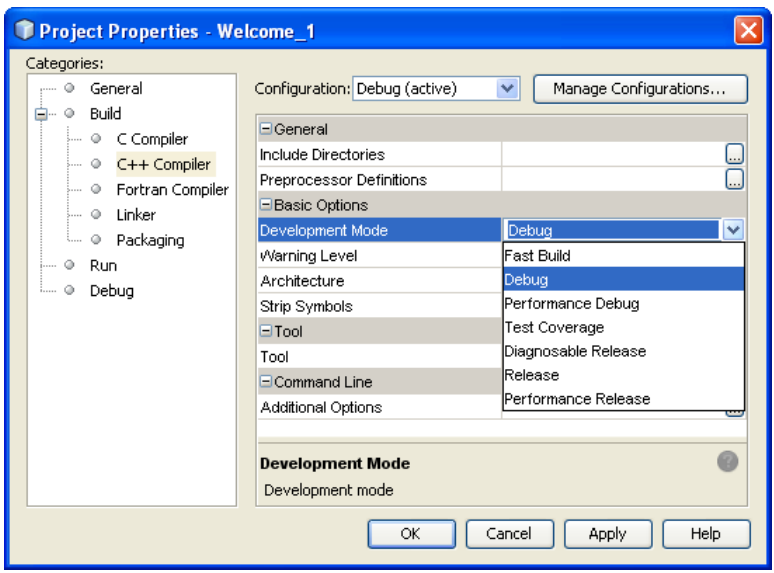


Рис. 16.6. Настройка свойств проекта C/C++ в категории **C/C++**

- **Architecture** (Архитектура) — разрядность целевой платформы, 32- или 64-разрядная;
- **Tool** (Инструмент) — файл компилятора g++, чтобы в разных проектах использовать разные версии файлов или файлы из разных коллекций;
- **Command Line** (Командная строка) — дополнительные опции компилятора.

♦ **Build**, узел **Linker** (Компоновщик) (рис. 16.7), пункты позволяют задавать:

- **Output** (Вывод) — имя выходного файла (в нашем случае — динамической библиотеки). По умолчанию это файл с именем проекта с префиксом lib (сокращение от library) и расширением dll (для платформы Windows). Но префикс (то, что стоит в начале) можно убрать редактированием либо непосредственно в строке, либо в окне редактора, появляющегося при нажатии кнопки [...], стоящей справа от пункта **Output**;
- **Additional Options** (Дополнительные установки) — параметры компоновщика. При компиляции библиотеки DLL, предназначенной для работы с native-методами, обязательно задание в этом пункте опции **-Xlinker --add-stdcall-alias**, иначе вызовы методов библиотеки не будут удовлетворять спецификатору sdtcall.

♦ **Build**, узел **Packaging** — позволяет задавать тип выходного архива (tar, zip и др.), а также программу, с помощью которой создается архив.

♦ **Run** — позволяет задавать аргументы командной строки приложения; папку, из которой запускается приложение; переменные окружения; необходимость компилировать приложения перед запуском (по умолчанию флажок установлен) и др.

♦ **Debug** — позволяет задавать путь к отладчику gdb.

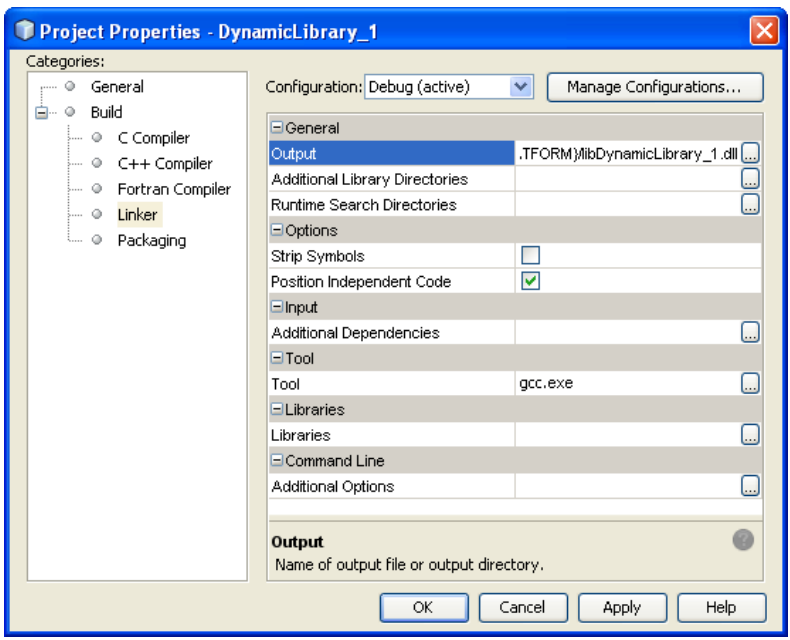


Рис. 16.7. Настройка свойств проекта C/C++ в категории **Linker**

Особо подчеркнем, что для режимов отладки (конфигурация **Debug**) и окончательной компиляции (конфигурация **Release**) все упомянутые настройки действуют совершенно независимо, что предоставляет удобную возможность использования в этих случаях разных опций и даже разных компиляторов (причем для каждого проекта самостоятельно). Имеется возможность выбрать один из семи режимов окончательной компиляции (выпадающий список **Development Mode**, рис. 16.6) — от самого быстрого до медленного и наиболее оптимизированного:

- **Fast Build** — быстрая компиляция (ключ оптимизации отсутствует);
- **Debug** — отладка (ключ оптимизации отсутствует);
- **Performance Debug** — отладка с качественной компиляцией (ключ оптимизации `-O`);
- **Test Coverage** — тестирование (ключ оптимизации отсутствует);
- **Diagnosable Release** — релиз с диагностикой (ключ оптимизации `-O2`);
- **Release** — релиз (ключ оптимизации `-O2`);
- **Performance Release** — релиз с высокой оптимизацией (ключ оптимизации `-O3`).

В проектах C/C++ имеется еще одна дополнительная возможность — устанавливать для отдельных файлов проекта пути к include-файлам, опции препроцессора, путь к файлу компилятора и т. д. Это делается с помощью пункта **Properties** контекстного меню, вызываемого для данного файла.

Еще одно замечание, важное для разработки больших проектов. По умолчанию размер динамической области памяти (кучи — heap) среды NetBeans составляет 128 Мбайт. Этого бывает достаточно для компиляции проектов примерно со 100 файлами исходных кодов. Для больших проектов размер кучи следует увеличить (см. в справочной системе NetBeans раздел **C and C++ Development/Getting Started/Memory Requirements**).

Наконец, отметим, что в редакторе исходных кодов C, C++, Fortran реализованы все те возможности показа ошибок, предупреждений и подсказок, о которых говорилось ранее по отношению к языку Java.

16.4. Создание приложений C++ и C

После создания приложения C++ возникает проект с пустыми списками файлов исходных кодов и автоматически созданной заготовкой make-файла, используемого программистами C/C++ для управления операциями компиляции и сборки файлов. В make-файле содержится набор инструкций утилите make.exe, занимающейся компиляцией и сборкой файлов. Для того чтобы работать с проектами C/C++ в среде NetBeans, знания командного языка утилиты make.exe не требуется — NetBeans создает и меняет этот файл автоматически.

Новые файлы исходных кодов задают с помощью пункта контекстного меню **New** (рис. 16.8).

Листинг 16.4 иллюстрирует созданный заголовочный файл.

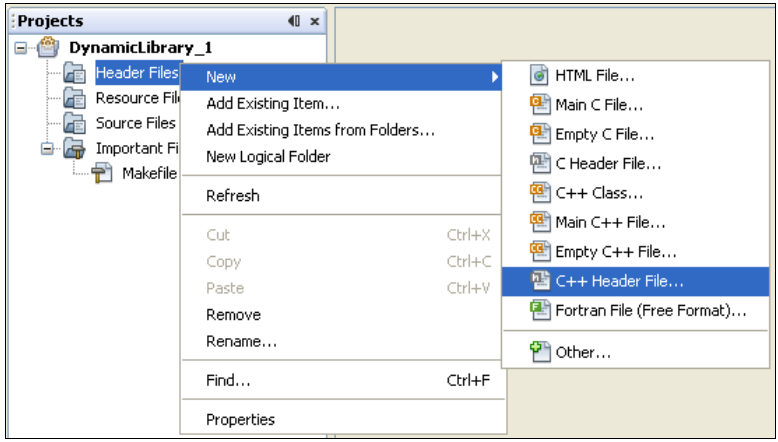


Рис. 16.8. Добавление в проект h-файла (заголовка)

Листинг 16.4. Созданный NetBeans заголовочный файл

```
/*
 * File:   newfile.h
 * Author: V.V.Monakhov
 *
 * Created on 23 ������ 2009 ., 23:53
 */
#ifndef _NEWFILE_H
#define _NEWFILE_H
#endif
/* _NEWFILE_H */
```

Символы с вопросиками отражают проблему с русскоязычной кодировкой (в проектах Java эта проблема уже несколько лет не возникает).

Аналогично можно создать и главный файл C/C++.

Но, как мы видим, это недостаточная помощь начинающим программистам. Гораздо удобнее воспользоваться примерами посредством команды **File/New Project/Samples/ "C/C++"**.

В частности, можно создать простейшее приложение C, вызвав команду **File/New Project.../Samples/ "C/C++"/Arguments**.

В результате в окно консоли выведутся аргументы, с которыми запущено приложение. Исходный код в данном случае (как и во всех последующих) начинается с информации о правах использования примеров (листинг 16.5).

Листинг 16.5. Информация о правах использования примеров C/C++

```
/*
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
 *
 * Copyright 1997-2007 Sun Microsystems, Inc. All rights reserved.
```

```
 *
* The contents of this file are subject to the terms of either the GNU
 * General Public License Version 2 only ("GPL") or the Common
 * Development and Distribution License("CDDL") (collectively, the
 * "License"). You may not use this file except in compliance with the
 * License. You can obtain a copy of the License at
 * http://www.netbeans.org/cddl-gplv2.html
 * or nbbuild/licenses/CDDL-GPL-2-CP. See the License for the
 * specific language governing permissions and limitations under the
 * License.  When distributing the software, include this License Header
 * Notice in each file and include the License file at
 * nbbuild/licenses/CDDL-GPL-2-CP.  Sun designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Sun in the GPL Version 2 section of the License file that
 * accompanied this code. If applicable, add the following below the
 * License Header, with the fields enclosed by brackets [] replaced by
 * your own identifying information:
 * "Portions Copyrighted [year] [name of copyright owner]"
 *
 * Contributor(s):
 *
 * The Original Software is NetBeans. The Initial Developer of the Original
 * Software is Sun Microsystems, Inc. Portions Copyright 1997-2007 Sun
 * Microsystems, Inc. All Rights Reserved.
 *
 * If you wish your version of this file to be governed by only the CDDL
 * or only the GPL Version 2, indicate your decision by adding
 * "[Contributor] elects to include this software in this distribution
 * under the [CDDL or GPL Version 2] license." If you do not indicate a
 * single choice of license, a recipient has the option to distribute
 * your version of this file under either the CDDL, the GPL Version 2 or
 * to extend the choice of license to its licensees as provided above.
 * However, if you add GPL Version 2 code and therefore, elected the GPL
 * Version 2 license, then the option applies only if the new code is
 * made subject to such option by the copyright holder.
*/
```

В лицензии говорится:

- о том, что исходный код примера распространяется на основе лицензии GPL 2 (свободное использование и модификация) или лицензии CDDL (свободное использование без права модификации) — по выбору пользователя;
- о возможности пометки оригинальных участков кода, разработанных программистами на основе примера, фразами Portions Copyrighted (частичный копирайт);
- о том, что первоначальной средой разработки является NetBeans, а первоначальным разработчиком NetBeans — Sun Microsystems.

Данные лицензии подразумевают свободное бесплатное распространение программных продуктов, поэтому никаких ограничений на учебные примеры они не накладывают.

Содержимое примера args.c без комментариев о правах использования приведено в листинге 16.6.

Листинг 16.6. Основное содержимое примера `args.c`

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char**argv) {
    int i;

    // Prints arguments
    printf("Arguments:\n");
    for (i = 0; i < argc; i++) {
        printf("%i: %s\n", i, argv[i]);
    }

    return 0;
}
```

В разделе **File/New Project/Samples/ "C/C++"/** содержится еще несколько примеров. В частности, несколько более сложная заготовка **Subproject Application** (приложение с подпроектами). Это вариант примера **Hello, World!**, написанный на языке C++. Автоматически создаваемый исходный код этого консольного приложения, использующего статически подключаемые к нему библиотеки hello1lib, hello2lib, hello3lib и hello4lib, приведен в листинге 16.7.

Листинг 16.7. Пример Hello, World! с использованием DLL

```
#include <iostream >

#include "../hello1lib/hello1.h"
#include "../hello2lib/hello2.h"
#include "../hello3lib/hello3.h"
#include "../hello4lib/hello4.h"

int main(int argc, char**argv) {
    // Print welcome messages...
    cout << "Hello from main...\n";
    cout << hello1();
    cout << hello2();
    cout << hello3();
    cout << hello4();

    // Prints arguments...
    if (argc > 1) {
        cout << "\nArguments:\n";
```

```
        for (int i = 1; i < argc; i++) {
            cout << i << ": " << argv[i] << "\n";
        }
    }

    return 0;
}
```

Исходный код остальных библиотек аналогичен коду первой из них.

Листинг 16.8 содержит основной код заголовочного файла hello1.h, а листинг 16.9 — главного файла hello1.cc.

Листинг 16.8. Код заголовочного файла hello1.h

```
#ifndef _hello1_H
#define _hello1_H

extern char *hello1();

#endif  /* _hello1_H */
```

Листинг 16.9. Основной код главного файла hello1.cc

```
#include "hello1.h"

char *hello1() {
    return (char*)"Hello from hello1...\n";
}
```

В NetBeans 6.5 в мастерах создания проектов в категории **Samples | C/C++** имеется еще несколько примеров, иллюстрирующих работу с языками C и C++. Хотя, конечно, они настолько уступают по уровню и количеству примерам для Java, что могут служить только для освоения простейших программ, написанных на языках C и C++.

В заключение хотелось бы поблагодарить разработчиков модуля C/C++ за оперативное исправление ошибок и недоработок по сделанным автором замечаниям, а также за добавление в NetBeans поддержки компилятора Fortran, что помогает сделать NetBeans базовым средством разработки научного и учебного программного обеспечения.

16.5. Примеры приложений с native-методами

Рассмотрим пример создания приложения с native-методом. Этот процесс можно разделить на несколько этапов:

1. Создание исходного кода Java для native-методов.
2. Генерация h-файла заголовка C++, содержащего декларацию методов.
3. Написание на C++ реализации методов, заголовки которых заданы в сгенерированных h-файлах.
4. Компиляция кода C++ в динамическую библиотеку (DLL).

5. Запуск приложения вместе с DLL в NetBeans.
6. Распространение приложения вместе с DLL.

Опишем эти этапы подробнее на примере класса с native-функцией, вычисляющей произведение двух чисел типа `double`. Затем усовершенствуем пример, добавив native-метод с перемножением матриц и возвратом результата в виде матрицы.

Сначала создадим приложение **DesktopApplication**, в котором и будем проверять работу класса с native-методами.

В дальнейшем для краткости будем обозначать как <build-classes> путь к корневой папке classes, в которой хранятся папки пакетов со скомпилированными классами проекта, например:

C:\Documents and Settings\User\Мои документы\NetBeansProjects\16_5_JNI_example1\build\classes.

Первый этап. Создание исходного кода Java для native-методов

В узле **Source Packages** дерева проектов создадим пакет `jnipackage`, а в нем — класс `Multiply` (листинг 16.10).

Листинг 16.10. Класс `Multiply` с native-методом

```
package jnipackage;
public class Multiply {

    static {
        System.loadLibrary("CPP_MultiplyLibrary");
    }

    public static native double getJniProduct(double a, double b);
}
```

Библиотека CPP_MultiplyLibrary.dll загружается при запуске приложения, т. к. использован блок статической инициализации. Native-функция `getJniProduct` возвращает результат перемножения двух чисел, выполняемого функцией, написанной на языке C++, вызываемой из данной библиотеки DLL.

На прилагаемом к книге DVD это пример 16_5_JNI_example1.

Второй этап. Генерация h-файла заголовка C++, содержащего декларацию методов

Для того чтобы воспользоваться утилитой javah.exe, скомпилируем проект и в консольном окне перейдем в папку <build-classes>.

Автор рекомендует бесплатный для жителей бывшего СССР файловый менеджер Far. Это удобнее стандартных средств Windows, поскольку можно наблюдать результаты диагностики при возникновении ошибок (комбинация клавиш <Ctrl>+<O> для про-

смотра содержимого консоли). Но можно пользоваться и другими аналогичными программами (Total Commander, Windows Commander и т. п.).

В папке <build-classes> будут располагаться вложенные папки — папка с пакетом нашего приложения jniexample, папка пакета jnipackage, в котором расположен класс с native-методами, и папка META-INF.

В режиме командной строки выполним команду

```
"C:\Program Files\Java\jdk1.6.0_11\bin\javah.exe" jnipackage.Multiply
```

Чтобы каждый раз не набирать путь C:\Program Files\Java\jdk1.6.0_11\bin при вызове javah.exe (или java.exe при запуске из консоли jar-файлов), полезно включить путь к этой папке в путь операционной системы. Для этого нужно вызвать: **Панель управления/Система/Дополнительно/Переменные среды/Системные переменные**, выбрать в списке переменную **Path** (Путь) и добавить в конец имеющейся строки:

```
C:\Program Files\Java\jdk1.6.0_11\bin
```

После этого команду генерации h-файла можно будет делать проще:

```
javah jnipackage.Multiply
```

После вызова `javah.exe jnipackage.Multiply` в папке classes появится файл jnipackage_Multiply.h. К сожалению, пока в NetBeans отсутствует автоматическая генерация заголовков (хотя такую возможность следовало бы добавить).

Часто встречающиеся ошибки:

- попытка запуска команды не из папки classes, а из папки пакета (в данном случае jnipackage), в которой находится файл class скомпилированного класса (в данном случае Multiply.class);
- указание разделителя / или \ вместо точки в имени обрабатываемого класса: нужно писать jnipackage.Multiply (а не jnipackage/Multiply или jnipackage\Multiply).

В результате работы утилиты javah.exe мы получим сгенерированный код h-файла с заголовками C++ (листинг 16.11).

Листинг 16.11. Содержание файла jnipackage_Multiply.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class jnipackage_Multiply */

#ifndef _Included_jnipackage_Multiply
#define _Included_jnipackage_Multiply
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     jnipackage_Multiply
 * Method:    getJniProduct
 * Signature: (DD)D
 */
```

```
JNIEXPORT jdouble JNICALL Java_jnipackage_Multiply_getJniProduct
  (JNIEnv *, jclass, jdouble, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

Третий этап. Написание на C++ реализации методов, заголовки которых заданы в сгенерированном файле jnipackage_JniMultiply.h

Создаем проект, выполнив последовательность действий.

Вызываем пункт меню **File/New Project...**, в открывшейся форме выбираем категорию **C/C++**, после чего в окне **Projects** этой формы выбираем тип проекта **C/C++ Dynamic Library** (Динамическая библиотека C/C++). Нажимаем кнопку **Next>**, вводим в качестве имени библиотеки **CPP_MultiplyLibrary** и нажимаем кнопку **Finish**.

В получившемся проекте **CPP_MultiplyLibrary** для файла **Header Files** правой кнопкой мыши вызываем контекстное меню, где выбираем команду **Add Existing Item...** (Добавить существующий элемент). В файловом диалоге выбираем сгенерированный h-файл (но прежде его лучше вручную скопировать в головную папку проекта). Отметим небольшую проблему с размером диалоговой формы выбора файла — форма слишком широкая, но кнопка **Select** для выбора файла находится справа и скрывается при попытке уменьшить размер формы. Тем не менее файл можно выбрать двойным щелчком, так что особых трудностей это не вызывает.

Затем аналогичным образом вызываем контекстное меню для раздела **Source Files** (файлы исходных кодов C/C++) и выбираем команду **New/Empty C++ File...** (Новый/Пустой файл C++). В появившейся форме вводим имя: `jnipackage_Multiply`.

В появившемся пустом файле C++ пишем реализацию библиотеки, содержащей всего один метод (листинг 16.12).

Листинг 16.12. Содержимое файла jnipackage_Multiply.cc

```
#include "jnipackage_Multiply.h"
JNIEXPORT jdouble JNICALL Java_jnipackage_Multiply_getJniProduct (
                JNIEnv * env, jclass cls, jdouble a, jdouble b) {
   return a*b;
}
```

На прилагаемом к книге DVD это примеры 16_5_CPP_ex1_Cygwin и 16_5_CPP_ex1_MinGW. В первом приведены настройки для работы с коллекцией Cygwin, во втором — с коллекцией MinGW (но с подключенной утилитой make.exe из коллекции Cygwin).

Четвертый этап. Компиляция кода C++ в динамическую библиотеку (DLL)

Возможна компиляция DLL непосредственно в среде NetBeans, если поставить соответствующие опции в свойствах проекта *(см. разд. 16.3)*. При этом в свойствах проекта в разделе **Build/C++ Compiler** обязательно следует прописать пути к include-папкам.

Для коллекции Cygwin:

C:/Program Files/Java/jdk1.6.0_11/include/win32

C:/Program Files/Java/jdk1.6.0_11/include

C:/cygwin/usr/include/mingw

Для коллекции MinGW:

C:/Program Files/Java/jdk1.6.0_11/include/win32

C:/Program Files/Java/jdk1.6.0_11/include

Не забудьте про задание опции `-Xlinker --add-stdcall-alias` в пункте **Additional Options** компоновщика (Linker) в свойствах проекта.

Особое внимание обратите на задание этих параметров при переключении режимов **Debug** и **Release**: настройки одного режима никак не влияют на настройки другого.

После компиляции в NetBeans скомпилированная DLL возникает в папке проекта:

для коллекции Cygwin:

dist\Release\Cygwin-Windows — при компиляции в режиме **Release**,

dist\Debug\Cygwin-Windows — при компиляции в режиме **Debug**;

для коллекции MinGW:

dist\Release\ MinGW-Windows — при компиляции в режиме **Release**,

dist\Debug\ MinGW-Windows — при компиляции в режиме **Debug**.

Но, напомним, возможны сбои среды, если вы при создании проекта работали с одной коллекцией, а потом переключились на другую.

Также можно проводить компиляцию в режиме командной строки. Отметим, что как при компиляции с помощью NetBeans, так и при компиляции в режиме командной строки существует опасность получить неработающую библиотеку, хотя она и скомпилируется. Поэтому библиотеку обязательно нужно проверить путем вызова из нее функций в проекте Java.

В случае неработоспособности DLL полезно просмотреть содержимое DLL: имеются ли внутри требуемые методы, и те ли у них имена. Для такого просмотра можно использовать свободно распространяемую программу Dependency Walker (Искатель зависимостей), которая доступна на сайте **http://www.dependencywalker.com/**.

Для платформы Win32 и компиляторов реализации GCC MinGW компиляция в режиме командной строки выполняется так:

```
C:\MinGW\bin\g++ -I"C:\Program Files\Java\jdk1.6.0_11\include" -I"C:\Program
Files\Java\jdk1.6.0_11\include\win32" -O3 jnipackage_Multiply.cc -shared -o
CPP_MultiplyLibrary.dll -Xlinker --add-stdcall-alias
```

Для комплекта Cygwin в режиме командной строки работающую DLL скомпилировать не удалось.

После компиляции в режиме командной строки в папке с C++-кодом возникает файл CPP_MultiplyLibrary.dll.

Пятый этап. Запуск приложения вместе с DLL в NetBeans

Помещаем DLL в головную папку проекта (NetBeansProjects\16.5_JNI_example1). Теперь после компиляции и запуска приложения 16.5_JNI_example1, нажав кнопку **Перемножить**, можно вызвать метод из этой библиотеки (вычисляется и показывается произведение двух чисел).

Шестой этап. Распространение приложения вместе с DLL

Помещаем DLL в папку проекта dist (туда, где находится jar-файл приложения), у нас это NetBeansProjects\16.5_JNI_example1\dist.

Копируем папку dist вместе со всем содержимым на тот компьютер, где должно работать наше приложение. Теперь приложение готово к работе и можно двойным щелчком по имени jar-файла запустить его на выполнение. Но, конечно, на компьютере должна быть установлена среда JRE (например, из комплекта JDK).

Чтобы различные приложения операционной системы Windows могли пользоваться одной и той же библиотекой DLL, ее можно поместить в системную папку system32. На большинстве компьютеров это C:\WINDOWS\system32. В данном случае хранить DLL в папке с jar-файлом не обязательно.

Пример 16.5_JNI_example2 на прилагаемом DVD иллюстрирует работу с массивами с помощью native-методов. Он реализует только консольный вывод.

Главный файл `Main` приложения приведен в листинге 16.13.

Листинг 16.13. Исходный код файла example.Main

```
package example;

public class Main {

    public static String matrixToString(double[][] M) {
        String s = "";
        for (int i = 0; i < M.length; i++) {
            for (int j = 0; j < M[0].length; j++) {
                s = s + M[i][j] + "\t";
            }
            s = s + "\n";
        }
        return s;
    }
```

```
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        System.out.println("Product of 2.5 and 4 is: " +
                                    Multiply.getProduct(2.5, 4) + "\n");

        double A[][] = {{1, 2, 3}, {4, 5, 6}};
        double B[][] = {{1, 2}, {3, 4}, {5, 6}};
        double C[][] = Multiply.getProduct(A, B);
        System.out.println("Matrix A: \n" + matrixToString(A));
        System.out.println("Matrix B: \n" + matrixToString(B));
        if (C != null) {
            System.out.println("Product of A and B is:\n" +
                                        matrixToString(C));
        } else {
            System.out.println("Cannot multiply A and B!\n");
        }
    }
}
```

Замечание

Метод `matrixToString` реализован неэффективно: для больших матриц он будет работать крайне медленно, будет занимать очень много памяти и ресурсов процессора. В следующей главе мы разберемся в причинах и узнаем, каким образом найти места в программах с подобными неэффективными участками кода и как решить данную проблему.

Листинг 16.14 иллюстрирует класс `Multiply`, имеющий native-методы.

Листинг 16.14. Класс `Multiply` с двумя native-методами

```
package example;
public class Multiply {
   static {System.loadLibrary("CPP_MultiplyLibrary2");}
   public static native double getProduct(double A, double B);
   public static native double[][] getProduct(double[][] A,double[][] B);
}
```

Создаем h-файл командой, запускаемой из папки classes, например, такой:

```
D:\Util\Java\jdk1.6.0_11\bin\javah.exe  example.Multiply
```

Сгенерированный файл example_Multiply.h приведен в листинге 16.15.

Листинг 16.15. Содержимое файла example_Multiply.h

```
/* DO NOT EDIT THIS FILE — it is machine generated */
#include <jni.h>
/* Header for class example_Multiply */
```

```
#ifndef _Included_example_Multiply
#define _Included_example_Multiply
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     example_Multiply
 * Method:    getProduct
 * Signature: (DD)D
 */
JNIEXPORT jdouble JNICALL Java_example_Multiply_getProduct__DD
  (JNIEnv *, jclass, jdouble, jdouble);

/*
 * Class:     example_Multiply
 * Method:    getProduct
 * Signature: ([[D[[D)[[D
 */
JNIEXPORT jobjectArray JNICALL
                        Java_example_Multiply_getProduct___3_3D_3_3D
                         ( JNIEnv *, jclass, jobjectArray, jobjectArray);

#ifdef __cplusplus
}
#endif
#endif
```

Создаем на его основе и наполняем содержимым файл библиотеки example_Multiply.cc, написанный на языке C++ (листинг 16.16), примеры 16_5_CPP_ex2_Cygwin и 16_5_CPP_ex2_MinGW на прилагаемом к книге DVD.

Листинг 16.16. Содержимое файла example_Multiply.cc

```
#include "example_Multiply.h"

JNIEXPORT jdouble JNICALL Java_example_Multiply_getProduct__DD
                       (JNIEnv * env, jclass cls, jdouble a, jdouble b)
{
       return a*b;
}

// Метод копирует содержимое двумерного массива Java в буфер C++
void jMatrix2cMatrix(JNIEnv * env, jobjectArray jM, int r, int c, double* M){
    double* X = M;
    for (int i = 0;i < r;i++){
      jdoubleArray jrow =
        reinterpret_cast<jdoubleArray>(env->GetObjectArrayElement(jM,i));
      env->GetDoubleArrayRegion(jrow, 0, c, X);
      X+=c;
    }
}
```

```
// Метод создает двумерный массив Java с rows строками и cols столбцами
// и копирует данные из буфера C++ в этот массив
jobjectArray cMatrix2jMatrix(JNIEnv* env, double* C, int rows, int cols)
{
   jdoubleArray row = env->NewDoubleArray(cols);
   jclass cls = env->GetObjectClass(row);
   jobjectArray jC = (jobjectArray)env->NewObjectArray(rows, cls, row);
   for (int i = 0;i < rows;i++){
       row = env->NewDoubleArray(cols);
       env->SetDoubleArrayRegion(row, 0, cols, C);
       env->SetObjectArrayElement(jC, i, row);
       C += cols;
   }
   return jC;
}

JNIEXPORT jobjectArray JNICALL
         Java_example_Multiply_getProduct___3_3D_3_3D
           (JNIEnv * env, jclass cls, jobjectArray jA, jobjectArray jB) {
// Определяет размеры массивов jA и jB и проверяет, разрешено ли
// произведение соответствующих им матриц. В Java двумерные массивы – это
// массивы массивов. Поэтому матрица — это одномерный массив типа
// jobjectArray,каждый элемент которого является массивом типа //jdoubleArray

 // Число строк матрицы A
 int rowsA = env->GetArrayLength(jA);
 if (rowsA < 1) return NULL;

 // Число столбцов матрицы A
 jdoubleArray jrow =
        reinterpret_cast<jdoubleArray>(env->GetObjectArrayElement(jA,0));
 int colsA = env->GetArrayLength(jrow);
 if (colsA < 1) return NULL;

 // Число строк матрицы B
 int rowsB = env->GetArrayLength(jB);
 if (rowsB < 1) return NULL;

 // Число столбцов матрицы B
 jrow = reinterpret_cast<jdoubleArray>(env->GetObjectArrayElement(jB,0));
 int colsB = env->GetArrayLength(jrow);
 if (colsB < 1) return NULL;

 // Умножение матриц запрещено
 if (colsA != rowsB) return NULL;

 // Без оптимизации (выравнивания в памяти) мы должны копировать входные
 // матрицы в буфер C++ для того, чтобы гарантировать их непрерывное
 // расположение в памяти
 double* A = new double[rowsA*colsA];
```

```
 jMatrix2cMatrix(env, jA, rowsA, colsA, A);
 double* B = new double[rowsB*colsB];
 jMatrix2cMatrix(env, jB, rowsB, colsB, B);
 double* C = new double[rowsA*colsB];

 // Выполнение умножения
 for (int i = 0;i < rowsA;i++){
     for (int j = 0;j < colsB;j++){
         double d = 0;
         for (int k = 0;k < colsA;k++)
             d += A[i*colsA + k]*B[k*colsB + j];         C[i*colsB + j] = d;
     }
 }
 jobjectArray jC = cMatrix2jMatrix(env, C, rowsA, colsB);
 delete []A;
 delete []B;
 delete []C;

 return jC;
}
```

Основная особенность данного примера по сравнению с предыдущим — необходимость выделения в библиотеке C++ памяти под массив, возвращаемый native-методом. При этом функция `env->GetArrayLength(jA)` вычисляет размер полученного из Java-приложения в качестве параметра native-метода массива.

В конце вызова мы преобразуем массив `C` из формата C++ в массив `jC` формата Java. Затем уничтожаются ненужные уже массивы `A`, `B` и `C`, и в качестве результата возвращается `jC`.

В случае неудачи в качестве результата мы возвращаем `null`.

Краткие итоги по главе

- Для получения доступа к библиотекам, написанным на других языках программирования, в Java используются методы, объявленные с модификатором native (платформо-зависимый). При выполнении такого метода вызывается специфический для конкретной платформы двоичный код, а не платформо-независимый байт-код, как во всех других случаях.
- Заголовок платформо-зависимого метода описывают в классе Java, а реализуют на языках программирования C++ или C, позволяющих создавать динамически подключаемые библиотеки.
- Java-код взаимодействует с методами из динамических библиотек на основе интерфейса JNI (Java Native Interface).
- NetBeans 6.5 обладает развитыми средствами для разработки программ, написанных на языках C и C++. Однако пока еще при разработке таких программ возникает ряд проблем.

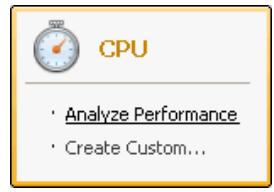
- При разработке native-кода под Linux можно использовать компиляторы GCC, а под Windows необходимо пользоваться коллекциями Cygwin или MinGW.
- Обязательна установка в свойствах проекта путей к include-папкам **<JDK>/include/win32** (для Windows), **<JDK>/include** и опции компоновщику `-Xlinker --add-stdcall-alias`. Для коллекции Cygwin также необходимо указать путь к include-папке <Cygwin>/usr/include/mingw.
- Настройки среды NetBeans для режимов **Debug** и **Release** независимы.
- То, что динамическая библиотека компилируется без ошибок и имеет правильный исходный код, еще не означает, что скомпилированный код библиотеки будет корректно работать. Обязательна проверка библиотеки в вызывающем ее Java-приложении.

Типичные ошибки

- При генерации h-файла утилитой javah пытаются запускать команду не из папки classes, а из папки пакета (в данном случае jnipackage), в которой находится файл class скомпилированного класса (в данном случае Multiply.class).
- При генерации h-файла утилитой javah ставят разделитель / или \ вместо точки в имени обрабатываемого класса: в нашем случае нужно писать `jnipackage.Multiply`, а не `jnipackage/Multiply` или `jnipackage\Multiply`.
- При компиляции DLL в свойствах проекта забывают указать пути к include-папкам.
- Пути к include-папкам перечисляют в неправильном порядке. Из-за чего проект не компилируется, хотя на первый взгляд все необходимые пути указаны.
- В путях к include-папкам указывают лишние папки перед теми, которые действительно необходимы, из-за чего в проект включаются неправильные фрагменты кода и он не компилируется.
- При компиляции DLL в свойствах проекта забывают указать опции `-Xlinker --add-stdcall-alias` в пункте **Additional Options** компоновщика (Linker), из-за чего вызов native-методов из скомпилированной библиотеки приводит к неправильной передаче параметров (они передаются со спецификатором вызова `C-call`, а не `stdcall`) и возбуждению исключительной ситуации или даже краху приложения.
- Забывают, что когда при компиляции операции **Build** и **Clean** перестают работать, выдавая диагностику `*** multiple target patterns.Stop`, помогает удаление в проекте папок build и dist (вручную в каком-либо файловом менеджере).

Задания

- Настроить опции подключения к NetBeans компиляторов C и C++.
- Создать приложение C++ на основе примера, имеющегося в NetBeans. Проверить работоспособность примера.
- Проверить работоспособность примера приложения с native-методом.



ГЛАВА 17

Отладка, тестирование и профилирование программ

При написании программ достаточно часто возникает необходимость искать ошибки в исходном коде. Только очень наивные начинающие программисты полагают, что способны написать программный код абсолютно правильно, не сделав никаких ошибок. Ошибки обнаруживают с помощью специального программного средства — *отладчика* (debugger).

Значительно повышают надежность разрабатываемых проектов *тестовые пакеты* — специальные программные элементы, основанные на использовании модуля JUnit. В этих пакетах программист должен создать по специальным правилам тесты, проверяющие, правильно ли работают разрабатываемые в основных пакетах классы после внесения каких-либо изменений.

Но даже после исправления ошибок и создания версии, кажущейся на первых порах окончательной, во многих случаях требуется повышение быстродействия программы и уменьшение требуемого объема памяти. Определить участки кода приложений, нуждающиеся в усовершенствовании, позволяет другое программное средство — *профилировщик* (profiler).

Вопросам *отладки*, *тестирования* и *профилирования* приложений посвящена данная глава.

17.1. Отладка приложений в среде NetBeans

Уже говорилось, что жесткость синтаксических конструкций языка программирования обеспечивает надежность программ, т. к. подавляющее большинство ошибок отслеживается еще на этапе компиляции. А в среде NetBeans такие ошибки (они называются *синтаксическими*) видны уже при редактировании, поскольку синтаксический анализатор проверяет исходный код приложения непосредственно во время редактирования.

Но есть *логические* ошибки, не связанные с синтаксическими конструкциями языка программирования. И их в принципе не может отследить никакой компилятор, т. к. они вызваны ошибочностью представлений программиста о том, как должна работать программа. Он думает, что программа должна делать одно, а она делает другое, причем непонятно, в какой ситуации и по какой причине.

Для пояснения рассмотрим пример 17_1_DesktopApplication, в котором проиллюстрирована ситуация, описанная в *разд. 5.7*. На рис. 17.1 (*а*—*г*) представлен результат вычисления суммы при различных значениях `n`. Из рис. 17.1, *в* видно, что при `n=10` наблюдается отклонение от тенденции — вычисленная сумма резко отличается от значений при других `n`. Допустим, что мы не знаем, почему возникла такая ситуация. Как нам установить причину?

Когда-то давным-давно применялись вставки отладочных фрагментов в основной текст программы с выводом на экран промежуточных значений, по которым можно было определить, что происходит в программе и почему она работает неправильно. В дальнейшем такие значения стали выводить в так называемые лог-файлы. Прочитав эти файлы можно было разобраться в том, что происходило в программе. В предыдущих главах мы фактически использовали консольный вывод в качестве лог-файлов.

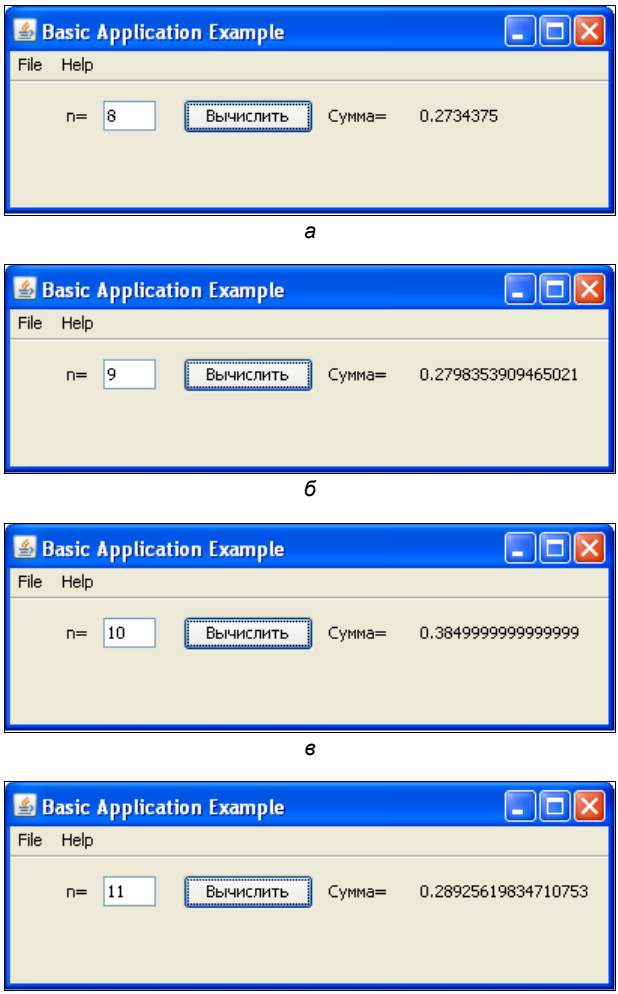


Рис. 17.1. Пример работы приложения: *а* — при `n` = 8; *б* — при `n` = 9; *в* — при `n` = 10 (с ошибкой); *г* — при `n` = 11

Но в современных средах разработки существует более эффективное средство для нахождения ошибок — debugger, т. е. программа для отыскания ошибок (bugs). Начинающие программисты обычно не понимают, зачем нужна отладка приложений и как она выполняется. А для профессиональных программистов отладчик — один из основных инструментов. В учебниках по программированию данному вопросу обычно не находится места, но мы попробуем восполнить этот пробел.

Чтобы работал режим отладки, требуется запустить приложение с помощью кнопки на панели инструментов, на которой нарисован маленький зеленый треугольник на фоне исходного кода программы (на рис. 17.2 на эту кнопку указывает стрелка).

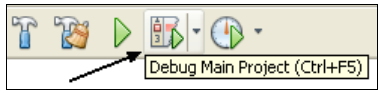


Рис. 17.2. Кнопка запуска приложения в режиме отладки

Отметим, что в большинстве других сред разработки одна и та же кнопка запускает приложение в обычном и отладочном режиме. А переключение этих режимов производится установкой опций компилятора. Подход, используемый в NetBeans, имеет некоторые преимущества: в обычном режиме можно независимо запускать произвольное количество приложений, их можно оставлять открытыми во время сеанса отладки, чтобы легче находить условия, которые необходимо создавать для отлаживаемого приложения.

Основные средства, используемые в режиме отладки, — *точки остановки* (breakpoints) и команды *режима пошагового выполнения*.

Чтобы установить точку остановки (их может быть произвольное число — по решению программиста), в редакторе исходного кода следует щелкнуть мышью в сером столбце слева от начала редактируемого кода (рис. 17.3). В поле появится розовый квадратик, а отмеченная строка подсветится розовым.

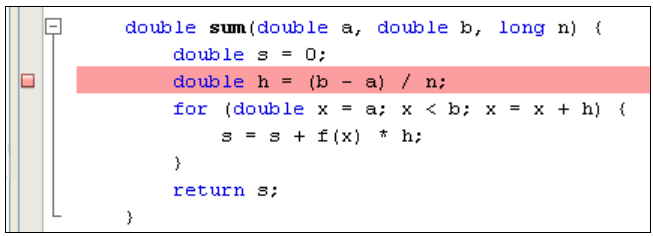


Рис. 17.3. Задание точки остановки

Снять точку остановки можно, щелкнув мышью по квадратику, отмечающему эту точку.

Сразу после запуска приложения в режиме отладки в левом окне рядом со вкладкой **Projects** возникает новая вкладка — **Debugging** (Отладка) (рис. 17.4). В ней показываются потоки выполнения нашего приложения, поэтому про нее мы поговорим после того, как разберем более простые возможности отладки.

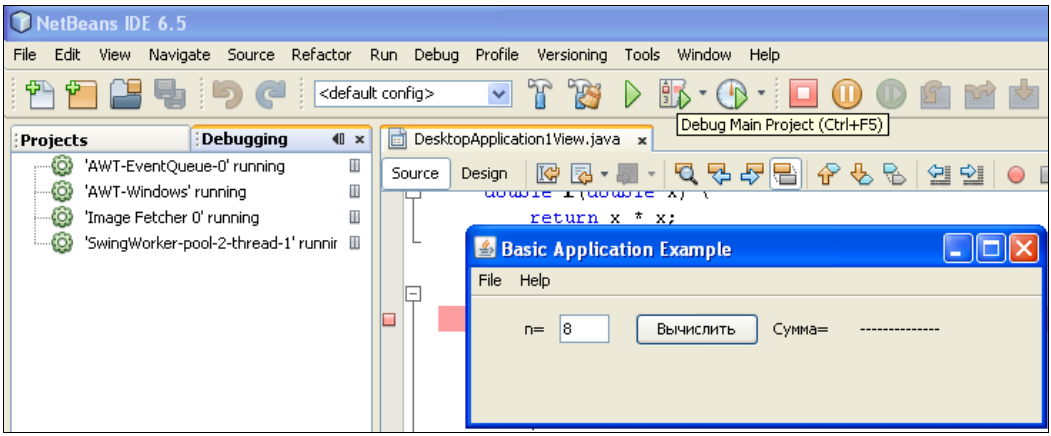


Рис. 17.4. Окно **Debug** после запуска приложения

В режиме отладки программа, дойдя до очередной точки остановки, прерывает обычную работу и останавливается, переключаясь в режим пошагового выполнения. Например, если мы нажмем кнопку **Вычислить**, произойдет вызов функции `sum`, в которой мы ввели точку остановки, и выполнение программы приостановится, а фокус переключится в редактор исходного кода (рис. 17.5).

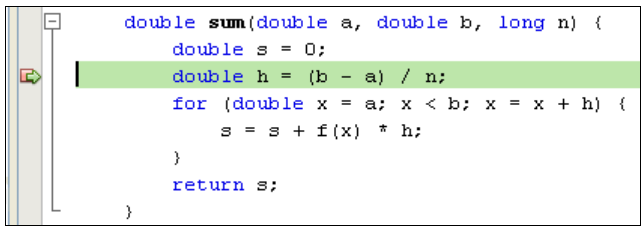


Рис. 17.5. Остановка программы в точке остановки

При этом операторы начинают выполняться по одному с остановкой после вызова очередного оператора (или группы операторов) и ожиданием от программиста команды запуска следующего оператора (группы операторов). Во время остановки программист видит в редакторе исходного кода маркер текущего оператора — стрелку, указывающую на строку в коде, где в данный момент находится программа (рис. 17.6), т. е. ту строку, с которой начнется выполнение очередного шага. Нажатие клавиши <F8> (Step Over — шаг вперед) осуществляет переход к следующей строке (следующему оператору).

Важная возможность отладчика — отображение текущих значений переменных. Для такого показа достаточно навести мышь на имя переменной — например, на переменную `h` (рис. 17.7).

Продолжить выполнение программы с выходом из пошагового режима можно, нажав клавишу <F5> (Continue — продолжить).

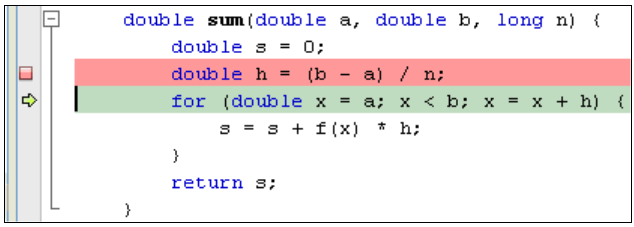


Рис. 17.6. Один шаг (Step Over) после точки остановки

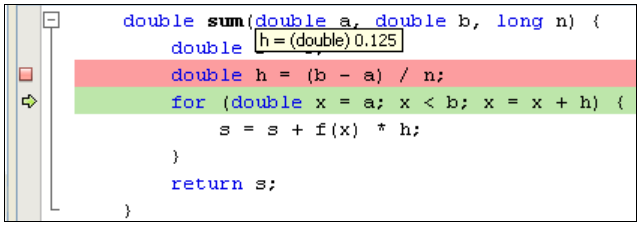


Рис. 17.7. Показ значений переменных

17.2. Команды режима пошагового выполнения

После запуска приложения в режиме отладки панель инструментов изменяет вид (рис. 17.8).

Рис. 17.8. Правая часть панели инструментов во время отладки

На панели появляются кнопки, дублирующие команды пошагового режима. Перечислим их в порядке следования кнопок.

- <Shift>+<F5> — **Finish Debugger Session** (Завершение сессии отладки). Отлаживаемое приложение снимается с исполнения. К такому же эффекту приводит закрытие отлаживаемого приложения.
- — **Pause** (Пауза). Кнопка активна только в обычном режиме выполнения отлаживаемого приложения (вне пошагового режима). Приостанавливает выполнение приложения в обычном режиме и переводит его в пошаговый режим. При этом в редакторе исходного кода маркер текущего оператора указывает на место, где произошла остановка.
- <F5> — **Continue** (Продолжить). Выход из пошагового режима и продолжение выполнения в обычном режиме с остановкой при достижении очередной точки остановки (breakpoint).
- <F8> — **Step Over** (Шаг вперед на один оператор).

- <Shift>+<F8> — **Step Over Expression** (Шаг вперед к вычислению очередного выражения). Очень полезная возможность отладчика, отсутствующая в большинстве других сред разработки. Если в операторах нет вызовов функций, то остановки происходят в начале каждого следующего оператора так же, как по команде **Step Over**. Но если в строке имеется выражение, то сначала происходит остановка перед выполнением строки, а затем остановка перед вычислением каждой функции. При этом можно с помощью наведения курсора мыши посмотреть значения параметров, передаваемых в функцию перед ее вычислением, а после вычисления очередной функции при наведении курсора мыши на имя функции можно увидеть вычисленное значение этой функции (имена вычисленных функций показываются подчеркнутыми).
- <F7> — **Step Into** (Шаг вперед с "проваливанием внутрь" каждого вызываемого метода). Для других операторов не отличается от **Step Over**. С помощью этой команды можно во время отладки зайти внутрь интересуемого метода и проследить, что происходит при его выполнении.
- <Ctrl>+<F7> — **Step Out** (Выход наружу из текущего метода). В частности, можно осуществлять выход из метода, в который до того произошло "проваливание внутрь" по команде **Step Into**. При этом после выхода маркер текущего положения остается на той строке, с которой был вызван метод, но вертикальной стрелкой помечается, что данный оператор уже выполнен (рис. 17.9).
- <F4> — **Run to Cursor** (Перейти к курсору). Фактически строке, на которой установлен курсор, назначается одноразовая точка остановки. Позиция курсора в строке не имеет значения, поскольку точка остановки назначается перед целой строкой, а не для какой-то ее части.

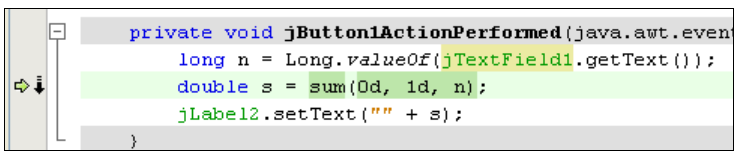


Рис. 17.9. Результат выхода из метода `sum` по <Ctrl>+<F7>

- — **Apply Code Changes** (Применить изменения в исходном коде). Если во время отладки программист отредактировал исходный код, то эти изменения не влияют на процесс отладки, т. к. отладчик вначале компилирует первоначальный вариант исходного кода в специальный вариант байт-кода, и именно этим кодом пользуется при выполнении. Данная команда используется для того, чтобы изменения исходного кода вступили в силу, т. е. чтобы соответствующий участок кода был перекомпилирован в отладочный вариант байт-кода.
- <Ctrl>+<Shift>+<F7> — **New Watch...** (Новое окно). По умолчанию данная кнопка на панели инструментов отсутствует, но ее можно добавить перетаскиванием из формы, открываемой пунктом меню **View/Toolbar/Customize...**. При вызове предлагается ввести выражение (рис. 17.10). В данный пункт можно вводить значения выражений, зависящих от локальных и глобальных переменных.

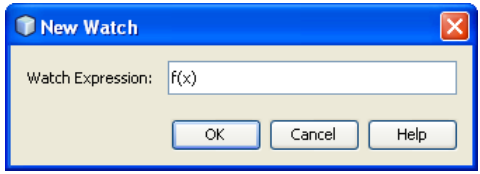


Рис. 17.10. Задание выражения для просмотра в окне **Watches**

- Более удобный вариант — добавление выражения в окно **Watches** непосредственно из редактора исходного кода. Для этого, находясь в режиме отладки, следует выделить необходимый фрагмент кода, вызвать правой кнопкой мыши контекстное меню и выбрать в нем пункт (рис. 17.11).

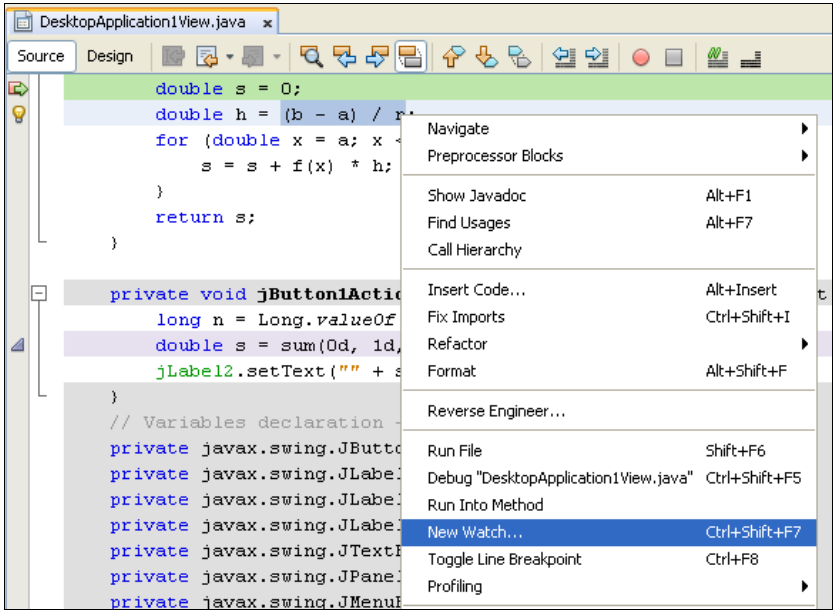


Рис. 17.11. Задание выражения для просмотра в окне **Watches**

- После ввода просмотр результатов доступен в окне **Watches** (Окна просмотра) в правой нижней части экрана (рис. 17.12).

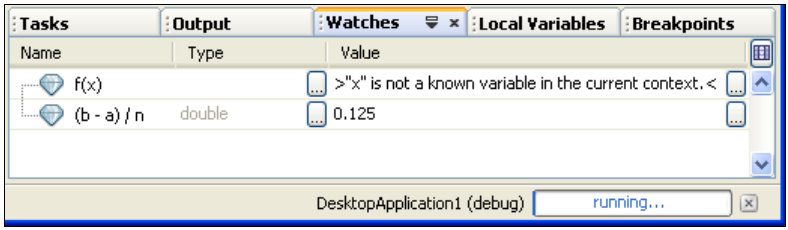


Рис. 17.12. Окно **Watches**

- Двойной щелчок по ячейке с выражением открывает форму, в которой это выражение можно изменить (аналогична форме, показанной на рис. 17.10). Также можно "на лету" вручную поменять значение любой переменной (при отладке это иногда бывает необходимо для того, чтобы понять, как себя ведет отлаживаемый фрагмент программы).

На вкладке **Local Variables** (Локальные переменные) (рис. 17.13) имеется возможность просмотреть значения всех локальных переменных метода, так что нет необходимости искать их имена в исходном коде и наводить на каждую из них курсор мыши, пытаясь запомнить значения. При необходимости можно изменить значение переменной непосредственно в той области, где оно показывается.

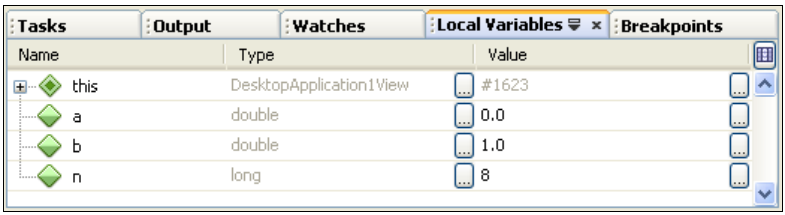


Рис. 17.13. Вкладка **Local Variables**

Объектная переменная (в нашем случае это переменная `this` — ссылка на текущий объект) показывается в виде свернутого дерева. Разворачивание дерева позволяет просмотреть значения полей данного объекта на произвольном уровне вложенности объектов (рис. 17.14).

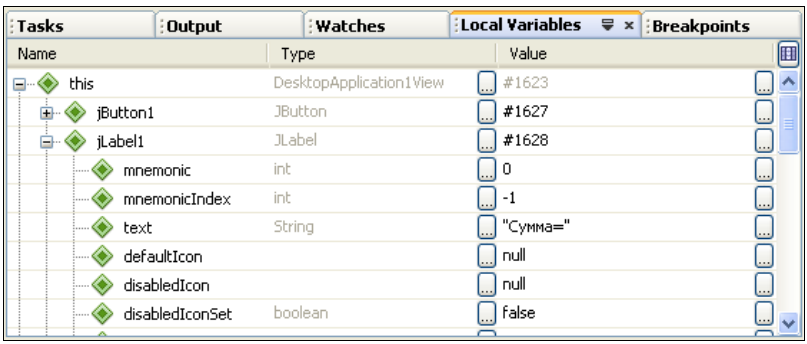


Рис. 17.14. Вкладка **Local Variables**, просмотр вложенных объектов

На вкладке **Call Stack** (Вызовы стека) (рис. 17.15) находится список имен всех вызовов, которые были сделаны за время выполнения приложения. После двоеточия указан номер той строки исходного кода, которой соответствует вызов. Двойной щелчок по имени метода позволяет перейти в редактор исходного кода в то место, где происходит вызов этого метода.

По умолчанию данная вкладка не показывается, но ее можно открыть с помощью пункта меню **NetBeans Window/Debugging/Call Stack** (рис. 17.16).

Эту вкладку в немного усложненном варианте дублирует вкладка **Debugging**, открывающаяся в том же окне, что и часто использовавшаяся нами вкладка **Projects**

(рис. 17.17). Поэтому вместо вкладки **Call Stack** можно работать с вкладкой **Debugging**, показывающей потоки выполнения и вызовы в каждом потоке.

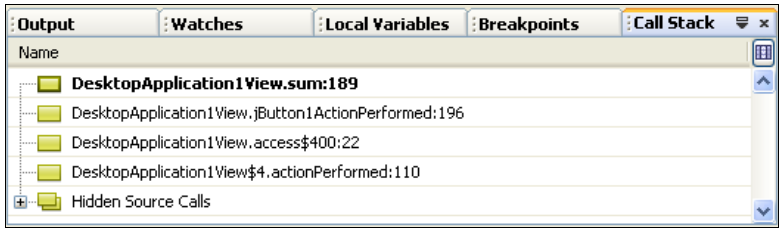


Рис. 17.15. Вкладка **Call Stack**

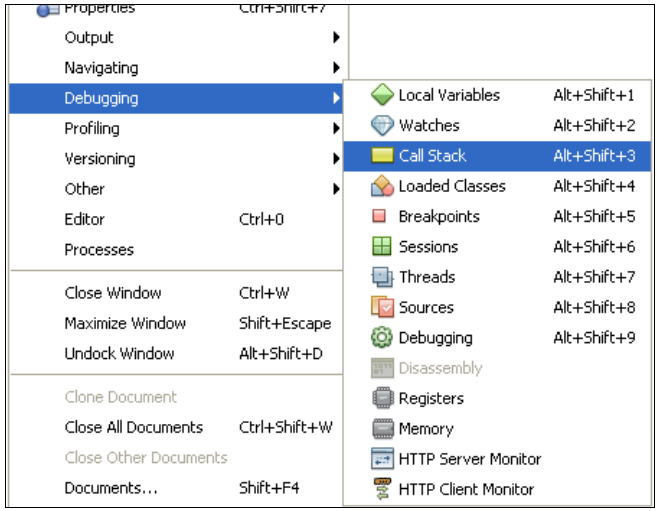


Рис. 17.16. Открытие вкладки **Call Stack**

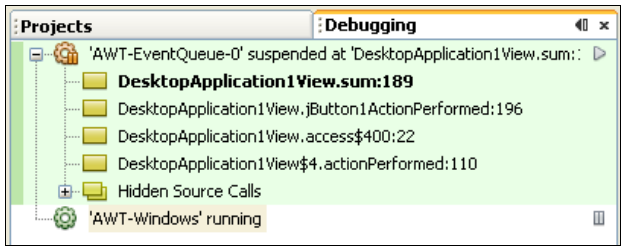


Рис. 17.17. Вкладка **Debugging**

На вкладке **Breakpoints** (рис. 17.18) видны все точки остановки программы. При двойном щелчке по имени точки в редакторе исходного кода происходит переход в соответствующее место проекта. Снятие флажка приводит к выключению соответствующей точки остановки, но без исключения ее из данного списка, чтобы оставалась возможность опять включить данную точку. Во многих случаях удобнее отключать и заново

подключать имеющиеся точки остановки не в редакторе исходного кода, а на вкладке **Breakpoints** — тогда нет необходимости переходить в разные места проекта, управление проверкой оказывается более оперативным. В редакторе исходного кода точку остановки отключают щелчком правой кнопкой мыши по значку точки остановки. В появившемся контекстном меню снимают флажок **Breakpoint/Enabled**.

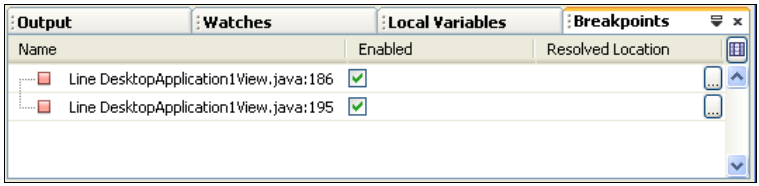


Рис. 17.18. Вкладка **Breakpoints**

В правом верхнем углу вкладок отладки (**Watches**, **Local Variables**, **Breakpoints** и др.) находится иконка с изображением таблицы. Щелчок по этой иконке открывает форму настроек для соответствующей вкладки. Вид формы для вкладки **Watches** показан на рис. 17.19.

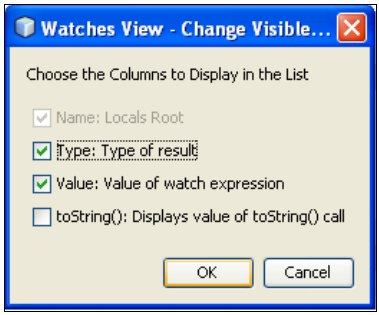


Рис. 17.19. Настройки вкладки **Watches**

Во время отладки приложения в окне **Output** (Вывод), кроме основного окна вывода, появляется вкладка **Debugger Console** (Консоль отладчика). В нее выводится служебная информация отладчика об установке и снятии точек остановки, остановке программы в этих точках, командах программиста, подаваемых отладчику. Пример такой информации приведен в листинге 17.1.

Листинг 17.1. Пример информации, выводимой в консоли отладчика

```
Listening on 1343
User program running
LineBreakpoint DesktopApplication1View.java : 202 successfully submitted.
Not able to submit breakpoint LineBreakpoint DesktopApplication1View.java : 191, reason:
No executable location available at line 191.
Invalid LineBreakpoint DesktopApplication1View.java : 191
Breakpoint hit at line 202 in class desktopapplication1.DesktopApplication1View by thread
AWT-EventQueue-0.
Thread AWT-EventQueue-0 stopped at DesktopApplication1View.java:202.
User program running
```

```
Thread AWT-EventQueue-0 stopped at DesktopApplication1View.java:207.
User program running
User program finished
```

Такая диагностика позволяет при необходимости восстановить последовательность действий, которые программист выполнял при отладке. Очистить выходное окно консоли отладчика можно так же, как и выходное окно основной консоли — вызовом пункта **Clear** (Очистить) контекстного меню для этого окна.

Если какие-то окна отладчика были закрыты или необходимо открыть дополнительные окна, то следует воспользоваться пунктом меню **Window/Debugging**.

Мы познакомились с основными возможностями отладчика. Отметим некоторые особенности режима отладки.

Считается, что при отладке приложение работает медленнее, чем в нормальном режиме. Но при проверке скорости вычислений с помощью примеров 05_8_effectiveCalc для вычислений в формате с плавающей точкой и 05_9_effectiveCalc это практически незаметно — отличие лежит в пределах погрешности измерения времени. Таким образом, можно сделать вывод, что даже на самых критичных к скорости участках кода отладчик NetBeans 6.5, работающий с исполняющей средой JDK 6, практически не влияет на скорость выполнения приложения, хотя перед запуском в режиме отладки приложение перекомпилируется для использования отладчика, что занимает заметное время (да и сам отладчик стартует не мгновенно).

Обсудим теперь влияние упомянутой ранее особенности отладчиков ставить точку остановки в начале строки. Именно из-за нее выработались многие правила форматирования исходного кода программы.

Пусть, например, у нас имеется условный оператор:

```
if (a>b)
    оператор1;
else
    оператор2;
```

При данном форматировании точку остановки легко поставить как на начало оператора (чтобы проконтролировать значения `a` и `b`), так и на любой из операторов, выполняющихся в альтернативных ветвях (на `оператор1` или `оператор2`). Причем все эти точки остановки независимы, что очень важно. Допустим, нас интересует только ветвь `else`, которая выполняется один раз из миллиона, а в остальных случаях выполняется ветвь `if`. Если у нас нет возможности поставить точку остановки на `else`, то совсем нереально проверить, что будет происходить в этой ветви.

Поэтому вариант:

```
if (a>b)оператор1; else оператор2;
```

хотя и синтаксически корректен, но совершенно неприемлем для отладки этого кода.

По той же причине недопустим вариант:

```
if (a>b)оператор1;
else оператор2;
```

17.3. Пример отладки неправильно работающего приложения

Воспользуемся полученными знаниями, чтобы выяснить, что же происходит в нашем приложении во время вычислений.

Установим две точки остановки: в первой строке метода обработчика, в котором осуществляется вызов функции `sum`, и в первой строке тела этой функции. Запустим приложение `17_1_DesktopApplication` в режиме отладки, введем `n=10` и нажмем кнопку **Вычислить**.

Произойдет остановка перед выполнением строки (рис. 17.20).

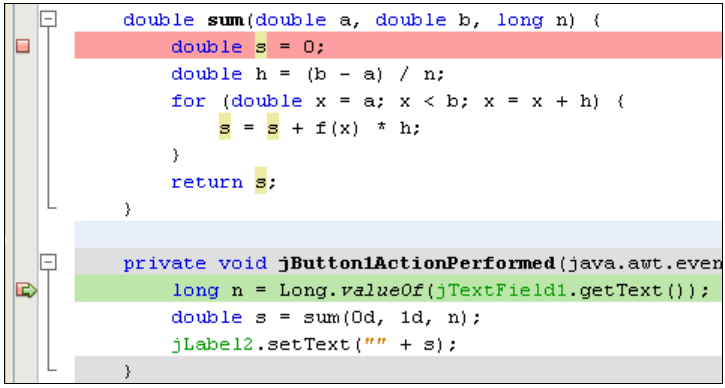


Рис. 17.20. Начало отладки

Сделаем шаг, нажав клавишу <F8>, и убедимся, что в переменную `n` занесено значение `10`. Войдем в функцию `sum` и проверим значения `a`, `b` и `n` — все в порядке.

Делаем шаг — `s` инициализируется нулем.

Еще шаг — вычисляется значение `h`, оно оказывается равно 0,1. Все правильно.

Еще шаг — и мы начинаем цикл.

Смотрим значения локальных переменных: вещественный счетчик цикла `x` инициализирован значением 0, `a=0.0`, `b=1.0`.

Шаг — и кончается первая итерация цикла. Результат: `s=0`, `x=0`.

Шаг — начинается вторая итерация (`x=0.1`).

В окно **Watches** добавляем функцию `f(x)*h` для того, чтобы можно было контролировать вычисление `s` на каждой итерации. Обнаруживаем, что ее значение равно `0.001`.

После добавления `s=0.0010000000000000002` — все правильно, вычисления в формате с плавающей точкой имеют погрешность.

Продолжаем суммирование в каждой следующей итерации. И уже после третьей итерации обнаруживаем, что `x=0.30000000000000004`.

Проверять все итерации цикла утомительно — у нас их 10 штук, но могло бы быть и 10 000. Как правило, достаточно проделать несколько первых и несколько последних

итераций циклов, в которых обнаружена ошибка. В подавляющем большинстве случаев ошибка связана с отличием на единицу от правильного числа итераций цикла или конечного значения индекса массива. Массивов у нас нет, так что следует проверить число итераций.

Мы пока не знаем, как воспользоваться средствами остановки после выполнения заданного числа итераций цикла. Так что попробуем остановиться после выполнения всех итераций на строке `return s;`, возвращающей значение функции. Поставим на эту строку курсор и выполним команду **Run to Cursor** (Перейти к курсору).

Мы обнаружим, что проконтролировать значение счетчика цикла `x` нам не удается — мы уже покинули цикл и локальная переменная `x`, заданная в цикле, перестала быть доступной. В учебниках по программированию часто рекомендуют использовать счетчик цикла `for`, заданный именно так, как у нас было сделано — внутри самого цикла. Но это порождает наблюдавшуюся нами проблему слежения за значением счетчика после окончания цикла. Поэтому пока в отладчиках не появится возможность отслеживать значения счетчиков (их может быть несколько разных!) после завершения цикла, лучше задавать локальные переменные счетчиков непосредственно перед циклом `for`, а инициализировать их лучше в секции инициализации цикла (если они имеют один тип).

Нам придется завершить сеанс отладки и соответствующим образом модифицировать код программы.

Заменяем

```
for (double x = a; x < b; x = x + h) {
```

на

```
double x;
for (x = a; x < b; x = x + h) {
```

и заново запускаем отладчик.

В пользовательском интерфейсе работающего приложения 17_1_DesktopApplication (см. рис. 17.1) вводим `n=10`, нажимаем кнопку **Вычислить**. Переходим на строку с возвратом значения функции `sum` и обнаруживаем, что `x=1.0999999999999999`. Это крайне подозрительно!

Нажимаем клавишу <F5> и переходим в приложение. Попробуем проверить результаты для значений `n`, ближайших к 10 (т. е. при `n = 9` и при `n = 11`).

Вводим `n=9`, нажимаем кнопку **Вычислить**. Переходим на строку с возвратом значения функции `sum` и обнаруживаем, что `x=1.0000000000000002`. Все в норме.

Нажимаем клавишу <F5> и переходим в приложение.

Вводим `n=11`, нажимаем кнопку **Вычислить**. Переходим на строку с возвратом значения функции `sum` и обнаруживаем, что `x=1.0000000000000002`. Все в норме.

Таким образом обнаружено, что при `n=10` последнее значение счетчика цикла `x` почему-то ошибочно — оно сильно отличается от `1.0`, далеко выходя за пределы погрешности чисел типа `double`. Если мы не догадались, что `x=b+h`, т. е. что сделана одна лишняя итерация, или не поняли причины, из-за которой это произошло, нам придется исследовать 2—3 последних итераций цикла. Обычно этого достаточно для того, чтобы понять возникающие проблемы.

Мы пока не знаем, как в отладчике пропустить ненужные итерации и наблюдать только последние итерации цикла.

Можно написать отдельную подпрограмму (пример 17_3_DesktopApplication) (листинг 17.2).

Листинг 17.2. Вспомогательный метод для отладки алгоритма

```
public void calc100(ActionEvent e) {
   int i;
   for(i=1;i<=100;i++){
    double s = sum(0d, 1d, i);
    System.out.println("n="+i+" s="+s);
   }
}
```

Либо написать отладочный фрагмент кода с выводом необходимой информации в окно консоли (потом мы его закомментируем).

Листинг 17.3 иллюстрирует цикл `for` в этом случае.

Листинг 17.3. Отладочный фрагмент кода с выводом информации в консоль

```
for (x = a; x < b; x = x + h) {
    s = s + f(x) * h;
    if(x>=b-2*h){
        System.out.println("n="+n);
        System.out.println("a="+b);
        System.out.println("b="+b);
        System.out.println("h="+h);
        System.out.println("x="+x);
        System.out.println("f(x)="+f(x));
        System.out.println("s="+s);
        System.out.println();
    }
 }
```

При запуске программы мы получаем в окне консоли следующую отладочную информацию (листинг 17.4).

Листинг 17.4. Отладочная информация

при n=9:

```
n=9
a=1.0
b=1.0
h=0.1111111111111111
x=0.7777777777777779
f(x)=0.6049382716049385
s=0.19204389574759947
```

```
n=9
a=1.0
b=1.0
h=0.1111111111111111
x=0.8888888888888891
f(x)=0.7901234567901237
s=0.2798353909465021
```

при n=10:

```
n=10
a=1.0
b=1.0
h=0.1
x=0.8999999999999999
f(x)=0.8099999999999998
s=0.2849999999999999

n=10
a=1.0
b=1.0
h=0.1
x=0.9999999999999999
f(x)=0.9999999999999998
s=0.3849999999999999
```

при n=11:

```
n=11
a=1.0
b=1.0
h=0.09090909090909091
x=0.8181818181818183
f(x)=0.669421487603306
s=0.21412471825694973

n=11
a=1.0
b=1.0
h=0.09090909090909091
x=0.9090909090909093
f(x)=0.8264462809917359
s=0.28925619834710753
```

И наконец, мы обнаруживаем причину ошибки — при `n=10` последнее значение счетчика цикла `x=0.9999999999999999`. Это крайне подозрительно! Ведь в тех случаях, когда "все нормально", последнее значение `x` отстоит от `b` примерно на `h`.

У нас проверяется условие `x < b`.

А ведь при `x=0.9999999999999999` и `b=1.0` получается, что проверяется, будет ли выполняться неравенство `0.9999999999999999<1.0`.

Ой, а что же мы сравниваем? Ведь выполнение этого неравенства — дело случая. С таким же успехом могла возникнуть ситуация с проверкой, будет ли выполняться неравенство `1.0000000000000001 < 1.0` (и такая ситуация действительно возникает).

Наконец, рассмотрим дополнительные возможности отладки, позволяющие останавливать выполнение приложения в точке остановки не каждый раз, а при выполнении некоторого условия. Если щелкнуть правой кнопкой мыши по квадрату, помечающему в исходном коде точку остановки, и в появившемся контекстном меню выбрать пункт **Breakpoint/Properties** (свойства точки остановки), то появится диалог настройки условий остановки, в котором можно задать:

- пункт **Condition** (Условие) — остановка при выполнении условия, заданного с помощью булевого выражения. В выражении разрешены любые конструкции Java, синтаксически правильные в области точки остановки, но для всех элементов из других пакетов требуется указание полного имени с указанием имени пакета;
- пункт **Break when hit count** — остановка по достижении заданного числа вызовов по условию **Condition**.

После задания условий внешний вид символа, помечающего точку остановки, изменится с квадрата на уголок.

Предоставляем читателю изучить данные возможности отладки самостоятельно.

17.4. Принцип "презумпции виновности" и тестирование классов

Мы выяснили, в чем дело, и теперь удивляемся, почему программа вообще так редко выдавала неправильные результаты, ведь по идее она должна была бы это делать в каждом втором случае. Иногда такие ошибки обнаруживаются через несколько лет работы. И остается только удивляться, почему их не заметили раньше, и как наши классы вообще кто-нибудь мог использовать.

В программировании действует принцип "презумпции виновности": если ты не можешь доказать, что предъявляемая тобой программа правильна, считается, что она содержит ошибки. И даже если она проверена в миллионе разных ситуаций, это вовсе не означает, что в миллион первом случае программа не начнет работать неверно, т. е. не обнаружится баг (ошибка).

Точно так же обстоит дело и с результатами научных экспериментов — пока независимые группы ученых не проверят и не подтвердят их, эти результаты считаются недостоверными (даже если их получил нобелевский лауреат). И так же обстоит дело с теориями — пока экспериментально не проверены все возможные следствия этих теорий, они находятся в ранге "подозреваемых". Например, если специальная теория относительности (СТО) Эйнштейна доказана (в границах своей применимости) и в научном сообществе не вызывает никаких сомнений, то разработанная тем же Эйнштейном общая теория относительности (ОТО) существует в ранге гипотезы, наряду с альтернативными моделями, авторы которых не так знамениты. Но в данной области может оказаться правым именно кто-нибудь малоизвестный.

Поэтому необходимо усвоить следующий подход: не доверяй словам, не подтвержденным доказательствами (как бы ни был высок авторитет того, кто эти слова произносит).

И не верь в правильность программного кода, как бы ни была высока квалификация того, кто его писал. Человек может ошибаться, и не следует исключать возможности ошибок даже в работающих и отлаженных программах. Просто, возможно, еще не возникла та ситуация, когда данная ошибка проявится.

Поэтому в современном программировании возникло целое направление — тестирование программных продуктов как некий стандарт качества.

Для такой деятельности в NetBeans имеется специальная система сопровождения проекта тестовыми библиотеками (Test Libraries), с помощью которых проверяют работу разработанных классов в самых разных условиях. Например, мы могли бы проверить результаты интегрирования на примере известных функций, задав различные наборы начальных данных (`a`, `b`, `n`, `f(x)`).

Конечно, и тут действует принцип презумпции виновности. Поскольку корректность полученных результатов проверена для конечного числа входных параметров, то гарантировать правильность работы наших классов можно только для этого набора значений (и то, если при формулировке условий проверки не было упущено какое-нибудь важное звено).

Тем не менее, протестированные проекты представляются более надежными, поскольку в них выполнена проверка выполнения самых важных требований к нашим классам. Хотя основная проблема в написании тестов — это вопрос: "а когда следует остановиться?". Чем больше разных тестов, тем выше надежность, но тем больше затраты на их написание. Если объем кода тестов превышает объем проверяемого кода, то затраты на тестирование становятся обузой как по времени, так и по деньгам. Да к тому же возникает проблема наличия ошибок в самих тестах. Так что где-то все равно придется остановиться.

Во многих фирмах, разрабатывающих программное обеспечение, требующее особой надежности, принято помимо рассматриваемой в данном разделе системы тестирования с помощью тестовых пакетов выполнять тотальный разбор программного кода — анализ независимыми специалистами каждой строчки кода. И при таком разборе достаточно часто обнаруживаются ошибки, незамеченные ранее. Хотя, конечно, это очень дорогостоящая и трудоемкая деятельность. А выдача предупреждений средой разработки при проведении тестирования позволяет вовремя обнаруживать большую часть ошибок, возникающих при модификации проекта, и повысить надежность кода до уровня, достаточного для большинства случаев.

Рассмотрим создание тестовых пакетов в NetBeans на примере простейшей программы (рис. 17.21), проект 17_4_tests_1 с прилагаемого к книге DVD.

Зададим в пакете `pkg` класс `Location` ("Местоположение"), инкапсулирующий задание координаты точки (листинг 17.5) и использующий его главный класс приложения (листинг 17.6).

Листинг 17.5. Класс `Location`

```
public class Location {

    private int x = 0;
    private int y = 0;
```

```
    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
     * @return the x
     */
    public int getX() {
        return x;
    }

    /**
     * @param x the x to set
     */
    public void setX(int x) {
        this.x = x;
    }

    /**
     * @return the y
     */
    public int getY() {
        return y;
    }

    /**
     * @param y the y to set
     */
    public void setY(int y) {
        this.y = y;
    }
}
```

Листинг 17.6. Класс `Main`

```
public class Main {

    public static void main(String[] args) {
        int x = (int) Math.round(Math.random() * 100);
        int y = (int) Math.round(Math.random() * 100);
        Location location = new Location(x, y);
        System.out.println("x=" + x + "  y=" + y);
    }
}
```

На рис. 17.21 видно, что в проекте NetBeans 6.5 существует раздел `Test Packages`, в котором имеется безымянный пустой пакет по умолчанию — он обозначен как `<default package>`, а также раздел `Test Libraries`, в котором находятся две тестовые библиотеки, `JUnit 3.8.2` и `JUnit 4.5`. Именно эти библиотеки обеспечивают среду NetBeans общими

возможностями автоматизированного тестирования классов. А пакет `Test Packages` используется для хранения классов, созданных для проверки данного конкретного приложения.

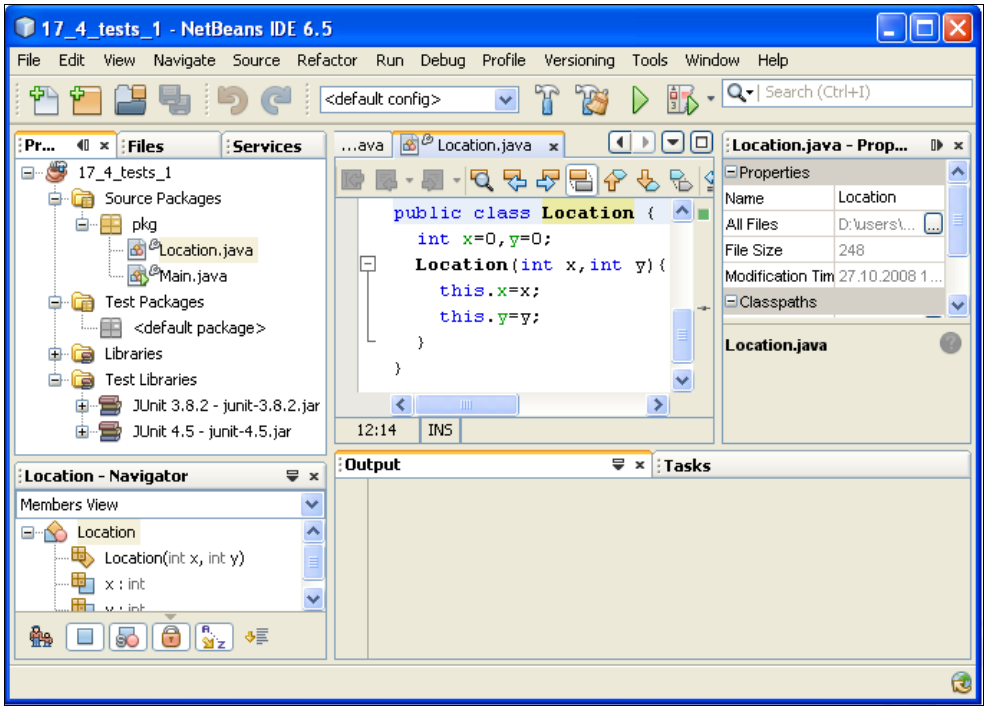


Рис. 17.21. Приложение, в котором будет создаваться тестовый пакет

Выбор пункта меню **Tools/Create JUnit Tests** приводит к появлению первой формы с двумя условиями генерации теста (рис. 17.22): предлагается выбрать либо современную версию тестов Java (JUnit 4.*x*), ориентированную на JDK 5 и JDK 6, либо устаревшую (JUnit 3.*x*, реально это JUnit 3.8.2), обеспечивающую совместимость с более ранними версиями JDK.

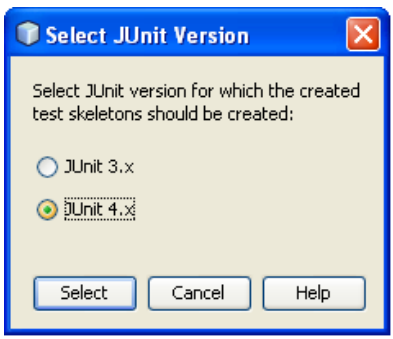


Рис. 17.22. Создание тестового пакета, шаг 1

Естественно, мы выбираем вариант по умолчанию — JUnit 4.*x* (реально это JUnit 4.5, самая новая на начало 2009 года версия).

Замечание

Технология JUnit не является достижением среды NetBeans, она развивается сообществом JUnit (сайт **http://JUnit.org**) и поддерживается всеми развитыми средами для Java-программирования. Тем не менее, в среде NetBeans 6.5 работа с тестами JUnit реализована очень удобно и качественно — она поддерживается как обязательная часть каждого проекта NetBeans.

После нажатия на кнопку **Select** (Выбрать) на форме, показанной на рис. 17.22, появляется следующая форма выбора условий генерации теста (рис. 17.23).

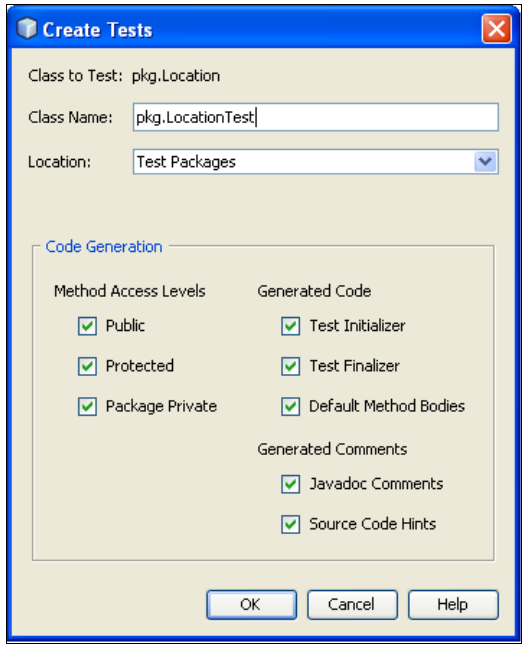


Рис. 17.23. Создание тестового пакета, шаг 2

Первый пункт (**Class Name**) позволяет при необходимости изменить автоматически генерируемое имя класса теста. Второй (**Location**) дает возможность выбрать пакет, в котором будет расположен тестовый класс приложения. Он нужен для сложных приложений, имеющих несколько тестовых пакетов. Остальные пункты относятся к автоматически генерируемому коду. Мы оставим все галочки отмеченными, не меняя установок по умолчанию. После нажатия **OK** получим заготовку тестового пакета (рис. 17.24).

Сгенерированный на основе проекта 17_4_tests_1 исходный код представлен в проекте 17_4_tests_2 (листинг 17.7).

Листинг 17.7. Заготовка теста, сгенерированная для класса `Location`

```
package pkg;

import org.junit.After;
import org.junit.AfterClass;
```

```
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class LocationTest {

    public LocationTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

}
```

В старых версиях JUnit требовалось, чтобы класс теста был наследником класса `TestCase` библиотеки JUnit, и чтобы имя любого метода в тесте начиналось с префикса `test`. В JUnit 4 вместо этого используются аннотации. Для того чтобы метод из класса в тестовом пакете стал тестом, достаточно пометить его аннотацией `@Test`.

В классе `org.junit.Assert` предусмотрены методы для различных тестовых проверок:

- `assertEquals(int1,int2)` — на равенство двух целых значений любых целых типов;
- `assertEquals(double1, double2,delta)` — на равенство двух вещественных значений, с отличием между ними не более `delta`;
- `assertEquals(obj1, obj2)` — на равенство двух объектов (методом `equals`, если первая ссылка не `null`; объекты считаются равными, если обе ссылки `null`);
- `assertArrayEquals(arr1,arr2)` — на равенство двух массивов;
- `assertFalse(condition)` — на то, что условие всегда `false`;
- `assertTrue(condition)` — на то, что условие всегда `true`;
- `assertNull(obj)` — на то, что объект всегда `null`;
- `assertNotNull(obj)` — на то, что объект всегда не `null`;
- `assertSame(obj1, obj2)` — на то, что оба объекта всегда идентичны.

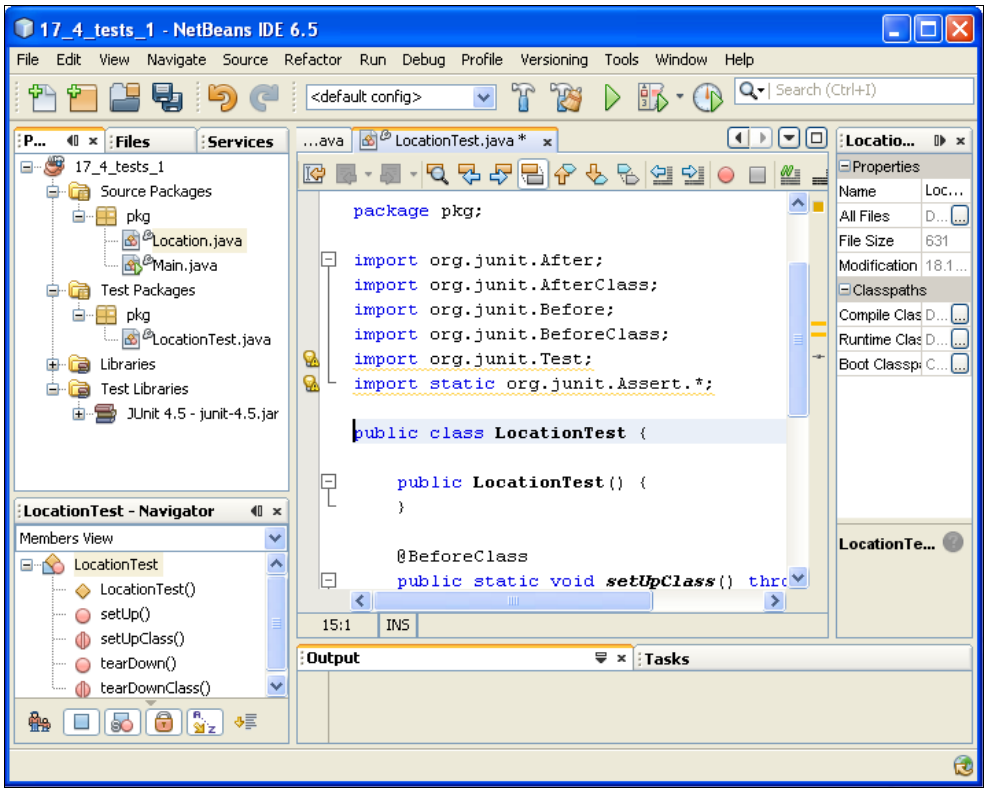


Рис. 17.24. Создание тестового пакета, полученная заготовка

Слово assert означает "тестировать", "проверять". Если условие не выполняется, тестовый метод возбуждает исключительную ситуацию, воспринимаемую как неудача при прохождении теста (test failed).

Например, метод `assertTrue` проверяет значение булевого выражения, стоящего в качестве второго параметра. Название метода может быть переведено как "Проверять на то, что условие равно true". Если условие не выполняется (т. е. равно false), возбуждается исключительная ситуация, сигнализирующая, что тест не пройден.

Помимо указанных методов имеются совершенно аналогичные им, но с дополнительным первым параметром — строковым сообщением, выдающимся при возникновении ошибки.

Также существует еще один полезный метод:

- `fail()` — возбуждает исключительную ситуацию, сигнализирующую, что тест не пройден. Обычно такой вызов делается при сложных проверках для повышения читаемости тестового кода.

Разработаем простейший тест (проект 17_4_tests_3) на основе полученной ранее заготовки (проект 17_4_tests_2), используя известные нам теперь возможности проверки. Допустим, объекты нашего класса правильно функционируют только при значении координат x и y, лежащих в пределах от 1 до 100, и мы хотим проверить, что это выполняется при всех вариантах запуска приложения методом `main`.

Создадим тест для такой проверки — листинг 17.8.

Листинг 17.8. Метод `test1()` теста для класса `Location`

```
@Test
public void test1() throws Exception {
    int x = (int) (Math.random() * 100);
    int y = (int) (Math.random() * 100);
    Location location = new Location(x, y);
    assertTrue("Ошибка, location.x==" + location.getX(),
               0 < location.getX() && location.getX() < 100);
    assertTrue("Ошибка, location.y==" + location.getY(),
               0 < location.getY() && location.getY() < 100);
    }
```

Запустим тест на выполнение через меню **Run/Test Project**, с помощью комбинации клавиш <Alt>+<F6> или щелкнув правой кнопки мыши по имени проекта в окне **Projects** и выбрав пункт контекстного меню **Test** (рис. 17.25).

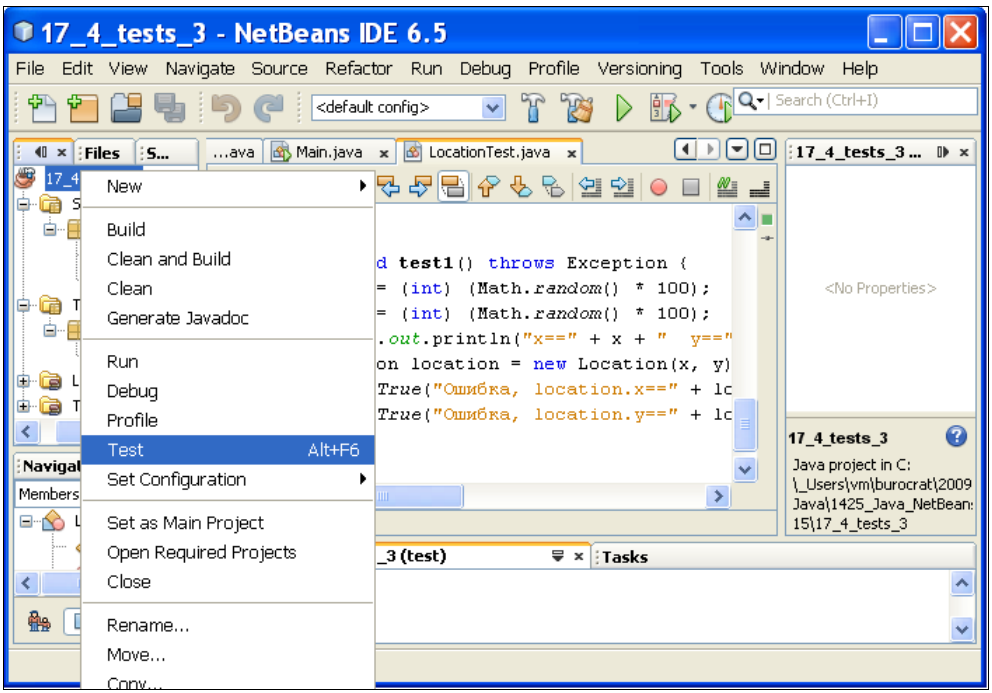


Рис. 17.25. Запуск теста на выполнение

Если тест прошел успешно, то в нижнем окне появляется соответствующее сообщение, написанное зеленым цветом (рис. 17.26).

Если же тест не прошел, выдается сообщение о неудаче (выделенное красным цветом). Заметим, что в JUnit 3 неудачи и ошибки разделялись и диагностировались по-разному. В JUnit 4 этот подход был заменен более логичным, когда тест либо проходит, завершаясь успехом (passed), либо не проходит — "проваливается" (failed).

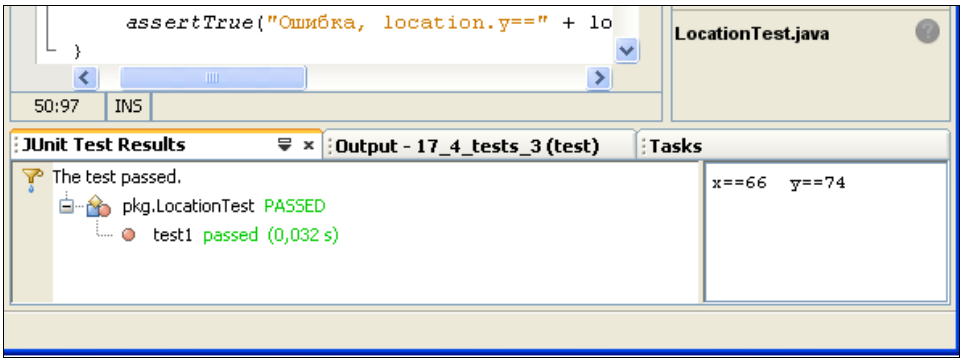


Рис. 17.26. Успешное прохождение теста

В нашем случае неудача возможна, но маловероятна (примерно один шанс из 50 из-за генерации значений `x==0` или `y==0` с помощью метода `random`). А вот если мы немного модифицируем наш тест, убрав аннотацию у метода `test1()` и написав циклический вызов этого метода в тестовом методе `test2` (листинг 17.9), вероятность получить диагностику неудачи станет достаточно большой (рис. 17.27).

Листинг 17.9. Метод `test2()` теста для класса `Location`

```
@Test
public void test2() throws Exception {
    for (int i = 0; i < 1000; i++) {
        test1();
    }
}
```

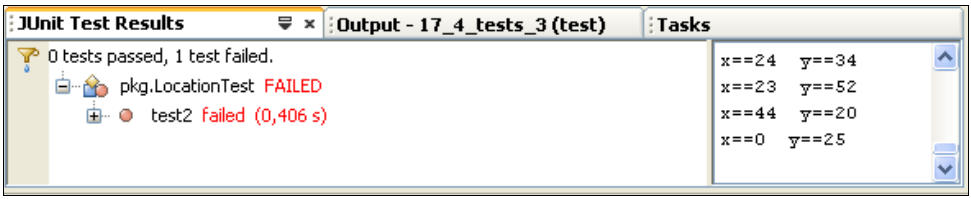


Рис. 17.27. Неудачное прохождение теста

Если в окне **JUnit Test Results** (рис. 17.27) развернуть узел дерева, относящегося к методу `test2`, мы увидим всю "подноготную" неудачи (рис. 17.28).

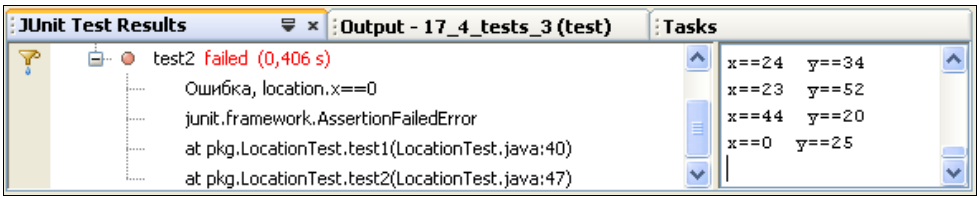


Рис. 17.28. Неудачное прохождение теста, развернутая диагностика ошибки

Обычно для тестирования используют не один, а несколько различных тестов. Для этого пишется столько методов, аннотированных как `@Test`, сколько нужно тестов. Тесты запускают по очереди в том порядке, в котором они заданы в тестовых пакетах и классах.

Модифицируем код класса `Location`, введя возбуждение исключительной ситуации при выходе за допустимые границы для методов `setX` и `SetY` (листинг 17.10).

Листинг 17.10. Модификация класса `Location`

```
public void setX(int x) throws IndexOutOfBoundsException {
    if (0 < x && x < 100) {
        this.x = x;
    } else {
        throw new IndexOutOfBoundsException("location.x==" + x);
    }
}
public void setY(int y) throws IndexOutOfBoundsException {
    if (0 < y && y < 100) {
        this.y = y;
    } else {
        throw new IndexOutOfBoundsException("location.y==" + y);
    }
}
```

Затем добавим в класс новый тест (т. е. тестовый метод, помеченный аннотацией `@Test`), показанный в листинге 17.11 (проект 17_4_tests_4).

Листинг 17.11. Добавление второго теста для класса `Location`

```
    @Test
    public void test3() throws Exception {
    String[] sa = new String[0];
    for (int i = 0; i < 100; i++) {
    Main.main(sa);
    }
    }
```

После запуска мы, вероятнее всего, обнаружим, что первый тест окончился неудачей (failed), а второй прошел удачно (passed) — рис. 17.29.

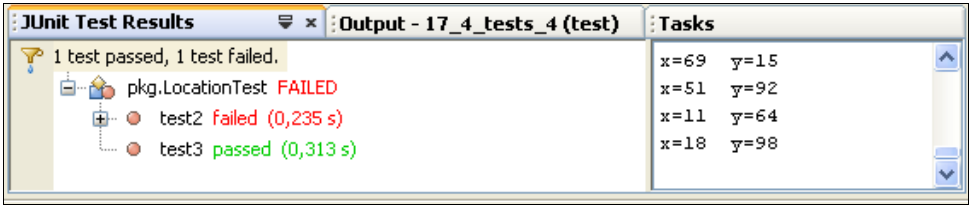


Рис. 17.29. Прохождение двух тестов — один из них не прошел

Второй тест прошел удачно потому, что наш класс устроен неверно — не проверяется допустимость начальных значений, передаваемых в конструктор при создании экземпляра, из-за чего нам приходится проверять значения полей объекта `location` в методе `test1()`.

Заменим непосредственный доступ к полям на сеттеры (листинг 17.12).

Листинг 17.12. Модификация конструктора класса `Location`

```
public Location(int x, int y) {
//  this.x=x;
    setX(x);
//  this.y=y;
    setY(y);
}
```

Такая замена обязательна — иначе нам придется дублировать код, вводя аналогичную проверку в конструкторе. Теперь в методе `test1()` нет необходимости проверять значения `location.x` и `location.y` с помощью операторов `assertTrue`, закомментируем их (проект 17_4_tests_5).

Заметим, что лучше всегда с самого начала использовать сеттеры, а не присваивания значений полям, т. к. проверки, похожие на сделанные нами, часто возникают в процессе усовершенствования классов. По той же причине лучше при инкапсуляции полей данных с помощью рефакторинга всегда осуществлять доступ к полям с помощью сеттеров даже в случаях, когда поле находится в непосредственной видимости (как у нас this.x и this.y). Единственное исключение из этого правила — ситуация, когда от метода требуется максимальное быстродействие, и работа ведется с целыми величинами. В этом случае время вызова сеттера может уменьшить быстродействие. Но такое редко бывает на практике.

Напротив, для операций чтения обычно безразлично, пользоваться геттерами или непосредственно обращаться к полям данных, поскольку в подавляющем большинстве случаев нормой является ситуация, когда объект имеет внутренне согласованные допустимые значения полей данных. А все проверки значений — внешние по отношению к объекту.

Запустив тест, мы убедимся, что он не проходит, но уже с новой диагностикой ошибки, связанной с возбуждением исключительной ситуации в сеттере (рис. 17.30).

Когда внутри тестового метода возбуждается исключительная ситуация, тест считается неудавшимся (failed). Но это не относится к случаям, когда необходимо тестирование программы при возникновении исключений. А для проверки работы нашей программы может быть необходима именно такая ситуация. Разрешить в тесте исключения можно, добавив в аннотацию `@Test` параметр `expected`:

```
@Test(expected=IndexOutOfBoundsException.class)
```

Если мы проверим пример 17_4_tests_6, который отличается от примера 17_4_tests_5 только таким добавлением параметров аннотации для метода `test3()`, то обнаружим, что тест проходит нормально (рис. 17.31).

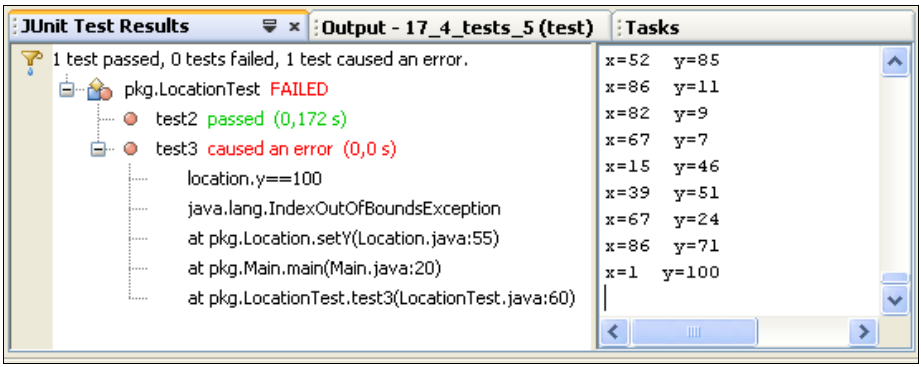


Рис. 17.30. Прохождение двух тестов — в одном ошибка выполнения

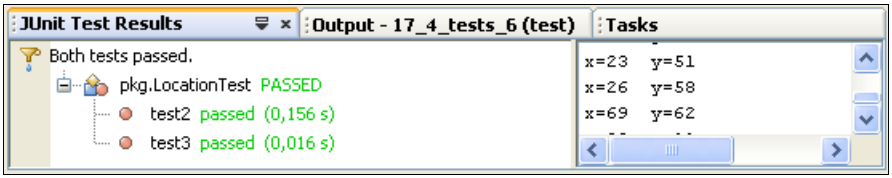


Рис. 17.31. Прохождение тестов при возникновении ожидаемых исключений

Проанализируем сделанные проверки. Они пока чисто демонстрационные. В нашем случае прохождение теста не гарантирует правильности работы класса `Location`, хотя иногда и позволяет обнаруживать ошибки. Конечно, тестирование в общем случае не может дать полной гарантии правильности. Тем не менее тесты очень полезны для проверки разработанных классов в нескольких характерных ситуациях, и мы пока эти ситуации не учли — перед тем, как выдавать разработанные классы пользователю, программист обязан испытать их как минимум в трех режимах:

- Простейшие пробные наборы данных, для которых очевидно поведение разработанной программной системы. Тесты должны отслеживать это известное поведение. Например, если написана программа, вычисляющая интеграл с помощью заданного численного метода, вычисление интеграла должно проверяться для пробного набора функций (для `f(x)=1`, далее для `f(x)=x`, затем для $f(x)=x^2$ и т. п.).
- Типичные ситуации, для которых поведение программной системы должно быть нормальным. В нашем случае это должны быть вычисления интегралов от функций вида `f(x)=sin(x)`, `f(x)=a1*exp(b1*x)` и т. п. от `a` до `b`, где пределы интегрирования `a` и `b` отличаются не очень сильно.
- Экстремальные ситуации, для которых поведение программной системы может быть ненормальным. В нашем случае это могут быть вычисления интегралов от функций вида `f(x)=a1*exp(b1*x)`, `f(x)=sqrt(x)` и т. п. Причем значения параметров `a1` и `b1`, а также пределы интегрирования `a` и `b` в этом тесте могут отличаться очень сильно. Обязательно захватывать границы разрешенных диапазонов, а также точку `x=0`, отрицательные значения тех параметров, которые могут быть отрицательными, и т. п. Если в качестве параметров передаются ссылки на объекты, обязательна проверка на то, как поведет себя программа в случае, если в качестве ссылки передать значение `null`. И т. п.

Еще одна возможность, которую дает JUnit, — задание для теста тайм-аута, т. е. интервала времени (в миллисекундах), в течение которого тест должен закончить свою работу:

```
@Test(timeout=10)
```

Если тайм-аут истек, тест считается неудавшимся. Например, если мы изменим в методе `test3` аннотацию на

```
@Test(expected = IndexOutOfBoundsException.class, timeout = 1)
```

тест будет проходить нормально (по крайней мере, на достаточно современном компьютере), т. к. хотя в тесте `test3` и показывается время прохождения 16 мс (рис. 17.31), это полное время, включающее все накладные расходы на вызов метода. А в тайм-ауте учитывается только чистое время выполнения самого метода, и оно оказывается значительно меньше.

Если же мы аналогичным образом изменим аннотацию в методе `test2` на

```
@Test(timeout = 1)
```

тест не пройдет (рис. 17.32).

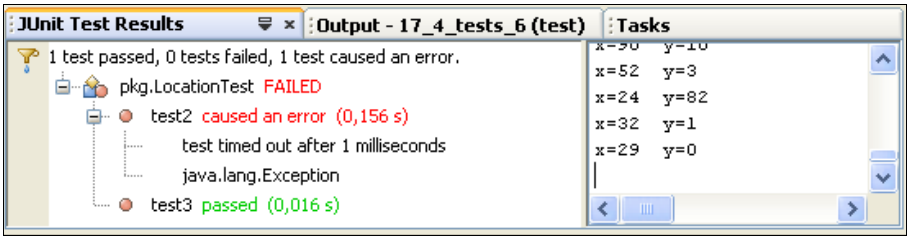


Рис. 17.32. Ошибка выполнения с прерыванием теста по тайм-ауту

Аннотация `@Ignore` позволяет проигнорировать помеченный ей тестовый метод — без необходимости комментирования его содержимого. Если необходимо выдать при запуске теста сообщение о причине игнорирования теста, в аннотации добавляют сообщение, например:

```
@Ignore ("Данный тест пока в стадии разработки")
```

Аннотации

```
@BeforeClass
@AfterClass
@Before
@After
```

ставят перед методами, которые обеспечивают подготовку программной среды к выполнению тестов и возвращающих ее в нормальное состояние после прохождения тестов, а также регистрирующих результаты тестирования. Такие методы называются *фикстурами* (fixtures).

Например, с помощью фикстур, запускаемых перед началом тестирования, можно подготовить данные для классов тестируемой программной системы и инициализировать

необходимые объекты. А с помощью фикстур, запускаемых после окончания тестирования, кончившегося неудачей, в сложных тестовых системах, содержащих большое количество тестов, можно провести анализ того, какие тесты не удались, и на основании этого выстроить логику итогового анализа.

В JUnit 3 фикстуры должны были иметь предопределенные имена. В JUnit 4 достаточно указать перед методом соответствующую аннотацию, хотя NetBeans создает заготовки методов, соответствующих соглашениям об именах JUnit 3.

- Метод с аннотацией `@BeforeClass` во время тестирования выполняется однократно, самым первым, еще до вызова конструктора класса `LocationTest`:

```
@BeforeClass
public static void setUpClass() throws Exception {
}
```

  Название before class означает "перед классом", а setup class в данном случае означает "сделать установки для класса".

- Метод с аннотацией `@Before` во время тестирования выполняется перед вызовом каждого тестового метода:

```
@Before
public void setUp() {
}
```

  Название setup в данном случае означает "сделать установки", при этом подразумевается, что они срабатывают для каждого тестового метода.

- Метод с аннотацией `@After` во время тестирования выполняется после вызова каждого тестового метода:

```
@After
public void tearDown() {
}
```

  Название tear down означает "сносить (постройку)", "разносить в пух и прах", и для теста идеологически играет роль деструктора.

- Метод с аннотацией `@AfterClass` во время тестирования выполняется однократно, самым последним:

```
@AfterClass
public static void tearDownClass() throws Exception {
}
```

  Название after class означает "после класса".

Пример 17_4_tests_7 с прилагаемого к книге DVD иллюстрирует работу этих методов. При работе с проектом следует учесть, что вывод методом `System.out.println` отображается в окне **JUnit Test Results** только для методов, относящихся к классу `LocationTest` и осуществляемых в нем вызовов. Поэтому вывод из методов, помеченных аннотациями `@BeforeClass` и `@AfterClass`, в этом окне не отображается. Но это не значит, что данные методы не вызываются — полный вывод в системную консоль, как всегда в NetBeans, отображается в окне **Output**. И, переключившись в это окно, можно убедиться, что данные методы вызываются в положенном порядке.

В JUnit 4 имеется аннотация `@RunWith`, предназначенная для запуска наборов тестов. Однако соответствующая методика пока является не устоявшейся и достаточно запутанной. Среда NetBeans 6.5 не создает для нее автоматического импорта, и для ее использования необходимо провести импорт из пакета org.junit.runner.

Аннотации JUnit ориентируются на идеи аспектно-ориентированного программирования — нового и достаточно интересного направления в теории программирования. Под аспектом понимается действие, которое должно быть выполнено без изменения кода классов или методов не для одного какого-то метода или класса, а для произвольных, помечаемых соответствующим образом. Были даже предприняты попытки создать аспектно-ориентированный диалект языка Java с названием AspectJ. Однако подход, основанный на аннотациях, оказался гораздо более удачным, что, как кажется автору, было достаточно убедительно продемонстрировано в данном разделе.

17.5. Профилирование приложений

Если про отладку приложений слышали многие люди, не являющиеся программистами, то про то, что такое профилирование, знают даже не все профессионалы.

Профилирование — это нахождение участков кода приложения, нуждающихся в оптимизации для повышения производительности и уменьшения потребляемых ресурсов. Профилирование производится путем нахождения "горячих" точек (hot spots), потребляющих основные ресурсы при работе приложения, и участков кода с неэффективным использованием памяти.

Таким образом, профилирование нужно только для приложений, потребляющих много ресурсов. Причем, в отличие от отладки, при профилировании не ищут ошибки, а только улучшают работу уже вполне работоспособного кода.

В качестве примера можно привести программное обеспечение для серверов, где увеличение скорости работы приложения на 20% может означать экономию десятков и сотен тысяч долларов (за счет отказа от покупки дополнительных серверов).

Или взять, например, программное обеспечение антивирусов, в режиме реального времени проверяющих все запускаемые программы на наличие вирусов и "троянцев". В связи с массовой рассылкой по электронной почте спама (вредоносных сообщений, содержащих рекламу и вирусы), а также вирусов, заражающих компьютеры при подключении к USB-порту карт флэш-памяти, компьютер без антивирусной программы является потенциальным (а в большинстве случаев реальным) рассадником вирусов. Автор предпочитает иметь на компьютере Антивирус Касперского (пакет Kaspersky Internet Security) — одну из наиболее эффективных программ защиты от вирусов. В настоящее время в этой антивирусной базе находится информация более чем о 1 700 000 вредоносных программах. Для того чтобы сделать 1 700 000 проверок и не занимать при этом все ресурсы компьютера на долгие часы, требуются изощренные варианты оптимизации программного кода, и здесь без профилирования не обойтись.

Точно так же и любые счетные приложения, предназначенные для многократного использования, нуждаются в профилировании. Например, если результат вычисляется в течение четырех часов, то повышение быстродействия на четверть сэкономит вам целый час! А ведь иногда с помощью профилирования и последующей оптимизации удается повысить быстродействие программ в десятки и даже сотни раз!

Наконец, профилирование программ, предназначенных для мобильных устройств, позволяет минимизировать требуемые ресурсы и максимизировать быстродействие, что для этих устройств крайне важно.

В NetBeans для профилирования главного приложения следует либо выбрать пункт меню **Profile/Profile Main Project...**, либо воспользоваться кнопкой на панели инструментов. Выберем в качестве профилируемого приложения проект 17_5_DesktopApplication с прилагаемого к книге DVD (предварительно его необходимо скопировать на жесткий диск вашего компьютера).

При первом запуске профилировщика возникает диалоговая форма с сообщением о том, что профилировщик не интегрирован с вашим проектом, и что сценарий (script) компиляции проекта будет модифицирован (рис. 17.33).

Задается вопрос, согласны ли вы выполнить данные действия, и что можно в любой момент отменить интеграцию с профилировщиком с помощью пункта меню **Profile/Advanced Commsnds/Unintegrate Profiler**. На вопрос следует ответить утвердительно, нажав кнопку **OK**.

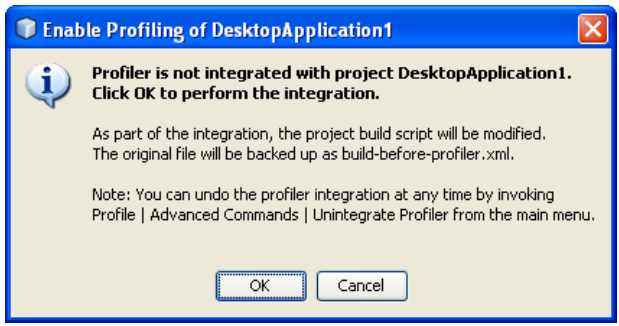


Рис. 17.33. Сообщение при первом запуске профилировщика

Появится диалог, позволяющий настроить параметры профилирования (рис. 17.34).

По умолчанию выбран анализ времени выполнения (**Analyze Performance** — раздел **CPU**). При этом анализ делается для всего приложения (выбрана позиция **Entire application**).

Индикатор **Overhead** (накладные расходы) показывает примерную долю времени (в процентах), которую занимает код профилирования по сравнению с основным кодом. В нашем случае это 5%. Важность этой величины связана с тем, что профилировщик должен тратить часть времени приложения на профилирование. В результате при профилировании приложение работает несколько медленнее по сравнению с обычным режимом. Дополнительные затраты и называются накладными расходами.

При запуске профилировщика нажатием на кнопку **Run** сначала идет калибровка — измеряется, какой вызов профилировщика сколько времени занимает.

Перед началом калибровки выдается сообщение о том, для чего она нужна (рис. 17.35).

При измерении времени выполнения вызовов приложения профилировщик учитывает свое влияние — из полного времени вызова вычитается время на вызовы самого профилировщика, что позволяет заметно повысить точность оценки (при этом учитываются также параметры конкретной Java-машины).

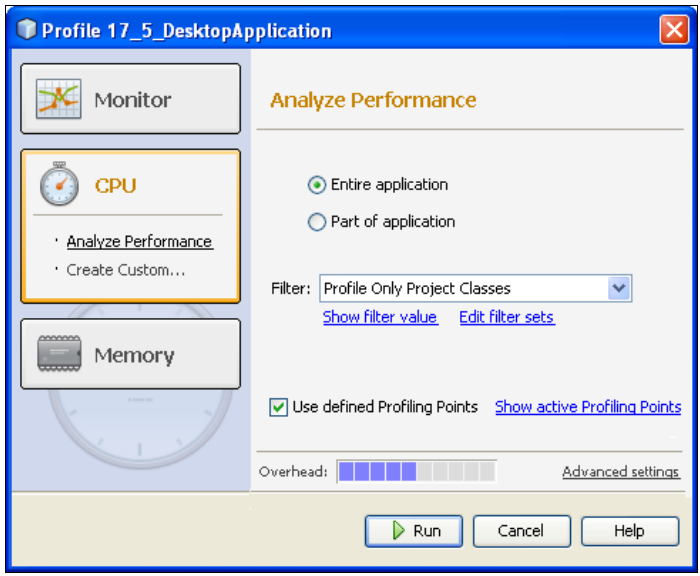


Рис. 17.34. Диалог настройки параметров профилирования — раздел **CPU** (процессор)

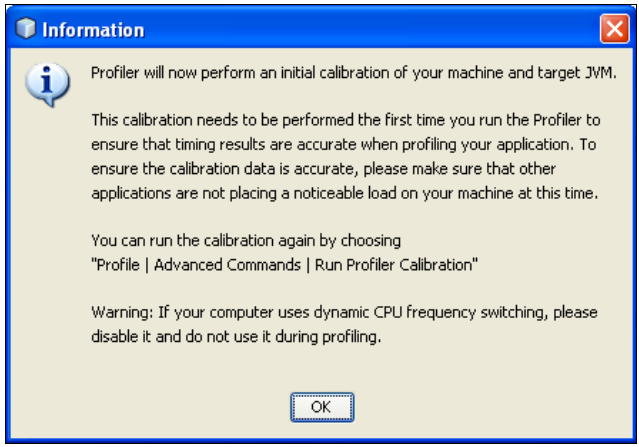


Рис. 17.35. Информация профилировщика о калибровке

Обычно накладные расходы составляют несколько процентов от общего времени и очень мало влияют на выполнение приложения. Но иногда они соизмеримы по величине со временем вызова программы, и без их учета результаты оказались бы сильно искаженными.

Еще один важный момент связан с использованием процессоров с переключаемой тактовой частотой (например, Intel Centrino). Для них результаты профилирования будут неправильными в случае, если калибровка проводилась на одной тактовой частоте, а во время профилирования частота изменилась. Для таких процессоров на время работы с профилировщиком следует отключить изменение частоты процессором.

Также важно, чтобы во время работы профилировщика ресурсы процессора не занимались другими приложениями, иначе это может повлиять на точность учета профили-

ровщиком накладных расходов. В частности, необходимо отключить резидентные антивирусные программы. Но при этом автор настоятельно рекомендует отключить компьютер от Интернета и локальных сетей (вытащив вилку линии связи из разъема!), в том числе от беспроводной сети Wi-Fi.

После нажатия кнопки **OK** выполняется калибровка (пару секунд), а затем появляется информационное сообщение об ее успешном окончании (рис. 17.36).

В любой момент времени можно заново провести калибровку, выбрав пункт меню **Profile/Advanced Commsnds/Run Profiler Calibration**.

Полезно посмотреть результаты калибровки — кнопка **Show Details** (Показать детали).

Для процессора Intel Core™ 2 Duo T5300 с тактовой частотой 1,73 ГГц примерное время затрат на пару "вход в метод/выход из метода" оказалось равно:

- при получении только абсолютного временного отсчета (timestamp): 4,41 мкс (1 мкс = 10^{-6} с);
- при получении только временного отсчета потока выполнения (thread) процессора: 1,43 мкс;
- при получении обоих временных отсчетов: 5,36 мкс.

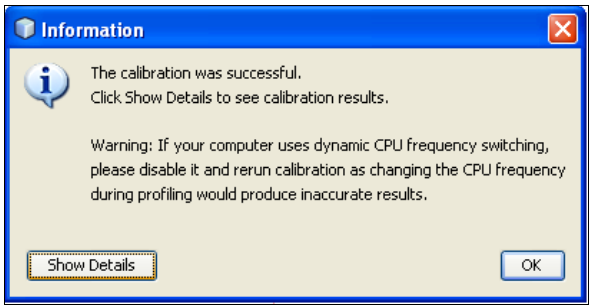


Рис. 17.36. Информация об окончании калибровки

Примерное время затрат на сбор данных профилировщиком на пару "вход в метод/выход из метода" равно 0,27 мкс.

Таким образом, наши накладные расходы на профилирование оказались около 5% (рис. 17.37).

Аналогичные результаты, полученные для процессора Intel Core™ 2 Quad Q8200 с тактовой частотой 2,33 ГГц.

Примерное время затрат на пару "вход в метод/выход из метода" оказалось равно:

- при получении только абсолютного временного отсчета (timestamp): 0,653 мкс;
- при получении только временного отсчета потока выполнения (thread) процессора: 0,969 мкс;
- при получении обоих временных отсчетов: 1,45 мкс.

Примерное время затрат на сбор данных профилировщиком на пару "вход в метод/выход из метода" равно 0,19 мкс.

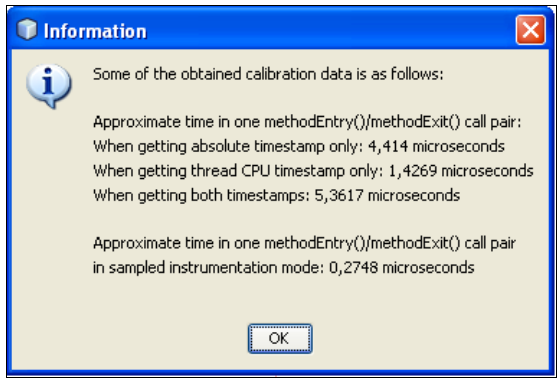


Рис. 17.37. Информация о результатах калибровки

После запуска приложения, выполнения им всех действий, а потом и закрытия, выдается сообщение, представленное на рис. 17.38.

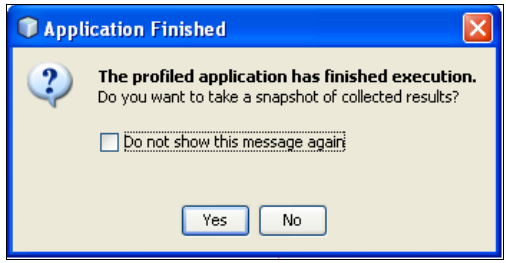


Рис. 17.38. Предложение показать результаты профилирования

На заданный вопрос следует ответить утвердительно, нажав кнопку **Yes**. Чтобы это сообщение не показывалось всякий раз, следует установить флажок **Do not show this message again** (Не показывать больше это сообщение).

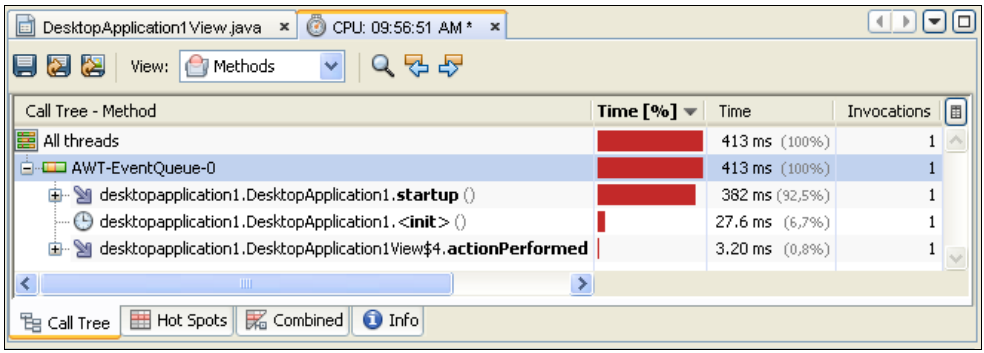


Рис. 17.39. Результат профилирования (дерево вызовов)

В окне **CPU** (сокращение от "Центральное процессорное устройство") на вкладке **Call Tree — Method** (Дерево вызовов) отображается время выполнения методов приложения по каждому из потоков выполнения (рис. 17.39). В графе **Time [%]** строится диа-

грамма, показывающая процент от общего времени для данного вызова в виде столбца соответствующей высоты. В графе **Time** та же величина указывается в виде числа (в миллисекундах). И наконец, в последней графе **Invocations** показывается число сделанных вызовов для каждого из методов.

Любой из узлов дерева можно развернуть и посмотреть распределение по времени вложенных вызовов (рис. 17.40).

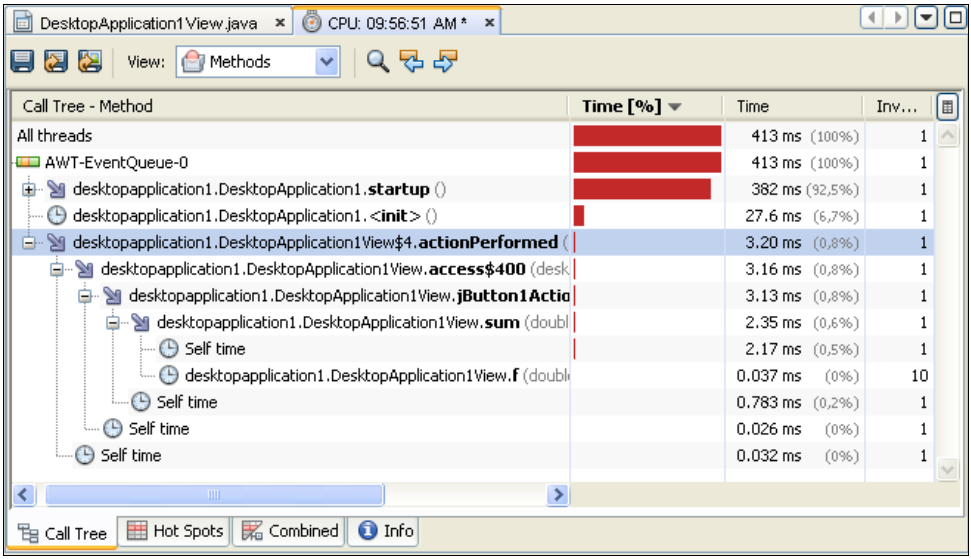


Рис. 17.40. Развернутое дерево вызовов

Двойной щелчок по имени метода, как всегда, приводит к тому, что в редакторе исходного кода показывается соответствующий метод либо сворачивается/разворачивается соответствующий узел. В последнем случае перейти в исходный код вместо задания метода можно с помощью команды **Go to Source** контекстного меню.

Обратим внимание на то, что с помощью профилировщика можно выполнять некоторые действия, которые относятся к отладке приложения (например, проверять число вызовов методов). Профилировщик мгновенно выдает, что при `n=8` (в нашем примере 17_3_DesktopApplication) число вызовов (invocations) метода `f` равно `8`, при `n=9` равно `9`, при `n=10` равно `11`, а при `n=11` также равно `11`. Таким образом легко установить, что сбой происходит из-за наличия лишнего вызова.

В верхней части окна находится панель инструментов.

Кнопка **Save Snapshot to Project** (Сохранить моментальный снимок в проекте) позволяет сохранить результаты в проекте в виде данных, чтобы эти результаты можно было в дальнейшем не только просматривать, но и обрабатывать. Слово Snapshot означает моментальный снимок, т. е. результат профилирования в некоторый момент времени.

Кнопка **Save Snapshot to Custom File** (Сохранить результат профилирования в заданный файл) дает возможность сохранить результаты в файле.

Кнопка **Saves current view to image** (Сохранить показываемое изображение в виде графического файла) позволяет сохранить изображение (а не сами данные!) в виде файла в формате png.

Вкладка **Hot Spots** (в нижней части окна) обеспечивает переключение в режим показа "горячих" точек (hot spots) (рис. 17.41). При этом профилировщик самостоятельно анализирует, какие методы сколько времени тратили на полезную деятельность (задержки из-за обращения к внешним устройствам и системные вызовы не учитываются). Поэтому значения затрат по времени для данной вкладки и для вкладки **Call Tree** заметно различаются.

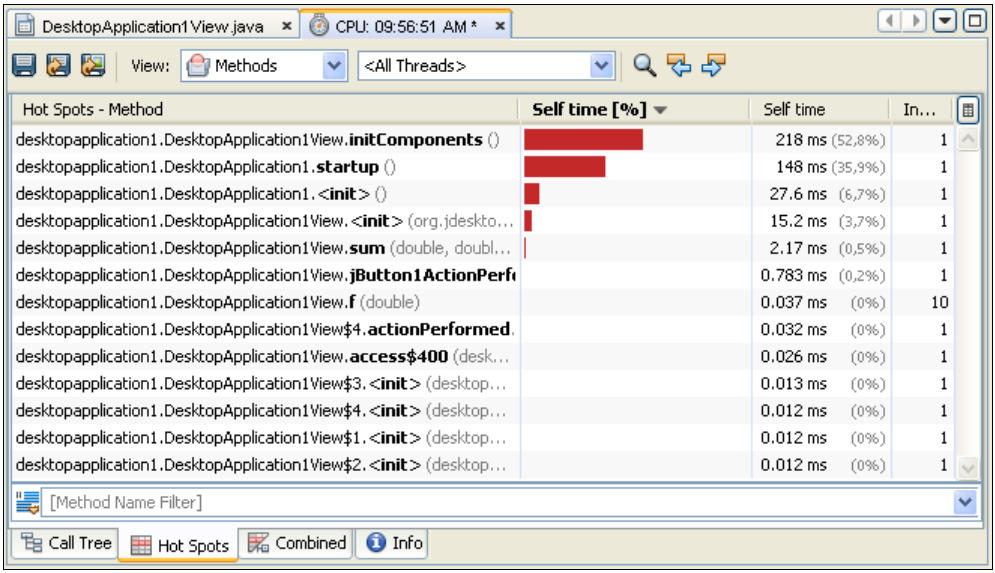


Рис. 17.41. Показ "горячих" точек (hot spots)

Вкладка **Combined** (Комбинированная) позволяет выводить в окно сразу обе диаграммы: в верхнюю часть — дерево вызовов, в нижнюю — "горячие" точки (рис. 17.42).

При анализе того, сколько раз в примере 17_3_DesktopApplication вызывался какой метод, нам приходилось каждый раз запускать приложение в режиме профилирования, делать вызов для заданного значения `n` и закрывать приложение. Только после этого показывались результаты, что не слишком удобно. Хотелось бы получать результаты каждый раз после нажатия на кнопку **Вычислить**.

Для этого следует воспользоваться окном **Profiler** (Профилировщик) (рис. 17.43).

При нажатии на иконку **Take Snapshot** (Получить результат профилирования, получить срез данных) мы получаем картину для момента времени, соответствующего нажатию.

А при нажатии на иконку **Live Results** (Живые результаты) мы получаем картину для автоматически обновляющихся результатов через заданные моменты времени. Правда, это не очень помогает, т. к. показывается общее число вызовов методов, поэтому при каждом повторном нажатии в приложении на кнопку **Вычислить** число в графе **Invocations** увеличивается на количество сделанных после этого нажатия вызовов.

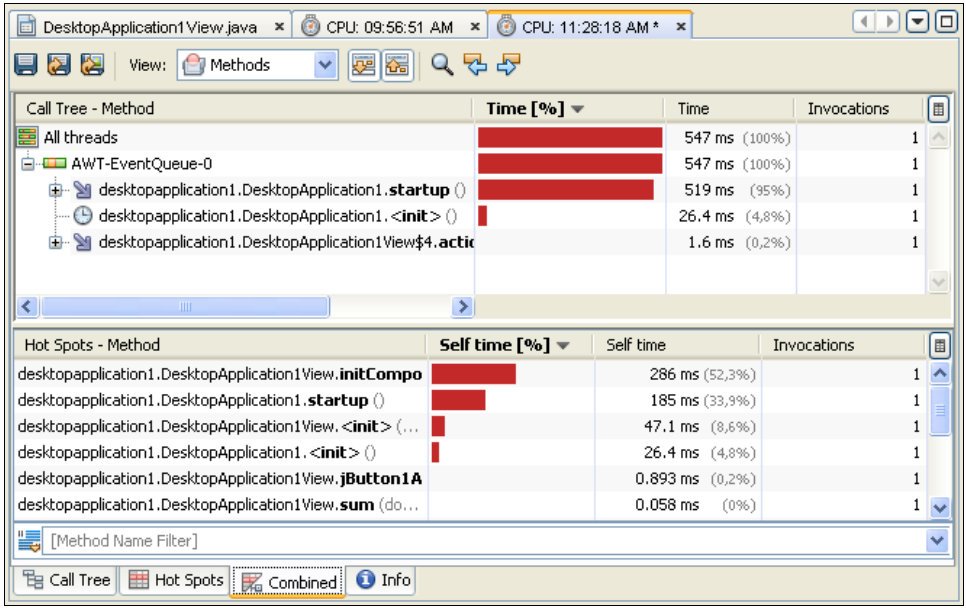


Рис. 17.42. Вкладка **Combined**

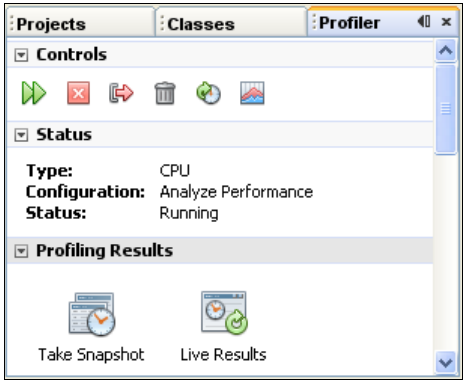


Рис. 17.43. Окно **Profiler**

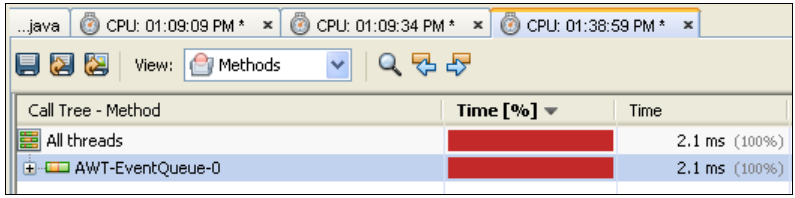


Рис. 17.44. Вкладки для разных временных срезов

Чтобы провести измерения с нуля без перезапуска приложения, нужно очистить буфер профилировщика, нажав на кнопку **Reset Collected Results Buffer** (Переустановить буфер собранных результатов) (рис. 17.43), после чего можно в приложении нажимать кнопку **Вычислить** и делать срез **Take Snapshort** для просмотра числа вызовов. Каж-

дый срез отображается в своем окне, между которыми можно переключаться, выбирая нужную вкладку (рис. 17.44).

17.6. Профилирование отдельных методов

Нас может заинтересовать профилирование конкретного метода или группы методов.

Для изучения особенностей работы со строками добавим в реализацию функции `sum` вывод в консоль (листинг 17.13, пример 17_6_DesktopApplication на прилагаемом к книге DVD).

Листинг 17.13. Исходный код функции `sum`

```
double sum(double a, double b, long n) {
    double s = 0;
    double h = (b − a) / n;
    double x;
    for (x = a; x < b; x = x + h) {
        s = s + f(x) * h;
        if(x>=b-2*h){
            System.out.println("n="+n);
            System.out.println("a="+b);
            System.out.println("b="+b);
            System.out.println("h="+h);
            System.out.println("x="+x);
            System.out.println("f(x)="+f(x));
            System.out.println("s="+s);
            System.out.println();
        }
    }
    return s;
}
```

Чтобы выяснить, как работает обработчик нажатия на кнопку, запустим профилировщик и переключимся из режима профилирования всего приложения (**Entire application**) в режим профилирования части приложения (**Part of application**) (рис. 17.45), при этом индикатор накладных расходов уменьшит показания до 2%.

После такого переключения в той же строке, где находится пункт выбора **Part of application**, возникнет надпись **No root methods, define...** (Нет корневых методов, задайте...), т. е. отсутствуют методы, которые мы хотим профилировать. Они называются корневыми (root), потому что будут профилироваться не только они, но и все вложенные методы, так что в дереве вызовов они окажутся на самом верхнем уровне — корневом.

Нажмем на гиперссылку **define...** и в форме **Specify Root Methods** (Задать корневые методы) (рис. 17.46) нажмем кнопку **Add From Project...** (Добавить из проекта...).

Развернем дерево пакетов, классов и методов нашего приложения и отметим в нем интересующий нас обработчик **jButton1ActionPerformed** (рис. 17.47).

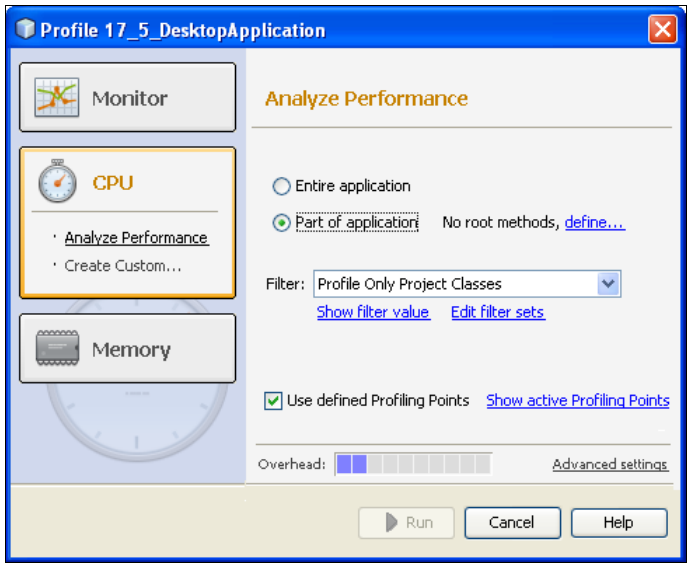


Рис. 17.45. Профилирование отдельного метода

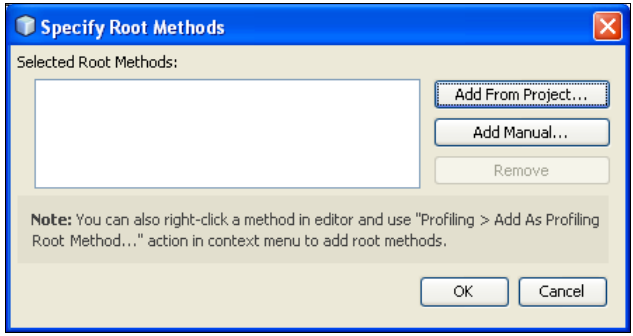


Рис. 17.46. Форма задания корневых методов

Автоматически отметятся необходимые узлы дерева более высокого уровня. Нажмем кнопку **OK** и увидим, что в форме задания корневых методов появился метод **desktopapplication1.DesktopApplication1View.jButton1ActionPerformed**.

Другой путь назначения корневого метода — вызвать в исходном коде контекстное меню и в нем выбрать пункт **Profiling/Set As Profiling Root Method...**.

Нажмем кнопку **OK**, закрыв тем самым форму задания корневых методов, и попадем обратно в главную форму профилировщика. В ней против пункта выбора **Part of application** возникнет надпись **1 root method, Edit...** (1 корневой метод, редактировать).

Теперь можно запускать приложение на профилирование.

В процессе снятия среза после нажатия в приложении на кнопку **Вычислить** мы получим результат при просмотре дерева вызовов (рис. 17.48).

Если переключиться в режим показа "горячих" точек, окно будет выглядеть так, как изображено на рис. 17.49.

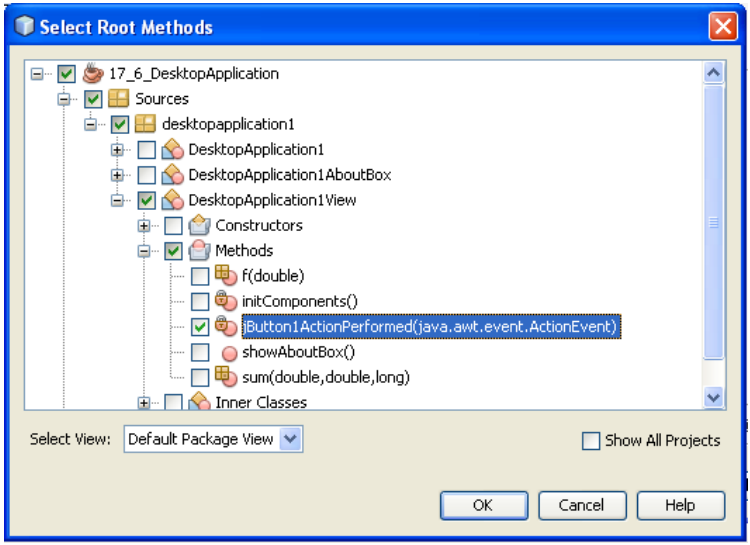


Рис. 17.47. Дерево методов, выбираемых для профилирования

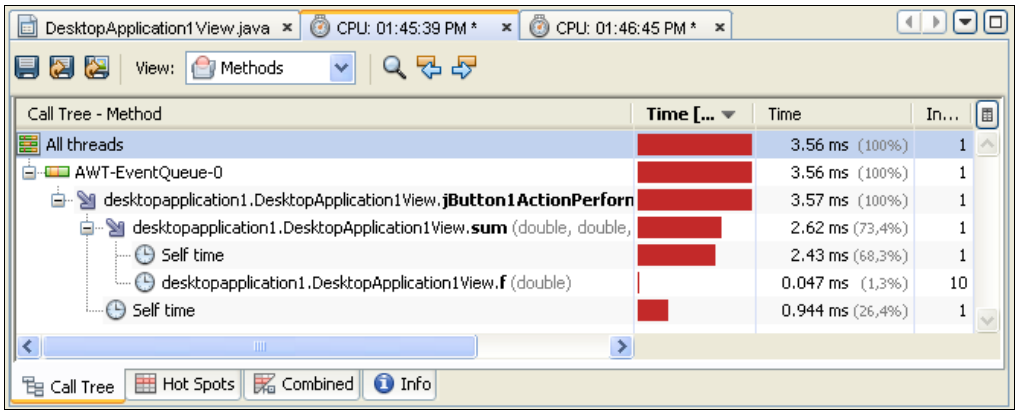


Рис. 17.48. Дерево вызовов при профилировании метода

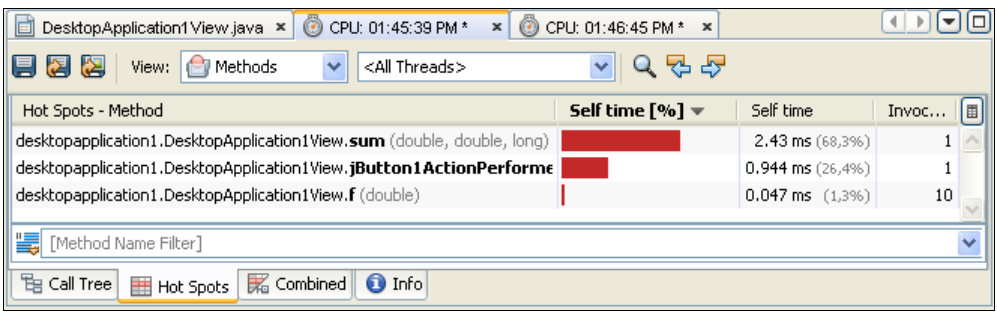


Рис. 17.49. Показ "горячих" точек при профилировании метода

Проанализируем наши "горячие" точки.

Самое большое время занимает вызов метода `sum` (68%).

Следующий по затратам — сам метод обработчика нажатия на кнопку (24%).

И меньше всего времени тратится на вызовы функции `f` (1,4%), хотя этих вызовов в нашем случае целых 10 штук (это намного меньше, чем 2,43 мс — время вызова функции `sum`). Поэтому можно сделать вывод, что наибольшее внимание следует обратить на операторы именно этой функции.

В нашем списке показывается только информация по методам из классов проекта. Чтобы получить более детальные сведения, необходимо изменить настройку фильтров профилировщика.

По умолчанию стоит фильтр **Profile Only Project Classes** (Профилировать только классы проекта). Если мы выберем в выпадающем списке вариант **Profile All Classes** (Профилировать все классы), то накладные расходы (overheads) сразу возрастут с 2 до 7%.

В качестве корневого метода мы выберем `sum`, поскольку иначе "утонем" в большом объеме ненужных нам данных о вызовах метода **jButton1ActionPerformed**.

Как видно из диаграммы (рис. 17.50), основное время тратится на вызов метода **java.io.PrintStream.println**. Откуда он взялся?

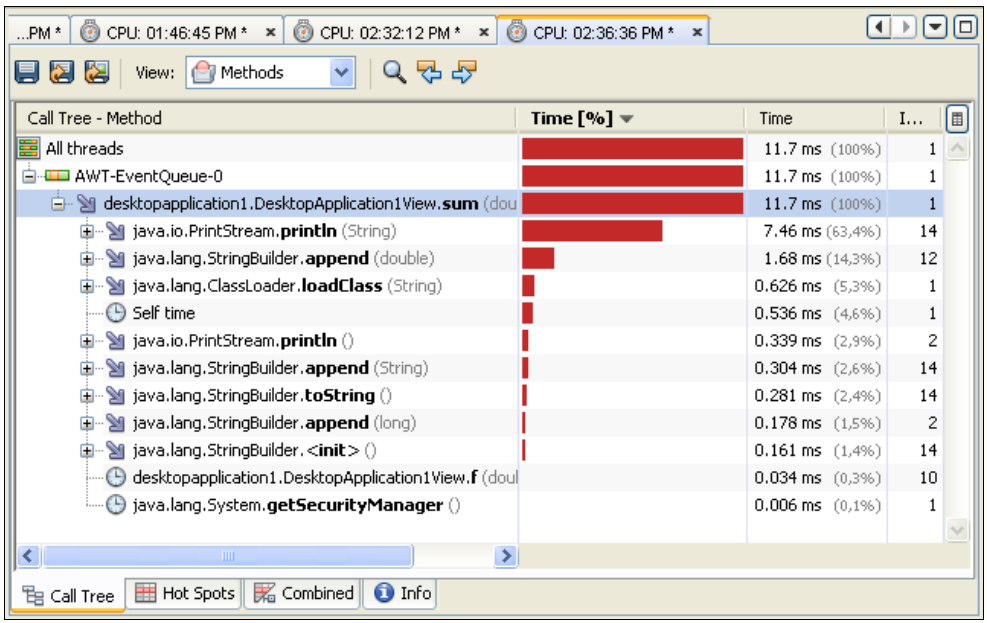


Рис. 17.50. Дерево вызовов при профилировании метода с учетом всех классов

Наш метод `sum` выглядит так, как представлено в листинге 17.13. После анализа кода данного метода становится понятно, что вызов **java.io.PrintStream.println** мог появиться только из вызовов **System.out.println**.

Также ясно, что метод **java.lang.StringBuilder.append** появляется из-за наличия сложений в вызовах **System.out.println**.

Замечание

Нам приходится делать такого рода выводы и предположения самостоятельно, хотя это должен был бы показывать сам профилировщик среды NetBeans. А попытка перейти во фрагмент исходного кода, откуда был вызван метод, отсылает нас в класс `StringBuilder`. Это происходит из-за того, что средства профилировщика среды NetBeans пока не так развиты, как, например, в профилировщике Intel VTune™. Тем не менее VTune — достаточно дорогостоящее средство, а NetBeans — бесплатная среда разработки. Так что учиться пользоваться профилировщиками можно в NetBeans. К тому же среда NetBeans очень быстро совершенствуется, поэтому будем надеяться, что в последующих версиях NetBeans профилировщик станет лучше.

Закомментируем вывод в окно консоли и заново запустим профилировщик.

Мы получим совсем другие результаты: теперь у нас резко уменьшилось число показываемых профилировщиком методов и существенно возросла скорость работы приложения (рис. 17.51). Вместо 11,7 мс метод `sum` стал выполняться 0,111 мс (т. е. скорость работы увеличилась на два порядка!). А время восьми вызовов функции `f` оказалось равно 33 мкс (0,033 мс) (т. е. примерно половине времени работы метода `sum`!).

Увеличим в десять раз число точек для расчета и посмотрим, как изменятся диаграммы (не забудьте очистить буфер!).

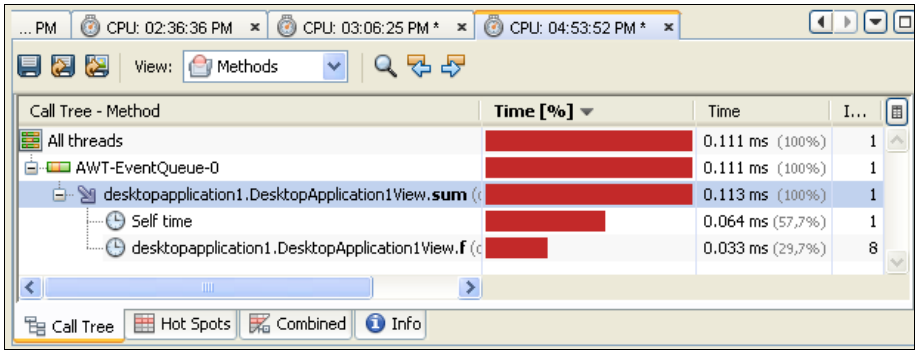


Рис. 17.51. Дерево вызовов после удаления вывода в окно консоли

Во-первых, сразу видно, что точек 81, а не 80 (ведь мы так и не исправили ошибку!).

Во-вторых, время, потраченное на вызовы `sum` и `f`, увеличилось вовсе не в 10 раз.

Для `sum` было 0,113 мс, стало 0,696 мс (что в 6,2 раза больше).

А для `f` было 0,033 мс, а стало 0,25 мс (что в 7,6 раз больше).

Таким образом, в функции `sum` при `n=8` на цикл тратится всего половина времени.

При `n=80` почти все время занимает цикл — на него тратится 85% времени, а на все остальное всего 15% (рис. 17.52).

Посмотрим, как изменится работа нашего приложения после исправления ошибки — замены цикла с вещественным счетчиком на целый (листинг 17.14).

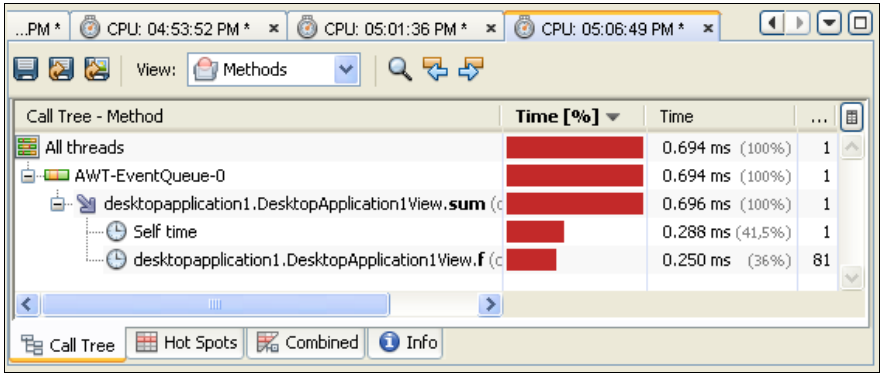


Рис. 17.52. Дерево вызовов для 80 точек

Листинг 17.14. Вещественный счетчик цикла заменен на целый

```
double sum(double a, double b, long n) {
    double s = 0;
    double h = (b – a) / n;
    double x;
    int i;
    for (i = 0; i < n; i++) {
        s = s + f(a + i * h) * h;
    }
    return s;
}
```

Время работы метода `sum` для `n=8` уменьшается до 0,107 мс, что по сравнению с 0,113 мс не очень заметно.

Для `n=80` имеем 80 вызовов функции `f`. Скорость не главное, если результат неверен! В предыдущем алгоритме наблюдался 81 вызов вместо 80. Время вычисления `sum` становится 0,475 мс вместо 0,696 мс. Так что мы не только исправили алгоритм, но и сделали его быстрее.

Попробуем еще немного усовершенствовать метод, уменьшив количество умножений, которые делаются внутри цикла. Не будем каждый раз домножать на `h`, а выполним это умножение в самом конце. Листинг 17.15 иллюстрирует новый вариант метода.

Листинг 17.15. Уменьшение числа умножений внутри цикла

```
double sum(double a, double b, long n) {
    double s = 0;
    double h = (b – a) / n;
    int i;
    for (i = 0; i < n; i++) {
        s = s + f(a + i * h);
    }
    return s*h;
}
```

Для n=8 время уменьшилось до 0,102 мс, а для n=80 время уменьшилось до 0,643 мс.

Итак, с помощью профилировщика нам не только удалось исправить ошибку, но и ускорить нашу программу примерно на 8%. Это не слишком заметное, но вполне приемлемое увеличение скорости. А если учесть, что до этого мы обнаружили фрагмент, замедлявший работу программы на два порядка, то становится понятным, почему наибольший эффект профилирование дает именно в случае обнаружения такого рода "ляпов". В этом случае реорганизация программы позволяет повысить быстродействие в разы, а иногда на порядки. А вот оптимизация грамотно написанных фрагментов кода позволяет за счет разного рода трюков повысить быстродействие на проценты, в лучшем случае на десятки процентов.

Стоит подчеркнуть, что оптимизировать нужно только "горячие" точки. Даже если вы повысите на два или три порядка быстродействие метода, занимающего 0,1% времени, эффект будет ничтожно мал (всего 0,1%).

А если вы повысите на 10% быстродействие метода, занимающего при работе 90% времени, то суммарное быстродействие повысится на 9%. Поэтому лучше биться за отдельные проценты для "горячих" точек, чем за порядки для методов, реально не влияющих на производительность приложения.

Приведем пример случая, когда оптимизация может быть более существенной.

Предположим, что у нас функция f была задана так:

```
double f(double x) {
    double r = x * x;
    return r;
}
```

Оптимизированный вариант выглядит следующим образом:

```
double f(double x) {
    return x * x ;
}
```

При n=8 отличия в скорости работы этих вариантов незаметны. Но при n=80 для первоначального варианта f время вызова функции sum на 20% больше, чем для оптимизированного варианта.

17.7. Настройка профилировщика "на лету". Точки профилирования

До сих пор мы говорили только о двух функциях профилировщика: измерении времени вызовов и их количестве. Но в ряде случаев возникает необходимость устранить проблемы с использованием памяти.

У этой проблемы несколько аспектов.

- Ситуация, когда в цикле создается много короткоживущих объектов. Неправильно построенный алгоритм способен вызвать значительные проблемы при работе с памятью, хотя, конечно, сборщик мусора в конце концов удалит все ненужные объекты. Но если программа будет генерировать объекты слишком активно, ее быстро-

действие может резко снизиться. Ведь время создания объектов относительно велико, поэтому каждый новый объект приводит к лишним затратам времени. Кроме того, занимается дополнительная память. А ведь во многих случаях экономия памяти может быть не менее важной, чем быстродействие (например, для мобильных устройств). И наконец, часто появление ненужных объектов свидетельствует о неправильном подходе к решению задачи.

- Наличие так называемых утечек памяти (memory leakage). Довольно распространено мнение, что в Java такая проблема отсутствует, поскольку сборщик мусора автоматически удаляет неиспользуемые объекты. Однако это не так — если ссылка на ненужный объект сохраняется, объект не будет удален из памяти. За подробностями отсылаем к главе 5 "Уничтожайте устаревшие ссылки" в книге Джошуа Блоха [11].

Чаще всего подобного рода ошибки допускают при работе в цикле с массивами или списками объектов, а также при операциях со строками.

Рассмотрим ошибку такого рода.

В Java имеется средство для замены части символов строки на другие, но эти действия производятся именно с массивом символов, а не со строками. Для того чтобы понять, почему так сделано, мы и рассмотрим наш пример. В нем самостоятельно напишем альтернативный вариант такой функции и сравним эффективность этих двух алгоритмов, а заодно и научимся искать проблемы с использованием памяти.

Расположим на нашей форме область ввода текста `jTextArea1` и пункты ввода текста `jTextField1` и `jTextField2`. Допустим, что мы хотим заменить в тексте символы, заданные пользователем в `jTextField1`, на символы, заданные пользователем в `jTextField2`. Например, у нас имеется большой набор данных со строковым представлением чисел, где десятичный разделитель имеет вид запятой (представление, характерное для России и Китая), а мы хотим получить набор, в котором запятые заменены на точки (представление, характерное для США и многих стран Европы).

Написанная нами функция `replaceSubS` (листинг 17.16, пример 17_7_DesktopApplication1 на прилагаемом к книге DVD) выполняет такую замену. У нее на входе строка `s`, в которой должна быть произведена замена, подстрока `subS` с набором заменяемых символов, а также строка `newSubS` с набором новых символов, на которые происходит замена.

Листинг 17.16. Исходный код функции `replaceSubS`

```
String replaceSubS(String s, String subS, String newSubS) {
     String newS = "";
     int i = 0;
     int old_i = 0;
     i = s.indexOf(subS, i);
     while (i > -1) {
         newS = newS + s.substring(old_i, i) + newSubS;
         old_i = i + subS.length();
         i = s.indexOf(subS, old_i);
     }
     newS = newS + s.substring(old_i, s.length());
     return newS;
}
```

В функции сначала ищется позиция заменяемой подстроки. Если такая подстрока найдена (`i > -1`), то к формируемой новой строке добавляется участок от прежней позиции до новой (причем, не включающий символы подстроки), а в конец добавляется строка `newSubS` с набором замещающих символов.

После этого старая позиция увеличивается, становясь равной текущей позиции первого символа после окончания заменяемого набора. А затем, начиная с этого значения, ищется новое вхождение подстроки `subS`. По завершении цикла (когда нового вхождения не найдено) к формируемой строке добавляется набор символов строки `s` от первого символа после окончания последнего вхождения подстроки до конца строки `s`.

В листинге 17.17 приведен пример функции `replaceSubS` для замены в компоненте `jTextArea1` последовательности символов, введенных пользователем в поле `jTextField1`, на новую последовательность, введенную пользователем в поле `jTextField2`, например, для замены запятой (вводим этот символ во время выполнения приложения в поле `jTextField1`) на точку (вводим в поле `jTextField2`).

Листинг 17.17. Использование функции `replaceSubS` в обработчике событий `jButton1ActionPerformed`

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
     String s = jTextArea1.getText();
     String subS = jTextField1.getText();
     String newSubS = jTextField2.getText();
     jTextArea1.setText(replaceSubS(s, subS, newSubS));
}
```

Данный алгоритм работает, и на первый взгляд, — хорошо. Можно, конечно, заменить присваивания вида `a=a+...` на `a+=...`, но не это основной недостаток нашего кода. Попробуем профилирование функции `replaceSubS` на тестовом наборе данных, формируемых во время выполнения приложения по нажатию кнопки **jButton2** (листинг 17.18).

Мы ограничимся генерацией 1000 значений, поскольку уже при 10 000 начинаются проблемы, причину которых скоро поймем. А ведь это совсем небольшой размер текста, в котором нужно проводить замену!

Листинг 17.18. Генерация строки с данными в обработчике событий `jButton2ActionPerformed`

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
     int i = 0;
     int COUNT = 1000;
     String s = "";
     for (i = 1; i <= COUNT; i++) {
         s = s + " 1," + i;
     }
     jTextArea1.setText(s);
}
```

Проведем профилирование для метода `replaceSubS` с фильтром **Profile All Classes**. Из рис. 17.53 видно, что основное время занимает сложение строк, выполняемое с по-

мощью метода `StringBuilder.append`. На втором месте — метод `StringBuilder.toString`, связанный с выполнением вызова `s.substring(old_i, i)`.

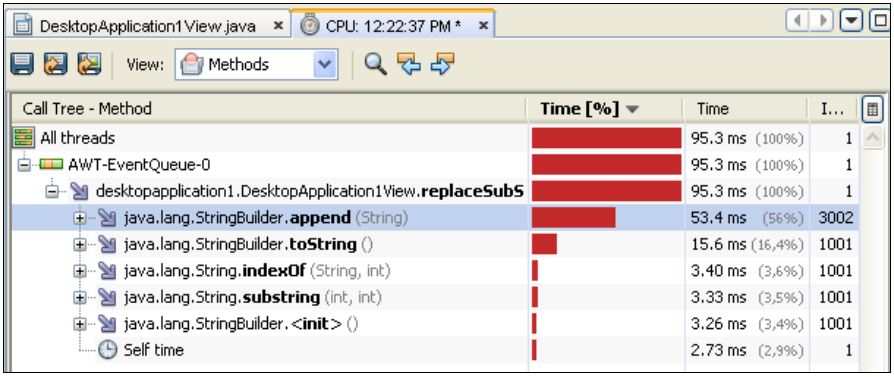


Рис. 17.53. Профилирование метода `replaceSubS`

Откуда взялись эти вызовы в методе `replaceSubS`, нам приходится только догадываться — профилировщик не дает указаний о том, каким строкам нашего исходного кода они соответствуют. Причем это не связано с настройками фильтров профилирования. Для того чтобы убедиться в этом, изменим настройки профилирования "на лету" (on fly) без выхода из сессии: в окне **Profiler** нажмем кнопку **Modify Profiling Session...** (<Alt>+<Shift>+<F2>) (Изменить сессию профилирования) (на рис. 17.54 она находится в верхнем ряду).

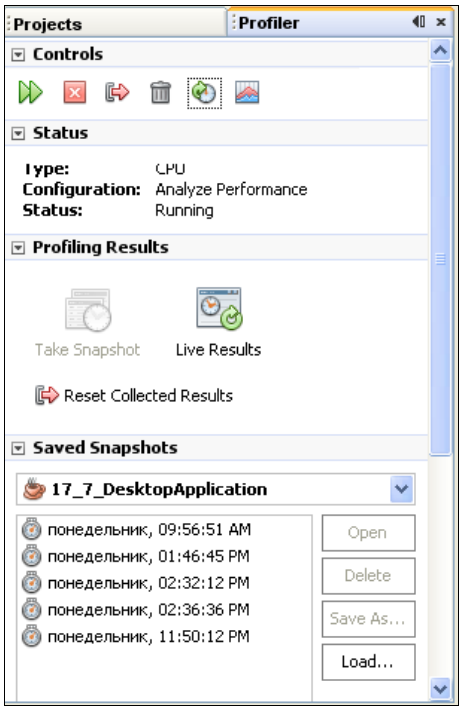


Рис. 17.54. Изменение "на лету" сессии профилирования

Появится диалог задания параметров профилирования, который мы рассматривали в предыдущем разделе. В этом диалоге можно установить фильтры **Profile only Project Classes** (Профилировать только классы проекта) или **Exclude Java core Classes** (Исключить классы ядра Java).

Очистим буфер (**Reset Collected Results**) и проведем профилирование. Убедимся, что на диаграммах показываются только параметры для профилируемого метода `replaceSubS`. Значит, таким способом мы не сможем выяснить, какой из участков или какая из строк данного метода дольше всех выполняется. Но решение все-таки существует, хотя "на лету" его применить не удастся. Мы можем использовать точки профилирования — **Profiling points**. Идеология работы с ними очень похожа на идеологию применения точек остановки, только вместо остановки выполняются некоторые действия, после чего приложение продолжает работу.

Завершим сессию профилирования и поставим курсор на нужную строку исходного кода внутри метода `replaceSubS`:

```
i = s.indexOf(subS, i);
```

Правой кнопкой мыши вызовем в нем контекстное меню, выберем в этом меню пункт **Profiling/Insert Profiling Point...**, появится диалог, в котором необходимо установить тип точки профилирования (рис. 17.55).

При достижении в приложении данной точки могут выполняться следующие варианты действий:

- **Reset Results** — очистка накопленных до того данных профилирования. Такой точкой следует отмечать начало профилируемого фрагмента кода, если требуется, чтобы учитывались данные только от этого фрагмента;

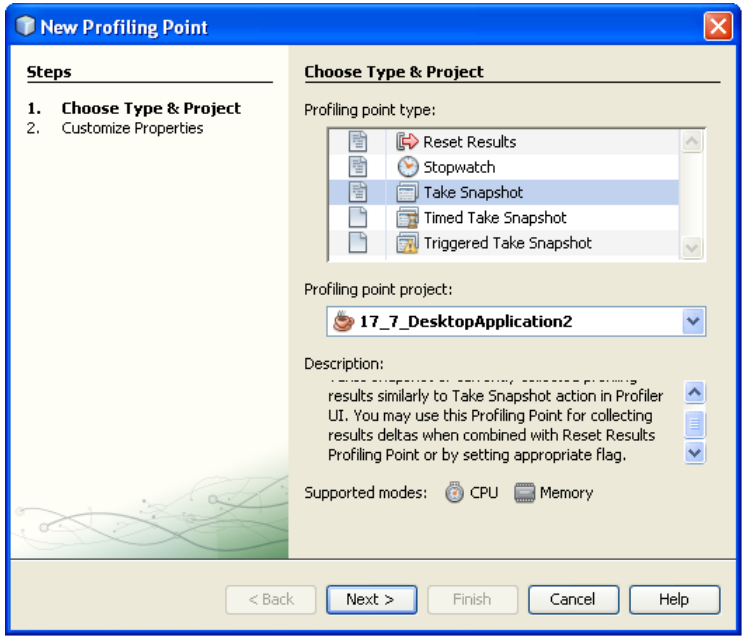


Рис. 17.55. Установка точки профилирования (шаг 1)

- **Stopwatch** — остановить часы. Используется для измерения временных интервалов вместо вызова непосредственно в коде метода `System.getCurrentTimeMillis()`;
- **Take Snapshot** — получить срез с результатами профилирования. Служит для получения результатов профилирования при достижении нужного места программы. Именно этот вариант мы сейчас выберем;
- **Timed Take Snapshot** — то же, что в предыдущем случае, но относится не к конкретному методу, а ко всей сессии профилирования, а срез профилирования снимается в заданный момент времени или будет сниматься периодически;
- **Triggered Take Snapshot** — то же, что в предыдущем случае, но вызывается не для определенных моментов времени, а по выполнению задаваемого программистом условия, относящегося к превышению допустимых ресурсов, например, размера занимаемой динамической области памяти — кучи (heap), числа загруженных классов или числа сгенерированных поколений объектов (generations). Под поколением понимают объекты, сгенерированные во время одного вызова метода. Во время следующего вызова могут либо использоваться объекты, созданные в предыдущем вызове, либо генерироваться новые — второе поколение и т. д. Если в цикле во время каждого вызова в большом количестве генерируются новые объекты и не высвобождаются, то наблюдается утечка памяти.

К сожалению, в профилировщике в варианте **Triggered Take Snapshot** отсутствует возможность вызова среза при выполнении условия, задаваемого пользователем с помощью значений переменных и функций. Также невозможно вывести в окно **Watches** значения переменных и функций в точке профилирования. Если бы такие возможности были добавлены, это сделало бы профилировщик хорошим дополнением к отладчику.

Итак, выберем вариант **Take Snapshot** и нажмем кнопку **Next >**.

Имеет смысл установить флажок **Reset results after taking snapshot** (Сбросить результаты после получения среза профилирования) для того, чтобы в следующей точке профилирования не происходило суммирования с результатами, полученными для первой точки (рис. 17.56). Но если мы хотим накапливать статистику по разным точкам профилирования, то этот флажок ставить не нужно.

В нижней части диалога указывается строка исходного кода (line), которой соответствует точка профилирования. По умолчанию стоит опция **End**, означающая, что профилирование будет происходить после окончания отмеченной строки. Но если мы хотим, чтобы профилирование проходило перед началом вызова строки, нужно переключить на вариант **Begin**. Вариант с профилированием перед началом вызова строки обычно используется для точки типа **Reset Results**, которая устанавливается перед точкой типа **Take Snapshot**, относящейся к концу той же строки. Причем для варианта **Reset Results** по умолчанию задан вариант **Begin.**

После нажатия на кнопку **Finish** в редакторе исходного кода в служебном столбце (в котором показываются точки остановки и предупреждения) появляется значок точки профилирования (рис. 17.57).

А рядом с окном **Output** появляется новая вкладка **Profiling Points** (рис. 17.58), где выводится список точек профилирования.

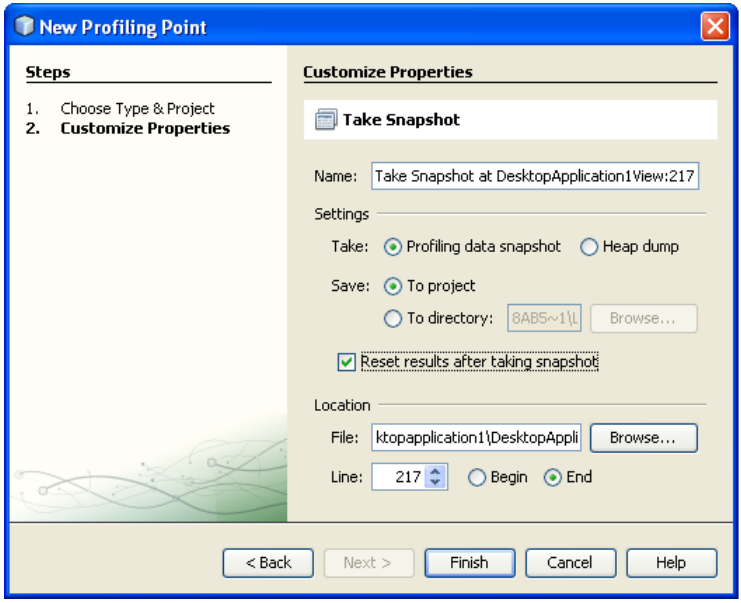


Рис. 17.56. Установка точки профилирования (шаг 2)

```
String replaceSubS(String s, String subS, String newSu
    String newS = "";
    int i = 0;
    int old_i = 0;
    i = s.indexOf(subS, i);
    while (i > -1) {
        newS = newS + s.substring(old_i, i) + newSubS;
        old_i = i + subS.length();
        i = s.indexOf(subS, old_i);
    }
    newS = newS + s.substring(old_i, s.length());
    return newS;
}
```

Рис. 17.57. Точка профилирования в редакторе исходного кода

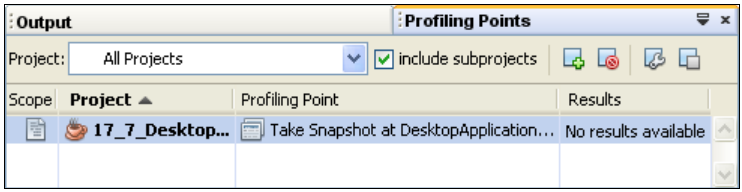


Рис. 17.58. Вкладка **Profiling Points**

И именно в этом окне можно менять параметры точек профилирования:

- **Add Profiling Point** — добавлять новые;
- **Remove Profiling Point(s)** — удалять ненужные (выделив их в списке);

- **Edit Profiling Point** — редактировать имеющиеся;
- **Enable/Disable Profiling Point(s) —** отключать (без удаления) и снова подключать имеющиеся точки.

Этой панелью инструментов пользоваться гораздо удобнее, чем добираться до соответствующего пункта в контекстном меню. Если вы редактируете какую-то строку исходного кода, достаточно одного щелчка мыши по кнопке , чтобы вызвать диалог добавления точки профилирования для этой строки. И одного щелчка по кнопке , чтобы убрать точку профилирования.

После успешного прохождения профилирования в поле **Results** появляются гиперссылки report (отчет) на отчеты по результатам для каждой точки профилирования (рис. 17.59).

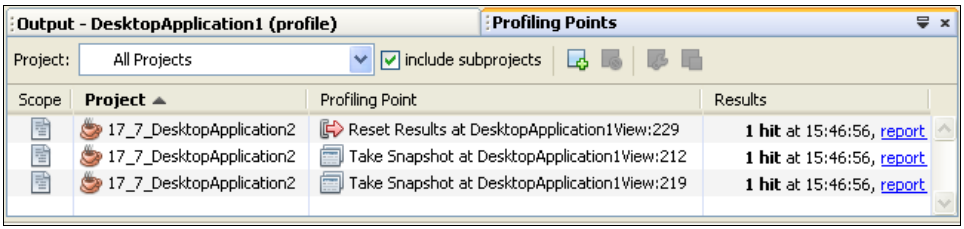


Рис. 17.59. Вкладка **Profiling Points** после профилирования

Нажатие на гиперссылку report приводит к появлению отчета. Для точки Reset этот отчет малоинформативен и содержит только информацию о времени достижения данной точки с точностью до 1 мс. А для моментального снимка (snapshot) отчет содержит как информацию о времени достижения данной точки, так и гиперссылку **open snapshot** (рис. 17.60). Нажатие на данную гиперссылку открывает окно с результатами профилирования (рис. 17.61).

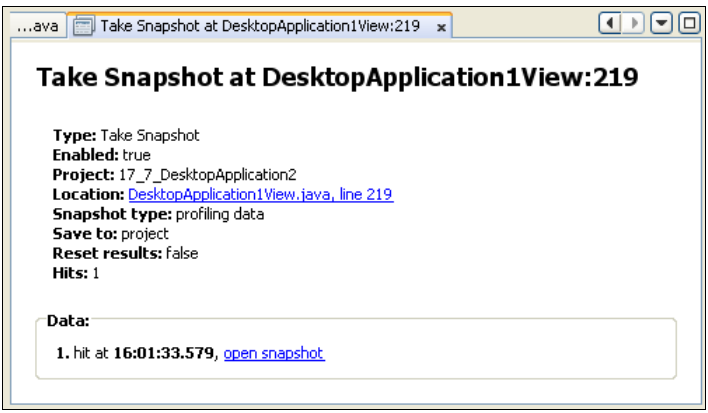


Рис. 17.60. Отчет для точки профилирования

К сожалению, профилировщик в NetBeans 6.5 не всегда работает с точками профилирования. При наличии точек профилирования иногда во время профилирования возникают исключительные ситуации, иногда моментальные снимки начинают генерироваться

в цикле (хотя должны получаться один раз), а также бывает, что среда NetBeans вообще закрывается. Если же не устанавливать точки профилирования, профилировщик функционирует нормально.

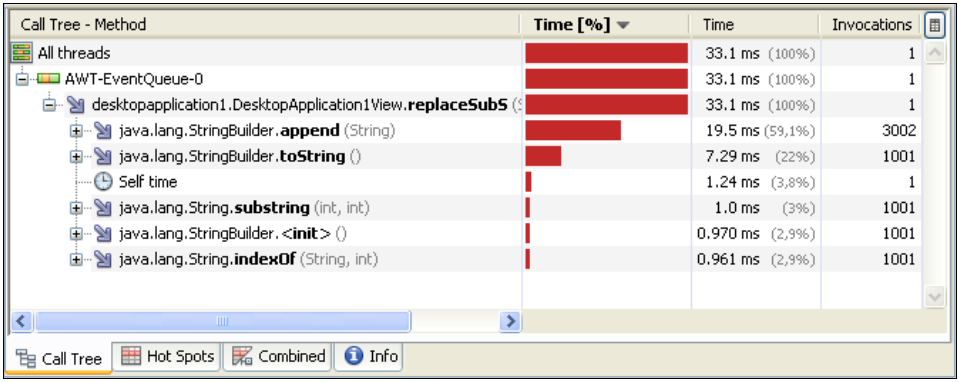


Рис. 17.61. Результаты для точки профилирования

17.8. Нахождение проблем с использованием памяти — телеметрия

Перейдем, наконец, к изучению проблем с использованием памяти в нашем методе.

Упростим задачу. Не будем показывать сгенерированный и замененный текст в компоненте `JTextArea`, а введем глобальную переменную `generatedS`, в которой станем хранить сгенерированный текст и проводить в нем замены. Но при этом усугубим проблему с потреблением памяти, сгенерировав 10 000 чисел, разделенных запятыми (пример 17_8_DesktopApplication1, листинг 17.19).

Листинг 17.19. Вариант обработчиков нажатия кнопок для генерации и замены символов в переменной `generatedS`

```
String generatedS = "";

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    String subS = jTextField1.getText();
    String newSubS = jTextField2.getText();
    replaceSubS(generatedS, subS, newSubS);
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    int i = 0;
    int COUNT = 10000;
    String s = "";
    for (i = 1; i <= COUNT; i++) {
        s = s + " 1," + i;
    }
    generatedS = s;
}
```

Запустим профилировщик и выберем режим **Monitor** (Просмотр), а не **CPU**, который мы применяли раньше. Запустим генерацию данных и сразу после ее окончания остановим профилирование нажатием на кнопку остановки.

После окончания сессии профилирования в окне **Profiler** нажмем кнопку **Telemetry Overview** (Просмотр результатов телеметрии) (на рис. 17.62 иконка телеметрии выделена пунктирной рамкой).

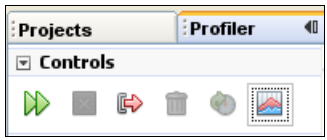


Рис. 17.62. Окно телеметрии

В результате рядом с окном **Output** появится вкладка **VM Telemetry Overview**, в которой показывается картина с обзором трех окон (рис. 17.63):

- использование динамической области памяти компьютера (кучи — heap);
- работа сборщика мусора (Garbage Collector — GC);
- потоки выполнения и загруженные классы.

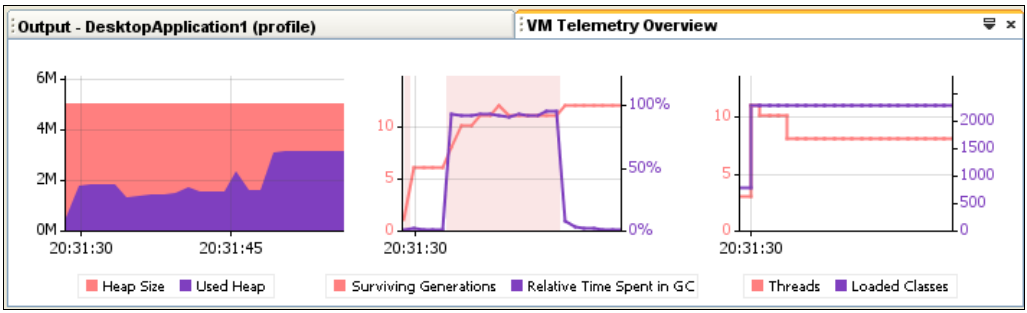


Рис. 17.63. Просмотр результатов телеметрии

Каждое из этих окон можно увидеть в главном окне среды NetBeans, если в окне **Profiler** прокрутить полосу прокрутки и в разделе **View** (Просмотр) нажать иконку **VM Telemetry** (Удаленное наблюдение за виртуальной машиной) (рис. 17.64).

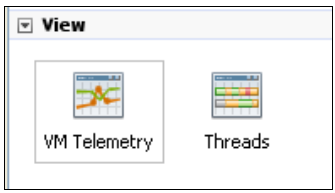


Рис. 17.64. Раздел **View**

Для нас сейчас наиболее интересно среднее окно, где отображается время, которое тратит сборщик мусора. Можно заметить, что в момент нажатия на кнопку генерации данных активность сборщика мусора скачком возрастает с близкого к нулю значения до большой величины и остается практически постоянной вплоть до окончания генерации данных, после чего скачком снижается практически до нулевого уровня. Это означает,

что при генерации данных постоянно возникают объекты, становящиеся мусором, и сборщик мусора их активно удаляет. Растет и число выживших поколений (Surviving Generations).

Какие объекты мы создаем и тут же теряем? Строки! Ведь каждое присваивание

```
s = s + " 1," + i;
```

порождает новый объект-строку, и мы создаем 10 000 новых объектов! А вот причина роста числа выживших поколений не может быть связана с данным присваиванием, т. к. ссылка на прежний объект-строку при присваивании теряется, а этот объект попадает в распоряжение сборщика мусора. Но по сравнению с 10 000 десяток поколений — совсем небольшое число, и такая малая утечка памяти нас может не волновать (хотя если бы она постоянно увеличивалась, то со временем это могло бы стать проблемой). Проведем эксперимент, несколько раз подряд нажимая на кнопку генерации данных после окончания очередной генерации.

Мы видим, что ситуация стабилизируется — число поколений не имеет тенденции к неограниченному росту с увеличением числа нажатий на кнопку генерации данных (рис. 17.65). Значит, утечки памяти нет.

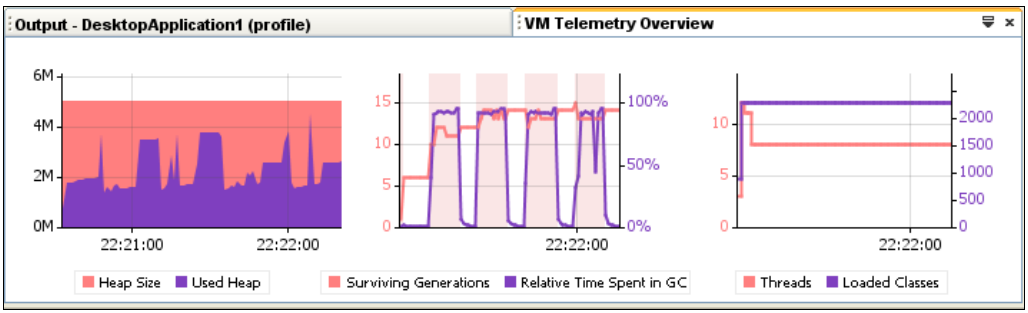


Рис. 17.65. Телеметрия при многократном повторении генерации строковых данных

Попробуем теперь проверить наше предположение о генерации объектов-строк. Заменим строковую переменную, в которой накапливается результат, численной, и добавлять будем не строки, а числа (пример 17_8_DesktopApplication2, листинг 17.20).

Листинг 17.20. Функция суммирования целых чисел и обработчик события для ее вызова

```
int generatedI = 0;

int generateSum() {
    int i;
    int COUNT = 10000;
    int sum = 0;
    for (i = 1; i <= COUNT; i++) {
        sum = sum + 1;
    }
    return sum;
}
```

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    generatedI = generateSum();
}
```

Результаты весьма убедительны — телеметрия показывает, что сборщик мусора почти не загружен, а память расходуется в два раза экономнее, без пиковых нагрузок и ее активного высвобождения сборщиком мусора (рис. 17.66).

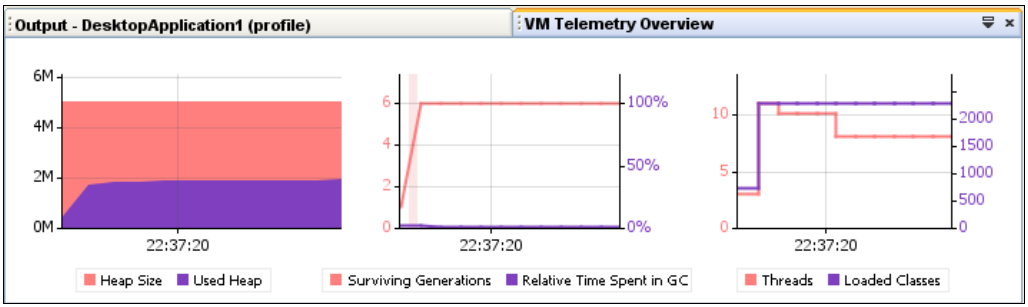


Рис. 17.66. Телеметрия при многократном повторении генерации числовых данных

Кроме того, мы обнаружим, что программа стала работать на несколько порядков быстрее. Если при использовании строк генерация данных занимала около 12 с, то теперь она происходит настолько быстро, что визуально это уже невозможно заметить, и нам необходимо воспользоваться профилировщиком (мы уже умеем выяснять, сколько времени выполняется тот или иной метод). Обработчики событий в NetBeans 6.5 могут быть корневыми методами, но мы вынесли генерацию данных в отдельную подпрограмму для того, чтобы не смешивать численный алгоритм (бизнес-логику) и визуализацию (работу с кнопкой как элементом пользовательского интерфейса).

Результат профилирования оказывается ошеломляющим — 10 000 чисел суммируются всего за 0,59 мс. Таким образом, скорость выполнения алгоритма увеличилась в 20 000 раз!

17.9. Профилирование использования памяти классами

Теперь рассмотрим еще одну методику профилирования — срез профиля использования памяти. Он виден в режиме профилирования **Memory** (Память). Можно предложить три способа нахождения объектов, генерирующихся на интересующем нас участке кода.

Первый вариант — задать две точки профилирования: **Reset Results** в начале метода и **Take Snapshot** в конце и получить срез автоматически.

Второй вариант — после запуска приложения нажать иконку **Reset Collected Results**, сбросив все данные в нуль. Затем нажать в приложении кнопку **Сгенерировать данные**, а после выполнения этой генерации выполнить срез памяти.

Третий вариант — сделать два среза вручную в режиме профилирования памяти и найти разность между ними. Первый срез можно сделать после запуска приложения (икон-

ка **Take Snapshot** в окне **Profiler** в разделе **Profiling Results**), второй — после нажатия в приложении кнопки **Сгенерировать данные**.

Но сначала нам необходимо отключить сборщик мусора, чтобы он не мешал наблюдениям. Это делается так: после запуска профилировщика создается пользовательская конфигурация путем нажатия на гиперссылку **Create Custom...** (Задать по выбору) (рис. 17.67). Появляется диалоговая форма (рис. 17.68), в которой выбираем создание конфигурации на основе уже имеющейся **New Memory Analysis**. После нажатия на кнопку **OK** мы возвращаемся в первоначальную главную форму настроек профилировщика (рис. 17.67), но теперь нам доступна гиперссылка **Advanced settings** (Дополнительные настройки), находящаяся в правом нижнем углу. После нажатия на нее мы попадаем в форму дополнительных настроек, где и отключаем сборщик мусора — снимаем флажок **Run garbage collection when getting memory results** (Запускать сборщик мусора во время получения результатов по памяти).

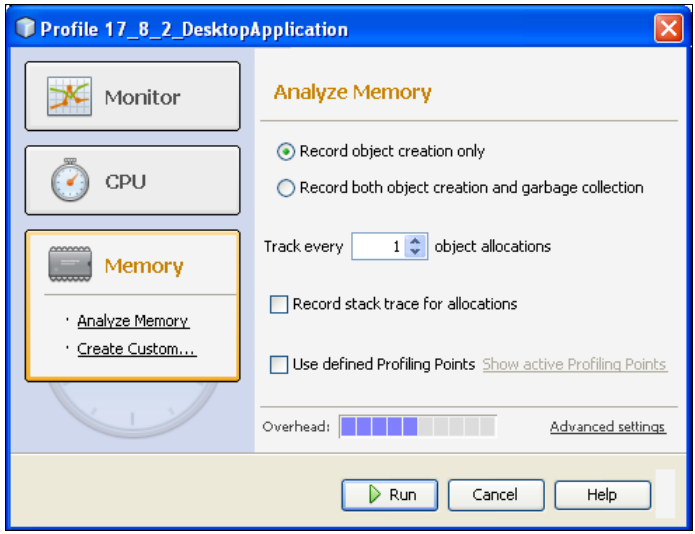


Рис. 17.67. Создание пользовательской конфигурации профилировщика (шаг 1)

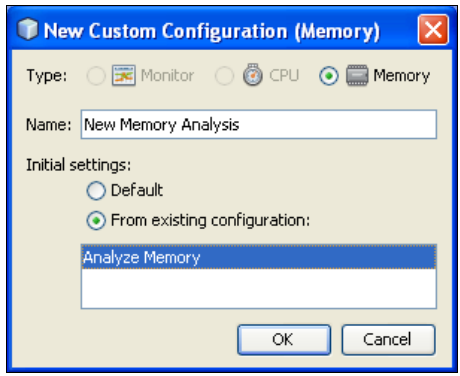


Рис. 17.68. Создание пользовательской конфигурации профилировщика (шаг 2)

Рассмотрим первый вариант профилирования — с заданием точки профилирования **Reset Results** в начале метода и **Take Snapshot** в конце (рис. 17.69).

```
private void jButton2ActionPerformed(java.awt.event.Act
    int i,j = 0;
    int COUNT = 10000;
    String s = "";
    for (i = 1; i <= COUNT; i++) {
        s = s + " 1," + i;
    }
    generatedS = s;
}
```

Рис. 17.69. Точки профилирования

Зададим точки профилирования и запустим приложение в режиме профилирования. Нажмем кнопку **Сгенерировать результаты** и закроем приложение.

Будет автоматически создан срез памяти на момент закрытия приложения (рис. 17.70). Но он не имеет никакого отношения к заданным нами точкам профилирования — по-видимому, разработчики профилировщика сочли полезным каждый раз при выходе из приложения напоминать программистам о том, как выглядит картина распределения классов по объему занимаемой памяти.

DesktopApplication1 View.java | Memory: 01:54:56 PM

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		284 298 112 B (100%)	41 834 (57,4%)
java.lang.**String**		28 608 B (0%)	11 232 (15,4%)
java.lang.**StringBuilder**		17 184 B (0%)	10 227 (14%)
java.lang.reflect.**Method**		8 560 B (0%)	1 009 (1,4%)
java.lang.**Class[]**		3 296 B (0%)	1 788 (2,5%)
java.lang.**Object[]**		2 384 B (0%)	732 (1%)
byte[]		2 280 B (0%)	273 (0,4%)
java.lang.ref.**WeakReference**		2 160 B (0%)	817 (1,1%)
java.util.**HashMap$Entry**		1 464 B (0%)	591 (0,8%)
java.lang.ref.**SoftReference**		1 376 B (0%)	407 (0,6%)
java.beans.**MethodDescriptor**		1 344 B (0%)	273 (0,4%)
java.util.**HashMap$Entry[]**		1 064 B (0%)	55 (0,1%)
java.util.**ArrayList**		960 B (0%)	368 (0,5%)
java.lang.reflect.**Constructor**		704 B (0%)	95 (0,1%)
java.beans.**PropertyDescriptor**		704 B (0%)	108 (0,1%)
java.nio.**HeapByteBuffer**		672 B (0%)	137 (0,2%)
java.nio.**HeapCharBuffer**		624 B (0%)	119 (0,2%)

Рис. 17.70. Срез профиля занимаемой классами памяти на момент выхода из приложения

Мы видим, что налицо аномалия — за время работы приложения создано более 41 000 объектов, причем они занимают 284 Мбайт памяти! Все эти объекты имеют тип `char[]` — массив символов. Откуда они берутся? Именно это мы узнаем по точкам профилирования (рис. 17.71).

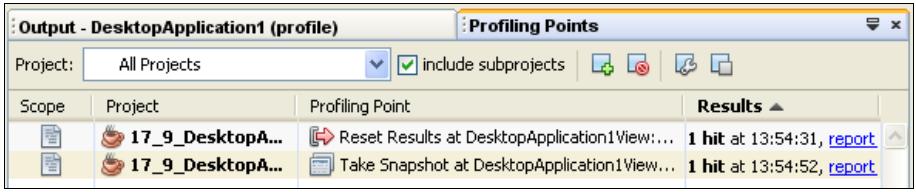


Рис. 17.71. Просмотр результатов для точек профилирования

После достижения очередной точки профилирования во время работы приложения в окне **Output** на вкладке **Profiling Points** против имени соответствующей точки профилирования вместо надписи **No results available** (Нет доступных результатов) возникает текст с информацией о полученной для этой точки информацией — числе вызовов этой точки (сделанных отчетов). При нажатии на гиперссылку **report** (отчет) в главном окне NetBeans появляется отчет (рис. 17.72).

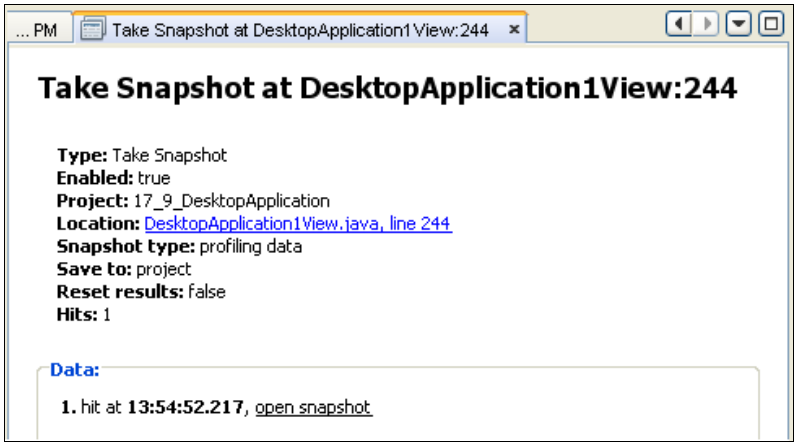


Рис. 17.72. Отчет по точке профилирования

При нажатии на гиперссылку **open snapshot**, стоящую после указания даты получения отчета, открывается срез профиля (рис. 17.73).

Данный срез похож на то, что мы имели при выходе из приложения, но в нем несколько отличаются цифры и содержится гораздо меньше классов — нет ничего лишнего, не связанного с изучаемым фрагментом кода.

Попробуем теперь второй вариант профилирования. Для этого нужно заменить тип первой точки профилирования с **Reset Results** в начале строки на **Take Snapshot**. Нам сначала придется отключить точку **Reset Results**: выделить в окне со списком точек профилирования строку с ее именем и нажать кнопку **Enable/Disable Profiling Point(s)** на панели инструментов окна **Output/Profiling Points** (см. рис. 17.58). Двойной щелчок по имени точки в этом списке приведет к тому, что откроется редактор исходного кода с курсором, установленным на соответствующей строке. Теперь в этой строке установим новую точку профилирования нужного типа и запустим приложение в режиме профилирования памяти.

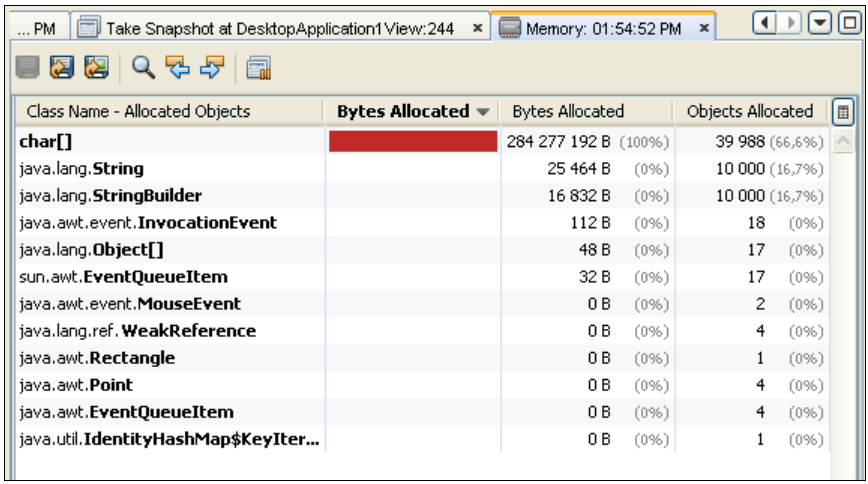


Рис. 17.73. Срез профиля использования памяти для точки профилирования

Первый срез имеет правильную форму без аномалий (рис. 17.74). Зато второй практически не отличается от среза при выходе из программы, и мы не приводим эту диаграмму.

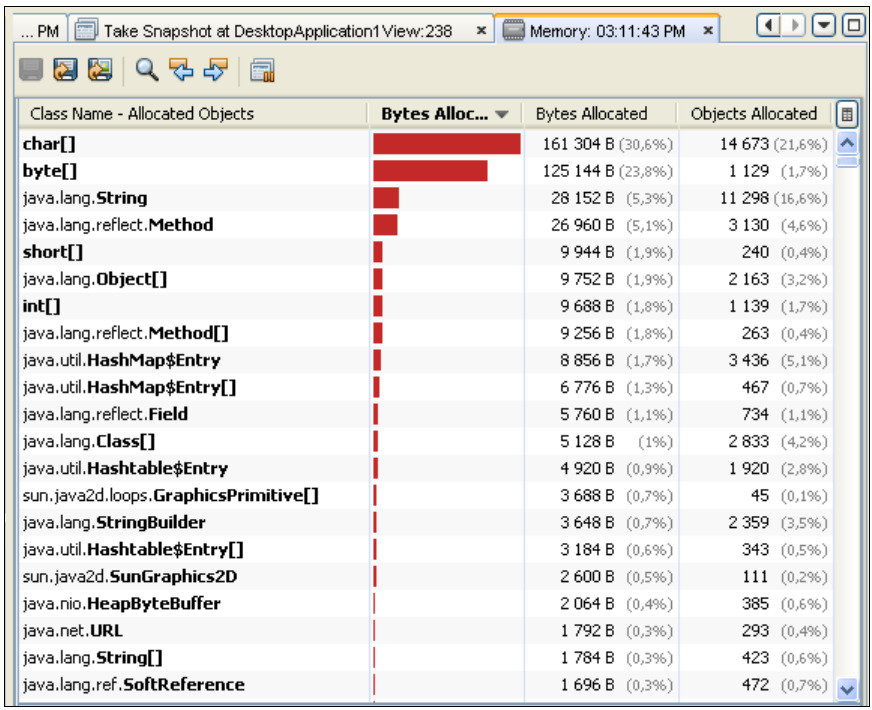


Рис. 17.74. Срез профиля в начале выполнения метода

В палитре инструментов данного окна имеется очень полезная кнопка **Computes the difference between two comparable memory snapshorts** (Вычислить разницу между

двумя сравнимыми срезами профилей памяти). Нажмем ее, выберем профиль (рис. 17.75), с каким нужно сравнивать, и получим разностное распределение (рис. 17.76). Причем вызывать сравнение необходимо из более раннего профиля, чтобы прирост показывался в виде положительных величин в верхней части диаграммы, а не отрицательных — в нижней.

Итак, мы освоили основные приемы профилирования и убедились, что работа со строками-объектами — не самая лучшая особенность Java.

Очевидно, что при грамотной работе с динамическими массивами удалось бы избежать перечисленных проблем. Появление сгенерированных в большом количестве объектов `char[]`, а не `String`, связано с тем, что компилятор Java большинство операций со строками неявно проводит с массивами символов, автоматически преобразуя в них строки. Это также совсем не лучшее решение — именно эти потерянные объекты мы и обнаруживаем в куче.

Использование класса `StringBuffer` существенно улучшает положение. Код, приведенный в листинге 17.21 (пример 17_9_DesktopApplication, обработчик нажатия на кнопку **StringBuffer**), работает несоизмеримо быстрее (0,48 мс) и занимает намного меньше памяти под лишние объекты (всего 74 Кбайт).

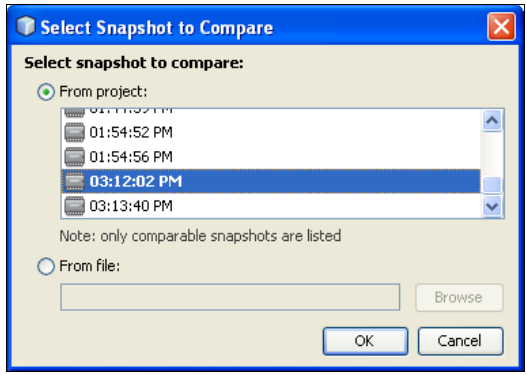


Рис. 17.75. Выбор среза для нахождения разницы с текущим

Листинг 17.21. Правильная реализация работы со строками

```
int i = 0;
int COUNT = 10000;
StringBuffer s = new StringBuffer("");
for (i = 1; i <= COUNT; i++) {
    s.append(" 1,").append(i);
}
generatedS = s.toString();
```

Таким образом, профилирование приложений позволяет выявить "больные места" и добиться лучшего понимания того, как работает приложение и что в нем действительно необходимо усовершенствовать.

Правильно выбрав средства, нам удалось повысить быстродействие программы примерно в 30 раз и уменьшить потребление памяти в 4000 раз.

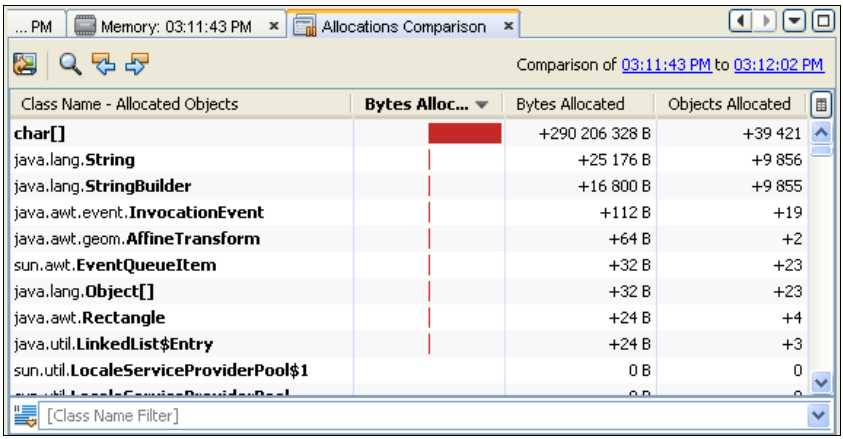


Рис. 17.76. Результат вычитания данных по двум срезам

Краткие итоги по главе

- Отладчик (debugger) — это программа для нахождения ошибок (bugs).
- Основные средства режима отладки: *точки остановки* (breakpoints) и команды *режима пошагового выполнения*.
- В режиме отладки программа, дойдя до очередной точки остановки, прерывает обычный режим работы и останавливается, переключаясь в режим пошагового выполнения.
- В окне **Watches** во время отладки можно просматривать значения переменных и выражений.
- В программировании действует *принцип "презумпции виновности"*: если ты не можешь доказать, что предъявляемая тобой программа правильна, то считается, что она содержит ошибки.
- *Профилирование* — это оптимизация приложения, выполняемая для повышения его производительности и уменьшения потребляемых им ресурсов, путем нахождения узких мест — "горячих" точек (hot spots), потребляющих основные ресурсы при работе приложения, и участков кода с неэффективным использованием памяти.
- В NetBeans имеются три основных режима профилирования:
  - телеметрия;
  - профилирование загрузки процессора (CPU) и выявление "горячих" точек;
  - профилирование использования памяти.
- Возможно профилирование:
  - всего приложения в целом;
  - отдельных методов;
  - участков кода с использованием точек профилирования.

Типичные ошибки

- Начинающие программисты применяют только режим перехода к следующему оператору (<F8> — **Step Over**) и забывают про режим трассировки (<F7> — **Step Into**) для входа внутрь вызываемой функции.
- На начальной стадии освоения отладчика NetBeans программисты, привыкшие работать в других средах, не учитывают существование режимов <Shift>+<F8> — **Step Over Expression** (шаг вперед к вычислению очередного выражения) и <Ctrl>+<F7> — **Step Out** (выход наружу из текущего метода).
- Точка зрения, что в Java вообще нет проблемы утечки памяти.
- Точка зрения, что в Java вообще отсутствует проблема создания "мусора" в памяти, т. к. этот "мусор" автоматически удаляется.
- Одна из самых распространенных ошибок при работе с памятью — создание в цикле промежуточных объектов (они обычно возникают при присваивании переменным типа `String` новых значений).
- При оптимизации приложения по скорости и потребляемым ресурсам вместо нахождения с помощью профилировщика "горячих" точек строят чисто умозрительные предположения о том, что должно влиять на эффективность работы приложения. После чего пытаются усовершенствовать те его части, которые практически не влияют ни на скорость выполнения, ни на потребляемые ресурсы. В результате программа изобилует "трюками" и из-за этого ненадежна, трудна для восприятия и плохо модифицируется.

Задания

- Осуществить разные варианты профилирования метода `replaceSubS`.
- Выполнить профилирование метода `setText` компонента `JTextArea` для текста, сгенерированного для 10 000 целых чисел, если после каждого числа добавляется перевод на новую строку. Проверить, сколько времени занимает установка теста, появляются ли при вызове метода промежуточные объекты и какой объем памяти требуется.

Заключение

Цель данного курса по языку Java и среде NetBeans — дать основные знания, необходимые для перехода к профессиональному программированию на языке Java. Он, конечно, не охватывает всех сторон огромной области человеческой деятельности, основанной на Java-технологиях, но после освоения материалов, изложенных в книге, и выполнения заданий к каждой главе читатель сможет самостоятельно продвигаться дальше. Для него уже не будут звучать загадочно слова "перегрузка метода", "полиморфизм", "рефакторинг", "обратное проектирование", "паттерн проектирования", "мидлет", "сервлет", "тестовая фикстура" или "корневой метод профилирования".

Мы надеемся, что читатель сможет оценить красоту хорошо написанных программ, научится самостоятельно принимать правильные решения о построении иерархии классов, использовать интерфейсы и композицию, писать полиморфный код, группировать наборы классов в пакеты, что он будет не простым кодировщиком, а настоящим архитектором программных проектов, пусть пока еще не имеющим достаточного опыта.

И если на начальной стадии изучения книги вам казалось, что Java и NetBeans — это блистательный мир сложного, но постепенно постигаемого совершенства, а к концу стало выясняться, что в этой области далеко не все идеально — не расстраивайтесь! Здесь многое можно и нужно совершенствовать. Причем автор считает, что это как раз очень хорошо — значит, в данной области кипит жизнь и есть куда двигаться дальше.

Вы можете выбирать лучшее и отвергать то, что не оправдывает ожиданий, можете предлагать свои решения проблем, причем самое главное — чтобы учеба и работа доставляли удовольствие, а не были тяжелой скучной обязанностью!

ПРИЛОЖЕНИЕ

Описание прилагаемого DVD

Прилагаемый к книге DVD содержит свободно предоставляемые Sun Microsystems (Oracle) наиболее свежие на момент верстки книги дистрибутивы JDK 6u22 и NetBeans 6.9.1; документацию к JDK 6; соответствующие лицензии для дистрибутивов и документации; разработанные автором проекты, приведенные в книге в качестве примеров, а также дистрибутивы JDK 6u11 и NetBeans 6.5, прилагавшиеся на компакт-диске ко второму изданию.

Полужирным выделены те дистрибутивы и примеры, которые отсутствовали в предыдущих изданиях (примеры к *главе 15* во втором издании книги упоминались, но на сопроводительном компакт-диске не были представлены).

- Папка 1.jdk-6u11 содержит дистрибутив JDK 6, обновление номер 11 для соответствующих операционных систем и аппаратных платформ:
  - файл jdk-6u11-windows-i586-p.exe — дистрибутив для 32-разрядных i586-совместимых процессоров (Pentium 4, Celeron, Core 2 Duo и т. п.) и операционной системы Windows. Работоспособен и для 64-разрядных процессоров;
  - файл jdk-6u11-linux-i586.bin — дистрибутив для 32-разрядных i586-совместимых процессоров и операционной системы Linux. Работоспособен и для 64-разрядных процессоров.
- Папка 2.jdk-6-doc содержит zip-архив с документацией к JDK 6 (файл jdk-6u10-docs.zip).
- Папка 3.netbeans-6.5 содержит дистрибутивы NetBeans 6.5 для операционных систем Windows (файл netbeans-6.5-ml-windows.exe) и Linux (файл netbeans-6.5-ml-linux.sh), а также файл uml6.zip с архивом модуля UML-проектирования.
- **Папка 4.netbeans-6.9** содержит дистрибутивы NetBeans 6.9.1 для операционных систем Windows (файл netbeans-6.9.1-ml-windows.exe) и Linux (файл netbeans-6.9.1-ml-linux.sh), инструкцию по инсталляции на английском языке "NetBeans IDE 6.9.1 Installation Instructions.html" (необходима только для пользователей Linux), а также файл uml.zip с архивом модуля UML-проектирования для NetBeans (после инсталляции NetBeans из архива uml.zip необходимо извлечь папку uml и скопировать в папку среды NetBeans, например, под Windows в папку C:\Program Files\NetBeans 6.9).
- **Папка 5.jdk-6u22** содержит дистрибутив JDK 6, обновление номер 22 для соответствующих операционных систем и аппаратных платформ:

- файл jdk-6u22-windows-i586.exe — дистрибутив для 32-разрядных i586-совместимых процессоров (Pentium 4, Celeron, Core 2 Duo и т. п.) и операционной системы Windows;
- jdk-6u22-windows-x64.exe — дистрибутив для 64-разрядных версий операционной системы Windows (установленных на компьютерах с 64-разрядным процессором);
- файлы jdk-6u22-linux-i586-rpm.bin — дистрибутив для 32-разрядных i586-совместимых процессоров и операционной системы Linux, jdk-6u22-linux-i586.bin — другой вариант того же дистрибутива;
- файл jdk-6u22-linux-x64-rpm.bin —дистрибутив для 64-разрядных версий операционной системы Linux (установленных на компьютерах с 64-разрядным процессором), jdk-6u22-linux-x64.bin — другой вариант того же дистрибутива.
- Папка NetBeansProjects содержит примеры проектов NetBeans, разработанные В. В. Монаховым.

Каждый проект (табл. П1) располагается в отдельной папке, находящейся внутри папки NetBeansProjects и папки, соответствующей номеру главы (Chapter02, Chapter04, Chapter07 и т. д.).

Имя папки проекта начинается с номера главы, после чего через символ подчеркивания идет номер раздела, к которому относится пример. Если в главе более 9 разделов, то номера, соответствующие первым девяти разделам, начинаются с нуля (01, 02 и т. д.). Затем через точку или подчеркивание идет название примера. Если у одного и того же примера имеется несколько вариантов (последовательные стадии разработки примеров), то названия примеров заканчиваются номерами 1, 2, 3 и т. д. (например, папки с проектами 02_09_LookAndFeel_1, 02_09_LookAndFeel_2 и т. п.).

Помимо файлов, образующих проект, в папках NetBeansProjects\Chapter16\16_6_JNI_example1 и NetBeansProjects\Chapter16\16_6_JNI_example2 находится файл CPP_MultiplyLibrary.dll скомпилированной библиотеки DLL, что позволяет проверить работу приложения Java с native-методами при его запуске непосредственно из среды NetBeans.

Кроме того, в папке NetBeansProjects есть несколько файлов с изображениями, нужными для использования в некоторых примерах (например, в 02_11_GUI_example3).

Для активной работы с примерами (чтобы иметь возможность компилировать или вносить в них изменения) скопируйте каталоги проектов на жесткий диск компьютера.

Внимание!

Нельзя одновременно открывать два и более проектов с UML-диаграммами. Перед открытием нового проекта предыдущий следует закрывать, т. к. при одновременном открытии двух и более UML-проектов среда NetBeans может работать некорректно.

Таблица П1. *Проекты*

Папка проекта	Описание
02_03_JavaApplication1	Простейшее приложение Java с выводом в окно консоли фразы "Привет!"
02_04_console	Простейшее приложение Java с выводом в окно консоли фразы "Привет!" и ожиданием нажатия пользователем клавиши <Enter>

Таблица П1 *(продолжение)*

Папка проекта	Описание
02_06_GUI_example	Пример приложения Java с графическим пользовательским интерфейсом
02_07_GUI_example1	Пример приложения Java с графическим пользовательским интерфейсом (кнопка)
02_07_GUI_example2	Пример приложения Java с графическим пользовательским интерфейсом (кнопка с HTML-форматированной надписью)
02_07_GUI_example3	Пример приложения Java с графическим пользовательским интерфейсом (назначенные обработчики событий и диалоги с сообщениями)
02_09_LookAndFeel_1	Пример простого приложения Java с графическим пользовательским интерфейсом (стиль Metal)
02_09_LookAndFeel_2	Пример простого приложения Java с графическим пользовательским интерфейсом (стиль System и позиционирование по центру экрана)
02_09_LookAndFeel_3	Пример простого приложения Java с графическим пользовательским интерфейсом (стиль Motif и позиционирование по центру экрана)
02_09_LookAndFeel_4	Пример приложения Java с графическим пользовательским интерфейсом и большим числом компонентов (стиль Metal)
02_09_LookAndFeel_5	Пример простого приложения Java с графическим пользовательским интерфейсом и большим числом компонентов (стиль System)
02_09_LookAndFeel_6	Пример простого приложения Java с графическим пользовательским интерфейсом и большим числом компонентов (стиль Motif)
02_11_GUI_example1	Редактирование меню экранной формы (начальное состояние). Пунктам меню назначены акселераторы. Назначена горячая клавиша <S> (комбинация клавиш <Ctrl>+<S>)
02_11_GUI_example2	Редактирование меню экранной формы (удалены лишние пункты и добавлены новые с элементами независимого и зависимого выбора)
02_11_GUI_example3	Редактирование меню экранной формы (пунктам меню назначены изображения)
02_11_GUI_example4	Контекстное меню, назначенное кнопке
02_12_GUI_example1	Начальный вариант проекта с графическим пользовательским интерфейсом до добавления к проекту пользовательского класса
02_12_GUI_example2	Проект с графическим пользовательским интерфейсом после добавления к проекту пользовательского класса `Figure`
02_13_EnumExample	Пример относительно сложного приложения с разнообразными документационными комментариями
02_14_OptionPane	Пример вывода диалогов с сообщениями
02_15_Layout1_Free	Форма с менеджером размещения `Free Design`

Таблица П1 *(продолжение)*

Папка проекта	Описание
02_15_Layout2_Absolute	Форма с менеджером размещения `Absolute`
02_15_Layout3_GridBag	Форма с менеджером размещения `GridBag`
02_15_Layout4_Null	Форма с менеджером размещения `Null`
02_16_DesktopApplication1	Пример приложения Java с графическим пользовательским интерфейсом на основе заготовки DesktopApplication
04_1_integerCalc	Пример, иллюстрирующий проблему целочисленного переполнения и зависимости результата от порядка множителей
05_5_consol_example	Пример, иллюстрирующий работу оператора инкремента (постфиксная и префиксная формы)
05_7_integral	Пример, иллюстрирующий проблему вещественного счетчика цикла
05_8_effectiveCalc	Сравнение эффективности организации циклов при вычислениях в формате с плавающей точкой
05_9_effectiveCalc	Сравнение эффективности организации циклов при вычислениях с целыми числами
07_1_UMLProject1	Пример UML-диаграммы классов `Figure` и `Dot`
07_3_UML_Figures0	UML-диаграмма иерархии фигур в варианте одного поколения наследования
07_3_UML_Figures1	UML-диаграмма иерархии фигур в варианте иерархического наследования с уровнями абстрактных классов
07_3_UML_Figures2	Предыдущая диаграмма после применения `Hierarchical Layout`
07_4_1_UML_Figures	Диаграмма иерархии фигур с показом атрибутов и методов
07_4_2_UML_Figures	Диаграмма иерархии фигур с показом атрибутов и методов (появление геттеров и сеттеров после добавления полей)
07_4_3_UML_Figures	Диаграмма, используемая для генерации кода Java-проекта 07_4_3_JavaLibrary1
07_4_3_JavaLibrary1	Код Java, сгенерированный по диаграммам проекта 07_4_3_UML_Figures
07_5_JavaLibrary_Figure	Код Java, используемый для генерации диаграммы проекта 07_5_JavaLibrary_Figure-Model
07_5_JavaLibrary_Figure-Model	Диаграмма, сгенерированная по коду Java-проекта 07_5_JavaLibrary_Figure
07_6_refactoring1	Проект, предназначенный для рефакторинга
07_6_refactoring2	Проект после рефакторинга
08_3_for_each	Пример использования оператора `for...each`
08_5_JavaEnumExample	Пример работы с перечислениями
08_6_DateTime	Пример работы с календарем, датой и временем
08_7_Graphics1	Пример работы с графикой

Таблица П1 *(продолжение)*

Папка проекта	Описание
08_7_Graphics2	Пример динамического добавления компонентов и работы с графикой (переопределение метода `paint()`)
08_9_File1	Заготовка приложения для примеров работы с файлами
08_9_File2	Пример выбора файла с помощью файлового диалога и задания фильтра к этому диалогу
08_9_File3	Пример выбора нескольких файлов или папок, а также пример чтения списка файлов и папок, содержащихся в папке
08_9_File4_Stream	Пример работы с файловыми потоками и файлами с произвольным доступом
08_9_File5_Stream	Пример задания текущей папки для диалога, а также чтения и записи данных в различных форматах
08_9_File6_Zip	Пример работы с архивами gzip и zip, а также разработанный автором класс `Files` для поддержки эффективной и удобной работы с файлами
10_6_Threads	Пример работы с потоками выполнения
10_7_Lock	Пример класса `FIFOMutex`
11_2_applets	Пример исходного кода трех апплетов и HTML-документов, использующих эти апплеты
11_2_Applet1	Результат создания проекта с апплетом
11_3_ServletsExamples_1	Пример простейшего англоязычного сервлета
11_3_ServletsExamples_2	Пример русификации сервлета
11_4_JSPExamples	Пример серверного проекта JSP
11_5_JSTL_Examples	Пример серверного проекта JSTL
13_01_DesktopApplication1	Заготовка DesktopApplication
13_06_DesktopApplication1	Реализация в приложении DesktopApplication метода `doInBackground()`
13_07_DesktopApplication1	Заготовка DesktopApplication (начальное состояние создаваемого проекта)
13_07_DesktopApplication2	В заготовке DesktopApplication пункту меню **Save** и кнопке **Save** панели инструментов назначена акция `save`. Акции `save` назначены иконки. Реализован метод `doInBackground()`
13_07_DesktopApplication3	Видимость работы индикации в приложении DesktopApplication
13_08_DesktopApplication1	Частично локализованное приложение DesktopApplication
13_08_DesktopApplication2	Начало полноценной локализации приложения DesktopApplication
13_09_DesktopApplication1	Измерение времени и досрочное прекращение выполнения задания в DesktopApplication
13_10_DesktopApplication1	Пример копирования текстовой и графической информации из буфера обмена в приложение

Таблица П1 *(продолжение)*

Папка проекта	Описание
13_10_DesktopApplication2	Пример копирования изображения из приложения в буфер обмена
13_11_DesktopApplication1	Полноценная работа индикации в многопоточном приложении DesktopApplication
14_2_MobileApplication1	Пример простейшего мидлета "Hello, World!"
14_4_MobileApplication1	Русифицированный мидлет "Здравствуй, мир!"
14_5_CrossesZeros1	Заготовка мидлета "Крестики-нолики"
14_5_CrossesZeros2	Заготовка мидлета "Крестики-нолики" (добавлена кнопка **Ходить**)
14_6_CrossesZeros1	Мидлет "Крестики-нолики" (добавлен код бизнес-логики)
14_6_CrossesZeros2	В мидлете "Крестики-нолики" решена проблема обновления экрана (но не очень хорошим путем)
14_7_MobileItemsLibrary	Создание собственного компонента в Java ME
14_7_CrossesZeros1	Создание собственного компонента непосредственно в мидлете "Крестики-нолики" (заготовка)
14_7_CrossesZeros2	Полноценное использование разработанного компонента в мидлете "Крестики-нолики"
14_9_MobileApplication	Пример мидлета с переключением экрана и списком выбора
15_2_DesktopApplication1	Заготовка собственного компонента
15_2_DesktopApplication2	Заготовка собственного компонента и сопроводительный класс BeanInfo
15_3_DesktopApplication	Собственный компонент `JTitledPanel` со свойством `title`
15_4_DesktopApplication	Компонент `JTitledPanel` с добавленным свойством `titleShift`
15_5_DesktopApplication1	Компонент `JTitledPanel` с добавленным событием `TextValueChanged` для свойства `title`
15_5_DesktopApplication2	Предыдущий пример с назначенным обработчиком события `TextValueChanged`
16_5_JNI_example1	Пример приложения, в котором имеется простейший native-метод
16_5_CPP_ex1_Cygwin	Проект C++, в котором в библиотеке DLL реализован native-метод для приложения 16_5_JNI_example1. Настроен для использования коллекции Cygwin
16_5_CPP_ex1_MinGW	Проект C++, в котором в библиотеке DLL реализован native-метод для приложения 16_5_JNI_example1. Настроен для использования коллекции MinGW
16_5_JNI_example2	Пример приложения, в котором имеется два native-метода, один из которых требует выделения места под массив в динамической области памяти

Таблица П1 *(окончание)*

Папка проекта	Описание
16_5_CPP_ ex2_Cygwin	Проект C++, в котором в библиотеке DLL реализованы native-методы для приложения 16_5_JNI_example2. Настроен для использования коллекции Cygwin
16_5_CPP_ ex2_ MinGW	Проект C++, в котором в библиотеке DLL реализованы native-методы для приложения 16_5_JNI_example2. Настроен для использования коллекции MinGW
17_1_DesktopApplication	Пример приложения с ошибочной реализацией алгоритма (для иллюстрации отладки приложения)
17_3_DesktopApplication	Усовершенствованный пример приложения с ошибочной реализацией алгоритма (для иллюстрации отладки приложения)
17_4_tests_1	Приложение, для которого будут создаваться тесты
17_4_tests_2	Приложение со сгенерированной заготовкой тестового пакета
17_4_tests_3	Приложение с простейшим тестом
17_4_tests_4	Приложение с двумя тестами
17_4_tests_5	Приложение с двумя тестами после исправления класса в основном приложении (доступ к полям только через сеттеры)
17_4_tests_6	Добавление параметров аннотации для тестового метода
17_4_tests_7	Пример использования тестовых фикстур
17_5_DesktopApplication	Профилируемое приложение
17_6_DesktopApplication	Добавление вывода в консоль в профилируемом приложении
17_7_DesktopApplication1	Профилируемое приложение с неправильно организованной работой со строками
17_7_DesktopApplication2	То же приложение с назначенным корневым методом для профилирования
17_8_DesktopApplication1	Профилируемое приложение с неправильно организованной работой со строками (пример для просмотра телеметрии)
17_8_DesktopApplication2	Профилируемое приложение с заменой цикла со строками на цикл с числами
17_9_DesktopApplication	Пример правильной организации работы со строками

Литература

Основная

(рекомендуется для изучения в качестве дополнения к данному учебному пособию)

1. **http://java.sun.com/** — сайт Sun по технологии Java для разработчиков.
2. **http://www.netbeans.org/** — официальный сайт среды NetBeans (в том числе свободная загрузка новых версий и документация).
3. **http://barsic.spbu.ru/www/comlan/java_r.html/** — сайт автора, посвященный языку Java. В том числе информация о найденных опечатках и ошибках в книге "Язык программирования Java и среда NetBeans".
4. Ken Arnold, James Gosling, David Holmes, The Java Programming Language, Fourth Edition, Addison-Wesley Professional, 2005. — 928 p.
5. На сайте **http://java.sun.com/docs/books/jls/index.html** имеется свободно распространяемая (официально!) версия предыдущего издания книги: James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification, Third Edition. Addison Wesley, 2005. — 688 p.
6. К. Арнольд, Дж. Гослинг, Д. Холмс. Язык программирования Java, 3-е издание / Пер. с англ. — СПб.: Вильямс, 2002. — 624 с.
7. П. Нотон. Java. Справочное руководство / Пер. с англ. — М.: Восточная книжная Компания, 1996. — 448 с.
8. Н. И. Смирнов. Java 2: Учебное пособие. — М.: "Три Л", 2000. — 320 с.
9. Б. Эккель. Философия Java, 4-е издание / Пер. с англ. — СПб.: Питер, 2009. — 640 с.

Дополнительная

(рекомендуется для изучения после чтения данного учебного пособия)

10. М. Фаулер. Рефакторинг: улучшение существующего кода / Пер. с англ. — СПб.: Символ-Плюс, 2003. — 432 с.
11. Джошуа Блох. Java™. Эффективное программирование / Пер. с англ. — М.: Издательство "Лори", 2002. — 224 с.

Предметный указатель

J

Java Class Library — библиотека классов 25, 71
JIT-компиляция 23

N

NetBeans Module — модуль платформы NetBeans 25, 71

А

Аварийное завершение приложения 105
Агрегация 383
Адаптеры слушателей 473
Акселераторы 123
Акторы 280
Аннотация @Action 491, 523
Анонимный класс 490
Апплет (applet) 24, 427
Арифметическое переполнение 175, 211, 214
Архивирование 94
Ассоциация 383
Атрибут объекта 40

Б

Базовый класс 224
Базовый класс иерархии 223, 267
Байт 30, 171
Байт-код 23
Библиотека JSTL 460
Бит 30, 170
Битовые маски 180
Блок очистки ресурсов 346
Булевы величины 150

В

Виртуальная машина 20
Всемирная паутина 417
Всплывающая подсказка 108
Встроенные классы 465
◊ анонимные (anonimous) 466
◊ вложенные (nested) 465
◊ внутренние (inner) 465
◊ локальные (local) 465

Г

Геттер 251, 287, 288, 305, 582
Гиперсвязь 418
Гиперссылка 417
Гипертекст 417
Горячие клавиши 122
Графический интерфейс 100

Д

Декларация метода в интерфейсе 377
Декремент 200
Дерево проектов 97
Деструкторы 260

Динамическая объектная модель 43
Динамическое выделение памяти 40
Динамическое связывание 253
Дополнительный код 173
Дочерний поток 388

З

Заглушка 553
Заготовка методов 91
Знаковый бит 183

И

Идентификатор приложения 505
Идентификаторы 35
Иерархия классов 223
Иконка 486, 501
Инициализация 201
Инкапсуляция 38, 39, 223
Инкремент 200, 240
Интегрированная среда 75
Интерпретация 18
Исключительные ситуации 345
Исполняемый код 20
Итератор 315

К

Калибровка 657
Класс 40, 42
Классы-потомки 245
Классы-прародители 245
Клонирование 255
Код
◊ бизнес-логики 338
◊ визуализации 338
Кодировка документа 420
Компиляция 18
Композиция 383
Компонент 56, 100, 579, 593
Компоненты NetBeans 58
Конкатенация 326
Консольный ввод/вывод 91
Константы 154
Конструктор 40, 91, 256
Контекстное меню 96, 101, 128
Контракт метода 248, 380
Конфигурация 526, 577
Конфликт имен 379
Критическая секция кода 391

Л

Логические ошибки 627

М

Мантисса 183
Массивы 308
◊ анонимные 312
◊ базовый тип 308
◊ быстрое заполнение 313
◊ вложенные 311
◊ двумерные 311
◊ длина 309
◊ инициализация 312
◊ иррегулярные 311
◊ коллекции 314
◊ многомерные 311
◊ поэлементное сравнение 313
◊ регулярные 311
◊ сортировка 314
◊ функции копирования 314
◊ элементы массива 310
◊ ячейки массива 309, 373
Масштабируемая фигура 229
Масштабируемые системы 397
Машинное слово 171
Машинный нуль 188
Менеджер размещения 144
Метаданные 90, 132
Метаобъекты 43
Метка 218
Метод 40
◊ класса 41, 90
◊ компонента 579
◊ левых прямоугольников 202
◊ объектов 41
◊ средних прямоугольников 205
Мидлет (midlet) 24, 71, 527, 578
Многопоточное программирование 519
Многоядерная технология 396
Множественное наследование 375, 380
Модификатор 42, 48, 232
◊ native 596
◊ видимости 90
Модификаторы видимости
◊ private 235
◊ protected 235
◊ public 236
Модуль 96

Мультиплатформенность 58
Мусор 47

Н

Наследование 38, 223, 267
Ненормализованное представление чисел 188
Непарные теги 419
Неразрывный пробел 419
Нормализованное число 186

О

Оболочечные классы 162
Обработчик события 104, 108, 472, 476
Обратное проектирование 271
Объект 40, 42
Объект-блокиратор lock 408
Объектно-ориентированный дизайн 100
Объект-синглетон 481, 523
Объект-слушатель 491
Объявление параметров 233
Окно редактирования свойств 110, 282
Оператор
◊ for 318
◊ выбора switch 198
◊ цикла do...while 217
◊ цикла for 200, 201
◊ цикла while 215
◊ прерывания 347
◊ прерывания цикла 217
Операции отношения 151
Оптимизирующий компилятор 176
Основание системы счисления 169
Основная панель 493
Отладчик (debugger) 629

П

Пакетный вид доступа 251
Пакеты 47
◊ вложенные 48
Палитра
◊ инструментов 282
◊ компонентов 105
Параметризованный класс 494
Параметры
◊ командной строки 90
◊ фактические 234
◊ формальные 234
Парные теги (контейнеры) 419
Паттерн проектирования 281
Первоначальный класс 267
Перегрузка
◊ конструкторов 258
◊ методов 248
Передача
◊ по значению 234
◊ по ссылке 240
Перекрытие области видимости (hiding) 248
Переменная 36
◊ глобальная 43, 233
◊ классов 42
◊ локальная 43
◊ объектного типа 42
◊ объектные 40, 237
◊ объектов 42
Пиксел 116
Платформо-зависимый метод 596
Побитовые операции 180
Побитовый сдвиг 172, 182
Поведение объекта 40, 226
Подключение компонента 593
Подпрограмма 39
Подпрограмма-процедура 39
Подпрограмма-функция 39
Подпрограммы-функции 19
Подпроцессы (threads) 387
Подсветка синтаксических конструкций 91
Полиморфизм 38, 223, 252
Полиморфный код 224, 375
Поля данных 40
Порядок числа 183
Поставщики сервисов 478
Поток
◊ вычислений 521
◊ индикации 522
◊ калькулятора 520
◊ выполнения (threads) 387, 414
Потомки (descendants) 223
Прародитель (ancestor) 223
Преобразование типа 266
Приведение типа 190, 265
Приоритет операторов 164
Программа 39
Программа-приложение 19
Программирование
◊ объектное 20
◊ объектно-ориентированное 38
◊ прикладное 18
◊ системное 18

Программное возбуждение
исключительной ситуации 352
Программные события 476
Программный сервер GlassFish 25
Прогрессбар 486
Проектирование иерархии 228
Прозрачность алгоритмов 206
Пространства имен (namespaces) 47
Профилирование 656
Профилировщики 389
Профиль 526, 577
Процессы 387
Прямое проектирование 271

P

Разделяемые объекты 523
Разделяемые объекты приложения 488
Разрядность 171
Растягиваемая фигура 229
Редактор
◊ исходного кода 437
◊ свойств 121
Режим
◊ выполнения приложения 125
◊ дизайна приложения 125
◊ отладки 629
◊ пошагового выполнения 630
Рефакторинг 298
Ромбовидное наследование 375

C

Сборщик мусора 47
Свойства компонента 579
Свойство 105
Сенсоры 280
Серверное приложение 24, 71
Сервлет (servlet) 24, 442
Сервлет-контейнер 443
Сеттер 251, 287, 288, 305, 582
Сигнатура
◊ метода 248
◊ функции 261
Синглетоны-спутники 487
Синтаксический анализатор 627
Синхронизация
◊ объекта 389
◊ по ресурсам 389
◊ по событиям 391
◊ потоков 389
Система поддержки версий 120
Система счисления
◊ восьмеричная 169
◊ двоичная 169
◊ десятичная 169
◊ непозиционная 170
◊ позиционная 169
◊ унарная 170
◊ шестнадцатеричная 169, 177
Скомпилированный модуль 96
Слушатели (listeners) событий 472
Событие 472, 579
Совместимость типов по присваиванию 160
Создание сервлета 444
Составной оператор 193, 195
Состояние объекта 226
Список параметров 41, 233
Ссылка this 267
Старший бит 172
Статическая объектная модель 43
Статическое связывание 253
Статусбар 523
Стили форматирования текста 423
Структура папок проекта NetBeans 98
Субкласс (subclass) 224
Суперкласс (superclass) 224
Счетчик цикла 201

T

Таблица
◊ map 488
◊ ресурсов 488
Тактовая частота 397
Таскбар 479
Теги 418
Тело
◊ документа 420
◊ функции 233
◊ цикла 201
Тестирование программных продуктов 643
Тестовые библиотеки 643
Технология
◊ ASP 59
◊ JSP 59
◊ PHP 59
◊ JSP 456
Тип переменной
◊ boolean 36, 37
◊ byte 37
◊ char 37

◊ double 37
◊ float 36, 37
◊ int 36, 37
◊ long 37
◊ object 36
◊ short 37
◊ пользовательский 36
◊ предопределенный 36
◊ примитивный 37
◊ ссылочный 37
Тип-перечисление 333
Типы источников событий 590
Трансляция 18

У

Унарное наследование 375
Управляющая последовательность 32
Условный оператор if 194

Ф

Фактические параметры 267
Фоновый поток индикации 522
Формальные параметры 267
Формат
◊ JPEG 283
◊ PNG 283
◊ SVG 284
◊ с плавающей точкой (floating point) 183
Формирование проекта с апплетами 434

Х

Хэш-код 243, 325

Ц

Целочисленное переполнение 177
Цикл 201

Э

Экранная (диалоговая) форма 100

Я

Языковый формат 452
Ячейка памяти 30