Кёрк Скотт

JAVA

ДЛЯ СТУДЕНТА

bhv®

+ CD

Кёрк Скотт

JAVA

ДЛЯ СТУДЕНТА

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.068+800.92java
ББК 32.973.26-018.1
С44

Скотт К.

С44 Java для студента. — СПб.: БХВ-Петербург, 2007. — 448 с.: ил. + CD-ROM

ISBN 978-5-94157-968-6

Книга написана на базе курса лекций, читаемых автором на протяжении многих лет в США, России и Казахстане. В краткой и доступной форме описаны основные особенности объектно-ориентированного программирования на языке Java, иллюстрируемые многочисленными примерами. Первая часть книги знакомит читателей с основным синтаксисом языка и используемыми обозначениями. Вторая часть посвящена программированию графического интерфейса пользователя. Изложение материала сопровождается упражнениями и заданиями к каждой главе. Промежуточные задания представляют собой написание программы для древних восточных игр, а итоговое — графической версии программы, моделирующей простой микропроцессор. Исходные тексты программ и упражнения располагаются на компакт-диске и сайте http://math.uaa.alaska.edu/~afkas/BHV.

Для начинающих программистов и студентов

УДК 681.3.068+800.92java
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Перевод с английского	*Андрея Резникова*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.02.07.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 28.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 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 978-5-94157-968-6

© Скотт К., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление

Введение **9**

Благодарности **10**

Глава 1. Основы языка Java и объектно-ориентированного программирования **11**

1.1. Java 11
Выводы 14
1.2. Объектно-ориентированное программирование (ООП) 15
Выводы 18
1.3. Язык UML 18
Выводы 21
1.4. Загрузка и установка Java и TextPad 22
Выводы 24
1.5. Практические советы 24
Выводы 27

Глава 2. Начало программирования **28**

2.1. Пример первой программы 28
2.2. Структура кода и результат вывода программы на экран 30
2.3. Документация Java API 33
2.4. Создание и использование объектов 37
2.5. Ошибки 40

Глава 3. Типы данных, операторы и переменные **44**

3.1. Типы данных, объявление переменных и инициализация 44
3.2. Присваивание и отношения между числовыми типами 47

3.3. Присваивание и простые арифметические операции 49
3.4. Класс *Math* 52
3.5. Класс *String* 55
3.6. Класс *MyTerminalIO* 60

Глава 4. Как написать и использовать коды для классов 62

4.1. Переменные экземпляра, конструкторы и методы 62
4.2. Копирование и использование ссылок на объект 71
4.3. Ссылка на *null* 75

Глава 5. Апплеты и графика 82

5.1. Web-страницы и просмотр апплетов 82
5.2. Информация о геометрических и графических классах 83
5.3. Пример простого апплета 86
5.4. Черчение и заполнение геометрических форм 89
5.5. Цвета 90
5.6. Текст 93
5.7. Размещение апплетов на Web-страницах 95
5.8. Web-документация Java 97

Глава 6. Оператор *if* — условное выполнение 102

6.1. Синтаксис оператора *if* и условия вычисления 102
6.2. Множественные альтернативы и вложенный оператор *if* 105
6.3. Логические операторы, сложные выражения и булевы переменные 108
6.4. Сравнение объектов 115
6.5. Последний пример, символьный тип и некоторые типичные ошибки 118

Глава 7. Циклы 123

7.1. Циклы *while* и *do* 123
7.2. Цикл *for* 126
7.3. Область действия 128
7.4. Примеры 131
7.5. Вложенные циклы 135
7.6. Графические примеры 137

Глава 8. Методы, параметры и переменные 147

8.1. Передача параметров простых типов 147
8.2. Передача объектных ссылок в качестве параметров 152

8.3. Методы доступа, модифицирующие методы и побочные эффекты 157
8.4. Статические методы 160
8.5. Статические переменные 162
8.6. Полная классификация переменных в Java 166
8.7. Еще о синтаксисе 170

Глава 9. Массивы, класс Vector и класс ArrayLists 173

9.1. Объявление и использование массивов 173
9.2. Копирование массивов, передача массивов в качестве параметров и возвращение массивов методами 178
9.3. Примеры массивов 181
9.4. Структура коллекций в языке Java 187
9.5. Синтаксис коллекций 189
9.6. Примеры коллекций 193

Глава 10. Файл ввода/вывода 199

10.1. Вывод на экран и чтение из командной строки 199
10.2. Символы и Unicode 202
10.3. Обработка исключений 207
10.4. Запись символов в файл 210
10.5. Чтение символов из файла 212
10.6. Запись строк в файл 217
10.7. Файл ввода/вывода с произвольным доступом 218

Глава 11. Рекурсия 222

11.1. Описание рекурсии 222
11.2. Составление и выполнение рекурсивного кода 225
11.3. Особенности рекурсии 230
11.4. Рекурсия и циклы, рекурсия с двумя параметрами 232

Глава 12. Подклассы и наследование 237

12.1. Что такое наследование? 237
12.2. Написание кода для подклассов 241
12.3. Доступ к переменным экземпляра и переопределение методов 245
12.4. Ключевое слово *super*, методы и конструкторы 249
12.5. Ссылки на классы и подклассы в программах и методах 252

Глава 13. Абстрактность, полиморфизм и интерфейсы 258

13.1. Абстрактность в конструировании классов 258

13.2. Полиморфизм и динамическая диспетчеризация....263
13.3. Множественное наследование и интерфейсы 268

Глава 14. Методы *toString()*, *equals()* и *clone()* 273

14.1. Метод *toString()*....273
14.2. Метод *equals()*....276
14.3. Клонирование....278
14.4. Реализация метода *clone()*....284
14.5. Важное использование клонирования....289

Глава 15. Проекты игровых программ 290

15.1. Исторический обзор 290
15.2. Wari....293
15.3. Togiz Kumalak 295
15.4. Варианты реализации....298

Глава 16. Простые схемы UML с помощью программы Visio....300

16.1. Схемы статической структуры для классов и объектов 300
16.2. Ассоциации, агрегации и композиции....304
16.3. Наследование, интерфейсы и внутренние классы....307
16.4. Схемы последовательностей....311

Глава 17. Контейнеры, рисование и диалоговые окна....313

17.1. Контейнеры и компоненты 313
17.2. Рисование 317
17.3. Диалоговые окна и строковый вывод на панель 320
Echo1 320
Echo2 322
Echo3 325

Глава 18. Внутренние классы, блоки прослушивания и классы-адаптеры 329

18.1. Внутренние классы, блоки прослушивания и интегрированные внутренние поля....329
Echo4 330
Echo5 332
Echo6 336
18.2. Панели, классы-адаптеры и анонимные внутренние классы 339
Echo7 339

Echo8 345
Echo9 348

Глава 19. Объекты, реагирующие на щелчок мыши, апплеты и jar-файл 355

19.1. Объекты, реагирующие на щелчок мыши 355
ClickCup 355
ClickDot 360
ClickHand 364
19.2. Преобразование приложения в апплет и размещение его на Web-странице 367
19.3. Преобразование приложения в самовыполняющийся jar-файл 371

Глава 20. Меню, множественные фреймы, сериализация и текстовые поля 373

20.1. Ввод с клавиатуры, меню и множественные фреймы 373
ClickKey 373
ClickMenu 375
ClickMany 379
20.2. Сериализация, выборщики файлов, текстовые поля и полосы прокрутки 381
ClickSave 381
ClickChooser 385
ClickTextScroll 387

Глава 21. Задание для итогового проекта 390

21.1. Регистры и память: общие сведения 390
21.2. Примеры программ 393
MiscButton1 393
MiscButton2 394
MiscButton3 394
MiscRegister 394
MiscText 396
MiscLabel 397
21.3. Фокус 397
MiscFocus 397
21.4. Новые примеры программ 404
MiscAction 404
MiscMulti 404
MiscMemory 405
MiscRegAndMem 406

21.5. Потоки 407
MiscThread 407

Глава 22. Компьютер с минимальным набором команд 412

22.1. Использование MISC 412
22.2. Структура MISC 416
22.3. Машинный класс 418
22.4. Машинные команды 420
Команда *MOVE* 421
Команда *ADD* 422
Команда *SUB* 422
Команда *JUMP* 423
22.5. Программы машинного языка 423
22.6. Машинная арифметика 426
22.7. Подробное описание проекта 430
Схема зависимостей 433

Приложение. Описание компакт-диска 435

Предметный указатель 443

Введение

Уважаемые читатели!

Вашему вниманию предлагается вводный курс в язык программирования Java, в котором в краткой и доступной форме (в отличие от иных изданий, ориентированных на профессиональных программистов или требующих от читателей определенного уровня подготовки) описаны основные особенности объектно-ориентированного программирования, иллюстрируемые многочисленными примерами. Изложение материала сопровождается упражнениями к каждой главе, вместе с кодами программ, где это необходимо, а также кодами для примеров; весь этот материал вы сможете найти на прилагаемом компакт-диске, а также в Интернете по адресу: **http://math.uaa.alaska.edu/~afkas/BHV**.

В первых главах, как правило, полные листинги приведены в самом тексте книги, но даже в этих случаях понять разбираемые примеры будет значительно проще, если загрузить, составить и попробовать в работе коды, представленные на компакт-диске или сайте. В последующих главах примеры становятся настолько громоздкими и сложными, что нам представлялось нецелесообразным помещать листинги полностью в тексте книги, поэтому необходимо будет взять их с компакт-диска или загрузить с указанного сайта. На этом же сайте будут выкладываться и соответствующие исправления и дополнения.

Благодарности

Пользуясь случаем, я хотел бы выразить благодарность тем людям, которые помогли осуществить настоящее издание.

Я выражаю благодарность программе "Фулбрайт" за поддержку. Первые 15 глав книги были написаны с использованием грантов этой программы, во время моей исследовательской и преподавательской работы в России и Казахстане.

Я также хочу выразить благодарность переводчику книги, А. Резникову. Без его заинтересованного отношения к работе, мой проект не дошел бы до стадии опубликования.

Я хочу поблагодарить мой университет, Университет штата Аляски, Анкоридж, за финансовую поддержку работы над *главами 16—22* издания.

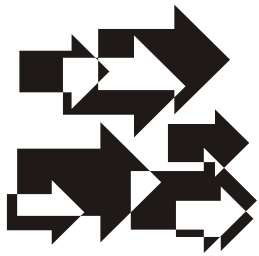
Я хочу также поблагодарить моих студентов, К. Хаванского и Д. Коробова, за помощь в подготовке материалов заданий для диска и Web-сайта.

Я выражаю благодарность всем сотрудникам редакции издательства "БХВ-Петербург", которые принимали участие в работе над текстом книги.

И последнее: вся ответственность за любые недочеты, которые могут быть в книге, лежит исключительно на мне. При этом я надеюсь, что издание будет полезно всем читателям, желающим научиться программировать на языке Java.

Керк Скотт
Анкоридж, январь 2007

ГЛАВА 1



Основы языка Java и объектно-ориентированного программирования

1.1. Java

Каковы характеристики языка Java, отличающие его от других языков программирования? Какими особенностями обладает этот язык? Что позволяет ему быть полезным в тех ситуациях, когда другие языки не подходят?

С технической точки зрения, Java обладает одним главным свойством, которым объясняются все остальные особенности данного языка: Java — это язык, который работает на виртуальной машине. В принципе можно создать архитектуру, которая будет непосредственно выполнять код Java, однако в стандартной среде, такой как Windows или UNIX, установка Java на компьютере означает, что установлена виртуальная машина. Когда программы работают, считается, что они функционируют на виртуальной машине. В других языках, таких как C или C++, в результате компиляции мы получаем выполняемый файл, который содержит машинные команды для той архитектуры, для которой был создан компилятор. Для языка Java этот процесс также называется *компиляцией*, но он не является компиляцией в буквальном смысле слова. Когда компилируется код для Java, то результатом будет байт-код, т. е. код, который работает в виртуальной Java-машине.

Язык программирования такого типа является просто еще одним слоем программного обеспечения и не зависит от основных характеристик архитектуры компьютера. В итоге, по крайней мере в теории, Java является потенциально "универсальным" языком программирования. Версии Java существуют для различных операционных систем, механизм их действия отличается друг от друга. Однако результат работы кода Java, запущенного в разных операционных системах, будет один и тот же. Таким образом, программы Java должны быть свободно переносимы между ними. Если программа написана в одной среде, то нет необходимости производить какие-либо изменения в исходном коде, чтобы скомпилировать или напрямую запустить ее в другой среде. В идеале, это можно выразить следующим образом: "Написал один раз, работает везде". Однако на практике тонкие различия в осуществлении механизма Java приводят к проблемам, которые заключаются во фразе: "Написал один раз, находишь ошибки везде".

Как и во всех системных и программных продуктах, в Java существуют компромиссы. Одной из движущих сил развития языка такого типа является Интернет. Если существует "универсальный" язык программирования, то тогда программы могут быть установлены и будут работать на машинах пользователей независимо от характеристик платформы. С помощью Java создаются *апплеты* — программы, которые могут быть загружены из Интернета. Не имеет значения, из-под какой операционной системы происходит подключение к Интернету: если в ней установлен Java, то она будет работать с апплетами, которые доступны во Всемирной сети. Большим компромиссом в данном случае является то, что программы, написанные на Java, работают значительно медленнее, чем программы, которые скомпилированы для конкретной архитектуры. В среднем код Java будет работать в 10 раз медленнее. Но по мере того как процессоры становятся все быстрее, а Интернет становится средой распространения программ, недостаток в скорости, за счет которого достигается универсальность, выглядит все более приемлемым компромиссом.

Есть еще одна особенность языка Java, обусловленная желанием загружать готовые программы. Некоторые языки, такие как C и C++, дают программисту возможность написать код, который

напрямую обращается к адресам в памяти компьютера. Это сделано путем использования указателей. Этого нет в Java. Может показаться странным, что в современном языке невозможно работать с указателями. Можно написать код для связанных структур на Java, но нельзя написать код с использованием адресов памяти. Программисты знают, что программы, которые напрямую имеют доступ к памяти, могут, преднамеренно или случайно, навредить системе. По этой причине прямой доступ к памяти нежелателен в Java. Когда пользователи запускают программы, они хотят быть уверенными, что эти программы не будут разрушать их системы из-за прямого доступа к памяти. Таким образом, Java поддерживает и универсальность, и безопасность программ.

Цель нашей книги — познакомить читателя с основами программирования на языке Java. Java — это объектно-ориентированный язык программирования. Он смоделирован для того, чтобы поддерживать событийно-действующее программирование. Другими словами, он может быть использован для создания графического пользовательского интерфейса, который поддерживает действия по типу "укажи и щелкни" (point and click). Такие приложения могут быть самостоятельными программами или апплетами, загружаемыми из Интернета. Java обладает основными синтаксическими особенностями структурированных языков программирования, а именно последовательным выполнением блоков кода, условным выполнением (`if`-операторы) и повторяющимся выполнением (циклы). Эти элементы будут рассмотрены в следующих главах, так же как и описание объектно-ориентированного механизма, который поддерживает более сложные виды программирования.

Эксперты расходятся во мнениях относительно технических достоинств Java, однако необходимо сразу отметить, что, с точки зрения изучающих, язык Java обладает рядом недостатков. Java не рассчитан для обучения программированию. Он предназначен для построения сложных приложений. В дополнение ко многим свойствам в самом языке, он имеет библиотеки готовых компонентов кода, которые могут использовать опытные программисты, вместо самостоятельного написания компонентов с нуля. Ни полностью синтаксис, ни библиотеки невозможно изучить за ко-

роткий срок. Многое из того, что изучается на первом этапе, может показаться не слишком полезным. Необходимо рассмотреть основы объектно-ориентированного программирования прежде, чем станет возможным создание интересных приложений, включающих интерактивные графические свойства.

Как только вы познакомитесь с этими основами, то убедитесь, что Java прекрасно соответствует своим целям. Вы сможете создавать графические приложения, которые просто невозможно написать на таких языках программирования, как C. Кроме того, код, написанный на Java, может быть значительно проще, чем код, который написан на C++, чтобы выполнить то же самое. Java с самого начала был создан как объектно-ориентированный язык. В этом его преимущество перед языком C++, который является объектно-ориентированной эволюцией языка C. В заключение нужно сказать, что язык C# был создан для той же самой цели, что и язык Java, поэтому C# является его прямым конкурентом, который по своим характеристикам может быть не хуже или даже способен превосходить Java в некоторых аспектах.

Выводы

1. Язык Java интерпретируется на виртуальной Java-машине.
2. Исходный код и код заданий Java легко переносятся с компьютера на компьютер.
3. Существует возможность загружать код Java из Интернета, несмотря на то, что программы будут не так быстро работать.
4. Java не использует доступ напрямую к памяти для обеспечения безопасности загруженной программы.
5. Java — это объектно-ориентированный язык, имеющий свойства, необходимые для того, чтобы написать графические приложения с управлением событиями.
6. Java предназначен для программирования, но не обучения языку. Он обладает многими свойствами и содержит много библиотек.
7. В Java можно создавать такие программы, которые невозможно или трудно написать на других языках.

1.2. Объектно-ориентированное программирование (ООП)

Языки программирования и парадигмы развиваются с течением времени. Так, появилось новое аппаратное обеспечение, которое способно поддерживать новые парадигмы программирования; возникла необходимость решать новые проблемы, которые, в свою очередь, требуют новых возможностей программирования; при разработке крупных или сложных проектов выявились недостатки в существующих системах. Объектно-ориентированное программирование — одна из новых парадигм программирования. Его развитие и особенности заключаются в следующих трех свойствах: это принцип программирования, который не смогли бы поддержать примитивные машины; программирование графического пользовательского интерфейса — это одна из областей, где его применение очевидно; и поскольку ООП решает, по крайней мере, некоторые проблемы, связанные со структурным программированием — его непосредственным предшественником, оно в значительной мере вытеснило структурное программирование в развитии сложных систем программного обеспечения.

Java — это объектно-ориентированный язык. Есть много объектно-ориентированных языков, и каждый осуществляет этот принцип программирования по-своему. Полное объяснение ООП не входит в круг задач настоящего пособия. Невозможно даже дать полное объяснение ООП в рамках языка Java. Однако желательно начать с некоторых общих идей, относящихся к ООП, и определения терминов и основных понятий. Более полное понимание возникнет тогда, когда применение этих идей в языке Java будет изучено и когда к основным понятиям будут добавлены более сложные понятия и их применение.

ООП имеет два фундаментальных термина, которые будут повторяться снова и снова. Эти термины "класс" и "объект". *Класс* — это модель для объектов. Можно также сказать, что класс — это шаблон или образец для объекта. Точнее, класс — это часть компьютерного кода, которая дает полную спецификацию для объектов. Определение класса — это часть программного исходного кода. Во время работы программы можно создать объект согласс-

но спецификациям данного класса. *Объект* может быть рассмотрен как экземпляр класса. Программа может использовать более одного класса, и более одного экземпляра классов может быть создано во время работы программы. Работа программы состоит из манипуляций такими объектами.

В ООП включены два дополнительных понятия. Компьютерные программы содержат последовательность инструкций, которые действуют вместе и могут выполнять конкретные задания. В объектно-ориентированном программировании они называются методами. *Методы* — это часть определения класса. Компьютерные программы также включают объявления переменных, таких как числа, последовательность символов и т. д., с которыми производятся манипуляции. *Переменные* — это символьные имена, которые используются для указания на сохраненные количества или значения в программе. Различные виды переменных могут быть использованы в различных частях программы. Переменные, связанные с классами и объектами, являются *переменными экземпляра* (или *экземплярными переменными*), и так же, как и методы, они являются частью определения класса.

Хотя оба этих понятия приводятся в определении класса, методы и переменные экземпляра рассматриваются в ООП по-разному. В программе может быть более одного экземпляра более чем одного класса. Следующее отличие между методами и переменными экземпляра заключается в том, что нет необходимости для каждого экземпляра класса иметь свой код метода. Методы определяются в классе и доступны всем объектам данного класса, существующим в программе в данное время. Таким образом, методы для всех объектов класса являются абсолютно одинаковыми. Однако, когда создаются экземпляры класса, они получают свои уникальные копии переменных экземпляра. Каждый объект класса получает тот же самый набор переменных, но значения переменных для каждого объекта могут быть различны.

Итак, каковы взаимоотношения между методами и переменными экземпляра? Фундаментальным аспектом ООП является инкапсуляция. Каждый объект имеет свои копии экземплярных переменных. Значения этих переменных могут быть доступны или изменены во время работы программы только одним способом: вызо-

вом метода, целью которого является доступ к переменным экземпляра или их изменение. Это обеспечивает строгую структуру программы. Невозможно в каких-либо местах программного кода изменить значение переменных экземпляра так, как это удобно в данный момент. Изменения могут быть произведены только в рамках инкапсуляции. Свободный доступ к значению переменных считался одним из недостатков структурного программирования. Эта проблема решена в объектно-ориентированном программировании.

На рис. 1.1 представлена простая графическая схема объекта, показывающая переменные экземпляра, которые он содержит, и методы, которые он использует.

Рис. 1.1

Простой способ понять значение этого рисунка — это представить объект в виде яичницы-глазуньи. Как только объект создан, программа имеет доступ к желтку только через белок. Важно понять, что программисту, пользующемуся объектами, нужно знать, через какие методы доступны переменные. Однако нет необходимости знать, как этот код работает. Хотя наличие некоторых переменных экземпляра может быть очевидно, программисту не нужно знать, как они применяются, и не нужно знать все переменные, чтобы использовать объект. В этом смысле объект можно представить в виде "черного ящика". Программист может производить с ним определенные операции с конкретным ожидаемым результатом, но каким образом достигается данный результат, мы не видим. Методы предоставляют собой интерфейс для доступа к переменным экземпляра и иным значениям, находящимся внутри объекта.

Выводы

1. ООП полагается на современные возможности аппаратного обеспечения, решает проблемы, которые не могли быть решены предшествующими концепциями программирования, и преодолевает некоторые недостатки предыдущих концепций программирования.
2. Java осуществляет концепцию ООП; основные термины и идеи ООП полезны при объяснении языка Java.
3. Классы и объекты представляют собой два основных понятия ООП.
4. Методы и переменные — это два дополнительных ключевых понятия ООП.
5. Когда создаются объекты класса, они используют его методы совместно, но каждый объект получает свои копии переменных экземпляра.
6. Объект инкапсулирует переменные. Значения переменных могут быть доступны только при условии, если существует метод в классе, который позволяет это сделать.
7. Программист, использующий объект, имеет доступ к нему только через методы и ему не нужно знать, как методы работают и какие переменные экземпляра они используют.

1.3. Язык UML

В разных источниках аббревиатуру UML расшифровывают по-разному: "unified modeling language" (единый язык моделирования) или "universal modeling language" (универсальный язык моделирования). Как бы то ни было, это набор условных обозначений для графического изображения объектно-ориентированных программ. Можно нарисовать такие вещи, как классы и их методы, экземпляры объекта, объекты классов, отношения между классами и т. д. В нашей книге мы не будем давать полное описание UML. Однако при описании Java в некоторых случаях графическое изображение идеи или отношения очень помогает. В таких случаях мы будем использовать соответствующие символы UML.

Кроме собственных графических символов, в UML имеется и свой словарь, который используется при представлении графически-ориентированных компонентов. Ранее в этой главе уже упоминались термины *"класс"* и *"объект"*. Эти термины используются в таких же значениях и в рамках UML. Также ранее были объяснены термины *"переменная экземпляра"* и *"метод"*. В UML соответствующими терминами являются *"атрибут"* и *"операция"*. Существует более чем один объектно-ориентированный язык программирования, однако язык UML является нейтральным. Его терминологию можно использовать для описания любой объектно-ориентированной программы, созданной на любом из таких языков.

Поскольку в данной книге рассматривается только язык программирования Java, то мы будем продолжать использовать термины *"переменная экземпляра"* и *"метод"*. Следует также отметить, что в официальной документации языка Java термин *"поле"* (field) используется в значении "переменная экземпляра". Поскольку данный термин выбран разработчиками языка, то, вероятно, он является более правильным. Однако этот термин может сбивать с толку, т. к. он имеет и иные значения в программировании. Поэтому мы будем использовать более описательный термин *"переменная экземпляра"*.

Для иллюстрации употребления UML необходим пример. Такой пример можно рассмотреть и обсудить с его помощью идеи объектно-ориентированного программирования, даже не касаясь языка Java. Пример, который мы будем использовать на протяжении всей книги, основан на следующем. Предположим, что мы будем хранить семена в чашке. На первый взгляд, это может показаться не слишком практичным. Обычно мы представляем себе чашку как сосуд, содержащий какую-либо жидкость, да и не совсем понятно, как можно написать интересные программы, используя чашки и семена. Но любой вводный пример неизбежно будет не слишком практичным. Пример с чашками и семенами приводится здесь потому, что в дальнейшем он используется в качестве компонента игровой программы.

Классы и объекты изображаются с помощью прямоугольников. Отношения указываются с помощью надписей, линий, стрелок и

иных символов. В своей простейшей форме класс может быть изображен прямоугольником, содержащим имя класса. Имя пишется жирным шрифтом с большой буквы, как на рис. 1.2.

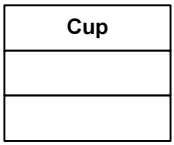


Рис. 1.2

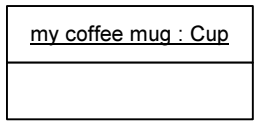


Рис. 1.3

Любой объект данного класса будет иметь собственное имя. Предположим, экземпляр известен как "моя кофейная кружка". Он также представлен прямоугольником. Имя не пишется с большой буквы. За ним ставится двоеточие и название класса, экземпляром которого он является. Все это подчеркивается (рис. 1.3).

Отношения между классами и объектами графически могут быть изображены в виде стрелок. В UML различные виды отношений указываются с помощью различных стрелок, так что форма стрелки имеет значение. Стрелка направлена от объекта к классу (рис. 1.4).

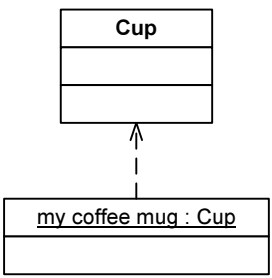


Рис. 1.4

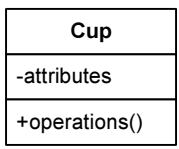


Рис. 1.5

Более подробная информация о классе дается с помощью указания атрибутов и операций на рисунке. В целом, изображение такого класса принимает следующий вид (рис. 1.5).

Пусть наш класс имеет одну переменную экземпляра, `seedCount`, и два метода: `setSeedCount()` и `getSeedCount()`. Пусть переменная будет целым числом, и когда создается новый экземпляр класса, то пусть она инициализируется со значением 0. Метод `setSeed()` приписывает значение экземпляру объекта, а метод `getSeed()` возвращает значение, приданное переменной `seedCount` в данный момент, каким бы оно ни было. Эта информация показана более подробно в следующем изображении класса на рис. 1.6.

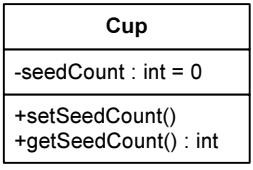


Рис. 1.6

Можно дать еще более подробную информацию, например информацию о параметрах метода, диапазоне значений переменной и иные сведения. Такие схемы используются при документировании систем программного обеспечения, программ или их компонентов. Уровень детализации должен быть рассчитан так, чтобы быть полезным для того, для кого создается графическое изображение, будь то дизайнер, программист или пользователь.

Выводы

1. UML используется для графического изображения объектно-ориентированных программ.
2. UML имеет свой словарь для наименования компонентов классов.
3. Язык Java имеет собственный, параллельный словарь для обозначения компонентов классов.
4. Пример с семенами и чашками используется для иллюстрации.
5. В UML классы и объекты изображаются в виде прямоугольников, а отношения указываются с помощью стрелок и иных символов и надписей.

6. Простейшая схема класса представляет собой прямоугольник, содержащий название объекта, написанное с заглавной буквы жирным шрифтом.
7. Простейшая схема объекта представляет собой прямоугольник, содержащий название объекта, за которым через двоеточие ставится название класса; все это набирается жирным шрифтом.
8. Отношения между объектом и его классом показываются с помощью пунктирной стрелки, направленной от объекта к классу.
9. Более полное изображение класса включает атрибуты и операции класса, а также, возможно, и более подробную информацию.
10. Уровень детализации рисунков может быть разным.

1.4. Загрузка и установка Java и TextPad

Для того чтобы программировать на языке Java, необходимо иметь компилятор. Вы можете бесплатно загрузить компилятор фирмы Sun Microsystems с Web-сайта этой фирмы. Редактор TextPad может быть загружен бесплатно для ознакомления и последующей покупки, при необходимости. Этот редактор рассчитан на работу с Java-программами фирмы Sun. Если сначала вы установили Java, а затем TextPad, следуя инструкции по инсталлированию, то в меню редактора должны появиться активные опции для компиляции и выполнения Java-приложений и апплетов.

Java находится в Интернете по данному адресу:

http://java.sun.com/javase/downloads/index.jsp

Часть страницы с желательным компилятором выглядит так, как показано на рис. 1.7.

Когда вы щелкните мышью по ссылке **Download**, то сначала откроется страница с лицензионным соглашением. После этого вам

будет представлен список различных версий программ для разных типов компьютеров. Первой дается стандартная версия для компьютеров с Windows. Вам потребуется установка программы в режиме автономной работы (offline). Это значит, что программа будет скачана на ваш компьютер, а затем вам нужно два раза щелкнуть мышью по ее пиктограмме, чтобы установить программу на вашем компьютере. Если вы работаете с каким-либо иным компьютером, то для вас может подойти одна из других ссылок. Вы можете изучить сайт компании Sun в поисках дополнительной информации, такой как документация к языку или обучающие программы. Здесь мы не можем дать описание всего, что находится на сайте.

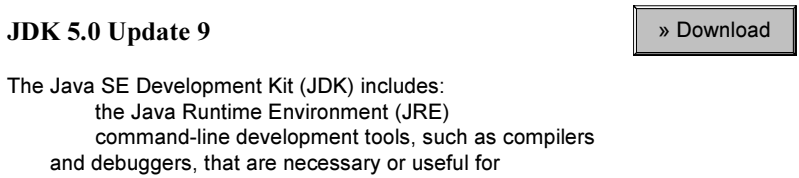


Рис. 1.7

Редактор TextPad находится в Интернете по адресу:

http://www.textpad.com/download/index.html

Следуйте инструкциям, данным на этой странице. Если возникнут проблемы, можно следовать этим ссылкам: **Support**, **FAQ**, **java**.

Вы найдете ответы на любые вопросы, которые касаются возможных проблем при работе редактора TextPad с Java. Например, если вы загрузили и установили TextPad и Java не в том порядке, здесь вы найдете инструкцию, как сделать так, чтобы они заработали вместе без переустановки. Также имеются ответы и на другие вопросы. Если инструкции вам непонятны или вы столкну-

лись с неразрешимыми проблемами, то, возможно, вам следует использовать интегрированный пакет, который можно скачать из Интернета или купить. Данная книга не содержит полных сведений по установке и работе с редактором TextPad. Если вы используете последнюю версию языка Java, то сможете сделать все, что описано в книге, независимо от типа редактора.

Выводы

1. Как Java, так TextPad и можно скачать из Интернета.
2. Если их установить в указанном порядке, то TextPad будет содержать инструменты для компилирования и выполнения программ на Java.

1.5. Практические советы

Существует несколько деталей, касающихся наименования файлов программ Java, которые следует запомнить. Они сводятся к следующему:

- Файлы программ на Java должны сохраняться с расширением java.

  Мы еще не ознакомились с кодом для Java-программ. Тем не менее в начале каждой программы будет появляться строка такого вида:

  ```
  public class Name_of_program
  ```

  Имя файла, с которым программа сохранена, должно точно совпадать с названием, которое появляется в первой строке внутри программы. Первая буква названия программы пишется заглавной, и первая буква названия файла также должна быть заглавной. Другими словами, для того чтобы быть работающей, программа должна быть сохранена с таким именем:

  ```
  Name_of_program.java
  ```

- В языке Java, в отличие от Microsoft Windows, всегда имеет значение, употребляете ли вы заглавную (прописную) или строчную букву. Это может привести к проблемам. Если вы

сохранили файл в Windows с именем, записанным со строчной буквы, а затем попытаетесь изменить название файла, чтобы использовать заглавную букву, то в этом случае Windows может не позволить это сделать, сообщая вам: "Файл с этим именем уже существует". Если это произошло, сохраните файл под временным названием, удалите копию с неправильным именем и переименуйте временный файл, используя заглавную букву.

Если вы используете редактор TextPad, то вам необходимо иметь в виду следующее.

В меню **Tools** редактора TextPad вы найдете 3 опции: **Compile Java**, **Run Java Application** и **Run Java Applet**. Сейчас мы рассмотрим первые две опции: компиляцию и запуск программы. Работу апплетов мы рассмотрим позже.

В меню **Tools** редактора TextPad вы также увидите опцию **Run**. Мы упоминаем о ней только потому, что вы наверняка по ошибке попытаетесь хотя бы раз ее выбрать. Используя эту опцию для того, чтобы запустить программу Java, вы либо откроете диалоговое окно, спрашивающее, какое приложение необходимо запустить, либо получите сообщение об ошибке. Вы должны быть уверены, что запускаете программу с помощью опции **Run Java Application**.

Чтобы проверить компилирование и запуск программ Java в TextPad, необходимо сделать следующее: загружаете программы MyTerminalIO.java и FirstProg.java и сохраняете их в одном месте — на дискете, жестком диске, в какой-либо папке и т. д. на вашем компьютере. Вы можете также ввести имя программы, используя TextPad. Далее приведен код для справки. Сохраните его в файле с именем FirstProg.java.

```
public class FirstProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
```

```
    myterminal.println("Hello, World!");
  }
}
```

Затем выберите в меню опцию **Compile Java**. На это может уйти несколько секунд. Успешная компиляция просто закончится, показывая на экране файл программы. В противном случае на экране появится сообщение об ошибке. Приведенный пример не содержит ошибок, поэтому, если возникла ошибка, проверьте, находится ли копия программы MyTerminalIO.java в том же месте, правильно ли вы набрали код, а также сохранили ли вы файл под правильным именем, а затем попытайтесь снова скомпилировать программу.

После успешной компиляции вы можете выбрать в меню **Run Java Application**. Это приведет к открытию окна с результатом работы программы.

Вы можете случайно сделать ошибку при входе в программу или же сделать ошибку намеренно, пытаясь посмотреть, что делает компилятор. В редакторе появится список ошибок с номерами строк, а также достаточно непонятные объяснения. Если установленная программа функционирует нормально, то у вас должна быть возможность перескочить прямо в строку программы, в которой находится ошибка, щелкнув мышью на сообщении об ошибке. Если эта функция программы не работает, то просто запомните номер строки, перейдите в файл кода, выбрав из списка в левом верхнем меню программы TextPad, и затем вручную найдите нужную строку. Номер текущей строки показан внизу окна редактора.

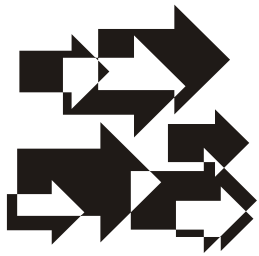
Если ваш компьютер соединен с принтером, то вы можете напечатать программу исходных файлов прямо из редактора TextPad через выбор пункта **File** в меню.

Для того чтобы распечатать результаты работы приложения, необходим еще один шаг: дело в том, что окно с выводом программы не содержит меню. Поэтому нужно выделить и скопировать в буфер обмена содержимое окна, затем открыть новый документ в TextPad и вставить содержимое буфера. Теперь вы можете распечатать файл, используя опции меню как обычно.

Выводы

1. Имена файлов Java-программ должны совпадать с названиями классов и иметь расширение java. Java различает прописные и строчные буквы.
2. Меню TextPad включает опции для компилирования и запуска программ на Java.
3. Имеется возможность распечатать как код, так и результат программы.

ГЛАВА 2



Начало программирования

2.1. Пример первой программы

Цель данного раздела — дать первоначальные пояснения к первому примеру программы, приведенному в конце предыдущей главы. Напомним эту программу:

```
public class FirstProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    myterminal.println("Hello, World!");
  }
}
```

Далее приводится пошаговое описание синтаксиса программы вместе с некоторыми пояснениями того, как и почему код написан именно таким образом. На первой стадии самое главное — познакомиться с синтаксисом, чтобы вы могли использовать его при написании собственных программ. Пояснения даются потому, что для большинства людей легче понять что-либо с опорой на пояснения. Однако, если некоторые из комментариев покажутся вам непонятными или недостаточными, не волнуйтесь: в последующих главах вы найдете более подробные пояснения.

1. Первая строка кода показывает синтаксис для объявления класса:

   ```
   public class FirstProg
   ```

   Все программы, написанные пользователями, являются классами. Класс `public` означает, что он доступен для общего пользования. Название класса — `FirstProg`.

2. Вторую строку кода довольно легко объяснить, и это же касается последней строки кода. Все определения класса заключаются в фигурные скобки `{}`.

3. В третьей строке кода указывается метод.

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

   Название этого метода — `main()`. Самостоятельная программа должна иметь этот метод. Когда вы просите систему Java выполнить программу, она смотрит внутрь класса программы и вызывает работу данного метода. Ключевое слово `public` означает, что этот метод свободно вызывается системой. Ключевое слово `static` означает, что метод является общим методом, а не таким, который используется только с данным объектом. Эти понятия будут более подробно объяснены позже. Ключевое слово `void` показывает, что выполнение этого метода не вызывает возвращения каких-либо значений.

   Каждое имя метода сопровождается соответствующей парой круглых скобок. Когда в книге будет идти речь о методах, они всегда будут указываться именем со скобками, например `main()`. Подобное написание указывает на то, что имя `main` — название метода, а не чего-то еще. Метод может содержать запись в скобках. Это параметры для аргументов командной строки, в случае если программа использовала их. Взятая для примера программа не использует их, но объявление в скобках все же необходимо. Что обозначает такое объявление, будет объяснено позже.

4. Четвертую и седьмую строки кода довольно легко объяснить. Все определения метода должны быть заключены в соответствующий набор фигурных скобок `{}`.

5. В пятой строке происходит создание объекта. Классом объекта является `MyTerminalIO`. Название объекта — `myterminal`. Ключевое слово `new` находится непосредственно перед вызовом конструктора, представляющего собой имя класса, за которым следуют круглые скобки.

   ```
   MyTerminalIO myterminal = new MyTerminalIO();
   ```

   Этот объект, `myterminal`, является экземпляром класса `MyTerminalIO` и может использовать метод, который осуществляет вывод на консоль. Вы увидите позже, что этот класс также поддерживает ввод с клавиатуры.

6. Шестая строка кода выполняет реальную работу программы. Этот пример показывает, как можно выполнить полезную работу, используя объекты. Имя объекта — `myterminal`. Затем следует точка, после которой — имя метода. Метод `println()` вызывается для объекта с параметром "Hello, World!".

   ```
   myterminal.println("Hello, World!");
   ```

   Если вам удалось скомпилировать и запустить программу, рассмотренную в *главе 1*, вы знаете, что она делает. Она выводит сообщение "Hello, World!" на экран. Это результат вызова метода `println()`. Последовательность символов в двойных кавычках называется *строкой*, и эта последовательность может быть выведена на экран. Возможность сделать это является отправной точкой для изучения того, что можно сделать с помощью языка Java.

2.2. Структура кода и результат вывода программы на экран

Комментарии представляют собой пояснения, которые программист может включить в программу. Они не являются частью кода и не влияют на работу программы. В любой строке, где появляются символы `//`, чтобы ни следовало за этой парой символов, рассматривается как комментарий. Другими словами, компилятор не будет пытаться компилировать этот фрагмент кода, и он не будет выполняться, когда программа заработает. Аналогичным

образом, если что-либо появляется между символами `/*` и `*/`, то это будет рассматриваться как комментарий. Вот версия первой программы с комментариями:

```
/*  Это моя первая программа.  */
public class FirstProg
{
  public static void main(String[] args)
  {
    // Здесь создается объект.
    MyTerminalIO myterminal = new MyTerminalIO();
    // Здесь вызывается метод к объекту.
    myterminal.println("Hello, World!");
  }
}
```

При написании программы полезно выделять каждый блок программы отступом фигурных скобок так, чтобы было ясно, где начинается и где заканчивается каждый блок кода. Редактор TextPad сделает это для вас автоматически, и все приведенные примеры будут показаны в этой форме. Обратите внимание на то, что внутри блоков кода могут быть еще и вложенные блоки, и каждый из них пишется с отступом.

Приведем краткое описание некоторых аспектов вывода на экран с использованием метода `println()` и связанных с ним методов.

1. Метод `println()` выводит строку на экран и осуществляет переход на новую строку или возврат каретки. Вы также можете использовать метод `print()`, который просто печатает строку без возврата каретки. Полный вызов метода имеет следующий вид:

   ```
   myterminal.print("Some String");
   ```

2. Вы можете печатать арифметические величины, так же как и строки символов. Следующий вызов метода напечатает число 3:

   ```
   myterminal.println(3);
   ```

   В результате следующего вызова на экране также появится 3, но имеются существенные различия в значении этих двух вы-

зовов. Если в предыдущем примере был показан вывод числа, то в следующем — это вывод строки, содержащей символ 3:

```
myterminal.println("3");
```

3. Вы можете вывести на экран более одного параметра за один раз. Для этого используется знак +. Печатая строки, если вы хотите получить пробелы между словами, вы должны их указать. Если вы сделаете так:

   ```
   myterminal.println("Hello," + "World!");
   ```

   то увидите:

   ```
   Hello,World!
   ```

   Однако вы можете также сделать:

   ```
   myterminal.println("Hello," + " World!");
   ```

   или

   ```
   myterminal.println("Hello," + " " + "World!");
   ```

   и тогда вы увидите:

   ```
   Hello, World!
   ```

4. Знак +, когда вы выводите на экран числа, имеет другое значение. Если вы набрали:

   ```
   myterminal.println(3 + 4);
   ```

   то увидите:

   ```
   7
   ```

5. Вы также можете комбинировать параметры в кавычках с числовым значением без кавычек, используя знак +. В таких случаях система автоматически конвертирует числовое значение в строку и осуществляет конкатенацию. Если вы сделаете так:

   ```
   myterminal.println("Hello" + 7)
   ```

   то вы увидите следующее:

   ```
   Hello7
   ```

   Более полное объяснение способов употребления знака + будет дано позже. Пока вы должны просто уметь пользоваться им правильно со строками и числами.

6. Хотя точка имеет специальное синтаксическое значение в разных местах кода Java, в тех случаях, где допускаются простые параметры или переменные, например при выводе на экран, она имеет свое привычное значение — точка десятичной дроби. Таким образом, если в вашей программе есть строка кода:

   ```
   myterminal.println(3.4);
   ```

   то вы увидите такое значение:

   ```
   3.4
   ```

7. В Java символ `\` используется для вывода на экран escape-последовательности. Это означает, что любой символ, следующий непосредственно за обратным слэшем, рассматривается как выводимый на экран символ. Это позволяет распечатывать символы, которые в противном случае были бы двусмысленными или синтаксически неверными в параметре печати. Например, если вы хотите напечатать двойные кавычки, то можете сделать следующее:

   ```
   myterminal.println("\"");
   ```

   Если вы хотите напечатать сам символ `\`, то можете указать строку кода:

   ```
   myterminal.println("\\");
   ```

   Символ `\` может быть также использован в некоторых командах для вывода строк на экран. Так, `\n` напечатает новую строку. Таким образом, если вы зададите строку кода:

   ```
   myterminal.println("Hello,\nWorld!");
   ```

   то увидите:

   ```
   Hello,
   World!
   ```

2.3. Документация Java API

На рис. 2.1 приводится отрывок из документации Java API (Java Application Programming Interface) для имеющегося в системе класса `Point`. Полную документацию Java API можно найти в Интернете по адресу: **http://java.sun.com/j2se/1.5.0/docs/api/**. Это лучший доступный справочник в тех ситуациях, когда у вас есть

вопросы по программированию на языке Java. После рисунка дано объяснение класса `Point` и некоторых положений документации.

java.awt

Class Point

```
java.lang.Object
  └java.awt.geom.Point2D
      └java.awt.Point
```

All Implemented Interfaces:

Serializable, Cloneable

```
public class Point
extends Point2D
implements Serializable
```

A point representing a location in (x, y) coordinate space, specified in integer precision.

Since:

JDK1.0

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.geom.Point2D
Point2D.Double, Point2D.Float

Field Summary

int	**x** The *x* coordinate.
int	**y** The *y* coordinate.

Рис. 2.1. *(См. продолжение)*

Constructor Summary

Point() Constructs and initializes a point at the origin (0, 0) of the coordinate space.
Point(int x, int y) Constructs and initializes a point at the specified (*x*, *y*) location in the coordinate space.
Point(Point p) Constructs and initializes a point with the same location as the specified Point object.

Method Summary

boolean	**equals**(Object obj) Determines whether or not two points are equal.
Point	**getLocation**() Returns the location of this point.
double	**getX**() Returns the X coordinate of the point in double precision.
double	**getY**() Returns the Y coordinate of the point in double precision.
void	**move**(int x, int y) Moves this point to the specified location in the (*x*, *y*) coordinate plane.
void	**setLocation**(double x, double y) Sets the location of this point to the specified double coordinates.
void	**setLocation**(int x, int y) Changes the point to have the specified location.
void	**setLocation**(Point p) Sets the location of the point to the specified location.
String	**toString**() Returns a string representation of this point and its location in the (*x*, *y*) coordinate space.
void	**translate**(int dx, int dy) Translates this point, at location (*x*, *y*), by dx along the *x* axis and dy along the *y* axis so that it now represents the point (x + dx, y + dy).

Рис. 2.1. *Окончание*

Геометрически точка имеет некоторое расположение в пространстве. Класс `Point` в Java содержит информацию, необходимую для представления точки в двумерном пространстве. Для объектов этого класса переменными экземпляра будут координаты *x* и *y*. В Java координата *x* изменяется слева направо, координата *y* — сверху вниз. Позднее вы сможете графически изобразить точку на экране. Пока же экземпляры этого класса не будут иметь графического представления. Однако можно сконструировать экземпляры данного класса и совершать манипуляции ими посредством методов.

Информация в начале документации касается имени класса и его расположения. Классы, предоставляемые системой, находятся в пакетах. Если вы посмотрите информацию в начале, то увидите следующее: `Java.awt.Point`. Сокращение awt означает abstract windowing toolkit. Это имя пакета в Java, который включает классы, связанные с созданием графических интерфейсов. `Point` является одним из этих классов. Если в программе используется один из классов, предоставляемых системой, то подобная строка будет стоять в шапке кода:

```
import java.awt.Point;
```

Идея заключается в том, что данная строка делает класс доступным для использования в программе. Это будет показано в приводимых примерах программ. Можно импортировать весь пакет программ одновременно. Тогда нужно будет использовать знак `*`:

```
import java.awt.*;
```

Следующим разделом документации, который представляет для нас интерес, является раздел ***Field Summary***. То, что мы в *главе 1* называли переменными экземпляра, в документации называется *fields*. Другими словами, в документации говорится об объекте класса `Point`, имеющем 2 переменные экземпляра: *x*-координату и *y*-координату. Эти переменные имеют тип `int`, который означает, что указанные координаты могут принимать любые значения целых чисел.

Затем, в документации идет раздел ***Constructor Summary***. Конструкторы представляют собой специальные разделы кода, которые используются для создания экземпляров классов. Конструкторы

имеют форму, напоминающую форму методов — у них есть имя, за которым следуют скобки, содержащие или не содержащие параметры. Однако они не являются методами, и имя конструктора является таким же, как имя класса, к которому он принадлежит. Как видите, один класс может иметь более одного конструктора. Система различает их, т. к. у них разные списки параметров. В документации показаны типы параметров. Пока мы ограничимся примерами с параметром типа `int`.

Последней частью документации является ***Method Summary***. Здесь приводятся имена методов, их параметры и типы. Могут существовать разные методы с одним именем, которые различаются тем, что они имеют разный список параметров. Только через эти методы программа может влиять на переменные экземпляра — координаты *x* и *y* — объекта `Point`, который был создан. Как вы видите, `int` и `double` являются двумя различными числовыми типами. Значения типов будут описаны в следующей главе. Пока же все примеры будут ограничены использованием типа `int`, т. е. типа, принимающего целые числовые значения.

2.4. Создание и использование объектов

Написание кода для ваших собственных классов будет рассмотрено позже. Вы уже создали и использовали объект класса `MyTerminalIO`. Сейчас у нас есть возможность понять и правильно написать строки кода и маленькие логически завершенные программы, которые создают объекты классов, имеющихся в системе, и затем использовать их.

После того как класс `Point` был импортирован в программу, следующие две строки кода позволят вам:

- объявить имя, которое может быть использовано для объекта `Point`;
- сконструировать объект и придать ему имя, используя знак `=`.

```
Point mypoint;
mypoint = new Point(10, 20);
```

Таков синтаксис для вызова конструктора и передачи ему параметров:

```
new Point(10, 20);
```

Для того чтобы вызвать конструктор, необходимо использовать ключевое слово `new`. В показанном вызове координаты *x* и *y* инициализируются значениями 10 и 20 соответственно.

```
Point mypoint = new Point(10, 20);
```

Когда вы создаете объект и даете ему имя, это служит своего рода рукояткой для объекта. Имя `mypoint` является такой рукояткой. С ее помощью можно ссылаться на объект и манипулировать позже им в программе. Чтобы пояснить эту мысль, рассмотрим пример, в котором создается объект, но ему не придается никакого имени:

```
new Point(10, 20);
```

Синтаксически здесь все правильно. Эта строка не вызовет ошибки компилирования или запуска программы. Однако данный объект не может быть доступен в программе, т. к. у него нет имени.

Ранее мы говорили о приписывании объекту имени. Более точный термин, который будет использоваться в дальнейшем, — это *ссылка на объект*. Вызов конструктора возвращает безымянную ссылку на объект. Когда вы объявляете имя, например `mypoint`, и приписываете ему безымянную ссылку, то это имя и становится ссылкой на объект.

Представление о названных и безымянных ссылках можно проиллюстрировать еще одним примером. Рассмотрим следующую синтаксически правильную строку кода:

```
myterminal.println(new Point(10, 20));
```

Этот вызов работает, т. к. код, содержащий вложенные скобки, выполняется изнутри наружу. Сначала создается новая точка, и когда ее создание завершено, безымянная ссылка к этому объекту служит параметром для метода `println()`. Этот пример показывает, что вызов конструктора вызывает появление объекта, даже если он остается безымянным. Обратите внимание на то, что, несмотря на свою компактность, коды, в которых один вызов вы-

полняется внутри другого, или коды с безымянными ссылками, могут быть достаточно трудными.

Как только объект был создан и назван, к нему могут быть применены методы. Класс `Point` имеет метод `translate()`, который перемещает расположение точки на плоскости путем прибавления или вычитания значений к ее координатам *x* и *y*. Рассмотрим строку кода, которая иллюстрирует вызов метода к объекту:

```
mypoint.translate(30, 40);
```

Вызов имеет следующий вид: ссылка на объект, точка, имя метода и список параметров. Именно точка указывает на вызов метода к объекту. Важно отметить синтаксическую разницу между вызовом конструктора и вызовом метода. Вызов метода не использует ключевое слово `new`.

Результатом выполнения этой строки кода будет перемещение точки, которая выше была построена с координатами *x* и *y*, равными 10 и 20, в новое место с координатой *x*, равной 10 + 30, или 40, и координатой *y*, равной 20 + 40, или 60. Строка кода, которая выполняет вызов метода, не принимает вид приписывания. Значения приписываются переменным экземпляра в результате вызова, но работа по созданию приписываний скрыта внутри кода метода. Класс `Point` является черным ящиком, поскольку не обязательно видеть код метода для того, чтобы правильно применить этот метод.

Вот модель, по которой производится вызов метода и которая будет повторяться снова и снова, с изменениями:

```
object.method(parameters);
```

На этом этапе мы также можем вывести объект на экран. Методы вывода на экран будут принимать ссылки объекта в качестве параметров. Если данный объект существует, то следующая запись будет работающей строкой кода:

```
myterminal.println(mypoint);
```

Мы еще не программируем графику, поэтому на экране вы увидите:

```
Java.awt.Point[x=10,y=20]
```

Система просто говорит вам, какой это тип обращения и каковы текущие значения переменных экземпляра.

Теперь можем написать пример программы, которая включает все описанные ранее свойства.

```
import java.awt.Point;
/*  Это пример второй программы.  */
public class SecondProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Point mypoint = new Point(10, 20);
    myterminal.println(mypoint);
    mypoint.translate(30, 40);
    myterminal.println(mypoint);
  }
}
```

2.5. Ошибки

Ошибки программирования можно классифицировать различными способами. Один из них — это классифицировать ошибки в зависимости от того, когда они обнаруживаются: до выполнения программы, во время ее выполнения или после выполнения. Иным способом, связанным с первым, будет деление ошибок на синтаксические и логические. Синтаксические ошибки — это ошибки в использовании правил языка программирования. Логические ошибки — это ошибки в понимании проблемы и в предложении путей ее решения. Далее приводится общая информация об этих категориях ошибок.

- *Когда обнаруживается ошибка*: во время составления программы.

  Тип ошибки: синтаксические.

  Пояснение. Компилятор будет пытаться обнаружить синтаксические ошибки. Программа, которая содержит такие ошибки, не может быть выполнена.

- *Когда обнаруживается ошибка*: во время выполнения программы.

  Тип ошибки: синтаксические или логические.

  Пояснение. Система также может обнаружить ошибки во время выполнения и прекратить работу программы с сообщением об ошибке.

- *Когда обнаруживается ошибка*: после выполнения программы.

  Тип ошибки: логические.

  Пояснение. Программа, которая выполняется до конца, может выдать явно неверные результаты. Программа также может выдать неверные результаты, которые не будут столь очевидными. Задача программиста — тестировать программы и выверять результаты их работы. Компилятор и система не могут этого сделать.

Три самые распространенные, простые и наиболее часто досаждающие начинающим программистам синтаксические ошибки в Java — следующие:

- *отсутствие точки с запятой в конце строки кода*. Это будет раздражать вас, т. к. сообщение об ошибке будет говорить вам, что ошибка имеется в соседней строке, а не в строке, где пропущена точка с запятой;

- *отсутствие вторых скобок*. Это тоже будет мешать работе, потому что компилятор не будет распознавать строку, в которой произошла ошибка. Если вы будете располагать строки с отступом, то шансы сделать такую ошибку уменьшаются, а шансы заметить такую ошибку, в случае если вы ее допустили, увеличиваются;

- тот факт, что *Java различает прописные и строчные буквы*, также может быть причиной ошибок. Отказ от использования прописной буквы там, где это необходимо, или указание ее там, где она не нужна, может привести к проблемам, но человеку исключительно трудно заметить такие незначительные различия в коде.

  Ошибки в использовании прописной буквы могут быть особенно чреваты последствиями. В большинстве случаев они

происходят по невнимательности к деталям или в силу опечаток. Иногда они приводят к синтаксическим ошибкам, которые могут быть выявлены компилятором или которые будут выявлены во время работы программы. Однако они могут также привести к логическим ошибкам, которые станут очевидными только во время работы программы или когда проверяется результат программы. Иными словами, такие ошибки могут вызывать проблемы, которые, кажется, стирают грань между синтаксическими и логическими ошибками. По этой причине, а также потому, что трудно обнаружить пропущенную или ошибочно употребленную прописную букву, такие ошибки трудно обнаружить.

В языке, который использует ссылки на объекты, имеется еще один источник ошибок. Это происходит при попытке использовать ссылки на объект, который еще не был создан. Рассмотрим следующий фрагмент кода, содержащего подобную ошибку:

```
...
Point mypoint;
myterminal.println(mypoint);
...
```

В этом примере объявляется ссылка, но объект не создается. Ссылка ни к чему не отсылает. В коде эта ссылка передается методу `println()` в качестве параметра. Если запустить программу, то компилятор выдаст сообщение об ошибке, указывая, что объект `mypoint` не был инициализирован. Теперь рассмотрим следующий пример:

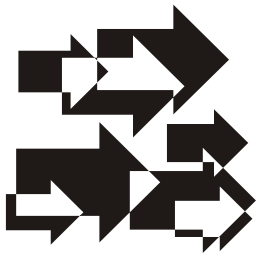
```
...
Point mypoint;
mypoint.translate(30, 40);
...
```

На этот раз вызывается метод по ссылке, которая не отсылает к объекту. Компилятор обнаружит и эту ошибку и сообщит вам, что `mypoint` не был инициализирован.

Компилятор Java и среда, в которой выполняется программа, стараются обнаружить ошибки во время выполнения и выводят на

экран сообщения об ошибках. Однако они не могут обнаружить все ошибки, и иногда сообщения об ошибках могут быть достаточно непонятными. По мере того как все более сложные понятия и синтаксис становятся доступными, а программы усложняются, ошибки могут становиться более трудноуловимыми, и программист должен быть осторожным во время программирования и внимательным при выявлении ошибок в программах. Ссылки на объект являются принципиально важной частью программирования на Java. Их правильное использование и ошибки, вытекающие из их неверного использования, будут более подробно рассмотрены в следующих главах.

ГЛАВА 3



Типы данных, операторы и переменные

3.1. Типы данных, объявление переменных и инициализация

В языке Java существует несколько типов числовых данных:

- `int` — целое число;
- `long` — длинное целое число;
- `float` — число с плавающей точкой с одинарной точностью;
- `double` — число с плавающей точкой с двойной точностью.

Существуют и другие числовые типы, но ни один из них пока не понадобится. Для большинства операций достаточно уметь использовать типы `int` и `double`. Однако знать о существовании других типов, таких как `long` и `float`, необходимо для более полного понимания языка Java.

Любое число с дробной частью, например `3.0`, будет рассматриваться как число с плавающей точкой, а не как целое значение. Также существует экспоненциальное представление очень больших и очень малых значений, но здесь это разбираться не будет.

Тип `double` иллюстрирует те ловушки, которые ожидают вас в связи со значением регистра букв в языке Java. В системе Java имеется класс `Double`. Если вы по ошибке употребите прописную букву в объявлении переменной, у вас возникнут проблемы в ко-

де, что приведет к непонятным ошибкам в программе. В таком случае очень трудно определить ошибку, когда это просто разница в прописной и строчной букве d в одной из строк кода.

Переменные являются контейнерами для значений. В результате, когда переменная объявлена, в памяти компьютера отводится определенное пространство для хранения значения такого типа. Существуют следующие правила и рекомендации использования переменных:

- при объявлении переменной необходимо указать ее тип;
- в идеале, переменные следует инициализировать с конкретными значениями;
- переменным следует давать описательное имя;
- имена переменных должны удовлетворять следующим требованиям:
  - они могут содержать буквы, цифры, знак `$` и знак подчеркивания (_);
  - они не могут начинаться с цифры;
  - имя переменной не может совпадать ни с одним из ключевых слов Java;
  - имена переменных чувствительны к регистрам букв;
  - имена переменных не могут содержать пробелов.

Третья часть четвертого правила является не совсем справедливой по отношению к тем, кто только начинает изучать язык. Невозможно заранее знать все ключевые слова. В нашей книге используется следующий прием: чтобы избежать использования ключевых слов или названий системных классов: названия классов, написанных авторами, начинаются с `my`. Так, `double` является ключевым словом, но `mydouble` не будет им. Таким образом, `mydouble` является приемлемым именем для переменной.

Вспомните, что названия классов должны начинаться с прописной буквы. Синтаксически имена переменных можно начинать с прописной буквы, но этого не следует делать, иначе вы спутаете переменные и классы. В предыдущей главе было введено понятие объектной ссылки. Следует отметить, что объявление типа и

приписывание значения указывает, что ссылка на объект функционирует как переменная. Особенно важно использовать принятые правила именования, которые позволят вам различать классы и экземпляры этих классов. Как и в случае с переменными, названия объектных ссылок также не следует писать с прописной буквы.

Чтобы сделать имя переменной описательным, довольно часто желательно использовать сложные слова. Хотя это требует большего набора символов, код легче понять и для него не требуются длинные пояснения. Синтаксически, возможны следующие виды сложных слов: `payRate`, `pay_rate`, `payrate` и т. д.

Дефис (-) нельзя употреблять в названии переменных, потому что компилятор интерпретирует его как знак минус или вычитание.

В нашей книге будет использоваться первый вариант из указанных выше, т. е. первая буква сложного слова пишется строчной, но последующие слова в составе сложного пишутся с прописной буквы. Именно такой стиль в основном используется в документации Java API. Вы также можете встретить имена переменных, которые используют один из других видов записи сложных слов.

Как уже было сказано, мы называем некоторые переменные с использованием элемента `my`, за которым следует имя типа. Это полезно в тех примерах, где интерес представляет именно тип переменной. В примерах, где имеется несколько переменных данного типа, в конце имени может стоять цифра для различения их, например `myinteger1`. В других примерах может использоваться префикс `your` или `their` вместо `my`. Этот стиль наименования был использован в именах `myterminal` и `mypoint`. В данной книге, если имя переменной состоит только из `my` (или `your` и т. д.) и еще одного слова, то мы будем писать и второе слово со строчной буквы.

Рекомендуется объявлять или обозначать переменные в начале блока кода, где они используются. Объявления переменных, которые включают инициализацию, являются предварительными примерами того, как присвоить значение переменным:

```
double payRate;
double taxRate = .05;
```

```
int daysInWeek;
int weeksInYear = 52;
int hoursInDay, minutesInHour;
```

Как было отмечено ранее, объявление переменной устанавливает место в памяти для ее значения. Если переменная не инициализирована, значением переменной будет то, что содержится в выделенном пространстве памяти, когда программа будет работать. Инициализация — это хорошая привычка, поскольку вы, вероятно, не захотите, чтобы вычисления зависели от значений, которые являются результатом случайных состояний системы. Компилятор попытается предупредить вас о переменных, которые не были инициализированы. Однако, для того чтобы избежать ошибок позже, полезно инициализировать переменные с подходящими исходными значениями.

Можно объявить и постоянные величины. Вот синтаксис для инициализации переменной с данным значением, которое не может быть позже изменено:

```
final int WEEKS_IN_YEAR = 52;
```

Если только вы не путешествуете с планеты на планету, продолжительность года не меняется. Если величина объявлена как `final` и ей присвоено некоторое значение, то любые последующие приписывания значения приведут к ошибке компилятора. Обычно используются прописные буквы при объявлении постоянной величины. Это — единственное исключение из правила о том, что имена переменных не следует начинать с заглавной буквы. Также обычно разделяют части имени таких величин знаком подчеркивания, если имя состоит из нескольких частей.

3.2. Присваивание и отношения между числовыми типами

Когда переменная объявлена, ей может быть присвоено значение. В типичной программе последовательность будет следующей:

```
int myinteger;  /* объявление */
...
myinteger = 7;  /* присваивание */
```

Язык Java использует знак = как *оператор присваивания*. Значение с правой стороны оператора сохраняется в памяти, отведенной именем переменной с левой стороны. Проблема возникает тогда, когда тип переменной и тип присваиваемого значения не совпадают. В некоторых случаях система автоматически конвертирует значение справа к типу переменной слева. В других случаях присвоение синтаксически неверно. Если тип значения справа не может содержать больше информации, чем тип переменной слева, то присваивание возможно, и произойдет автоматическая конверсия. Однако если тип значения справа может содержать больше информации, чем тип переменной слева, тогда автоматическое конвертирование может привести к потере информации. В этом случае присваивание невозможно.

В общем:

- число `int` может быть сохранено в `long`;
- число `float` может быть сохранено в `double`;
- число `int` или `long` может быть сохранено или во `float`, или в `double`;
- число `long` не может сохраняться в `int`;
- число `double` не может сохраняться во `float`;
- ни `float`, ни `double` не могут сохраняться в `int` или в `long`, потому что дробная часть числа была бы потеряна.

Даны следующие типы объявлений:

```
double taxRate = .05;
int weeksInYear = 52;
```

Вот два простых примера, показывающих, что верно и что неверно:

```
taxRate = weeksInYear;  /* верно */
weeksInYear = taxRate;  /* неверно */
```

Теперь изменим неверную запись так, чтобы сделать ее синтаксически верной. Этот новый синтаксис называется *приведением типов*, и он явным образом вызывает конверсию одного типа в другой:

```
weeksInYear = (int) taxRate;
```

Значение справа приводится к типу слева за счет того, что в скобках перед количеством справа указывается требуемый тип. Приведение типов требуется от программиста, потому что эта процедура функционирует как контракт. Когда программист производит приведение типов, он берет на себя ответственность за все, что, возможно, было утеряно из-за того, что значение слева не может содержать всей информации справа. Когда исходят от типа "плавающей точки" к типу целого числа, приведение типов заканчивается усечением. Округление не происходит, и все данные справа от десятичной точки теряются. Например:

```
double taxRate = .05;
int weeksInYear = 52;
weeksInYear = (int) taxRate;
```

Вся дробная часть значения `taxRate` потеряна, и конечным результатом является то, что `weeksInYear` содержит целое значение, равное 0.

3.3. Присваивание и простые арифметические операции

Чтобы понять роль переменных, можно представить их в виде контейнеров, которые могут быть использованы более одного раза и иметь разное содержимое во время выполнения программы. Переменные в основном используются для хранения результатов арифметических операций, и выражения присваивания применяются для выполнения арифметических операций в программе.

Простые *арифметические операторы* — это знаки сложения, вычитания, умножения и деления, со скобками для группирования. Если порядок выполнения действий не изменен скобками, умножение и деление выполняются перед прибавлением и вычитанием. При прочих равных условиях действия производятся слева направо.

Арифметика может включать и переменные, и числовые значения. Пусть даны следующие объявления переменных:

```
int myinteger1 = 10;
int myinteger2 = 20;
int myinteger3 = 30;
double mydouble1 = 10.0;
double mydouble2 = 20.0;
double mydouble3 = 30.0;
```

Следующие выражения возможны:

```
myinteger1 = 4 + myinteger3;
mydouble1 = mydouble2 * mydouble3;
```

Проблемы, связанные с преобразованием типов, также встречаются при арифметических операциях. В некоторых случаях допустимо смешивать типы операндов без проблем. В других случаях могут возникнуть проблемы. Например, следующее выражение возможно:

```
mydouble1 = myinteger1 — mydouble2;
```

Почему можно соединить целочисленную и вещественную переменные без проблем? Справа происходит автоматическая конверсия. При смешении типов, все они повышаются до того из присутствующих типов, который может вмещать наибольшее количество информации. Другими словами, `myinteger1` рассматривается как `double`, результат является `double`, и это может быть сохранено в `mydouble1`.

Однако следующее выражение невозможно:

```
myinteger1 = myinteger2 / mydouble;
```

Результаты справа переводятся в `double`, но это нельзя сохранить в переменной целого типа. В общем, лучше всего не полагаться на автоматическую конверсию. Если сомневаетесь, четко приведите все переменные к типу, который вам требуется в арифметических выражениях.

Теперь рассмотрим результат следующего выражения:

```
myinteger1 = myinteger2 / myinteger3;
```

Деление потенциально ведет к результату с дробной частью. Однако когда целые числа делятся в Java, то результатом будет только целая часть ответа. Другими словами, результат — это

сокращенное целое числовое значение, и правильно сохранить его в другой переменной целого числа.

Кроме того, при делении целого числа имеется возможность получить остаток. Арифметическим символом для такой операции будет `%`, называемый *модульным оператором*. В следующем примере значение 10 будет присвоено `myinteger1`, как остаток от деления 30 на 20:

```
myinteger1 = myinteger3 % myinteger2;
```

Преимущество переменных в том, что им могут быть присвоены новые значения, которые зависят от их текущих значений. Другими словами, следующие утверждения справедливы:

```
myinteger1 = myinteger1 + 1;
mydouble1 = mydouble1 - 5.0;
```

Такие инкрементирующие и декрементирующие операции так часты, что для них существует сокращенный синтаксис. Показанные далее примеры являются эквивалентными утверждениями к приведенным выше:

```
myinteger1++;
mydouble1 -= 5.0;
```

Такой сокращенный синтаксис работает одинаково для знаков `+` и `-`. Если вам нужно уменьшить значение `myinteger1` на `1` и увеличить значение `mydouble1` на `5.0`, то необходимо сделать, соответственно, следующее:

```
myinteger1--;
mydouble1 += 5.0;
```

Нам осталось рассмотреть несколько вопросов, касающихся простых арифметических операций. Из предыдущего материала можно было сделать вывод, что переменная может быть объявлена тогда, когда нужно. Таким образом, синтаксически приемлемо утверждение, подобное следующему:

```
int myinteger4 = myinteger2 + myinteger3;
```

Не касаясь сейчас недальновидности такого программирования, нужно отметить: подобная практика является причиной еще одного типа ошибок. Если вместо одновременного объявления всех

необходимых переменных в начале программы вы имеете привычку объявлять их в процессе работы, то таким образом вы можете объявить одну и ту же переменную более одного раза.

Кроме того, можно производить арифметические действия внутри выражений для вывода на экран. Полагая, что все переменные были объявлены и инициализированы, следующее выражение будет правильным:

```
myterminal.println(myinteger1 + myinteger2);
```

Произведение арифметических действий внутри таких выражений также является недальновидным; недостаток здесь заключается в том, что арифметический результат недоступен для будущего использования, поскольку он не был сохранен в другой переменной.

Нам остается сделать последнее, очень важное наблюдение относительно арифметических действий в Java. Как и в большинстве языков, числовые значения в Java хранятся в двоичной системе счисления. Результаты некоторых десятичных выражений могут не иметь точных представлений в двоичной системе счисления. Некоторые языки пытаются облегчить жизнь, показывая результаты в форме, которую ожидает пользователь. Java использует иной подход и показывает пользователю точное значение, использовавшееся в программе. Производя простое арифметическое действие и ожидая результат, скажем `0.5`, вы можете увидеть такое значение, как `0.49999999`. Вам не следует удивляться. Есть эксплицитные способы отформатировать результат так, чтобы вы получили значение `0.5`, но мы не будем их сейчас рассматривать. Пока же просто необходимо помнить, что Java никого не обманывает на этот счет.

3.4. Класс *Math*

Система Java делает доступными многие математические константы и функции. Она осуществляет это посредством класса, называемого `Math`. Для получения полной информации по этому классу и о его содержимом обратитесь к документации Java API. В этом разделе будут рассмотрены некоторые характеристики,

иллюстрирующие свойства данного класса. Прежде всего, чтобы использовать класс в программе, сделайте следующее:

```
import Java.lang.Math;
```

Класс `Math` содержит определенные математические константы, такие как π и *e*. Эти константы могут быть использованы в ваших программах под именами `Math.PI` и `Math.E`. Имена переменных написаны заглавными буквами, поскольку они являются *константами*. Полагая, что переменные объявлены и инициализированы должным образом, рассмотрим пример использования одной из этих констант:

```
area = Math.PI * radius * radius;
```

В данном примере за именем класса следует точка, за которой идет имя переменной. Хотя эта переменная находится в классе `Math`, она доступна для использования в программе без необходимости прибегать для этого к методу. Такая переменная называется *статической переменной*, которая отличается от переменных экземпляра, описанных ранее. Далее мы остановимся на статических переменных более подробно.

Также важно отметить, что данный класс существует без создания его экземпляров. Переменные класса `Math` доступны для использования без объекта. Класс может служить моделью для создания объектов, но он также может выступать контейнером для полезных вещей. Класс `Math` существует только как контейнер, и константы — это пример тех полезных вещей, которые он содержит.

Класс `Math` также содержит сложные математические операторы, такие как тригонометрические функции и т. д. Некоторые из них будут приведены в книге в качестве примеров. Так же как с математическими константами, совсем не обязательно наличие экземпляра класса `Math` для того, чтобы использовать его методы. В этом смысле методы класса `Math` похожи на метод `main()` в ваших программах. Такие методы называются *статическими методами*.

В предыдущих примерах вызова метода модель была следующей:

```
object.method(parameters);
```

При использовании статических методов нет объекта для вызова метода. Вся информация, которая передается в методы, передается как параметры, и ожидается возвращаемое значение, для того чтобы выявить результат. Используется следующая модель:

```
returnValue = ClassName.method(parameters);
```

Переменная `returnValue` должна быть правильно объявлена, чтобы возвращаемое значение метода совпадало с ее типом. Кроме того, типы передаваемых параметров должны согласовываться с типами параметров, использующихся в вызываемом методе.

Рассмотрим схематичное объяснение некоторых функций в классе `Math`. Заметьте, что объявление начинается с объявления типа возвращаемого значения, а параметры объявляются с их типами внутри скобок.

```
double exp(double x)
```

Этот метод возвращает число *e*, возведенное в степень `x`.

```
double pow(double x, double y)
```

Этот метод возвращает число `x`, возведенное в степень `y`.

```
double log(double x)
```

Этот метод возвращает логарифм числа `x`.

```
double sqrt(double x)
```

Этот метод возвращает квадратный корень числа `x`.

```
double abs(double x)
```

Этот метод возвращает абсолютное значение числа `x`.

Полагая, что класс `Math` доступен для использования и имеются все необходимые части программы, рассмотрим пример, иллюстрирующий использование одного из этих методов:

```
double someValue = 3.2;
double resultValue;
resultValue = Math.sqrt(someValue);
```

Ошибка, которую вы, возможно, сделаете, будет выглядеть примерно так:

```
resultValue = someValue.sqrt();  // Нет! Нет! Нет!
```

Это неправильно по двум причинам: методы класса `Math` не могут вызываться для объектов; то, над чем вы хотите произвести действия, должно быть передано как параметр. Кроме того, `someValue` — это переменная, а не объект, и в любом случае невозможно вызвать метод для переменной.

Вот еще одна иллюстрация использования одного из этих методов с помощью знакомой формулы кода Java: $A = \pi r^2$:

```
area = Math.PI * Math.pow(radius, 2.0);
```

Помните, что эти методы представляют собой черные ящики. Пользователю нужно только знать, как их использовать, а не как они действуют. Вы можете найти все необходимые типы параметров и типы возвращаемых значений в документации Java API, но там вы не найдете описание самого кода.

3.5. Класс *String*

Строка — это последовательность символов, которые рассматриваются вместе как группа. Они относятся к числу самых часто манипулируемых данных, наряду с простыми числовыми типами. Мы уже сталкивались со строковой константой. Все, что заключено в двойных кавычках, является строкой, поэтому следующее выражение является строковой константой:

```
"Hello, World!"
```

В Java имеется класс `String`. Можно построить экземпляры этого класса и получить ссылки на них. Поскольку строки широко используются, и двойные кавычки тесно связаны с ними, существует специальный синтаксис для построения строк. Например:

```
String mystring = "Hello, World!";
```

Или:

```
String mystring;
mystring = "Hello, World!";
```

Вы можете использовать и стандартный синтаксис для построения строки, хотя он применяется редко:

```
String mystring = new String("Hello, World!");
```

Ссылка на строку теперь может быть использована следующим образом, например, с предполагаемыми результатами:

```
myterminal.println(mystring);
```

После того как ссылка объявлена, она, как и переменная, может быть снова использована. Иными словами, теперь можно сделать следующее с `mystring`:

```
mystring = "A new string";
```

Первоначально мы видели эту модель для вызова метода на объектах:

```
object.method(parameters);
```

Затем мы видели эту модель для использования статических методов:

```
returnValue = Class.method(parameters);
```

Поскольку `String` является классом, он имеет методы, связанные с ним. Один полезный метод позволяет вам выяснить количество символов в строковом объекте в любой момент времени. Следующий пример иллюстрирует использование этого метода:

```
int stringLength;
stringLength = mystring.length();
```

Этот пример показывает, что методы, вызываемые для объектов, могут также возвращать значения. В данном случае возвращенное значение полностью зависит от объекта, для которого вызывается метод. Нет необходимости указывать какие-либо параметры в скобках. Это пример метода, который возвращает содержимое некоторых переменных экземпляра, содержащихся внутри объекта. В целом, вызовы метода для объектов, которые возвращают значения, также могут принимать параметры. Это будет рассмотрено более подробно далее. Модель для этих видов вызова следующая:

```
returnValue = object.method(parameters);
```

Хотя это и не сразу заметно, класс `String` имеет особое свойство. Даже изучив все методы этого класса, вы не найдете ни одного метода, который бы позволял вам изменить строку. Вы можете

получить информацию о строковом объекте, можете получить его содержание и присвоить указателю на строку новое значение, но вы не сможете изменить саму строку. Это свойство называется *неизменяемостью*.

В Java индивидуальные символы строковых переменных могут быть определены с помощью номера их позиции в строке, начиная слева от нуля. Например, пусть:

```
String mystring = "Hello";
```

Тогда

```
Hello
01234
```

В классе `String` имеется метод, который возвращает ссылку на выбранный участок строки. Этот метод передает два целочисленных параметра. Первый параметр указывает позицию первого выбранного символа в строке. Второй параметр указывает позицию за последним выбранным символом в строке. Вот пример вызова, демонстрирующий его использование:

```
String otherString;
otherString = mystring.substring(0, 4);
myterminal.println(otherString);
```

Вывод:

```
Hell
```

В документации Java API имена параметров — это *beginIndex* и *endIndex*. Модель для метода может быть дана так, как это показано, имея в виду, что *endIndex* относится к позиции, следующей за последней интересующей нас позицией:

```
otherString =
  someString.substring(int beginIndex, int endIndex);
```

Обратите внимание, что метод не укорачивает исходную строку: она остается без изменений. Метод возвращает ссылку на требуемую подстроку, которая может быть сохранена в другой строковой ссылке.

В дополнение к выбору подстрок, имеется возможность использовать несколько строк вместе. Это называется *конкатенацией*.

Фактически, мы уже видели это. Когда мы организовывали вывод на экран, то можно было сделать что-то вроде этого:

```
myterminal.println("Hello," + " " + "World!");
```

Параметры в том вызове являются просто строковыми константами, и знак + служит символом для конкатенации строк. В итоге, нас не должно удивлять, что происходит следующая последовательность действий с ожидаемым результатом:

```
String mystring1, mystring2, mystring3, mystring4;
mystring2 = "Hello,";
mystring3 = " ";
mystring4 = "World!";
mystring1 = mystring2 + mystring3 + mystring4;
myterminal.println(mystring1);
```

Кроме того, как видно в выведенном результате, во время создания конкатенации допускается смешивание строк и цифровых значений. Вы можете выполнить это:

```
myterminal.println("The result is: " + 27);
```

Следующая последовательность действий дает тот же результат:

```
int result = 27;
String mystring = "The result is: ";
String anotherString = mystring + result;
myterminal.println(anotherString);
```

Когда знак + показан как параметр метода `println()`, он поначалу кажется каким-то пунктуационным знаком, вроде запятой. Однако он является оператором, но имеет особую характеристику. Когда он используется только для числовых типов, то выполняет операцию сложения. Когда он используется со строковыми переменными, то выполняет конкатенацию. Когда он используется с числовыми и строковыми типами, то превращает числовой тип в строковый и производит конкатенацию. Когда один и тот же оператор выполняет различные процедуры, в зависимости от операндов, такое явление называется *перегрузкой*. Эта тема будет еще раз затронута позже.

Поскольку знак + работает таким образом, вы можете превращать числовые значения в строковые. Пара двойных кавычек, внутри

которых ничего нет, даже пробела, называется *пустой строкой*. Если вы соедините ее с числовой переменной, то получите строку, содержащую только последовательность символов, передающих это значение. Другими словами, представленная далее последовательность даст показанный результат:

```
double mydouble = 3.75;
String mystring1 = "" + mydouble;
String mystring2 = mystring1.substring(2, 3);
myterminal.println("The digit in the 2nd place is: " +
                   mystring2);
```

Вывод:

```
The digit in the 2nd place is: 7
```

Как было ранее замечено, существуют классы с именами `Integer` и `Double`. Как и класс `Math`, данные классы содержат некоторые полезные методы. Например, можно превратить числа в строки посредством вызова метода, а не с помощью конкатенации. Следующий пример иллюстрирует это:

```
import Java.lang.Double;
...
double mydouble = 3.75;
String mystring = Double.toString(mydouble);
```

Метод `toString()` конвертирует числовое значение в строку, содержащую число.

Классы `Integer` и `Double` также содержат методы `parseInt()` и `parseDouble()`, которые позволяют превращать содержимое строк в числовые значения при условии, что строки содержат последовательности символов, которые образуют допустимые числа. Например:

```
import Java.lang.Integer;
...
String mystring = "375";
int myinteger = Integer.parseInt(mystring);
```

Существуют похожие классы для других числовых типов, а не только `int` и `double`. Они все содержат схожие методы, позволяющие конверсию из строковых в числовые значения и наобо-

рот, а также другие методы. Подробно об этом можно прочитать в документации Java API.

Последние два примера иллюстрируют одно положение, касающееся написания имен. Имена методов `toString()` и `parseInt()` используют сложные слова. Обратите внимание на то, что первые буквы имен пишутся строчными, но в остальных частях первые буквы пишутся прописными, аналогично тому, как это принято в именах переменных. Так принято в документации Java API, и такая запись будет использоваться в названиях методов, рассматриваемых в примерах нашей книги.

3.6. Класс *MyTerminalIO*

Ранее было показано, как пользоваться классом `MyTerminalIO`, чтобы выводить результаты работы программы на экран. Теперь рассмотрим, как с его помощью можно осуществить ввод данных. Если ввод ожидается от пользователя во время работы программы, то необходимо отобразить подсказку, запрашивающую ввод данных. Класс `MyTerminalIO` имеет три метода для осуществления ввода: `getString()`, `getInt()` и `getDouble()`. Существуют версии этих методов, которые не принимают параметров. Для этих методов необходимо вызвать `println()` или `print()` для того, чтобы создать подсказку до вызова методов ввода. Эти методы вызывают диалоговое окно на экран, где можно ввести необходимый тип. Когда вы нажмете клавишу <Enter>, то появится ссылка или значение для программы, где они могут быть сохранены под именем переменной или ссылки.

Вот пример короткой программы, иллюстрирующей использование методов ввода данных без параметров:

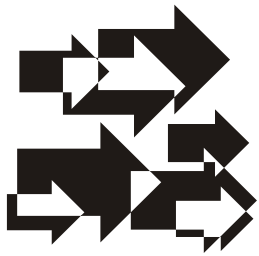
```
public class TestInput1
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    myterminal.println("Enter a string: ");
    String mystring = myterminal.getString();
    myterminal.println("Enter an integer: ");
```

```
    int myinteger = myterminal.getInt();
    myterminal.println("Enter a double: ");
    double mydouble = myterminal.getDouble();
  }
}
```

Также можно использовать версии методов ввода, которые принимают параметры. Параметр представляет собой строку с подсказкой. Если вы используете такие версии методов, то нет необходимости заранее выводить на экран подсказку. Она появится в диалоговом окне и автоматически повторится в области, где записываются действия программы. Вот пример короткой программы, иллюстрирующий методы ввода с параметрами:

```
public class TestInput2
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    String mystring =
      myterminal.getString("Enter a string: ");
    int myinteger =
      myterminal.getInt("Enter an integer: ");
    double mydouble =
      myterminal.getDouble("Enter a double: ");
  }
}
```

ГЛАВА 4



Как написать и использовать коды для классов

4.1. Переменные экземпляра, конструкторы и методы

Для определения класса необходимы три основных компонента:

- переменные экземпляра (которые называются полями (fields) в API-документации);
- конструкторы;
- методы.

В данной главе вы узнаете, как написать код для класса, определенного пользователем. Для этого мы рассмотрим каждый из трех компонентов по очереди, используя упоминавшийся в *главе 1* пример с чашкой и семенами.

Класс, определенный пользователем, должен получить имя, которое, как правило, пишется с большой буквы. Для того чтобы использовать такой класс в программе, его нужно объявить как общий (`public`). Код, определяющий класс, должен размещаться в фигурных скобках, следующих за объявлением. Таковы основы определения класса с именем `Cup1`:

```
public class Cup1
{
}
```

Термин "инкапсуляция" означает, что к некоторым частям объекта нет непосредственного доступа снаружи. Например, как правило, желательно, чтобы переменные экземпляра были доступны только с помощью методов. Для того чтобы ввести инкапсуляцию, переменные экземпляра объявляются частными (`private`). Далее показан код для класса `Cup1` с одной переменной экземпляра. Этот код синтаксически верен, но без методов мало, что можно сделать с этим классом:

```
public class Cup1
{
  private int seedCount;
}
```

Для того чтобы создать экземпляры класса, нужны конструкторы. Если программист не предоставляет конструкторы, как в приводившихся до сих пор примерах, то система предоставит конструктор по умолчанию. Для того чтобы использовать конструкторы или написать собственные, важно понимать их характеристики:

- они объявляются общими (`public`);
- у них нет типа; они всегда возвращают ссылку к экземпляру класса;
- они всегда имеют точно такие же имена, что и класс, включая регистры букв;
- они могут иметь 0 или более параметров; каждый из параметров должен получить свой тип;
- может быть более одного конструктора; они имеют одно имя, но система различает их по количеству и типу параметров;
- код внутри конструктора инициализирует переменные экземпляра объекта со значениями параметров, отправленных в момент вызова конструктора.

По мере развития нашего класса его новые версии будут обозначены изменением цифры в его имени. Следующая версия, `Cup2`, содержит конструкторы. Прежде чем дать полное определение класса, проиллюстрируем код для конструкторов. Эти конструк-

торы получат имя Cup2, чтобы отражать тот новый класс, в который они будут включены.

Код для конструктора пишется в виде отдельного блока в фигурных скобках внутри определения класса. Самый простой конструктор не содержит параметров и не имеет никакого кода. Такой конструктор инициализирует значение переменной экземпляра seedCount, равное 0. Программист может написать такой простой конструктор для своего класса:

```
public Cup2()
{
}
```

Если программист не напишет конструктор для своего класса, то в таком случае система предоставит конструктор, подобный указанному выше, сама. Код для такого конструктора не виден, но его, тем не менее, можно использовать.

Рассмотрим второй пример конструктора, написанного программистом. Он также не имеет параметров. Результат тот же самый, но в данном случае переменная экземпляра явно инициализирована со значением, равным 0.

```
public Cup2()
{
  seedCount = 0;
}
```

Рассмотрим третий пример конструктора, написанного программистом. У этого конструктора есть один параметр. Значение этого параметра используется для инициализации seedCount:

```
public Cup2(int initialValue)
{
  seedCount = initialValue;
}
```

Вот вторая версия класса Cup2, включающая конструкторы:

```
public class Cup2
{
  private int seedCount;
  public Cup2()
```

```
  {
    seedCount = 0;
  }
  public Cup2(int initialValue)
  {
    seedCount = initialValue;
  }
}
```

Без методов такой класс все еще имеет мало смысла, но мы уже можем создать экземпляры данного класса. В программе, использующей такой класс, оба следующих вызова создадут объекты класса:

```
Cup2 mycup = new Cup2();
Cup2 yourcup = new Cup2(4);
```

Чтобы пользоваться любым из этих двух объектов, необходимы методы. Методы делятся на две категории:

- методы доступа (Accessor). Это методы "получения" (get methods), которые возвращают значение переменной экземпляра, не изменяя ее;
- модифицирующие методы (Mutator). Это "установочные" методы (set methods), которые позволяют изменять значение переменной экземпляра.

Вот пример класса с добавлением к нему некоторых методов:

```
public class Cup3
{
  private int seedCount;
  public Cup3()
  {
    seedCount = 0;
  }
  public Cup3(int initialCount)
  {
    seedCount = initialCount;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
```

```
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
}
```

В отношении методов могут быть сделаны следующие наблюдения:

1. Для того чтобы было возможно использовать методы в программе вне определения класса, они должны быть объявлены общими (`public`).
2. После объявления `public` определяется тип. Методы могут возвращать значения, и тип метода зависит от того, что он возвращает. Тип может быть простым типом данных, ссылкой или пустым, если метод ничего не возвращает.
3. Метод может принимать или не принимать параметры.
4. Принято пользоваться какими-либо общими принципами наименования, которые позволяют быстро отличить методы доступа от модифицирующих методов.

Так, в отношении метода доступа `getSeedCount()` могут быть сделаны следующие наблюдения:

1. Он объявлен `public`.
2. Он имеет тип `int`. Чтобы быть синтаксически и логически правильным, код для такого метода должен заканчиваться возвращением константы или какой-либо переменной этого типа.
3. Он не принимает параметры.

По отношению к модифицирующему методу `setSeedCount()` могут быть сделаны следующие наблюдения:

1. Он объявлен `public`.
2. Он имеет тип `void`. Метод не возвращает значений, и код для метода не содержит выражения для возвращения.
3. Он принимает параметр, и, как и в конструкторе, параметр имеет определенный тип данных. Метод присваивает значение параметра переменной экземпляра объекта.

Не забывайте о том, что в программе можно получить доступ или изменить значение `seedCount` в объекте типа `mycup` только с помощью методов `getSeedCount()` и `setSeedCount()`. Однако в коде, находящемся внутри определения класса, вы имеете прямой доступ к переменным экземпляра данного класса. Следующая строка кода в методе `setSeedCount()` иллюстрирует это положение:

```
seedCount = newCount;
```

Теперь мы можем написать фрагмент кода программы, который иллюстрирует создание и использование объектов класса:

```
...
Cup3 mycup = new Cup3();
int mycount = mycup.getSeedCount();
myterminal.println(mycount);
mycup.setSeedCount(4);
mycount = mycup.getSeedCount();
myterminal.println(mycount);
...
```

Вы получите следующий результат из данного фрагмента:

```
0
4
```

Мы уже видели, что для классов, имеющихся в системе, можно вызвать метод `println()` для объекта такого класса в качестве параметра, и результатом вызова будет текстовая информация, говорящая о том, экземпляром какого класса является данный объект, вместе с именами и значениями переменных в квадратных скобках. А что произойдет, если мы попытаемся сделать то же самое, используя объект из класса, который вы сами создали?

```
myterminal.println(mycup);
```

Если бы класс `Cup3` был бы классом, предоставляемым системой, то вы получили бы следующее:

```
Cup3[seedCount=4]
```

Вместо этого вы получите, например:

```
Cup3@5d87b2
```

Класс должен иметь свое собственное выполнение метода `toString()` для того, чтобы вызов `println()` придал значения пе-

ременным экземпляра. Поскольку класс `Cup3` не имеет метода `toString()`, то вызов `println()` использует имеющийся в системе метод `toString()`, который и дает представленный результат. Эта версия метода показывает имя класса, далее следует символ `@`, за которым идет так называемый хэш-код объекта. В следующих главах вы узнаете, как созданный вами класс может получить доступ к имеющимся в системе версиям, и как написать собственную версию метода `toString()`.

Теперь рассмотрим фрагмент кода, где создается более одного экземпляра:

```
...
Cup3 mycup = new Cup3();
Cup3 yourcup = new Cup3();
...
mycup.setSeedCount(4);
yourcup.setSeedCount(5);
...
```

Код метода находится в классе. Каждый объект получает копию одного и того же кода. Каждый объект имеет свои переменные экземпляра. Однако названия переменных экземпляра в каждом объекте класса одинаковые. Посмотрите на строку кода в методе `setSeedCount()`:

```
seedCount = newCount;
```

Когда выполняется эта строка кода, в каком объекте изменяется `seedCount`? Чтобы ответить на этот вопрос, необходимо более подробно объяснить значение данной модели:

```
object.method(parameters);
```

При любом выполнении метода, объект, к которому вызывается метод, — это тот объект, у которого изменяются переменные экземпляра. До этого момента мы называли находящееся в скобках просто параметрами. Это те значения, которые используются при выполнении метода. Впредь мы будем называть их явными параметрами. Это вызвано тем, что объект, к которому вызывается метод, также представляет собой параметр. Это нечто, что также может быть использовано при выполнении метода. Мы будем называть такой параметр *скрытым параметром*. Именно скры-

тый параметр говорит системе, какая из переменных экземпляра объекта должна быть изменена вызовом метода, являющегося общим в определении класса. Существует ключевое слово, которое позволяет показать использование скрытого параметра. Посмотрите на строку кода из метода `setSeedCount()`, переписанную с этим ключевым словом:

```
this.seedCount = newCount;
```

Ключевое слово `this` является именем ссылки на скрытый параметр для любого случая выполнения метода. Например, если делается следующий вызов:

```
mycup.setSeedCount(4);
```

то внутри `seedCount()` ключевое слово `this` относится к `mycup`.

Методы могут быть более сложными, чем те, что мы рассматривали до сих пор. Простым примером может служить метод, позволяющий увеличить число семян на определенное количество:

```
public void increaseSeedCount(int addedNumber)
  {
    seedCount = seedCount + addedNumber;
  }
```

Класс может быть написан, сохранен и скомпилирован независимо от программы, в которой он используется. Так же, как и в случае с программными классами, имя файла, в котором сохранен класс, должно точно совпадать с именем класса (включая прописные буквы) и должно иметь расширение java. Класс, написанный пользователем, может быть использован в любой программе, в которой он может потребоваться, если он хранится в той же папке, что и программа. Посмотрите на полный код, который будет содержаться в файле Cup3.java, где класс теперь включает метод `increaseSeedCount()`.

```
public class Cup3
{
  private int seedCount;
  public Cup3()
  {
    seedCount = 0;
  }
```

```
  public Cup3(int initialCount)
  {
    seedCount = initialCount;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
  public void increaseSeedCount(int addedNumber)
  {
    seedCount = seedCount + addedNumber;
  }
}
```

Логика класса может быть проверена только программой, которая им пользуется. Полная тестовая программа должна использовать все конструкторы и методы, определенные в классе, и выдать на экран сообщение, которое позволит пользователю определить, достигнут ли желаемый эффект. Далее приводится тестовая программа для класса `Cup3`; после нее даны некоторые пояснения.

```
/* Эта программа проверяет все конструкторы
   и все методы класса Cup3 */
public class TestCup3
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Cup3 mycup;
    Cup3 yourcup;
    int someValue;
  /*  Проверяет конструктор, который не принимает
      параметров и методов set и get */
    mycup = new Cup3();
    myterminal.println("Enter an integer: ");
```

```
    someValue = myterminal.getInt();
    mycup.setSeedCount(someValue);
    someValue = mycup.getSeedCount();
    myterminal.println("mycup seedCount: " + someValue);
    /*  Проверяет конструктор, который принимает
        параметр.  */
    myterminal.println("Enter an integer: ");
    someValue = myterminal.getInt();
    yourcup = new Cup3(someValue);
    someValue = yourcup.getSeedCount();
    myterminal.println("yourcup seedCount: " + someValue);
    /*  Проверяет метод increaseSeedCount() */
    myterminal.println("Enter an integer: ");
    someValue = myterminal.getInt();
    mycup.increaseSeedCount(someValue);
    someValue = mycup.getSeedCount();
    myterminal.println("mycup seedCount: " + someValue);
  }
}
```

Следует отметить, что переменная `someValue`, использованная в вызове программы, не имеет то же имя, что и параметры `initialCount`, `newCount` и `addedNumber`, использованные в определении класса. Можно также использовать переменные в программе с такими же именами, что и параметры класса. Система различает параметры с одинаковыми именами, когда они объявлены в разных местах. Программисту может быть сложнее различать их, и следует называть переменные так, чтобы код не был сбивающим с толку.

4.2. Копирование и использование ссылок на объект

Когда объявляются типы простых переменных, в памяти отводится место, где значения такого типа могут сохраняться. Рассмотрим следующие строки кода:

```
double var1;
double var2;
```

```
var1 = 100.0;
var2 = var1;
```

В конце последовательности имеются две копии значения 100.0, сохраненные в двух разных местах в памяти, и каждое место обозначается разными именами в программе: `var1` и `var2`.

Объектные ссылки работают не совсем так. Рассмотрим следующие строки кода:

```
Cup3 mycup;
Cup3 yourcup;
mycup = new Cup3(4);
yourcup = mycup;
```

Создан только один объект, и поэтому только один объект существует. Однако существуют две правильные ссылки, и обе они относятся к этому единственному объекту.

Схема на рис. 4.1 может помочь понять, что происходит в данном случае. На рисунке прямоугольники слева представляют собой названные ссылки в программе. Прямоугольник справа представляет сам объект, как он хранится в памяти компьютера во время работы программы. Отрезки, заканчивающиеся точками, наглядно показывают, что оба имени, `mycup` и `yourcup`, относятся к одному и тому же объекту.

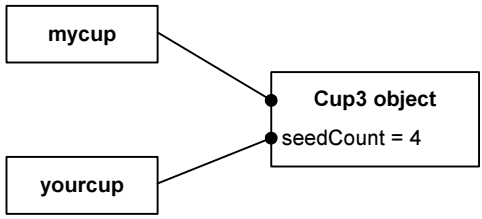


Рис. 4.1

Такие схемы будут использоваться и для других примеров в этом разделе, а также далее в книге в тех случаях, когда будет возникать тема объектов и ссылок. Эти схемы не являются частью языка UML; они используются только для объяснения материала в нашей книге.

В следующих примерах предположим, что были сделаны объявления. Теперь посмотрим на строки кода:

```
mycup = new Cup3(4);
mycup = new Cup3(5);
```

На этот раз созданы два объекта, но есть только одна ссылка. Ссылка не может относиться к двум объектам одновременно. В результате, когда создается второй объект и его ссылкой является `mycup`, первый объект утрачивает ссылку. На рис. 4.2 приведена схема такой ситуации. Пунктирная линия показывает ссылку, которая была выполнена в первой строке кода, но которая не существует после выполнения второй строки кода.

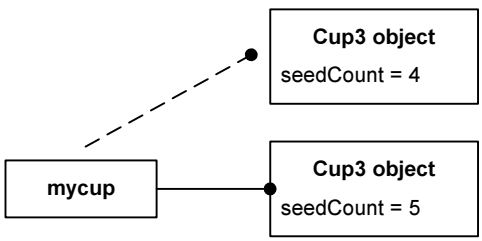


Рис. 4.2

Когда такое случается, то объект без ссылки все еще существует в системе, но его уже невозможно использовать в программе, потому что программа потеряла рукоятку от него. Это может произойти случайно, но такое можно сделать и намеренно. В языке Java система ищет объекты, которые уже более не доступны, и избавляется от них. Это называется *сборкой мусора*, и вам как программисту можно об этом не беспокоиться.

Теперь рассмотрим следующие строки кода:

```
mycup = new Cup3(4);
yourcup = new Cup3(5);
yourcup = mycup;
```

В этом случае есть два объекта и две ссылки, но когда происходит приписывание или копирование `mycup` на `yourcup`, ссылка на объект, к которому первоначально относилась `yourcup`, теряется. Теперь и `yourcup`, и `mycup` относятся к одному и тому же объекту.

На рис. 4.3 представлена схема, иллюстрирующая такую ситуацию.

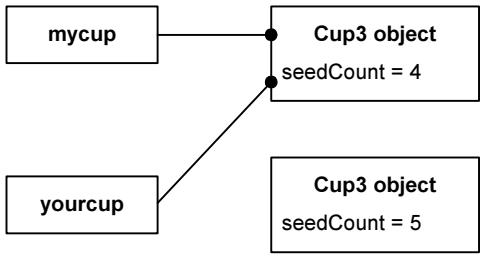


Рис. 4.3

В отличие от присвоения значений переменным, копирование ссылок на объект не приводит к появлению новой копии объекта. Оно просто создает несколько ссылок на тот же самый объект. Если же необходима копия, то существует возможность написать метод, который вернет ссылку на копию. Этот процесс называется *клонированием* и будет рассмотрен далее *(см. главу 14)*. Однако уже сейчас мы имеем возможность делать копии простых объектов, используя имеющиеся методы и конструкторы. Посмотрим на следующий фрагмент кода:

```
mycup = new Cup3(4);
yourcup = new Cup3(5);
int someValue = mycup.getSeedCount();
yourcup.setSeedCount(someValue);
```

В конце кода переменные экземпляра имеют одинаковые значения, но эти ссылки указывают на разные объекты. Оба объекта

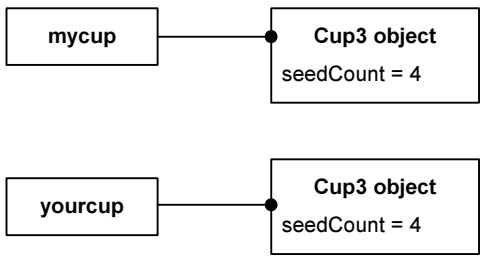


Рис. 4.4

являются точной копией друг друга. Однако они не связаны друг с другом, и дальнейшие действия могут привести к различию между их содержимым. На рис. 4.4 приведена схема, иллюстрирующая такую ситуацию.

4.3. Ссылка на *null*

Можно объявить имена объектных ссылок без назначения этим именам самих ссылок на объект. Такие имена в буквальном смысле слова ни к чему не отсылают. Также возможно присвоить ссылке специальное значение `null`. Хотя никакой объект не существует, это не то же самое, что ссылка, "ни к чему" не относящаяся. Программист может сделать такое присваивание, а в некоторых ситуациях система сама сделает это по умолчанию. Даже если вы решите не использовать ссылку на `null` в своем коде, вам следует о ней знать, т. к. система может использовать такие ссылки. Из-за неправильного использования таких ссылок могут возникать различные ошибки, и важно знать, как исправлять эти ошибки и отличать их от ошибок, возникающих вследствие ссылок на отсутствующий объект. Вот пример использования ссылки на `null`:

```
Cup3 mycup;
mycup = null;
```

Это может быть использовано, например, для того, чтобы удалить ранее созданный объект, когда вы создаете новый:

```
Cup3 mycup;
mycup = new Cup3(4);
...
mycup = null;
```

Теперь рассмотрим определение класса и примеры различных ситуаций, которые могут возникнуть внутри программ, использующих данный класс. Эти ситуации являются или результатом попытки использовать имя объекта, которое ни к чему не относится, или ссылки на `null`. Рассмотрим класс:

```
public class Shampoo
{
  private String kind;
```

```
  private int rating;
  public Shampoo(String initKind)
  {
    kind = initKind;
  }
  public Shampoo(int initRating)
  {
    rating = initRating;
  }
  public void setKind(String someKind)
  {
    kind = someKind;
  }
  public String getKind()
  {
    return kind;
  }
  public void setRating(int newRating)
  {
    rating = newRating;
  }
  public int getRating()
  {
    return rating;
  }
}
```

Рассмотрим первый пример. Он иллюстрирует, что использование в качестве параметра имени несуществующего объекта не работает.

```
public class NullTest1
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Shampoo myshampoo;
    myterminal.println(myshampoo);
  }
}
```

Компилятор определит проблему так:

```
NullTest1.java:7: variable myshampoo might not have
been initialized
myterminal.println(myshampoo);
                   ^
1 error
Tool completed with exit code 1
```

Если компилятор выводит подобную ошибку, программа не создается. Показанное далее сообщение об ошибке вы получите, если попытаетесь все равно запустить код. Оно ничего не говорит о необозначенной переменной. Оно просто сообщает, что класс в программе не существует.

```
Exception in thread "main"
java.lang.NoClassDefFoundError: NullTest1
Press any key to continue . . .
```

Вот второй пример, который иллюстрирует подобную ситуацию. Имя объекта объявлено, но никакой объект не существует, а метод вызывается для этого имени.

```
public class NullTest2
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Shampoo myshampoo;
    int someRating = myshampoo.getRating();
    myterminal.println(someRating);
  }
}
```

Второй пример генерирует точно такой же вид ошибки компилятора, как и первый пример, и работающая программа не создается.

Теперь рассмотрим третий пример. Он подобен первому примеру в том плане, что здесь рассматривается случай, когда объектная ссылка должна быть передана как параметр. Объект не существует, но в данном случае ссылке присвоено значение `null`.

```
public class NullTest3
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Shampoo myshampoo = null;
    myterminal.println(myshampoo);
  }
}
```

Программа компилируется и выполняется успешно. Она без проблем выводит на экран значение ссылки `myshampoo`. Вот результат:

```
null
```

Теперь рассмотрим четвертый пример. Он подобен второму примеру в том плане, что рассматривает случай, когда метод вызывается к объектной ссылке. Объект не существует, но в данном случае ссылке присвоено значение `null`.

```
public class NullTest4
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Shampoo myshampoo = null;
    int someRating = myshampoo.getRating();
    myterminal.println(someRating);
  }
}
```

Эта программа компилируется успешно, но не работает. Компилятор не может знать всего. Он определяет, что объектная ссылка инициализирована, и, следовательно, все хорошо. Он не определяет того, что вызов метода сделан к ссылке `null`. Невозможно осуществить вызов ссылки `null`, поэтому попытка запустить эту программу приведет к следующему сообщению об ошибке:

```
Exception in thread "main"
java.lang.NullPointerException
at NullTest4.main(NullTest4.java:7)
```

Первые четыре примера относительно просты. Они иллюстрируют проблемы, с которыми начинающий программист должен

уметь справляться. Однако существует более сложный источник ссылок на `null` в программах. Класс `Shampoo` имеет переменную экземпляра, которая является строкой. Поскольку `String` — это класс, эта переменная экземпляра является объектной ссылкой. Класс `Shampoo` имеет конструктор, который принимает единственный параметр типа `int`. В коде для конструктора нет ничего для того, чтобы явно инициализировать переменную `String`. Если и происходит инициализация, то по умолчанию. Дело в том, что система инициализирует переменную экземпляра `String` по умолчанию со значением `null`. Теперь рассмотрим пятый пример, который использует этот конструктор.

```
public class NullTest5
{
  public static void main(String[] args)
  {
    Shampoo myshampoo;
    MyTerminalIO myterminal = new MyTerminalIO();
    String someKind;
    int someRating = 5;
    myshampoo = new Shampoo(someRating);
    someKind = myshampoo.getKind();
    myterminal.println(someKind);
  }
}
```

Вот результат работы пятого примера:

```
null
```

Программа работает успешно, потому что можно иметь ссылку на `null` в качестве явного параметра. Однако эта программа показывает, что такие ссылки могут возникнуть в программе из неожиданных источников.

Шестой пример демонстрирует, как программа может закончиться неудачей в силу ссылки на `null`, полученной из иного источника.

```
public class NullTest6
{
  public static void main(String[] args)
```

```
  {
    Shampoo myshampoo;
    MyTerminalIO myterminal = new MyTerminalIO();
    String someKind;
    int someRating = 5;
    myshampoo = new Shampoo(someRating);
    someKind = myshampoo.getKind();
    myterminal.println(someKind);
    int length = someKind.length();
    myterminal.println(length);
  }
}
```

Программа успешно работает до следующей строки кода:

```
myterminal.println(someKind);
```

Вот что вы увидите на экране:

```
null
```

Затем программа встречается с проблемой в следующей строке кода:

```
int length = someKind.length();
```

Это приводит к следующей ошибке выполнения в окне командной строки:

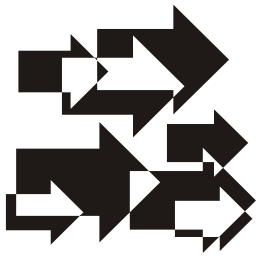
```
Exception in thread "main"
java.lang.NullPointerException
at NullTest6.main(NullTest6.java:18)
```

Появление `null` в результате работы программы происходит из первого, успешного, вызова `print()`. Ошибка исходит от вызова `someKind.length()`. Невозможно вызвать метод к ссылке на `null`. Обратите внимание, что компилятор не может предугадать подобные ошибки. Только программист может предвидеть эти проблемы и постараться избежать их или же разрешить их после обнаружения.

Ссылки на отсутствующий объект и ссылки на `null` допускаются в языке Java. Инициализация переменных экземпляра по умолча-

нию может привести к появлению ссылок на `null` в программах. Это сложный источник ошибок, о которых программист должен знать. Приведенные примеры не охватывают все возможные ситуации, но они показывают некоторые типичные случаи и ожидаемые результаты. Если вы получите сообщение об ошибке, подобное тем, что были приведены, то должны суметь разрешить проблему, проанализировав данные примеры еще раз.

ГЛАВА 5



Апплеты и графика

5.1. Web-страницы и просмотр апплетов

Апплеты являются фрагментами кода Java, которые могут быть включены в Web-страницы. Когда такая страница открывается, код апплета загружается автоматически и выполняется на локальном компьютере. Многие интересные апплеты имеют графический характер. В этой главе мы обзорно рассмотрим графические классы и методы, а также некоторые простые примеры того, как написать апплеты. Как апплеты, так и графика являются большими темами, и мы не сможем рассмотреть их во всех деталях. Более подробную информацию вы сможете найти в документации Java API. Написание графических приложений и превращение их в апплеты рассматривается в *главе 19*.

При создании апплетов имеется возможность проверить и просмотреть их без включения в Web-страницы. Если вы правильно установили на своем компьютере Java и TextPad, то в меню **Tools** должна появиться опция **Run Java Applet**. Вы можете скомпилировать апплет, используя опцию **Compile**, а затем запустить его с помощью первой опции. Когда вы запустите апплет таким образом, то откроется новое окно для программы просмотра апплета, и результаты его работы будут показаны в этом окне. Когда вы будете проделывать это в первый раз, то легко допустить ошибку

и щелкнуть мышью по опции **Run Java Application**. Если это случилось, необходимо попробовать еще раз. В зависимости от текущего состояния программы, HTML-файл мог быть создан в текущем каталоге, тогда откроется диалоговое окно с сообщением "Choose File", когда вы попытаетесь протестировать апплет. Если это окно открылось, просто нажмите кнопку **No**. Если и после этого у вас будут проблемы, то, возможно, вам следует удалить HTML-файл и затем повторить все снова.

5.2. Информация о геометрических и графических классах

В языке Java имеются различные графические классы. Некоторые из них использовались в предыдущих примерах, например, `Point` и `Rectangle`. До сих пор у нас была возможность использовать только метод `println()` для того, чтобы увидеть значения переменных экземпляра объектов этих классов. Сейчас нашей целью является реализация их графического представления на экране. Графические классы сравнительно сложны, поскольку язык Java постоянно совершенствуется. Для любой геометрической фигуры существуют как более старые, так и новые классы, которые обладают различными характеристиками.

Удобно начать изучение графических классов с класса `Point`. Если вы обратитесь к документации Java API, то найдете 4 класса, относящиеся к точкам:

```
java.awt.Point
java.awt.geom.Point2D
java.awt.geom.Point2D.Double
java.awt.geom.Point2D.Float
```

В начале раздела документации об этих классах вы увидите информацию, подобную представленной на рис. 5.1.

Эта запись показывает иерархические отношения между классами. Однако она дает и некоторую другую информацию, которая должна быть кратко объяснена здесь, чтобы отличить ее от иерархической информации. Такие части наименования, как `java.awt`, указывают на пакеты Java. Это контейнеры для хране-

ния классов, имеющихся в системе. Имена пакетов не отражают иерархические отношения между классами. Другая часть имени, такая как `Point2D.Double`, показывает, что существует класс точек, который принимает параметры типа `double`, и определение этого класса находится в классе `Point2D`. Это также не отражает иерархические отношения между классами.

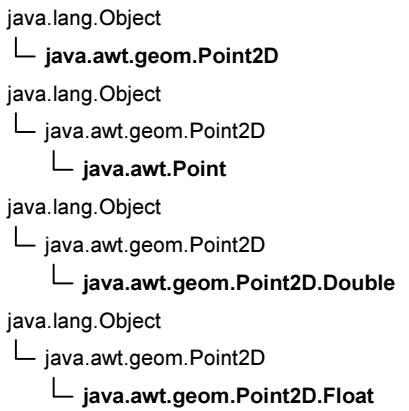


Рис. 5.1

Иерархически, класс `Point` находится ниже класса `Point2D`, класс `Point2D.Double` — ниже класса `Point2D`, и класс `Point2D.Float` также располагается ниже класса `Point2D`. Класс `Point2D` является суперклассом, а остальные три класса представляют собой его подклассы. Такой тип иерархических отношений называется в Java *иерархией наследования*. Подклассы наследуют переменные экземпляров, методы и конструкторы своих суперклассов. Объект подкласса может иметь и другое содержимое, которое было определено в самом данном классе, но он также будет иметь все переменные экземпляра и остальные характеристики своего суперкласса. Наследование — это большая тема, которая будет подробно рассмотрена в *главах 12* и *13*. Пока будет полезным объяснить это понятие с помощью схемы UML (рис. 5.2). Отношения наследования между классами показаны стрелками, направленными от подклассов к суперклассам. Это схема для классов, относящихся к точкам.

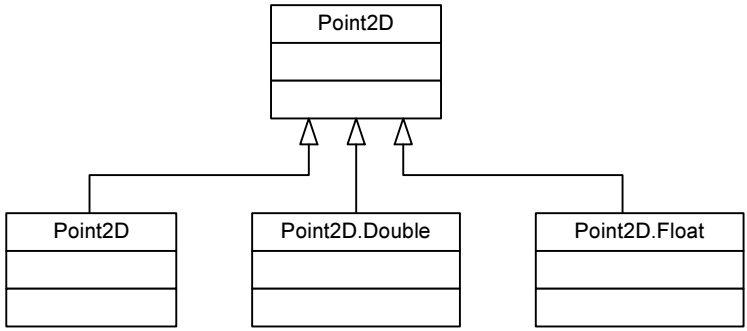


Рис. 5.2

Исторически, первым появился простой класс `Point`. Его переменные экземпляра являются целыми числами. `Point2D` является общей "шапкой" для всех остальных классов. Его определение написано таким образом, что невозможно построить экземпляр этого класса. Класс `Point2D.Double` имеет переменные экземпляра типа `double`, а класс `Point2D.Float` имеет переменные типа `float`. Важно иметь в виду, что автоматическая конверсия типа позволяет придавать целые значения переменным классов `Point2D.Double` и `Point2D.Float`. Модель, согласно которой простые геометрические классы интегрированы в более сложные наборы связанных классов, повторяется в Java. Кроме класса `Point2D.Double`, существуют классы `Rectangle2D.Double`, `Ellipse2D.Double` и `Line2D.Double` и т. д. С этого момента в примерах будут использоваться геометрические классы `Point2D.Double`.

Между геометрическими классами существует взаимосвязь еще одного свойства, что, в свою очередь, влияет на их использование. Класс с точкой в названии, например `Point2D.Double`, называется *внутренним классом* класса `Point2D`. Тема внутренних классов будет подробно затронута в следующей главе. Практическим результатом этого является то, что когда классы типа `Point2D.Double` используются в коде, достаточно импортировать внешний класс, `Point2D`. Однако в самом коде должно быть использовано полное имя внутреннего класса. Вот пример правильного синтаксиса:

```
import java.awt.geom.Point2D;
...
Point2D.Double mypoint = new Point2D.Double();
...
```

Механизм представления графики в Java также развивался со временем. В ранних версиях языка существовал класс `Graphics`. Теперь имеется класс `Graphics2D`. Для того чтобы приложения и апплеты, написанные на языке Java ранних версий, продолжали работать и в более поздних версиях, более простые классы остаются частью механизма представления графики. Новые программы, которые используют графику, должны включать в себя и более ранние способы. Все примеры в данной главе будут включать метод `paint()`, который имеет параметр типа `Graphics`. Первое, что происходит в начале кода метода, — это преобразование параметра в тип `Graphics2D`. Вот пример синтаксиса:

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D) g;
  ...
}
```

Вы, наверное, помните из курса геометрии, что точка — это абстрактное определение места, и она не имеет размеров. Это означает, что точку в буквальном смысле слова нельзя изобразить. В языке Java экземпляры точек невозможно показать на экране. В каком-либо приложении, если нужно показать точку, придется нарисовать маленькие окружности. В геометрии также считается, что линии, имеющие только одно измерение, тоже не могут быть изображены. Однако в этом случае язык Java не так строг. Линии являются самыми простыми геометрическими объектами, который можно изобразить на экране. Двухмерные фигуры также могут быть изображены.

5.3. Пример простого апплета

Рассмотрим пример простого апплета: представим изображение на экране, появляющееся в результате его работы, а затем дадим некоторые пояснения. В комментариях к коду указаны цифры,

соответствующие пунктам пояснений. Необходимо сразу отметить, что, как и приложения, апплеты представляют собой классы, но объявление таких классов включает дополнительное выражение: `extends JApplet`. Это означает, что апплет может воспользоваться преимуществами, определенными системой, для того чтобы поддерживать графические операции. Также следует отметить, что апплеты используют метод `paint()` вместо метода `main()`. Не все характеристики класса `JApplet` или метода `paint()` описываются в настоящей главе. На данный момент будет достаточным познакомиться с геометрическими классами и уметь пользоваться синтаксисом апплетов для создания ваших собственных простых апплетов.

```
import javax.swing.JApplet;                    // 1
import java.awt.Graphics;                      // 2
import java.awt.Graphics2D;                    // 3
import java.awt.geom.Point2D;                  // 4
import java.awt.geom.Line2D;                   // 5
public class LineApplet extends JApplet       // 6
{
  public void paint(Graphics g)                // 7
  {
    Graphics2D g2 = (Graphics2D) g;            // 8
    Point2D.Double startPoint =
      new Point2D.Double(10, 20);              // 9
    Point2D.Double endPoint =
      new Point2D.Double(110, 120);
    Line2D.Double myline =
      new Line2D.Double(startPoint, endPoint);
    g2.draw(myline);                           // 10
  }
}
```

На рис. 5.3 показано изображение на экране результата работы этой программы. Координатная система (x, y) в Java расположена следующим образом. Исходная точка (0, 0) расположена в левом верхнем углу. Значения по оси x увеличиваются слева направо, значения по оси y увеличиваются сверху вниз. Иными словами, начальная точка находится в левом верхнем углу, конечная точка — в правом нижнем.

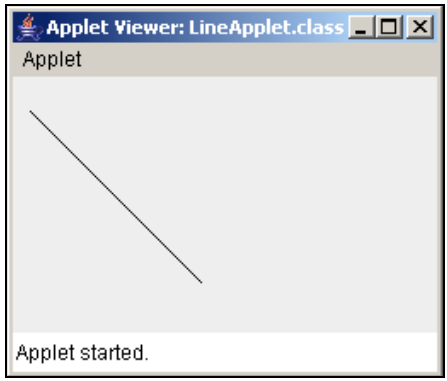


Рис. 5.3

Пояснения к коду.

- Строки 1—5. Если вы посмотрите код дальше, то увидите, что каждый из этих классов используется, поэтому каждый должен быть импортирован. Важно помнить, что использование классов `Point2D.Double` и `Line2D.Double` необходимо только для импорта внешних классов, `Point2D` и `Line2D`.
- Строка 6. Апплеты могут быть написаны как подклассы имеющегося в системе класса `JApplet`. Данная строка кода показывает правильный синтаксис для этого.
- Строка 7. Апплеты вызывают метод `paint()`, а не метод `main()`, который используют приложения. Внутри этого метода находятся строки кода, которые наносят изображение на экран.
- Строка 8. Метод `paint()` должен иметь параметр типа `Graphics`. По историческим причинам, этот параметр имеет такой вид, но он должен быть преобразован в `Graphics2D` в начале метода.
- Строка 9. В примере имеются три строки кода, где создаются начальная точка (`startPoint`), конечная точка (`endPoint`) и линия (`myline`). Точечные объекты создаются с помощью числовых параметров. Затем строится линейный объект, использующий эти точки в качестве параметров.

- Строка 10. Вызов метода `draw()` к графическому объекту `g2` с параметром `myline` вызывает появление `myline` на экране.

Хотя отдельные строки кода поддаются объяснению, может быть не понятно, как достигается конечный результат. В целом, когда программа просмотра апплетов Java получает такой класс, как этот, то он создает экземпляр класса. Так как этот класс является расширением класса `JApplet`, система знает, что он должен содержать метод `paint()`. Система вызывает этот метод для объекта и передает нужный параметр `Graphics`. Когда выполняется метод `paint()`, строки кода, содержащие вызовы к объекту `g2`, вызывают появление изображения на экране. Удобно представлять себе объект `g2` в виде шариковой ручки.

5.4. Черчение и заполнение геометрических форм

Классы прямоугольников принимают четыре параметра: координаты x и y для левого верхнего угла, а также ширина (w) и высота (h). Эллипсы также принимают эти же четыре параметра. Эллипсы определяются по прямоугольнику, в который они вписываются.

Рассмотрим простой пример использования прямоугольника в апплете. В дополнение к построению прямоугольника в заданном положении вводится также метод `fill()`, и в результате появляется второй прямоугольник. Различие между методами `draw()` и `fill()` должно быть очевидным.

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
public class RectangleApplet extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    Rectangle2D.Double myrectangle =
      new Rectangle2D.Double(10, 20, 30, 40);
```

```
    g2.draw(myrectangle);
    myrectangle =
      new Rectangle2D.Double(50, 20, 30, 40);
    g2.fill(myrectangle);
  }
}
```

Результат работы программы представлен на рис. 5.4.

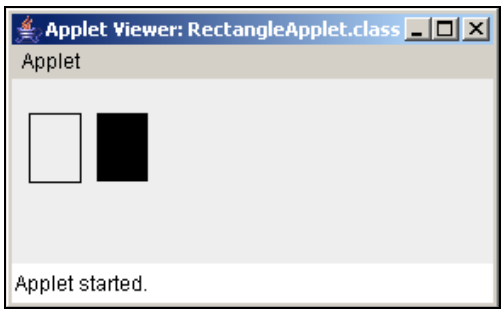


Рис. 5.4

5.5. Цвета

Java также включает механизм для использования цветов в графике. Существуют цвета, уже имеющиеся в системе. Кроме того, есть возможность создавать свои цвета. Для того чтобы использовать цвета, вам необходимо ввести класс `Color` с помощью следующей строки:

```
import Java.awt.Color;
```

Имена цветов, имеющихся в системе, являются общими константами типа `final` в классе `Color`. Примеры обращения к ним в программе приведены в табл. 5.1.

Таблица 5.1. *Цветовые константы класса* `Color`

Обращение	Название цвета
`Color.BLACK`	Черный
`Color.BLUE`	Голубой, синий

Таблица 5.1 *(окончание)*

Обращение	Название цвета
Color.CYAN	Зеленовато-голубой
Color.DARK_GRAY	Темно-серый
Color.GRAY	Серый
Color.GREEN	Зеленый
Color.LIGHT_GRAY	Светло-серый
Color.MAGENTA	Пурпурный
Color.ORANGE	Оранжевый
Color.PINK	Розовый
Color.RED	Красный
Color.WHITE	Белый
Color.YELLOW	Желтый

Цвет можно определить с помощью трех базовых компонентов: красного, зеленого или синего. Каждый компонент может принимать значения от `0.0` и `1.0`. Создание собственного цвета производится следующим образом:

```
Color mycolor = new Color(0.75F, 0.0F, 0.25F);
```

По историческим причинам, эти параметры являются скорее `float`, чем `double`. Символ `F`, следующий за каждым из параметров, преобразует тип константы, которая в противном случае считалась бы `double`, во `float`. Иначе могут возникнуть ошибки в компилировании.

Для использования цветов вызывают графический объект `g2`. Этот объект обладает методами, которые позволяют менять его характеристики. Как уже отмечалось, можно сравнить его с ручкой. Выбор цвета для графического объекта — это все равно, что выбор цвета чернил для ручки. Далее даны примеры того, как придать графическому объекту определенный цвет:

```
g2.setColor(Color.RED);
g2.setColor(mycolor);
```

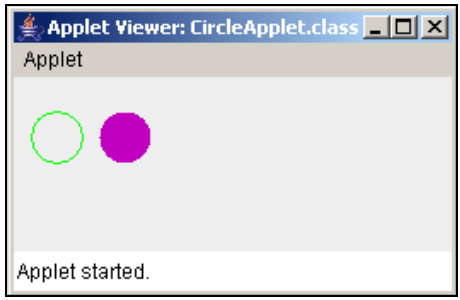


Рис. 5.5

Вот еще небольшой пример апплета с графикой, который иллюстрирует применение цветов, а также использование класса `Ellipse2D.Double`. В этом примере создаются окружности за счет того, что параметрам ширины и высоты присвоены одинаковые значения. Заметьте, что не существует удобного способа определения концентрических окружностей. Нет конструктора эллипсов, в котором в качестве точки отсчета передавался бы "центр". Точкой отсчета всегда является левый верхний угол описывающего прямоугольника.

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
public class CircleApplet extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    Ellipse2D.Double mycircle =
      new Ellipse2D.Double(10, 20, 30, 30);
    g2.setColor(Color.GREEN);
    g2.draw(mycircle);
    mycircle =
      new Ellipse2D.Double(50, 20, 30, 30);
    Color mycolor = new Color(0.75F, 0.0F, 0.75F);
    g2.setColor(mycolor);
```

```
    g2.fill(mycircle);
  }
}
```

Результат работы программы представлен на рис. 5.5.

5.6. Текст

Можно также добавить текст в графический режим. Изображение текста имеет долгую историю, у него своя терминология, и практически все его детали были разработаны задолго до появления компьютеров. Этот вопрос усложняется еще и тем, что имеются разные шрифты в различных видах компьютерных систем, и сложно пытаться обращаться с ними на одном компьютерном языке. Это тот аспект Java, где принцип "Скомпилировано один раз, работает везде" означает ограничение возможностей.

В отличие от геометрических форм, которые ориентируются согласно их левой верхней точке на плоскости *xy*, текст ориентируется соответственно своей "основной точке". Это левая конечная точка линии, на которой начерчен текст. Основная точка не зависит от нижней точки той части букв, которая опускается ниже линии, а также не зависит от того, в каком месте первая буква касается строки.

Для того чтобы выводить текст в простейших формах в Java, необходимо использовать другой метод в графическом объекте `g2`. Нужно указать текст в виде строки, а также установить координаты *x* и *y* для основной точки. Например:

```
g2.drawString("My text", 20, 30);
```

Если текст создан таким образом, то вы получите шрифт по умолчанию. Можно изменить шрифт. Для того чтобы сделать это, должен быть введен класс `Font`:

```
import Java.awt.Font;
```

В системе Java имеются следующие шрифты:

```
Serif
SansSerif
Monospaced
```

```
Dialog
DialogInput
```

Эти шрифты могут быть установлены с различными размерами, которые выражаются в пунктах. Также в системе Java имеется возможность изменения начертания шрифта на жирный и курсив. Вот пример строк кода, иллюстрирующий создание нового шрифта, назначение его графическому объекту, а затем вывод текста на экран:

```
Font myfont = new Font("Serif", Font.BOLD, 16);
g2.setFont(myfont);
g2.drawString("My text", 20, 30);
```

Существуют три параметра в конструкторе `Font`: название шрифта, имеющегося в Java, в виде `String`; характеристика шрифта, которая определяется как постоянная в классе `Font`; размер в пунктах, который является значением целого числа. Новый шрифт используется в качестве параметра в методе `setFont()` объекта `Graphics2D`. Наконец, строка напечатана. Обратите внимание, что графический объект `g2` может содержать характеристики цвета, характеристики шрифта и другие характеристики одновременно. Также не забывайте, что параметры могут быть заданы как переменные или как константы.

При создании графических изображений могут возникнуть сложности в определении того, как изображения появятся относительно друг друга в окне апплета. К сожалению, взаимосвязь между значениями осей координат для графических форм и для вывода текста не всегда ясна. Вместо того чтобы вдаваться в глубокие технические дискуссии об этом, рекомендуем воспользоваться методом проб и ошибок и исправлять положение текста на глаз. Далее приведен последний пример, где этот метод был использован для того, чтобы расположить объекты приятным глазу образом.

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.Color;
import java.awt.Font;
```

```
public class TextApplet extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    g2.setColor(Color.BLUE);
    Font myfont =
      new Font("Serif", Font.ITALIC, 24);
    g2.setFont(myfont);
    String blueString = "My Blue String";
    Rectangle2D.Double blueRectangle =
      new Rectangle2D.Double(30, 30, 170, 45);
    g2.draw(blueRectangle);
    g2.drawString(blueString, 40, 60);
  }
}
```

Вот что появится на экране в результате действия апплета (рис. 5.6).

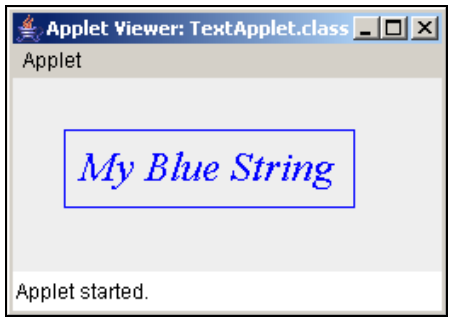


Рис. 5.6

5.7. Размещение апплетов на Web-страницах

Для того чтобы разместить апплет на Web-странице, нужно:

- скомпилировать апплет и сохранить файл с полученным классом;

- включить необходимые метки и другую информацию в исходный код HTML для страницы, на которой будет находиться апплет;
- загрузить файл HTML и файл класса в каталог на сервере, который предоставляет доступ к Web-страницам.

Вот минимальный код HTML, необходимый для создания Web-страницы, содержащей апплет:

```
<html>
  <applet
    code="MyText.class" width="500" height="500">
  </applet>
</html>
```

Теги `<html>` указывают на начало и конец страницы. Теги `<applet>` задают начало и конец вложенного апплета. Внутри тега апплета необходимо указать источник кода апплета, а также размер окна, в котором он появится. Имеется возможность установить размер апплета внутри кода вместо того, чтобы полагаться на размеры программы просмотра апплетов, установленные по умолчанию. При включении в Web-страницу размеры должны быть установлены в файле HTML. Все строки кода в апплете, которые устанавливают размеры, должны быть убраны, потому что они вступят в конфликт с тегом `<html>`. Когда вы будете писать собственные примеры, то можете скопировать вызов кода, ширину и высоту области вывода апплета из показанного образца. Обратите внимание, что указывается файл класса апплета, а не исходный файл с расширением java. В нашем примере файл класса должен быть в том же каталоге, что и файл HTML, поскольку не назван путь при указании имени файла класса. Помните о том, что если вы уберете строку, устанавливающую размеры, то апплет придется перекомпилировать и использовать новый файл класса.

Можно вставлять апплеты в сложные страницы. Также удобно создавать простые страницы, подобно той, что указана выше, и соединять их ссылками с другими страницами. За счет этого главная страница не становится загроможденной, и в то же время апплеты являются легкодоступными с помощью браузера.

5.8. Web-документация Java

Тема документации языка Java не связана напрямую с темой апплетов. Однако здесь есть связь в том смысле, что документация может быть включена в Web-страницы. Рекомендуется давать комментарии к написанному коду. В нашей книге в примерах нет развернутых комментариев, но это происходит лишь потому, что программы объясняются в тексте. Обычной практикой является предоставление комментариев к классам в целом и к конструкторам и методам внутри классов. Эти комментарии являются чем-то вроде внутренней документации. Язык Java также обладает свойством, благодаря которому комментарии, написанные в определенном формате, — это все, что требуется для создания Web-документации. Далее приведен наш класс `Cup` с комментариями именно в таком формате. Некоторые важные моменты относительно стиля документации поясняются после кода

```
/**
  Эта версия класса Cup содержит два конструктора
  и три метода. Цель данного примера — показать формат
  комментариев, которые создают Web-документацию.
*/
public class Cup3Commented
{
  private int seedCount;
  /**
  Это конструктор по умолчанию. У него нет параметров.
  Он явно устанавливает значение переменной seedCount на 0,
  инициализируя значение по умолчанию.
  Поскольку в этом классе имеется другой конструктор,
  написанный программистом, то система не предоставляет
  конструктор по умолчанию. Как и любой другой конструктор,
  данный конструктор возвращает ссылку на созданный объект.
  */
  public Cup3Commented()
  {
    seedCount = 0;
  }
```

```
  /**
  Этот конструктор инициализирует seedCount на
  значение параметра.
  @param initialCount, int - изначальное значение
  переменной экземпляра seedCount.
  */
  public Cup3Commented(int initialCount)
  {
    seedCount = initialCount;
  }
  /**
  Этот метод возвращает значение seedCount для
  чашки. Он не принимает никаких параметров.
  @return seedCount, int - значение переменной
  seedCount.
  */
  public int getSeedCount()
  {
    return seedCount;
  }
  /**
  Этот метод устанавливает seedCount для чашки. Он
  не возвращает значения.
  @param newCount, int - новое значение для
  переменной seedCount.
   */
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
  /**
  Этот метод добавляет значение параметра к текущему
  значению seedCount для чашки. Он не возвращает значения.
  @param addedNumber, int - значение, на которое
  должна быть увеличена переменная variable seedCount.
  */
  public void increaseSeedCount(int addedNumber)
  {
    seedCount = seedCount + addedNumber;
  }
}
```

Далее представлено краткое описание формата, использованного в приведенном примере, и шаги, необходимые для создания Web-документации из файла Java, который был прокомментирован подобным образом.

1. Комментарии должны идти непосредственно перед комментируемым фрагментом. Это значит, например, что импорт для класса будет идти перед комментарием для класса. Внутри класса комментарии для конструкторов и методов должны указываться непосредственно перед конструкторами и методами.
2. Комментарии, которые должны образовать Web-страницу документации, должны начинаться с `/**` и заканчиваться `*/`. Можно включить другие комментарии в код, которые не будут создавать Web-страницу с документацией. Такие комментарии должны отделяться обычными символами: `/*...*/`.
3. Для конструкторов и методов принято предоставлять общую информацию и специфическую документацию по параметрам и возвращаемым значениям, если такие есть. Комментарии, специфичные для параметров и возвращаемых значений, располагают на отдельных строках после общего комментария, и эти строки должны начинаться с `@`.
4. Документация образуется действием команды в командной строке `javadoc` против исходного файла Java, содержащего эти комментарии. Эта утилита также может запускаться для более чем одного файла в каталоге с помощью символа обобщения (`*`). Это приведет к тому, что документация будет формироваться для всех исходных файлов Java, которые будут в одном приложении.
5. Иногда оказывается, что наибольшее препятствие к созданию документации — это проблемы с выражением `PATH` на вашем компьютере, которое препятствует нахождению программы `javadoc`, когда вы пытаетесь использовать ее.

Вот что следует иметь в виду и с чем вы можете столкнуться, пытаясь создать документацию:

1. Вам нужно получить доступ к командной строке. Если вы работаете с Windows, то выберите **Start | Programs | Accesso-**

ries | Command Prompt (Пуск | Программы | Стандартные | Командная строка).

2. После этого вам нужно узнать путь к каталогу, где сохранена ваша программа. Как правило, командная подсказка придет по умолчанию при подсказке, которая начинается `C:\...>`. В зависимости от того, в сети ваш компьютер или нет и какие виды дисководов установлены на компьютере, подсказка может начинаться с другой буквы.

   В частности, ваша подсказка может выглядеть так: `C:\WINDOWS>` или `C:\WINNT>`. Если это так, то вам нужно ввести команду `cd..`, что выведет вас на один уровень выше в каталог, чтобы получить старую добрую подсказку `C:\>`.

   С другой стороны, если подсказка, например, такая: `H:\>`, или начинается с любой другой буквы, определяющей другой дисковод, первое, что вам следует сделать, — это напечатать `C:` и нажать клавишу <Enter>. Это должно привести вас к подсказке `C:\>`.

3. Теперь предположим, что ваша программа находится в папке на один уровень ниже на диске C:. Например, она может быть в папке myprogs. Другими словами, полный путь к этой программе выглядит так: C:\myprogs\Someprog.java.

   Вы должны суметь оказаться в этом каталоге, используя командную строку. Вы можете выполнить это следующим образом: при подсказке `C:\>` введите `cd myprogs`. Когда вы нажмете клавишу <Enter>, подсказка должна выглядеть так: `C:\MYPROGS>`.

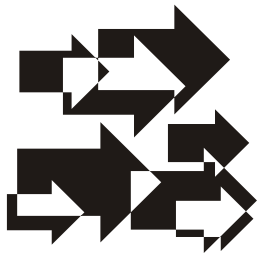
4. Теперь, если выражение `PATH` на вашем компьютере было установлено правильно, когда на нем была инсталлирована система Java, то вы должны суметь запустить следующую команду. Вот как она выглядит: `C:\MYPROGS>javadoc Someprog.java`.

5. Если это произошло, вы найдете в том же каталоге набор HTML-файлов, содержащих документацию Java для программы. Вы можете посмотреть и проверить их, открывая их браузером.

6. Если путь хранения файлов на вашем компьютере неправильный, то утилита `javadoc` не будет найдена и запущена. Вот два возможных подхода к решению этой проблемы.

   - Если вы хорошо разбираетесь в Windows или если у вас есть желание читать технические объяснения, вы можете начать изучение FAQ-страниц на сайтах TextPad и Sunweb, выясняя, как можно изменить путь на вашем компьютере, чтобы он нашел `javadoc`.
   - Второе возможное решение — быстрое и "грязное". Оно далеко от идеального, но это можно сделать один раз, чтобы выполнить задание. В компьютере, где инсталлирована система Java, должна быть папка на диске C: с названием типа jdk1.5.0_05 (в зависимости от того, какая версия была установлена). Внутри этой папки должна быть папка bin с утилитой `javadoc`. Сделайте копию вашего исходного файла кода Java в этом каталоге. Затем в командной строке измените каталог на bin и введите команду `javadoc Someprog.java`. Вы преодолели проблему пути, потому что система найдет `javadoc` в текущем каталоге наряду с вашим программным файлом. Будьте осторожны, чтобы не стереть или переместить любые системные файлы из каталога bin, иначе вам придется переустанавливать Java. При этом обязательно переместите вашу программу и все HTML-файлы, являющиеся результатом успешного выполнения утилиты `javadoc`, из этой папки. Именно это вы и искали.

ГЛАВА 6



Оператор *if* — условное выполнение

6.1. Синтаксис оператора *if* и условия вычисления

В этом разделе дано несколько примеров с оператором `if` — оператором условного перехода.

Итак, рассмотрим первый пример. Если некоторое условие верно, выполняется единственная строка кода, следующая сразу за условием. Если условие неверно, то следующая строка кода не выполняется. В любом случае, код, указанный за оператором `if`, выполняется.

```
if(condition)
  statement;
```

Обратите внимание, что ключевое слово `if` и скобки, содержащие условие, не образуют независимой строки кода. За ними не следует употреблять точку с запятой. Если за условием сразу стоит точка с запятой, это вызывает логическую ошибку, которую трудно обнаружить. Система проверяет истинность условия. При этом не имеет значения, каким будет результат, верным или неверным. Точка с запятой сигнализирует об окончании оператора `if`. Следующая строка кода не зависит от него и просто выполняется.

Второй пример показывает оператор `if` с двумя взаимно исключающими альтернативами. В процессе работы кода выполняется

либо одно, либо другое из двух утверждений, но оба не могут быть выполнены. Как и строка кода, содержащая `if`, `else`, не образует независимую строку кода, и за ней не нужна точка с запятой. Если же стоит точка с запятой, то любая строка кода, следующая далее, будет выполняться без всяких условий.

```
if(condition)
  statement1;
else
  statement2;
```

В обоих этих примерах одиночные выражения могут быть заменены блоком кода, заключенного в фигурные скобки:

```
if(condition)
{
  statement1;
  statement2;
  ...
}
else
{
  ...
}
```

Условия внутри `if`-выражений часто представляют собой сравнения числовых значений. Операторы сравнения в Java следующие:

- `<` — меньше чем;
- `<=` — меньше чем или равно;
- `>` — больше чем;
- `>=` — больше чем или равно;
- `!=` — не равно;
- `==` — равно.

Обратите внимание на разницу между оператором `==` и оператором присваивания `=`.

Обработка чисел с плавающей точкой может дать результат, который будет приблизительным. Важно не забывать об этом, выполняя числовые сравнения, особенно смешивая целочисленные и вещественные переменные. Если переменная содержит неточ-

ное двоичное представление десятичного значения, то проверка на равенство в отношении некоторого точного значения может дать неверный результат. По возможности, сравнения должны быть построены как проверки на неравенство, а не на равенство.

Можно записать любое число фрагментов кода, иллюстрируя приведенные выше модели. Полезно будет показать использование оператора в коде некоторого класса. Далее приводится код для класса. Он содержит метод для увеличения количества семян в чашке.

```
public class Cup3
{
  private int seedCount;
  public Cup3()
  {
    seedCount = 0;
  }
  public Cup3(int initialCount)
  {
    seedCount = initialCount;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
  public void increaseSeedCount(int addedNumber)
  {
    seedCount = seedCount + addedNumber;
  }
}
```

Обратите внимание на то, что хотя имеются такие имена, как `increase` (увеличить) и `added` (добавленный), существует возможность получить отрицательный параметр для метода `increaseSeedCount`. Можно использовать оператор `if` в коде мето-

да для того, чтобы исключить возможность действий с отрицательным значением. Простой способ сделать это выглядит так:

```
public void increaseSeedCount(int addedNumber)
{
  if(addedNumber > 0)
  {
    seedCount = seedCount + addedNumber;
  }
}
```

Это неполное решение. Удобно, когда оператор `if` не дает возможности получить нежелательный результат, но неудобно, что программист может вызвать метод и не получить предупреждения о том, что код с добавлением значения не был выполнен. Пути решения этой проблемы будут рассмотрены далее.

6.2. Множественные альтернативы и вложенный оператор *if*

Рассмотрим пример оператора `if` с множественными взаимно исключающими альтернативами:

```
if(condition1)
{
  ...
}
else if(condition2)
{
  ...
}
...
else
{
}
```

Может быть использовано любое количество выражений `else if`, но эта структура должна закончиться `else`. Последнее ключевое слово `else` представляет собой ситуацию по умолчанию, если ни один из операторов `if` не оказывается верным. Синтаксически

неверно завершить структуру на `else if`. Иногда нет отдельного выражения или блока выражений, который нужно выполнить в случае по умолчанию. Если так, тогда за `else` может сразу же следовать точка с запятой или фигурные скобки, которые не содержат ничего, но само слово `else` не может быть удалено.

Эта структура может быть использована для взаимно исключающего набора дискретных случаев, как это показано далее:

```
if(x == 1)
{
  // сделать что-либо
}
else if(x == 2)
{
  // сделать что-либо другое
}
else
{
  // сделать что-либо третье
}
```

Обратите внимание, что это также применимо для выполнения последовательностей числовых значений, возрастающих или убывающих:

```
if(x <= 1.0)
{
  // сделать что-либо
}
else if(x <= 2.0)
{
  // выполняется для значений, больших 1.0
  // и меньших либо равных 2.0
}
else if(x <= 3.0)
{
  // выполняется для значений, больших 2.0
  // и меньших либо равных 3.0
}
```

```
else
{
  // выполняется для всех значений, больших 3.0
}
```

В общем, нет ограничений на то, где `if`-операторы могут появляться в программном коде. Они могут возникнуть внутри пары скобок, разграничивая любой работающий блок. В частности, оператор `if`, а также `else` или `else if` могут появиться внутри блока кода, который принадлежит другому оператору `if`, `else` или `else if`. Такая структура называется *вложенным оператором* `if`. Вложенные выражения заключаются в фигурные скобки внутри других фигурных скобок. Для легкости прочтения рекомендуется, чтобы каждая пара фигурных скобок отделялась отступом от пары скобок, которая их содержит. Пример вложения только на один уровень будет выглядеть так:

```
if(condition1)
{
  if(condition2)
  {
    statement1;
  }
  else
  {
    statement2;
  }
}
else
{
  if(condition3)
  {
    statement3;
  }
  else
  {
    statement4;
  }
}
```

Выражение *statement1* выполняется, если условие *condition1* верно, и условие *condition2* тоже верно. Выражение *statement2* выполняется, если условие *condition1* верно, а условие *condition2* — нет. Выражение *statement3* выполняется, если условие *condition1* неверно, а условие *condition3* — верно. Выражение *statement4* выполняется, если условие *condition1* неверно, и условие *condition3* тоже неверно.

6.3. Логические операторы, сложные выражения и булевы переменные

В Java существуют логические операторы and (И), or (ИЛИ) и not (НЕ), обозначаемые `&&`, `||` и `!` соответственно. Также можно придумать логические выражения, которые имеют значение exclusive or (исключающее ИЛИ), которое означает одно или другое верно, но не оба. Скобки могут быть использованы в логических выражениях, чтобы прояснить группирование различных условий. Как только становятся доступными логические операторы, появляется возможность конструировать более сложные условия.

Напомним комбинации условий для вложенного оператора `if` в последнем примере предыдущего раздела.

Выражение *statement1* выполняется, если условие *condition1* верно, и условие *condition2* тоже верно. Выражение *statement2* выполняется, если условие *condition1* верно, а условие *condition2* — нет. Выражение *statement3* выполняется, если условие *condition1* неверно, а условие *condition3* — верно. Выражение *statement4* выполняется, если условие *condition1* неверно, и условие *statement3* тоже неверно.

Следующий код, использующий логические операторы, эквивалентен коду со вложенным оператором `if`:

```
if(condition1 && condition2)
{
  statement1;
}
else if(condition1 && !condition2)
```

```
{
  statement2;
}
else if(!condition1 && condition3)
{
  statement3;
}
else if(!condition1 && !condition3)
{
  statement4;
}
else
{
}
```

Программист может выбирать, использовать ли вложенный оператор `if` или логические операторы. В зависимости от наличия переменных и их значений, можно написать выражения, включающие более чем одну числовую переменную, следующего вида:

```
if(x < 5 && y > 12)
  ...
if(x < 5 || y > 12)
  ...
```

Вы также можете получить выражения сравнения с одной числовой переменной, которая привносит значения числовых строк из алгебры. Рассмотрим четыре примера.

Пример 1.

```
if(x < 5 && x > 12)
```

Всегда неверно (рис. 6.1).

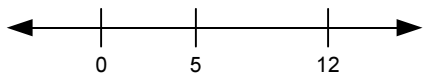


Рис. 6.1

Пример 2.

```
if(x < 5 || x > 12)
```

Верно для значений меньше 5 или больше 12 (рис. 6.2).

Рис. 6.2

Пример 3.

```
if(x > 5 && x < 12)
```

Верно для значений от 5 до 12 (рис. 6.3).

Рис. 6.3

Пример 4.

```
if(x > 5 || x < 12)
```

Всегда верно (рис. 6.4).

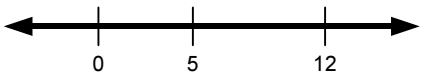


Рис. 6.4

С тем, чтобы получить лучшее представление о видах конструкций с логическими операторами, полезно напомнить таблицы истинности и законы де Моргана. Пусть `p` и `q` обозначают любые допустимые условия или утверждения, которые имеют некоторое логическое значение: либо верное, либо неверное. Пусть `T` (от англ. *true* — истина) и `F` (от англ. *false* — ложь) представляют "верно" или "неверно" соответственно. Две левые колонки табл. 6.1 и 6.2 показывают все возможные комбинации логических значений для `p` и `q`. Читая по горизонтали любой ряд, вы увидите логическое значение для более сложных выражений, включающих `p` и `q`.

Посмотрите на табл. 6.2. Для третей и четвертой колонок, также как и для пятой и шестой колонок, логические значения всегда

одинаковы. Это является иллюстрацией законов де Моргана, которые могут быть сформулированы следующим образом.

Таблица 6.1. *Таблица истинности логических операторов (часть 1)*

p	q	!p	!q	p && q	p || q
T	T	F	F	T	T
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	F	F

Таблица 6.2. *Таблица истинности логических операторов (часть 2)*

p	q	!(p && q)	!p || !q	!(p || q)	!p && !q
T	T	F	F	F	F
T	F	T	T	F	F
F	T	T	T	F	F
F	F	T	T	T	T

Логическое значение выражения `!(p && q)` не отличается от логического значения выражения `!p || !q`.

Логическое значение выражения `!(p || q)` не отличается от логического значения выражения `!p && !q`.

При использовании оператора `if` выполнение определенных блоков кода зависит от таких условий, как числовые сравнения на равенство или неравенство. Именно это представлено пропозициями `p` и `q` в предыдущем примере. Система не только оценивает условия, чтобы получить логические значения — Java также имеет простой тип, называемый *булевым*, который может содержать истинностные значения. Ключевые слова `true` и `false` — это два значения, которые может приобретать булева переменная.

В следующем примере `p` и `q` объявлены как булевы переменные. При первом присваивании `p` принимает значение `true`, т. к. 5 меньше 7. Во втором случае `q` принимает значение `false`, т. к. 7 не меньше 5.

```
boolean p, q;
boolean p = (5 < 7);
boolean q = (7 < 5);
```

В следующем примере булева переменная `someResult` принимает логические значения, которые присваиваются ей напрямую:

```
boolean someResult;
someResult = true;
someResult = false;
```

Следующий пример возвращает нас к случаю, который обсуждался ранее, когда выполнение кода внутри метода зависит от определенных условий. Полезно сообщать вызывающей программе, какой из возможных вариантов действий предпринимается методом. Сделать это можно с помощью задания методу типа и возвращаемого значения, которое указывает, какое действие было выполнено. Для метода, где существуют фактически две альтернативы — или некоторое действие, или отсутствие действия — разумным выбором будет булева переменная. Посмотрите на переписанный пример, где возвращаемое значение является `true` или `false`, в зависимости от того, увеличилось ли `seedCount` или нет:

```
public boolean increaseSeedCount(int addedNumber)
{
  if(addedNumber > 0)
  {
    seedCount = seedCount + addedNumber;
    return true;
  }
  else
    return false;
}
```

Вот класс чашек, переименованный в `Cup4`, включающий этот новый метод:

```
public class Cup4
{
  private int seedCount;
  public Cup4()
```

```
  {
    seedCount = 0;
  }
  public Cup4(int initialCount)
  {
    seedCount = initialCount;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
  public boolean increaseSeedCount(int addedNumber)
  {
    if(addedNumber > 0)
    {
      seedCount = seedCount + addedNumber;
      return true;
    }
    else
      return false;
  }
}
```

В коде, который использует данный метод, вызов примет вид, показанный в следующем фрагменте:

```
boolean callResult;
int moreSeeds = 7;
Cup4 mycup = new Cup4(5);
callResult = mycup.increaseSeedCount(moreSeeds);
```

После вызова можно выполнить некоторые действия в программе, в зависимости от того, является ли возвращаемое значение `true` или `false`. По структуре эти действия будут представлять собой новые операторы `if`. Обратите внимание, что когда вы используете булевы значения в условных выражениях с `if`, необязательно применять какой-либо символ для проверки равенства.

Достаточно использовать переменную, содержащую логическое значение. Так, для вызванного метода в предыдущем примере следующее выражение будет работающим условием:

```
if(callResult)
  ...
```

Также можно написать код так, как показано далее. Вызов заключен в скобки. Когда он производится, то возвращается логическое значение. Действие оператора зависит от этого возвращаемого значения. Такой вид кода компактен, но не всегда легок для понимания:

```
if(mycup.increaseSeedCount(moreSeeds))
  ...
```

В вызывающем коде не всегда нужно регистрировать возвращаемое значение. Оно может быть проигнорировано, и это не является синтаксической ошибкой. Например, следующий оператор будет работающей строкой кода:

```
mycup.increaseSeedCount(moreSeeds);
```

Обратите внимание на то, что код метода имеет два возвращаемых выражения. Они находятся в двух взаимно исключающих альтернативах оператора `if`. Как только встречается возвращаемое выражение, выполнение метода заканчивается. Можно консолидировать такую ситуацию через возвращаемое значение переменной и выполнить возврат переменной в конце метода. Метод, измененный подобным образом, будет выглядеть так:

```
public boolean increaseSeedCount(int addedNumber)
{
  boolean returnValue;
  if(addedNumber > 0)
  {
    seedCount = seedCount + addedNumber;
    returnValue = true;
  }
  else
  {
    returnValue = false;
  }
  return returnValue;
}
```

6.4. Сравнение объектов

Общий вопрос сравнения объектов, или ссылок на объекты, может быть рассмотрен на примере строк. Вспомните, что `String` — это класс, и сами строки являются экземплярами этого класса. Класс `String` обеспечивает выполнение метода, называемого `equals()`. Это метод, который проверяет, являются ли строки, содержащиеся в двух разных строковых объектах, одинаковыми. Пусть есть две строки, объявленные и инициализированные следующим образом (рис. 6.5):

```
String mystring = "Tra-la-la";
String yourstring = "La-de-dah";
```

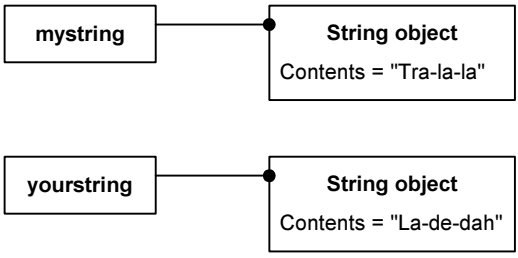


Рис. 6.5

Существуют два отдельных объекта, имеющих разную последовательность символов. Посмотрите на оператор `if`, содержащий вызов метода `equals()`. В этой ситуации условие примет значение `false`. Содержание двух объектов не одинаково:

```
if(mystring.equals(yourstring))
```

Теперь рассмотрим ситуацию, когда обе строки содержат одинаковое значение:

```
String mystring = "Hello, World!";
String yourstring = "Hello, World!";
```

Это можно схематично изобразить так, как показано на рис. 6.6.

Имеются два отдельных объекта, содержащие одинаковый набор символов. Оператор `if` содержит вызов метода. В такой ситуации условие примет значение `true`:

```
if(mystring.equals(yourstring))
```

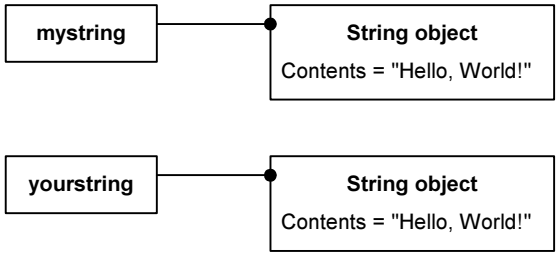


Рис. 6.6

Наконец, можно использовать оператор == для объектных ссылок. Он позволяет проверить равенство ссылок, а не равенство содержимого. Иными словами, проверяется одинаковость самих объектов, а не того, что в них находится. Посмотрим на предыдущий пример, в котором оператор `if` содержит вызов метода `equals()` на объекты. В этой ситуации условие примет значение `false`, т. к. хотя содержимое и одинаково, но ссылки относятся к двум разным объектам:

```
if(mystring == yourstring)
```

Теперь, пусть две строки будут объявлены и инициализированы, как это показано далее. Обратите внимание, что строка `yourstring` не создается отдельно, она просто объявляется. Ей присваивается значение `mystring`, которое создается:

```
String mystring = "Hello, World!";
String yourstring = mystring;
```

Такую ситуацию можно изобразить так, как показано на рис. 6.7.

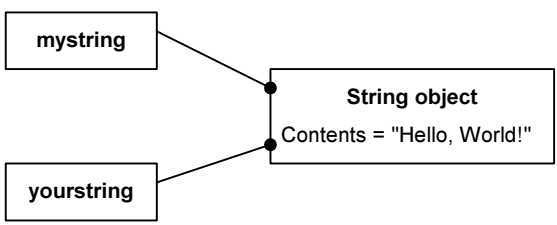


Рис. 6.7

В данном случае существует только один объект, но с двумя ссылками. Вот оператор `if`, содержащий проверку равенства ссылок. В этой ситуации условие принимает значение `true`, т. к. две ссылки относятся к одному и тому же объекту:

```
if(mystring == yourstring)
```

Существуют другие методы в классе `String`, которые поддерживают различные виды сравнений. Среди них есть `compareTo()` и `equalsIgnoreCase()`. Первый из них позволяет сравнивать содержимое строковых объектов, чтобы увидеть, который из них является первым по алфавиту. Второй метод позволяет проверить равенство содержимого строковых объектов там, где нет различий между строчными и прописными буквами. Это может быть полезно в ситуации, когда ввод совершается пользователями. Полную информацию о том, как работают эти методы, можно найти в документации Java API.

Такие же наблюдения, которые были сделаны для сравнения на равенство строковых объектов, могут быть сделаны и для сравнения на равенство объектов в целом. Предположим, имеются следующие два объявления и инициализации:

```
Point2D.Double mypoint =
  new Point2D.Double(10, 20);
Point2D.Double yourpoint =
  new Point2D.Double(30, 40);
```

Содержания объектов разные, да и сами объекты тоже разные. И метод `equals()`, и оператор `==` вернут значение `false`. Теперь предположим, что инициализации имели следующий вид:

```
Point2D.Double mypoint =
  new Point2D.Double(10, 20);
Point2D.Double yourpoint =
  new Point2D.Double(10, 20);
```

А теперь метод `equals()` вернет значение `true`, а оператор `==` будет давать значение `false`. Наконец, рассмотрим такую ситуацию:

```
Point2D.Double mypoint =
  new Point2D.Double(10, 20);
Point2D.Double yourpoint = mypoint;
```

А в этом случае и `equals()`, и оператор `==` вернут значение `true`.

Описание метода `equals()` и оператора == применяется к классам, имеющимся в системе. Однако будьте осторожны: `equals()` не будет работать таким способом с классами, которые вы написали. Например, предположим, что у нас есть следующие объекты, явно созданные без использования системных классов:

```
Shampoo myshampoo = new Shampoo();
Shampoo yourshampoo = new Shampoo();
```

Независимо от того, каково содержимое этих объектов, метод `equals()` будет работать точно так же, как и оператор ==. Другими словами, для вашего класса можно вызвать метод `equals()`, но он проверяет только равенство ссылки, но не равенство содержимого. Ваш класс наследует метод `equals()` у класса, который находится выше в иерархии. Однако унаследованный метод не работает так, как вам необходимо. О том, как написать метод `equals()`, будет рассмотрено в *главе 14*.

6.5. Последний пример, символьный тип и некоторые типичные ошибки

Некоторые авиакомпании имеют специальные бонусы для часто летающих пассажиров, в соответствии с которыми клиенты могут получить бесплатные билеты по некоторым маршрутам после того, как они налетают необходимое количество километров, используя платные билеты. Как правило, требуется налетать разное количество километров для получения бесплатного билета в разных направлениях. Кроме того, существуют и другие условия. Далее приводится описание такой программы.

1. Клиент, летающий из США, может выбрать между несколькими различными маршрутами, в том числе: США, Северная Америка (за пределами США), Европа, Индия, Африка и Ближний Восток, Азия и Тихий океан. В последующем обсуждении термин "маршрут" (destination) будет использоваться для указания на один из этих вариантов.
2. Путешественник должен пролететь различное количество километров, в зависимости от того, приходятся ли даты путеше-

ствия на пик сезона путешествий по этому направлению. В последующем обсуждении термин "пик сезона" (peak) будет использоваться в этом значении.

3. Наконец, вне зависимости от времени года существуют определенные праздники, когда невозможно получить бесплатные билеты до тех пор, пока клиент не налетал дополнительное количество миль. В последующем обсуждении термин "праздник" (holiday) будет использоваться в этом значении.

Информация о такой программе может быть представлена в табл. 6.3. Числа в таблице будут количеством километров, которое нужно набрать, чтобы полететь в данном направлении при существующих условиях.

Таблица 6.3. *Данные для бонусной программы полетов*

Маршрут	Пик сезона	Праздник	Количество километров
США	Нет	Нет	20 000
		Да	40 000
	Да	Нет	25 000
		Да	50 000
Северная Америка (за пределами США)	Нет	Нет	35 000
		Да	70 000
	Да	Нет	35 000
		Да	70 000
Европа	Нет	Нет	40 000
		Да	100 000
	Да	Нет	50 000
		Да	100 000
Индия, Африка и Ближний Восток	Нет	Нет	80 000
		Да	140 000
	Да	Нет	90 000
		Да	140 000

Таблица 6.3 *(окончание)*

Маршрут	Пик сезона	Праздник	Количество километров
Азия и Тихий океан	Нет	Нет	50 000
		Да	120 000
	Да	Нет	60 000
		Да	120 000

Имея эту информацию, можно написать программу, отвечающую на следующий вопрос: для определенных маршрутов и дат каково количество миль, необходимое для получения бесплатного билета?

Пусть `destination`, `peak` и `holiday` будут именами переменных класса `String` в программе.

Предположим также, что вы использовали следующие аббревиатуры для маршрутов: U, N, E, I, A, соответственно, для США, Северной Америки, Европы, Индии и Азии; O и P, соответственно, для слов off-peak (вне пика) и peak (пик); N и H, соответственно, для слов non-holiday (не праздник) и holiday (праздник). Тогда программа, предназначенная для определения значений в таблице для часто летающих путешественников, будет выглядеть следующим образом:

```
public class FrequentFlyer
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    String destination, peak, holiday;
    destination = myterminal.getString("Enter U, N, E, I,
                  or A for your destination.");
    peak = myterminal.getString("Enter P or O for peak
                  or off-peak");
    holiday = myterminal.getString("Enter H or N for holiday
                  or non-holiday.");
    if(destination.equalsIgnoreCase("U"))
    {
```

```
      if(!peak.equalsIgnoreCase("P"))
      {
        if(!holiday.equalsIgnoreCase("H"))
        {
          myterminal.println("20 000 kilometers .");
        }
        else
        {
          myterminal.println("40 000 kilometers.");
        }
      }
      else
      {
        if(!holiday.equalsIgnoreCase("H"))
        {
          myterminal.println("25 000 kilometers.");
        }
        else
        {
          myterminal.println("50 000 kilometers.");
        }
      }
    }
              /* и т. д. */
  }
}
```

Заметьте, что у вас имеются строковые объекты, для которых вы ожидаете ввод пользователем одного символа, а затем сравниваете содержимое этих строк с постоянными значениями строк, которые содержат единственный символ. В этом нет ничего плохого. Более того, это весьма желательно, поскольку вы знакомы с различными полезными методами класса `String`, которые позволяют вам производить сравнения.

В Java также имеется простой тип `char`. Класс `MyTerminalIO` не имеет метода для использования такого типа, поэтому мы пока не можем им воспользоваться. Мы упоминаем о нем здесь, поскольку в этом заключается причина возможных ошибок. Символьная константа отделяется апострофами, а не двойными кавычками, и

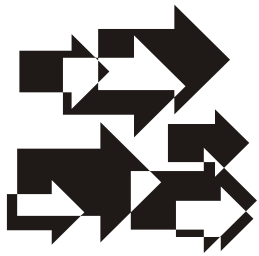
к ней не относятся строковые методы. Более детальное описание работы с символами будет дано в *главе 10*. А пока посмотрите, каков синтаксис для объявления и инициализации символьных переменных, которые четко отличаются от объявления и инициализации строк:

```
char mychar;
mychar = 'K';
```

Кроме возможной путаницы строк и символов, существуют другие типичные ошибки, совершаемые во время использования `if`-оператора и условий проверки:

- использование знака `=` вместо `==` во время проверки на равенство;
- использование символа `;` в конце `if`, `else if` или `else`;
- во время применения `else if` забывают закончить последовательность простым `else`;
- не используются фигурные скобки для выделения блока кода, который принадлежит условию;
- смешиваются скобки во время выделения блоков кода. Полезно всегда отмечать блоки отступом для того, чтобы можно было бы увидеть, где начинается один блок и где кончается другой;
- путают проверку равенства ссылок (`==`) с проверкой равенства содержимого (`equals()`);
- забывают, что метод `equals()` не проверяет равенство содержимого объектов классов, написанных программистом.

ГЛАВА 7



Циклы

7.1. Циклы *while* и *do*

Желательно, чтобы программы умели повторять определенные действия многократно. Такие действия называются повторениями, или *циклами*. Существует несколько структур в Java, которые позволяют сделать это. Одним из них является цикл `while`. Этот вид цикла выполняет действия с неизвестным заранее числом повторений. Он управляется таким же типом истинностных условий, которые рассматривались для оператора `if` в *главе 6*.

Цикл `while` может быть использован для получения вводимых данных от пользователя. Фрагмент кода, приводимый далее, иллюстрирует такое использование. Значение устанавливается и проверяется в условии цикла. Если условие верно, то действия цикла выполняются, производится выполнение тела цикла. Где-то в теле цикла переменная проверяется и условие изменяется. После того как тело выполнено, управление возвращается к началу цикла, и условие проверяется снова. Если условие ложное, тело цикла пропускается и управление передается операторам, следующим после цикла.

```
String answer = "yes";
while(!answer.equalsIgnoreCase("no"))
{
  //  Проделайте все, что необходимо, в теле цикла.
  myterminal.println("Go again?");
```

```
  myterminal.println("Answer yes or no.");
  answer = myterminal.getString();
}
```

До тех пор пока пользователь продолжает вводить `yes` при подсказке, цикл повторяется. Если введено `no`, это приведет к окончанию цикла и продолжению выполнения действий, следующих после тела цикла. Заметьте, что условие теста не чувствительно к регистрам букв.

Простые арифметические операции могут быть использованы для иллюстрации сути циклов. Деление целых чисел может быть определено как повторяющееся вычитание. Сколько раз число входит в другое — это то же самое, что сколько раз оно может вычитаться из другого до тех пор, пока результатом не окажется число, меньшее чем то, на которое вы делите. Цикл `while` является подходящей структурой для выполнения подобного алгоритма. Подсчет количества выполнений цикла и представляет собой ответ задачи на деление. Заранее неизвестно, какая это будет величина. Рассмотрим фрагмент кода, иллюстрирующий алгоритм, в котором желаемый результат хранится в переменной, называемой `quotient` (частное):

```
int dividend, divisor, quotient, remainder;
quotient = 0;
myterminal.println("Enter an integer dividend.");
dividend = myterminal.getInt();
myterminal.println("Enter an integer divisor.");
divisor = myterminal.getInt();
remainder = dividend;

while(remainder >= divisor)
{
  remainder = remainder - divisor;
  quotient++;
}
```

Такое простое исполнение не охватывает все случаи деления целых чисел. У него есть дефект, который иллюстрирует одну из ловушек программирования с циклами. Посмотрите, что случится, если делимое является положительным числом, а делитель —

отрицательным. Перед циклом остатку придается значение делимого, так что он является положительным. Условием цикла является *остаток >= делитель*, что, безусловно, будет определенно верным, так что цикл будет задействован. Внутри цикла остаток имеет отрицательное значение, вычтенное из него. Он остается положительным и только увеличивается. Условие цикла никогда не будет неверным.

Когда цикл существует в коде и условие цикла всегда верно, это приводит к ошибке, известной как *бесконечный цикл*. Компилятор не может заранее обнаружить такую ситуацию. Это ошибка выявляется в процессе работы, т. к. программа будет бесконечно выполнять цикл. Быстрым решением этой проблемы является прерывание программы, используя комбинации клавиш <Ctrl>+ +<C> или <Ctrl>+<Alt>+<Del> или закрыв окно, в котором работает программа. Иногда система настолько застревает в цикле, что приходится перезагружать компьютер. Остановив выполнение программы, вы должны найти и исправить ошибку в исходном коде, которая делает условие цикла всегда верным.

В общем, циклы, основанные на числовых значениях, базируются на проверке неравенства, где одно или другое значение увеличивается или уменьшается в теле цикла. Пример далее показывает общую форму, где значение увеличивается. В данном примере использовано строгое неравенство, но и строгое, и нестрогое неравенства возможны, в зависимости от логики конкретной ситуации.

```
while(x < y)
{
  /*  Увеличьте x на некоторое число.  */
}
```

При определенных условиях можно написать цикл, который выполнил бы то же самое в следующей форме, где `x` должен получить точное окончательное значение:

```
while(x != y)
{
  /*  Увеличьте x на некоторое число.  */
}
```

Это неразумно. Существует возможность того, что x плюс инкремент никогда не будет точно равен y. Это может произойти потому, что двоичное представление десятичных значений может быть неточным. В результате мы получим бесконечный цикл, где значение x превысит y. Если условие основывается на < или >, даже если точное значение не встречается, цикл остановится.

Циклы do схожи с циклами while. Вместо проверки условия цикла в начале циклов, они проверяют условия после того, как будет выполнено тело цикла. Цикл do может быть использован в ситуации, где что-либо должно быть сделано хотя бы один раз без написания специальной подсказки и ввода выражений до начала цикла. Вот фрагмент кода, иллюстрирующий такую ситуацию:

```
double rate;
String answer = "yes";
do
{
  myterminal.println("Enter a value for rate.");
  rate = myterminal.getDouble();
  /*  Произведите вычисления, основанные на
      значении переменной rate.  */
  myterminal.println("To quit, enter no");
  anwer = myterminal.getString();
} while(!answer.equalsIgnoreCase("no"));
```

Ключевое слово do отмечает шапку цикла, но не выполняет условие. Условие проверяется только после того, как основная часть цикла будет выполнена один раз. Обратите внимание, что нет никаких пунктуационных знаков перед первой фигурной скобкой или после второй фигурной скобки цикла, и стоит точка с запятой после условия цикла.

7.2. Цикл *for*

Цикл for — это цикл, в котором заранее известно, сколько раз будет выполняться тело цикла. Вообще, можно написать цикл while, который будет также управлять такой ситуацией. Однако обычно проще написать цикл for. Непрактично писать цикл for,

который предназначен для управления неизвестным количеством повторений.

Посмотрим на пример цикла `while`, который рассчитан на известное число повторений. Пусть переменные `startingValue` и `endingValue` содержат целые числа, при этом `startingValue < endingValue`.

```
int i = startingValue;
while(i <= endingValue)
{
  /* Сделайте все, что необходимо, в теле цикла. */
  i++;
}
```

Элементами данного цикла будут: инициализация контрольной переменной цикла со значением `startingValue`; условие, зависящее от верхнего предела `endingvalue`, и его инкрементация. Эквивалентный цикл `for` включает эти три вещи в одной компактной строке кода:

```
int i;
for(i = startingValue; i <= endingValue; i++)
{
  /* Сделайте все, что необходимо, в теле цикла. */
}
```

Часто в программировании полезнее начать индекс цикла с 0, а условием остановки цикла сделать строгое неравенство. Поскольку нумерация позиций в строке начинается с 0, то это представляет собой хорошую иллюстрацию данного положения:

```
int i;
String mystring = "Something";
for(i = 0; i < mystring.length(); i++)
{
  /*  Сделайте это для примера.  */
  myterminal.println(mystring.substring(i, i + 1));
}
```

При написании циклов `for` можно отсчитывать вверх или вниз, т. е. инкрементировать или декрементировать. Также можно от-

считывать величинами, отличными от 1, например, 2, 3 и т. д. Также индекс цикла может быть числом типа `double`, а не только целым числом, и можно отсчитывать дробными единицами. При счете вниз условие остановки должно быть "более чем" (greater than), а не "менее чем" (less than). Вот небольшой пример, иллюстрирующий эти положения:

```
double x;
for(x = 3.5; x >= -2.0; x = x — 0.5)
{
  ...
}
```

Обратите внимание, что выражение `for` содержит присваивание, которое выглядит так: `x = x - 0.5`. Программисты привыкают к сокращенному синтаксису для прибавления или вычитания на 1, `x++` и `x--`, и забывают, что можно иметь нормальное выражение присваивания, для того чтобы прибавить или вычесть значение, отличающееся от 1.

7.3. Область действия

Следующий ряд примеров относится к теме "область действия". Это касается вопроса о том, где в программе происходит определение переменной. В следующей главе область действия будет рассмотрена более подробно, но некоторые вещи, которые могут произойти с объявлением индексов цикла, требуют предварительного пояснения уже сейчас. В целом, невозможно объявить две переменные с одинаковым именем в той же самой программе. Вы получите ошибку компилятора, если в вашей программе будут подобные выражения:

```
int myint = 5;
...
int myint = 3 * 2;
```

Объявление одной переменной дважды называется *ошибкой повторного объявления*.

Следующие примеры иллюстрируют область действия, рассматривая фрагменты программ, где значение индекса цикла выво-

дится на экран внутри цикла или после цикла. Компилятор может пожаловаться, если переменная объявлена, но не инициализирована. Там, где это необходимо, в этих примерах переменная индекса инициализирована, так что единственное, что анализируется, — это область действия.

В языке Java имеется возможность объявить переменную, контролирующую цикл, называемую *индексом (или параметром) цикла*, в рамках оператора `for`. Если это сделано, то переменная определяется внутри пары фигурных скобок, содержащих тело цикла, например:

```
for(int i = 1; i <= 7; i++)
{
  /*  Переменная i существует в этих скобках.  */
  myterminal.println(i);
}
```

Теперь рассмотрим следующий пример, где переменная `i` объявлена только в операторе `for` и нигде больше в программе. За циклом следует `println()`, где производится попытка вывести на экран значение `i`. В таком случае компилятор пожалуется, что проблема появления `i` в вызове `println()` после цикла не может быть разрешена. Другими словами, объявление `i` внутри оператора `for` определяет эту переменную только внутри фигурных скобок, которые принадлежат циклу. Это можно выразить и так, что областью действия переменной является пара фигурных скобок, принадлежащих циклу:

```
for(int i = 1; i <= 7; i++)
{
  myterminal.println(i);
}
myterminal.println(i);
```

В следующем коде переменная `i` объявлена до цикла, а затем снова в операторе `for` . Это приведет к ошибке повторного объявления:

```
int i = 0;
for(int i = 1; i <= 7; i++)
```

```
{
  myterminal.println(i);
}
```

Следующий пример не приведет к ошибке компилятора. Объявление индекса снова после цикла не считается повторным объявлением. В рамках цикла `i` является индексом цикла. После цикла вновь объявленная `i` является совершенно новой и иной переменной. Переменная `i` после цикла не содержит значения индекса цикла, после того как цикл завершен:

```
for(int i = 1; i <= 7; i++)
{
  myterminal.println(i);
}
int i = 0;
myterminal.println(i);
```

Может показаться непоследовательным, что объявление переменной с тем же самым именем до цикла приводит к ошибке, а после цикла — ошибки нет. Но этому есть простое объяснение. Переменная `i` в операторе `for` синтаксически появляется до фигурных скобок, к которым она принадлежит. Это значит, что компилятор видит два объявления той же самой переменной `i` в пределах одной области действия, пары скобок (метода, например), содержащих оператор `for`. После закрывающей скобки пары скобок, принадлежащих оператору `for`, компилятор делает вывод, что переменная `i`, объявленная для оператора, принадлежит области действия этой пары скобок. Он сможет отличить нечто, следующее далее, с тем же именем, без всяких проблем.

Лучшим решением, как обычно, будет объявить и инициализировать все переменные, которые потребуются, в начале кода. В показанном далее коде переменная `i` будет доступна до, во время и после цикла. Значение, которое она примет сразу после цикла, будет последним значением, которое она получила в цикле:

```
int i = 0;
for(i = 1; i <= 7; i++)
{
  myterminal.println(i);
}
myterminal.println(i);
```

В общем, область действия можно понимать как скобки, в рамках которых объявляются переменные. Переменные доступны для использования внутри скобок в любой момент после того, как они объявляются. Если другая переменная с тем же именем объявляется внутри скобок, расположенных в данных скобках, тогда система может различать две переменные и придавать им разные значения. Это задача программиста — следить за тем, основываясь на области действия, которая из переменных задействована в разных частях кода. В последнем приведенном примере возможность путаницы была устранена за счет объявления переменной один раз в начале. В общем, программист может избавить себя от путаницы за счет того, чтобы не объявлять несколько переменных с одним и тем же именем.

7.4. Примеры

Рассмотрим пример использования цикла для того, чтобы реализовать математическую функцию. Вычисление целой степени какого-либо числа можно определить в терминах повторяющегося умножения. Основой является число, умножающееся само на себя. Степенью является количество раз, которое его необходимо умножить. Поскольку количество умножений известно заранее, такой алгоритм может быть выполнен с использованием цикла `for`. Степень служит верхним пределом для индекса цикла. Рассмотрим пример, иллюстрирующий эту идею.

```
public class PowerProgram
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    myterminal.println("Enter the base:  ");
    double mybase = myterminal.getDouble();
    myterminal.println("Enter a positive integral power:  ");
    int mypower = myterminal.getInt();
    double result = mybase;
    for(int i = 1; i < mypower; i++)
    {
      result = result * mybase;
    }
```

```
    myterminal.println(result);
  }
}
```

Обратите внимание, что по аналогии с примером, где деление было выполнено как повторяющееся вычитание, такая реализация вычисления степени не может быть применена во всех случаях, например, она не оперирует отрицательными или дробными степенями. Пример приведен просто для того, чтобы проиллюстрировать идею использования цикла, где ограничение на индекс цикла известно заранее.

Возможность производить какую-либо операцию много раз позволяет выполнить различные интересные алгоритмы. Рассмотрим задачу нахождения площади некоторой части плоскости. Предположим, вы хотите найти площадь области ниже кривой $y = x^2$ между точками $x = 0$ и $x = 1$. Для этих значений x значения y также находятся между 0 и 1. Интересующая нас область заштрихована на схеме (рис. 7.1).

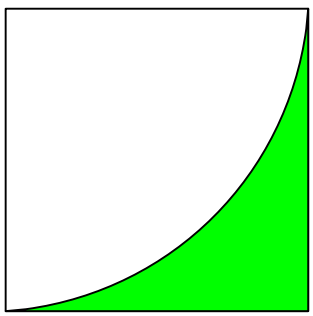


Рис. 7.1

Если вы изучали высшую математику, то знаете, что точный ответ можно получить, решив следующее интегральное уравнение:

$$\int_0^1 x^2 dx = \frac{x^3}{3}\Big|_0^1 = \frac{1}{3} - 0 = \frac{1}{3}.$$

Если же вы не изучали высшую математику, или не хотите ее использовать, или же у вас имеется задача, которая не имеет удов-

летворительного решения в высшей математике, то вы можете прибегнуть к другому способу нахождения площади. Предположим, что вы можете создавать произвольные точки, которые находятся в пределах всего квадрата. Координаты *x* для этих точек будут находиться между 0 и 1, а координаты *y* тоже будут находиться между 0 и 1. Если точки на самом деле произвольные, то соотношение, показанное далее, будет приближаться к площади области, находящейся ниже кривой. Чем больше будет точек, тем точнее будет данное отношение:

$$\frac{\text{Количество точек, у которых } y \text{ меньше, чем } x^2}{\text{Общее количество точек}}.$$

Возможный результат создания 18 случайных точек показан на рис. 7.2. Закрашенные точки находятся ниже кривой. Незакрашенные точки — выше кривой. Соотношение заштрихованных точек к общему количеству точек, 6/18, является аппроксимацией площади области, находящейся ниже кривой.

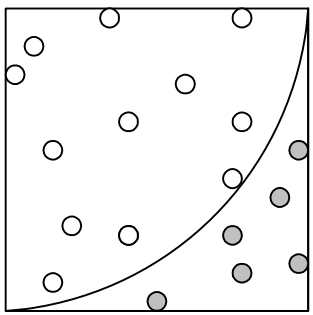


Рис. 7.2

Нахождение площади с использованием произвольных чисел известно как *метод Монте-Карло*.

В языке Java существует класс, который поддерживает генерирование произвольных чисел. Для получения полной информации об этом классе вам нужно обратиться к документации Java API. Далее приведены пояснения, касающиеся данного примера:

```
import java.util.Random;
...
Random mygenerator = new Random();
...
double myrandomDouble;
...
myrandomDouble = mygenerator.nextDouble();
...
```

Вызов метода `nextDouble()` вернет следующее значение:

```
0 <= myrandomDouble < 1
```

Это результат ситуации, когда используется конструктор по умолчанию класса `Random`. Если генератор случайных чисел создается с использованием параметра, этот параметр будет верхним пределом диапазона генерируемых случайных чисел.

Посмотрим на программу, которая решает уравнение методом Монте-Карло. Комментарии к программе объясняют, как выполняется алгоритм. Если вы запустите программу несколько раз, каждый раз указывая разное количество попыток, то увидите, что чем больше генерируется точек, тем ближе становится результат к точному решению:

```
import java.util.Random;
public class MonteCarlo
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    Random mygenerator = new Random();
    /*  trials равно общему количеству точек.  */
    int trials = 0;
    int count = 0;
    double area = 0.0;
    double randX, randY;
    myterminal.print("How may points do you want?  ");
    trials = myterminal.getInt();
    for(int i = 1; i <= trials; i++)
    {
```

```
      /*  Генерируются значения для x и y */
      /*  в интервале от 0 до 1. Вместе они */
      /*  представляют случайно сгенерированную точку */
      /*  на единице площади плоскости xy. */
      randX = mygenerator.nextDouble();
      randY = mygenerator.nextDouble();
      /*  Здесь проверяется, находится ли точка  */
      /*  ниже кривой. Если да, то количество   */
      /*  увеличивается.                */
      if(randY <= randX * randX)
        count++;
    }
    /*  Площадь приблизительно определяется путем деления */
    /*  количества точек ниже кривой на общее  */
    /*  количество точек (trials).  */
    area = (double) count / (double) trials;
    myterminal.println("The area under the curve is " +
                       area);
  }
}
```

7.5. Вложенные циклы

Циклы заключены в скобки, и любые скобки в Java могут содержать любой другой синтаксически верный набор выражений. Это значит, что циклы могут содержать другие циклы. Цикл `while` может содержать другой `while`-цикл или `for`-цикл, а `for`-цикл может содержать другой `for`-цикл или `while`-цикл. Каждый раз, когда один цикл содержит другой, это рассматривается как *вложение циклов* (размещение одного цикла внутри другого), и количество таких вложений может быть сколь угодно большим. Один цикл может содержать другой, который может содержать еще один, и т. д.

Вот простой пример программы, где цикл `for` содержится внутри цикла `while`. Цикл `while` управляет вводом пользователя, а `for`-цикл применяет алгоритм для нахождения суммы положительных целых чисел, меньших или равных данному целому числу.

```
public class NestedLoopEx1
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    int inputValue;
    int i;
    int mysum;
    myterminal.println("Enter a positive integer");
    myterminal.println("to find the sum of the");
    myterminal.println("integers less than or");
    myterminal.println("equal to it.");
    inputValue = myterminal.getInt();
    while(inputValue > 0)
    {
      mysum = 0;
      for(i = 1; i <= inputValue; i++)
      {
        mysum = mysum + i;
      }
      myterminal.println("The sum is:  " + mysum);
      myterminal.println("If you would like to");
      myterminal.println("go again, enter another");
      myterminal.println("positive integer.");
      myterminal.println("Otherwise, enter 0 or");
      myterminal.println("a negative integer.");
      inputValue = myterminal.getInt();
    }
  }
}
```

Мы увидим, что вложенные `for`-циклы представляют собой особый интерес из-за значений, которые принимают индексы циклов. Следующий пример программы иллюстрирует эти значения в виде таблицы. Внешний цикл, с индексом `i`, соответствует рядам в таблице. Внутренний цикл, с индексом `j`, соответствует колонкам. Обратите внимание на использование `print()` и `println()` в различных местах циклов для того, чтобы получить результат, напечатанный ряд за рядом в виде таблицы:

```
public class NestedLoopEx2
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    int i;
    int j;
    myterminal.println(
      "       Column 0   Column 1   Column 2   Column 3");
    for(i = 0; i < 4; i++)
    {
      myterminal.print("Row " + i + "  ");
      for(j = 0; j < 4; j++)
      {
        myterminal.print("(i=" + i + ", j=" + j +")  ");
      }
      myterminal.println();
    }
  }
}
```

Вот результат работы программы:

```
       Column 0   Column 1   Column 2   Column 3
Row 0 (i=0, j=0) (i=0, j=1) (i=0, j=2) (i=0, j=3)
Row 1 (i=1, j=0) (i=1, j=1) (i=1, j=2) (i=1, j=3)
Row 2 (i=2, j=0) (i=2, j=1) (i=2, j=2) (i=2, j=3)
Row 3 (i=3, j=0) (i=3, j=1) (i=3, j=2) (i=3, j=3)
```

7.6. Графические примеры

Каждый из следующих примеров содержит блок кода, подобный тому, что показан далее. Этот блок вызывает задержку в выполнении программы. Это желательно в случае применения анимации, чтобы ее скорость была такой, какую может уловить человеческий глаз. Параметром метода `sleep()` является количество миллисекунд задержки. В данном случае он вызовет задержку на четверть секунды. Пока не обращайте внимание на остальную часть синтаксиса и его значение. Если вы захотите использовать подобный код при написании собственных программ, то можете

просто скопировать пример и изменить параметр значения так, как вам нужно.

```
try
{
  Thread.sleep(250);
}
catch(InterruptedException e)
{
  System.out.println("InterruptedException: " + e);
}
```

Посмотрите на апплет, который анимирует предыдущий пример, NestedLoopEx2.java. Этот апплет содержит двойные вложенные циклы, в результате чего точки показываются в двухмерном изображении, ряд за рядом, и каждый индекс цикла соответствует одной из координат такого изображения:

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
public class NestedLoopApplet extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    int i, j;
    Ellipse2D.Double mycircle;
    double circleX = 75.0;
    double circleY = 75.0;
    Color mycolor = new Color(0.75F, 0.0F, 0.75F);
    g2.setColor(mycolor);
    Rectangle2D.Double myrectangle =  new
      Rectangle2D.Double(25.0, 25.0, 265.0, 265.0);
    g2.draw(myrectangle);
    for(i = 0; i < 4; i++)
    {
      for(j = 0; j < 4; j++)
```

```
      {
        mycircle = new
          Ellipse2D.Double(circleX, circleY, 15.0, 15.0);
        g2.fill(mycircle);
        circleX = circleX + 50.0;
        /*  Этот код вызывает задержку в четверть секунды
        для каждой точки, которая отображается.  */
        try
         {
          Thread.sleep(250);
        }
        catch(InterruptedException e)
        {
          System.out.println("InterruptedException:  " + e);
        }
      }
      circleX = 75.0;
      circleY = circleY + 50.0;
    }
  }
}
```

Результат работы программы представлен на рис. 7.3.

А вот апплет, который анимирует решение уравнения методом Монте-Карло. Интерактивный ввод с использованием класса `MyTerminalIO` не работает в апплетах, поэтому количество попыток жестко задано. Мы выбрали определенные числовые параметры так, чтобы результат выглядел красиво. В частности, параметры подобраны таким образом, чтобы окружности, представляющие случайные точки данных, оставались внутри пределов прямоугольника, и так, чтобы окружности правильно отображались со своими центрами выше или ниже линии $y = x^2$, в соответствии со значениями своих координат. Из-за такой регулировки код, возможно, выглядит немного сложным, но его основная идея точно такая же, как и для неграфического представления метода Монте-Карло. Используя знания, полученные после изучения всех предыдущих глав, вы можете написать сравнительно короткий апплет, который может помочь вам визуально

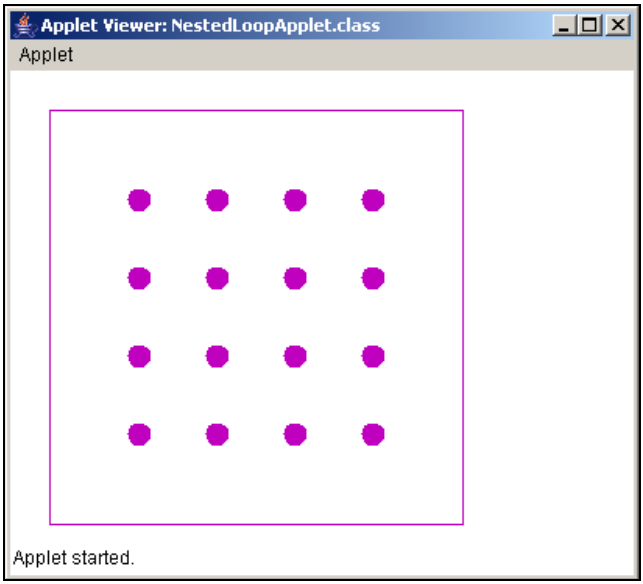


Рис. 7.3

представить алгоритм, воплощенный во фрагменте компьютерного кода:

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
import java.util.Random;
public class MonteCarloApplet extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    Random mygenerator = new Random();
    int trials = 100;
    int count = 0;
    double area = 0.0;
    double randX, randY;
```

```
int i, j;
Ellipse2D.Double mydot;
Ellipse2D.Double mycircle;
double dotX, dotY, circleX, circleY;
Color mycolor = new Color(0.75F, 0.0F, 0.75F);
g2.setColor(mycolor);
/*  Рисуется  прямоугольник.  */
Rectangle2D.Double myrectangle = new
Rectangle2D.Double(25.0, 25.0, 265.0, 265.0);
g2.draw(myrectangle);
/*  Рисуется кривая с использованием точек.
Индекс i получает различные значения для
размера x прямоугольника.  */
for(i = 0; i < 265; i++)
{
  /* Координата dotX должна быть сдвинута
  на 25 позиций направо, так как координата x
  левого верхнего угла прямоугольника равна 25.  */
  dotX = (25.0 + i);
  /*  В этом выражении, i * i представляет возведение
  в квадрат. Результат делится на 265 для того,
  чтобы он был пропорционален высоте прямоугольника.
  Поскольку ось y на графике идет сверху вниз,
  то если не внести дальнейшие поправки,
  кривая будет показана вверх ногами.
  Значение для нижнего края, вычтенное из 290,
  дает правильное, зеркальное изображение кривой.  */
  dotY = 290.0 - (i * i) / 265.0;
  mydot = new
    Ellipse2D.Double(dotX, dotY, 1.0, 1.0);
  g2.draw(mydot);
}
/*  Меняет цвет на синий и рисует точки,
изображая их в виде маленьких кругов.  */
g2.setColor(Color.blue);
for(i = 1; i <= trials; i++)
{
  randX = mygenerator.nextDouble();
  randY = mygenerator.nextDouble();
```

```
/*  Этот код вызывает задержку в одну десятую
секунды для каждой произвольной точки,
которая показывается.  */
try
{
  Thread.sleep(100);
}
catch(InterruptedException e)
{
  System.out.println("InterruptedException:  " + e);
}
/*  Для того чтобы уравнение Монте-Карло было верным,
randX и randY вместе являются координатами точек,
которые не видны. Для изображения этих точек
используются окружности, и произвольные точки
являются центрами таких окружностей.
Окружности ориентируются по верхнему левому углу
описывающих прямоугольников, так что координаты
верхнего левого угла должны быть выведены
из координат центра. Также необходимо внести поправки,
чтобы окружности полностью находились
внутри прямоугольника.  */

/*  В этом выражении умножение на 265
приводит значение x в соответствие с шириной
прямоугольника. Добавление 25 позиционирует его
в рамках левой границы прямоугольника.
Окружности имеют радиус 5, так что вычитание
5 перемещает верхний левый угол описывающего
прямоугольника на 5 единиц влево от центра.
В частности, эти поправки позволяют
удостовериться, что точки, находящиеся рядом с кривой,
будут верно изображаться или выше, или ниже ее.  */
circleX = 25.0 + randX * 265.0 - 5.0;

/*  Если, после того как к точке применены указанные
выше поправки, она не находится полностью внутри
прямоугольника, то тогда ее положение необходимо
изменить, чтобы она находилась внутри него.
```

```
Это означает,  что представление на экране
не точно соответствует сгенерированным значениям,
но визуально это допустимо и не влияет на нахождение
приблизительного ответа для уравнения
методом Монте-Карло.  */
if(circleX > 280.0)
  circleX = 280.0;
if(circleX < 25.0)
  circleX = 25.0;

/*  В этом выражении умножение на 265 приводит
значение y в соответствие с высотой прямоугольника.
Вычитание из 290 завершает необходимое отражение,
поскольку ось y перевернута на рисунке.
Также оно помещает точку внутрь нижней границы
прямоугольника. Окружности имеют радиус 5, поэтому
вычитание 5 смещает верхний левый угол описывающего
прямоугольника на 5 единиц выше центра. В частности,
эти поправки позволяют удостовериться, что точки,
находящиеся рядом с кривой, будут правильно
располагаться выше или ниже ее.  */
circleY = 290.0 - (randY * 265.0) - 5.0;

/*  Если, после того как к точке применены указанные
выше поправки, она не находится полностью внутри
прямоугольника, то тогда ее необходимо сместить,
чтобы она находилась внутри. Это означает,
что представление на экране не точно соответствует
сгенерированным значениям, но визуально это
допустимо и не влияет на нахождение приблизительного
ответа для уравнения методом Монте-Карло.  */
if(circleY > 280.0)
  circleY = 280.0;
if(circleY < 25.0)
  circleY = 25.0;
mycircle = new
  Ellipse2D.Double(circleX, circleY, 10, 10);

/*  Вы заполняете или рисуете окружности
в зависимости от того, находится ли точка выше
или ниже кривой.  */
```

```
        if(randY <= randX * randX)
        {
          count++;
          g2.fill(mycircle);
        }
        else
        {
          g2.draw(mycircle);
        }
      }
      /*  Высчитываете и показываете результат.  */
      area = (double) count / (double) trials;
      g2.drawString("The area under the curve is " +
                    area, 25, 15);
    }
}
```

Результат работы программы представлен на рис. 7.4.

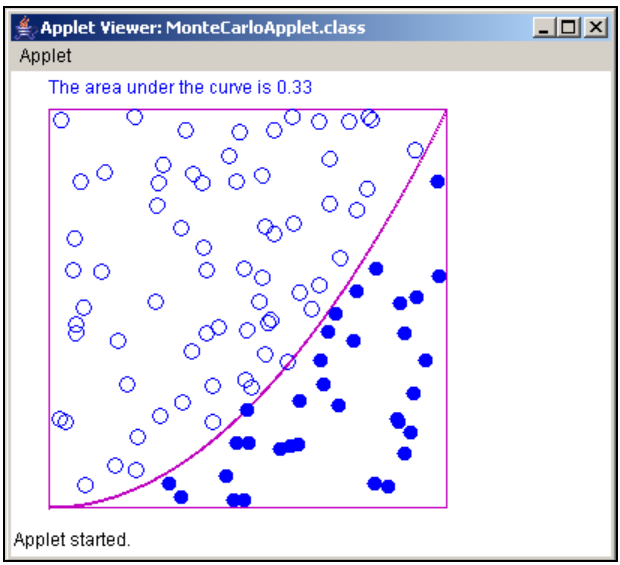


Рис. 7.4

Последний пример иллюстрирует тройной вложенный цикл, пересекающий трехмерное пространство. Пример также показывает

смешение красного, зеленого и синего цветов для того, чтобы получить различные цвета. Каждая из осей трехмерного пространства соответствует одному из трех основных цветов:

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
public class ColorCubeApplet extends JApplet
{
  public void paint(Graphics g)
  {
    float redPart, greenPart, bluePart;
    Color mycolor;
    double baseX = 0;
    double baseY = 0;
    double myx = 0;
    double myy = 0;
    double diam = 30;
    Graphics2D g2 = (Graphics2D) g;
    for(redPart = 0.0F; redPart <= 1.0; redPart += .25)
    {
      for(greenPart = 0.0F; greenPart <= 1.0;
          greenPart += .25)
      {
        for(bluePart = 0.0F; bluePart <= 1.0;
            bluePart += .25)
        {
          myx += 55;
          Ellipse2D.Double mycircle =
            new Ellipse2D.Double(myx, myy, diam, diam);
          mycolor = new
            Color(redPart, greenPart, bluePart);
          g2.setColor(mycolor);
          g2.fill(mycircle);
          /*  Этот код вызывает задержку в одну десятую
          секунды для каждой точки, которая показана.   */
          try
          {
```

```
            Thread.sleep(100);
          }
          catch(InterruptedException e)
          {
          System.out.println("InterruptedException:  " + e);
          }
        }
        baseX += 10;
        myx = baseX;
        myy += 10;
        diam -= 3;
      }
      baseX = 0;
      myx = baseX;
      baseY += 40;
      diam = 30;
    }
  }
}
```

Результат работы программы представлен на рис. 7.5.

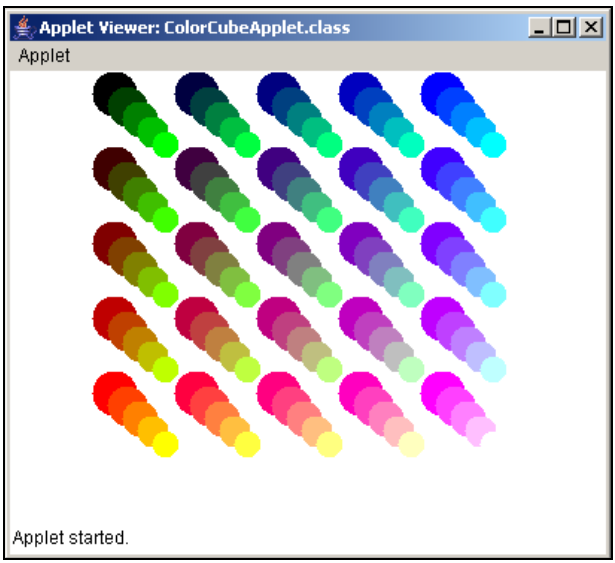
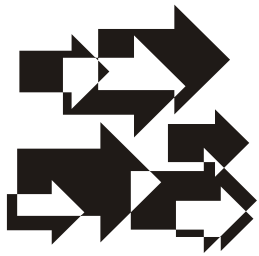


Рис. 7.5

ГЛАВА 8



Методы, параметры и переменные

8.1. Передача параметров простых типов

Полезно ввести еще некоторые термины, касающиеся передачи параметров, и обсудить их использование более подробно. Одной из основных моделей написания кода в Java является следующая модель:

```
object.method(parameters);
```

Объект указывается в качестве скрытого параметра, а любая переменная, объявленная внутри скобок, является явным параметром. Более одного параметра может быть объявлено внутри скобок:

```
public void mymethod(double x, double y)
{
  ...
}
```

Переменные `x` и `y`, объявленные в определении метода, указываются как *формальные параметры*. Этот термин обозначает, что они принимают значения во время работы программы. Эти переменные активны, когда код метода выполняется. Их имена действительны внутри скобок, которые заключают в себе код метода, подобно именам переменных, которые объявлены внутри этих же

скобок. Когда заканчивается вызов метода, независимо от того, возвращает ли он что-либо или нет, эти переменные не сохраняют никаких значений.

Существует только одна копия кода для каждого метода класса. Каждый объект получает свои копии переменных экземпляра, но код метода один для всех объектов класса, которые существуют. Во время работы различные объекты могут вызывать один и тот же метод. Несмотря на это, ни одно значение параметра из предыдущего вызова не будет существовать, когда происходит следующий вызов.

В программе вызов метода производится следующим образом:

```
double q, r;
...
object.mymethod(q, r);
...
```

Переменные `q` и `r` называются *фактическими параметрами*. Другими словами, в программе им придаются значения, которые затем передаются методу во время вызова. Эти фактические параметры должны быть такого же типа, как и формальные параметры, объявленные в методе. С точки зрения переменных, вот что происходит при вызове метода. Значение фактического параметра `q` копируется в формальный параметр `x`. Значение фактического параметра `r` копируется в формальный параметр `y`.

Передача параметра подобна присваиванию, которое пересекает границу между программой и методом. Нет связи между `q` и `x` и между `r` и `y`, кроме копирования значений из одного в другое. Никакие изменения `x` и `y` в методе не оказывают влияния на `q` и `r`. В конце вызова, когда метод возвращает управление, значения `x` и `y` теряются. Значения `q` и `r` в вызывающей программе остаются неизменными. Если желательно получить результат метода, то необходимо написать метод для возвращения такого результата.

Поскольку копирование значений фактического параметра в формальные параметры похоже на присваивание, то фактические параметры не обязательно всегда должны быть точно такого же типа, как и формальные параметры. Достаточно того, что фактические параметры будут такого типа, который может быть ус-

пешно присвоен формальным параметрам. Например, возможно присвоить значение `integer` переменной типа `double` без приведения типов (casting). Это означает, что значение параметра целого типа может быть передано формальному параметру типа `double` без проблем. Если же значение фактического параметра не может быть приписано формальному параметру без преобразования, то компилятор выдаст сообщение об ошибке.

Это иногда сбивает с толку начинающих программистов. Предположим, что объявлены формальные параметры таким образом:

```
public void mymethod(double x, double y)
{
  ...
}
```

Теперь посмотрите на следующее объявление и вызов в программе:

```
double x, y;
...
object.mymethod(x, y);
...
```

Существует ли здесь какая-нибудь связь между `x` и `y` в программе и между `x` и `y` в методе? Здесь есть только одна связь: когда выполняется метод, значение `x` из программы копируется в `x` в методе, и значение `y` из программы копируется в `y` в методе. Имеются две разные переменные `x` с одним и тем же именем, и две разные переменные `y` с одним и тем же именем. Они отличаются друг от друга, потому что объявлены внутри различных скобок, и по этой причине система может различить их. Изменения `x` или `y` в методе не изменяет `x` или `y` в коде программы.

Есть еще один связанный с этим вопрос, который иногда смущает начинающих программистов. Рассмотрим следующий код класса, который можно скомпилировать, но который является неверным:

```
public class CupX
{
  private int seedCount;
  public CupX()
```

```
  {
    seedCount = 0;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  /*  Нет!  Нет!  Нет!  */
  public void setSeedCount(int seedCount)
  {
    seedCount = seedCount;
  }
}
```

Проблема заключается в выражении присваивания в методе `setSeedCount()`: `seedCount = seedCount`. Посмотрите, что произойдет, если следующая программа, тестирующая данный класс, будет запущена:

```
public class TestCupX
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    CupX mycup = new CupX();
    mycup.setSeedCount(5);
    int result = mycup.getSeedCount();
    myterminal.println(result);
  }
}
```

Эта программа выведет на экран результат 0. Когда конструируется объект класса `Cup`, то `seedCount` установлен на 0. Вызов метода `setSeedCount()` не имеет никакого эффекта. Посмотрите, почему это происходит. Переменная экземпляра `seedCount` объявлена в начале скобок, содержащих определение класса. Она существует в рамках этих скобок. Формальный параметр `seedCount` определен в скобках в начале определения метода `setSeedCount()`. Он существует в рамках скобок, содержащих это определение. Система не будет жаловаться на наличие двух разных переменных

с именем `seedCount`, поскольку они различны. Внутри метода переменная, названная `seedCount`, является параметром. В любом другом месте в определении класса переменная, названная `seedCount`, является переменной экземпляра.

Строка кода в методе, `seedCount = seedCount`, включает только параметр. Она не влияет на переменную экземпляра с тем же именем. Когда выполняется программа, в коде метода текущее значение параметра `seedCount`, равное 5, снова присваивается параметру. Переменная экземпляра, которая содержит значение 0, не изменяется исполнением метода. Система может следить за переменными с одинаковыми именами и различать, где применяется каждая из них. Однако код класса, показанный выше, является и бесполезным, и сбивающим с толку, поскольку он имитирует некоторую работу, что на самом деле не происходит.

Подводя итог, можно сказать, что фактические и формальные параметры могут иметь одинаковые имена, в то время как формальные параметры и переменные экземпляра не могут иметь одинаковые имена. Вы можете задать вопрос: а допустимо ли фактическим параметрам и переменным экземпляра иметь одинаковые имена? Ответ будет положительным, поскольку система различает их. Фактические параметры объявляются и существуют только внутри скобок вызывающей программы. Переменные экземпляра объявляются и существуют в рамках скобок определения класса. Конечно, лучше давать им разные имена, чтобы не запутаться. Однако следующий код вызывающей программы будет абсолютно верным:

```
int seedCount = 5;
mycup.setSeedCount(seedCount);
```

Есть еще один вопрос, касающийся имен и типов формальных параметров. Мы уже видели, что можно иметь более одного конструктора с одинаковыми именами в одном и том же классе. Система различает их по списку параметров. Класс также может иметь более одного метода с одинаковыми именами. Тип и последовательность параметров позволяет системе различать разные конструкторы и методы. Имена формальных параметров, рассмотренных ранее, не позволяют системе различать разные

конструкторы с одинаковым именем или разные методы с одинаковым именем.

Например, не может быть двух разных методов в одном классе со списком параметров такого типа:

```
public void mymethod(double x, int y)
public void mymethod(double v, int w)
```

Система обнаружит подобную ошибку. Имена формальных параметров различны, но каждый из методов с одним и тем же именем содержит список параметров, в котором один параметр имеет тип `double`, а второй `int`, в одинаковом порядке.

С другой стороны, можно записать два метода следующим образом:

```
public void mymethod(int x, double y)
public void mymethod(double x, int y)
```

Или же можно записать их так:

```
public void mymethod(int x, double y)
public void mymethod(double y, int x)
```

В обоих случаях, система отмечает, что первый метод имеет список параметров, состоящий из переменной типа `int` и переменной типа `double`, в то время как у второго метода сначала идет переменная типа `double`, а затем переменная типа `int`. Это единственное, что имеет значение. Имена формальных параметров, `x` и `y`, не важны. Конечно, существуют ограничения на возможность разным методам иметь одинаковые имена и различение их по списку параметров. Главное, что нужно понять, — это то, что переменным экземпляра и формальным параметрам важно иметь разные имена. Однако невозможно различить два метода с одинаковым именем только на основании разных имен формальных параметров.

8.2. Передача объектных ссылок в качестве параметров

Посмотрим на вызов программы для приписывания значения переменной `seedCount` объекта:

```
mycup.setSeedCount(4);
```

Метод `setSeedCount()` выглядит так:

```
public void setSeedCount(int newCount)
{
  seedCount = newCount;
}
```

Объект, к которому вызывается метод, называется *скрытым параметром*. Он доступен внутри метода. В нашем примере код метода может изменить значение переменной экземпляра скрытого параметра `mycup`.

Мы также видели, что объектные ссылки могут передаваться в виде *явных параметров*:

```
Point2D.Double mypoint =
  new Point2D.Double(10, 20, 30, 40);
myterminal.println(mypoint);
```

Явный параметр — `mypoint`, и он становится доступным внутри метода `println()`. В данном случае код метода не изменяет значение явного параметра, но он имеет доступ к нему и вызывает вывод информации на экран.

Теперь настало время научиться писать методы класса, которые принимают объектные ссылки в качестве явных параметров. Класс `Cup4` в настоящее время имеет метод доступа, который принимает целое число в качестве параметра и присваивает это значение `seedCount`. Предположим, что вы хотите иметь возможность передать значение `seedCount` одной чашки переменной `seedCount` другой чашки. Это можно сделать с использованием методов доступа и модифицирующих методов. Это также можно сделать, написав новый метод для класса `Cup`. Поскольку мы добавляем новый метод, то пусть наш класс будет называться теперь `Cup5`. Посмотрите на этот метод:

```
public void
  assignSeedCountToAnotherCup(Cup5 anotherCup)
{
  int temp = this.getSeedCount();
  anotherCup.setSeedCount(temp);
}
```

В этом коде ключевое слово `this` используется в качестве скрытого параметра. Как работает код, должно быть очевидно: сначала мы получаем `seedCount` скрытого параметра и сохраняем его во временной переменной. Затем мы присваиваем `seedCount` явного параметра это значение.

Теперь рассмотрим использование такого метода в программе:

```
Cup5 mycup = new Cup5(0);
Cup5 yourcup = new Cup5(4);
mycup.assignSeedCountToAnotherCup(yourcup);
```

Этот фрагмент кода устанавливает `seedCount` скрытого параметра, `yourcup`, на 0. Обратите внимание, насколько это отличается от использования простых параметров, упоминавшихся в предыдущей главе. Когда явный параметр имеет простой тип, он не меняется в результате вызова. Когда явный параметр является объектной ссылкой, когда она передается, то ее значение просто копируется в формальный параметр. В результате образуются две ссылки на один и тот же объект: одна в вызывающей программе, а вторая — в методе. Любые изменения в формальном параметре влияют на данный объект. Когда вызов метода завершается и выполнение возвращается к вызывающей программе, ссылка все еще существует на тот же самый объект, который только что был изменен.

На рис. 8.1 представлена схема, иллюстрирующая ситуацию в начале вызова метода, после того как явный параметр был скопирован в формальный параметр. Существуют две ссылки, `yourcup` и `anotherCup`, на тот же самый объект, имеющие значение 4.

Следующая схема (рис. 8.2) показывает отношения между фактическим параметром, `yourcup`, и формальным параметром, `anotherCup`, после того как завершен вызов метода. В конце вызова метода формальный параметр не сохраняется. Он показан в виде пунктирной черты. Фактический параметр все еще относится к тому же объекту в вызывающей программе, что и раньше, но содержимое объекта изменилось. Вызов метода не оказывает влияния на скрытый параметр, так что нет необходимости иллюстрировать его статус после вызова.

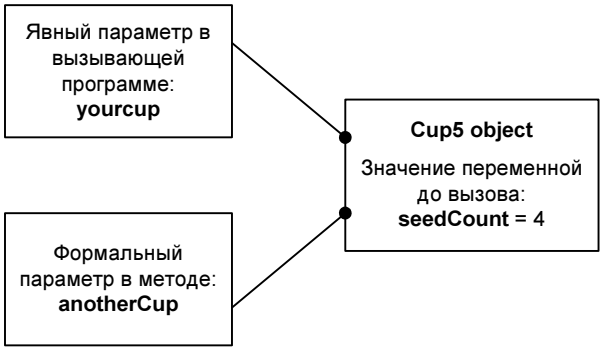


Рис. 8.1

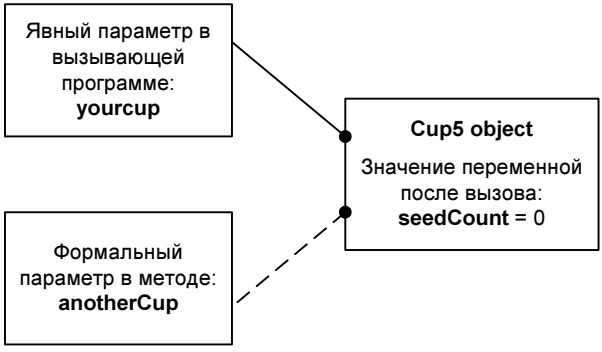


Рис. 8.2

В примере, приведенном выше, именно явный параметр изменяется вызовом метода. Можно также написать метод, вызов которого будет изменять скрытый параметр. Скрытый параметр всегда является ссылкой, и получить доступ к этой ссылке и изменить ее можно кодом внутри метода. Посмотрите на метод, который является противоположным по отношению к предыдущему. Он устанавливает `seedCount` скрытого параметра в соответствии с `seedCount` явного параметра:

```
public void
  setSeedCountFromAnotherCup(Cup5 anotherCup)
{
  int temp = anotherCup.getSeedCount();
```

```
  this.setSeedCount(temp);
}
```

Теперь рассмотрим использование такого метода в программе:

```
Cup5 mycup = new Cup5(0);
Cup5 yourcup = new Cup5(4);
mycup.setSeedCountFromAnotherCup(yourcup);
```

На рис. 8.3 представлена схема, иллюстрирующая отношения между скрытым параметром, `Mycup`, в вызывающей программе, и ссылкой на нее, `This`, в методе.

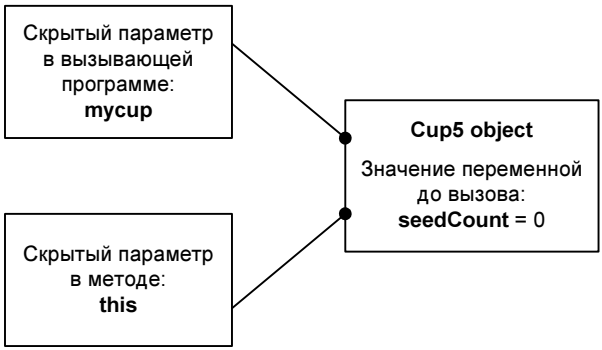


Рис. 8.3

Следующая схема (рис. 8.4) показывает отношения после того, как завершен вызов метода. В конце вызова метода `this` не со-

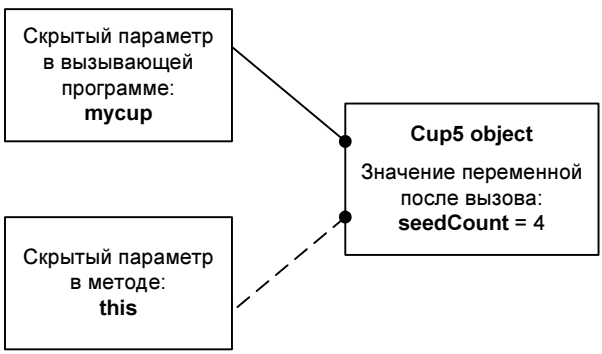


Рис. 8.4

храняется. Он показан в виде пунктирной черты. `mycup` все еще относится к тому же объекту в вызывающей программе, что и раньше, но содержимое объекта изменилось.

8.3. Методы доступа, модифицирующие методы и побочные эффекты

Вызовы методов могут иметь скрытые параметры. Явные параметры могут не потребоваться, т. к. у метода есть скрытые параметры для работы. Пока, не обращая внимания на возможность возврата значения, посмотрите на модель такого вызова:

```
object.method(необязательные параметры);
```

Методы, которые могут быть вызваны к объектам, в целом распадаются на две группы: методы доступа (или `get`) и модифицирующие методы (или `set`). *Методы доступа*, в своей простейшей форме, имеют тип и возвращают значение переменной экземпляра объекта. *Модифицирующие методы*, в своей простейшей форме, являются пустыми, они принимают параметр и изменяют значение переменной экземпляра объекта.

И методы доступа, и модифицирующие методы могут быть более сложными. Методы доступа могут возвращать значения, полученные из арифметических операций, основанных на значениях переменных экземпляра. Модифицирующие методы могут принимать больше одного параметра, и изменение значения переменной экземпляра может основываться на этих параметрах.

Вот пример того, как методы могут быть усложнены:

```
public void exchangeSeedCounts(Cup5 anotherCup)
{
  int thisCount = this.getSeedCount();
  int thatCount = anotherCup.getSeedCount();
  this.setSeedCount(thatCount);
  anotherCup.setSeedCount(thisCount);
}
```

В отношении скрытого параметра это модифицирующий метод. Он принимает параметр, и значение экземплярной переменной скрытого параметра изменяется. Явный параметр является объ-

ектной ссылкой, и он также изменяется вызовом метода. Таким образом, изменение скрытого параметра — не единственный результат. Вторичное или дополнительное влияние на скрытый параметр называется *побочным эффектом*. Если метод имеет побочный эффект, то желательно, чтобы он как-то показывал это, например, в имени метода. Если можно полностью избежать побочных эффектов в методах какого-либо класса, то это говорит о хорошей разработке программы.

В отношении скрытого параметра метод `increaseSeedCount()`, показанный далее, является модифицирующим методом. Если добавленный параметр `addedNumber` не является положительным, то его значение не добавляется к `seedCount`. Неопытный пользователь может подумать, что метод будет добавлять значение без всяких условий. В некотором смысле, условное выполнение добавления является побочным эффектом. За счет возвращения булевого значения метод сигнализирует о том, какое действие было предпринято. Этот оповещение можно также считать побочным эффектом, в данном случае, желательным, но не главным для деятельности метода:

```
public boolean increaseSeedCount(int addedNumber)
{
  if(addedNumber > 0)
  {
    seedCount = seedCount + addedNumber;
    return true;
  }
  else
    return false;
}
```

А вот следующий пример показывает, каким неприятным может быть побочный эффект:

```
public int getSeedCount()
{
  int temp = seedCount;
  seedCount = 0;
  return temp;
}
```

Этот метод понимает значение `get` буквально (в смысле "брать"). Он присваивает `seedCount` значение 0. Однако вряд ли такой исход ожидался или был желательным. В любом случае, метод должен иметь такое имя, чтобы его действия были ясны. Поскольку тот же результат можно получить вызовом обычного метода `getSeedCount()`, за которым следует вызов метода `setSeedCount()`, с параметром равным 0, то в подобном методе, который понимает значение `get` буквально, нет необходимости.

В завершение данного раздела посмотрите на код для класса со всеми рассмотренными ранее изменениями:

```
public class Cup5
{
  private int seedCount;
  public Cup5()
  {
    seedCount = 0;
  }
  public Cup5(int initialCount)
  {
    seedCount = initialCount;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public void setSeedCount(int newCount)
  {
    seedCount = newCount;
  }
  public boolean increaseSeedCount(int addedNumber)
  {
    if(addedNumber > 0)
    {
      seedCount = seedCount + addedNumber;
      return true;
    }
    else
      return false;
  }
```

```
  public void
    assignSeedCountToAnotherCup(Cup5 anotherCup)
  {
    int temp = this.getSeedCount();
    anotherCup.setSeedCount(temp);
  }
  public void
    setSeedCountFromAnotherCup(Cup5 anotherCup)
  {
    setSeedCount(anotherCup.getSeedCount());
  }
  public void exchangeSeedCounts(Cup5 anotherCup)
  {
    int thisCount = this.getSeedCount();
    int thatCount = anotherCup.getSeedCount();
    this.setSeedCount(thatCount);
    anotherCup.setSeedCount(thisCount);
  }
}
```

8.4. Статические методы

Статические методы — это те методы, которые не вызываются для объектов. Другими словами, они не имеют скрытых параметров. Методы класса `Math` могут служить типичным примером, где некоторые полезные вычисления производятся на явных параметрах и возвращается результат. Эти методы принадлежат классу, в котором они содержатся. Они могут быть написаны так, чтобы принимать простые типы или объектные ссылки любого класса в качестве явных параметров. Вызов статических методов имеет следующий вид:

```
variable = Class.method(parameter(s));
```

Вот пример, который мы уже видели:

```
...
double x, y;
...
y = Math.sqrt(x);
...
```

Ключевое слово `static` используется при объявлении таких методов. Вы уже видели примеры такой записи:

```
public class MyProg
{
  public static void main(String[] args)
  {
    ...
```

Метод `main()` в программе — это статический метод. Класс сам по себе является просто контейнером для этого полезного метода. У этого класса нет конструктора. Система просто вызывает непосредственно статический метод, чтобы выполнить программу.

Можно написать свои статические методы, которые принимают явные параметры и выполняют некоторые вычисления или производят какой-либо иной желаемый результат. Вот короткий пример:

```
public class MyMethodContainer
{
  public static double
    findPercent(double first, double second)
  {
    return ((first / second) * 100.0);
  }
}
```

Полагая, что класс `MyMethodContainer` находится в том же каталоге, что и программа, приведем пример небольшой программы, которая использует статический метод:

```
public class TestContainer
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    double x = 15;
    double y = 20;
    double percentResult =
      MyMethodContainer.findPercent(x, y);
```

```
    myterminal.println(x + " is " + percentResult +
                       "% of " + y);
  }
}
```

В результате должно получиться:

```
15 is 75.0% of 20
```

8.5. Статические переменные

Кроме статических методов, язык Java поддерживает *статические переменные*. Это переменные, которые объявляются в классе, но отличаются от переменных экземпляра следующим образом: есть только одна копия переменной, и она хранится в классе; объекты, т. е. экземпляры класса, не получают отдельных копий этой переменной. Математические константы `PI` и `E` в классе `Math` являются переменными `public static final double`. Они общие, статические переменные типа `double`, которые имеют постоянное значение, приданное им.

Статические переменные могут быть также объявлены частными (`private`), как и переменные экземпляра. Если это так, то к ним не будет прямого доступа из-за пределов класса. Однако к ним имеется прямой доступ для каждого объекта класса. Примером такой статической переменной будет минимальное `seedCount` для объектов класса `Cup`. Можно инициализировать статическую переменную с определенным значением при ее объявлении. Существует только одна ее копия, и нет смысла устанавливать ей значение в конструкторе, который вызывается каждый раз, когда создается новый объект класса.

Следующая строка кода иллюстрирует, как в классе `Cup` может быть задано минимальное значение `seedCount`, которое является неизменяемым:

```
private static final int minimumSeedCount = 0;
```

С добавлением статической переменной класс чашек будет изменен на новую версию, `Cup6`. Вот фрагмент кода для этого класса, показывающий, как может быть использована переменная `minimumSeedCount`. Конструктор без параметров устанавливает

значение `seedCount` на минимум. Метод `setSeedCount()` написан так, что он произведет изменение только в том случае, если вводимый параметр будет больше или равен минимальному:

```
public class Cup6
{
  private int seedCount;
  private static final int minimumSeedCount = 0;
  ...
  public Cup6()
  {
    seedCount = minimumSeedCount;
  }
  ...
  public boolean setSeedCount(int newSeedCount)
  {
    if(newSeedCount >= minimumSeedCount)
    {
      seedCount = newSeedCount;
      return true;
    }
    else
      return false;
  }
  ...
}
```

Также можно объявить и инициализировать статическую переменную, которая не является `final`:

```
private static int minimumSeedCount = 0;
```

Затем статический метод может быть добавлен к классу, что позволит изменить минимальное значение:

```
public static void changeMinimum(int newMin)
{
  minimumSeedCount = newMin;
}
```

Обратите внимание на то, что этот метод не изменяет переменную экземпляра для какого-либо объекта. Он изменяет перемен-

ную minimumSeedCount, которую поддерживает класс, для всех объектов данного класса.

Посмотрите на код класса Cup6 с изменениями для поддержки переменной minimumSeedCount, которая может быть изменена в процессе использования класса в программе. Одно из изменений — это изъятие конструктора по умолчанию. Поскольку минимальное количество семян может быть больше 0, то этот конструктор не подойдет для построения экземпляров с исходным значением, равным 0. Обратите также внимание на то, что на практике будет возможно создать объекты класса Cup с определенным количеством семян в них, и уже потом поднять минимальное значение. Не делается никаких попыток для того, чтобы ранее существовавшие чашки соответствовали новому минимуму.

```
public class Cup6
{
  private int seedCount;
  private static int minimumSeedCount = 0;
  public Cup6(int initialCount)
  {
    if(initialCount >= minimumSeedCount)
      seedCount = initialCount;
    else
      seedCount = minimumSeedCount;
  }
  public static void changeMinimum(int newMin)
  {
    minimumSeedCount = newMin;
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public boolean setSeedCount(int newCount)
  {
    if(newCount >= minimumSeedCount)
      {
      seedCount = newCount;
```

```
    return true;
  }
  else
    return false;
}
public boolean addToSeedCount(int addedNumber)
{
  if(this.seedCount + addedNumber
    >= minimumSeedCount)
  {
    seedCount = seedCount + addedNumber;
    return true;
  }
  else
    return false;
}
public boolean
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  if(this.seedCount >= minimumSeedCount)
  {
    int temp = this.getSeedCount();
    anotherCup.setSeedCount(temp);
    return true;
  }
  else
    return false;
}
public boolean
  setSeedCountFromAnotherCup(Cup6 anotherCup)
{
  int temp = anotherCup.getSeedCount();
  if(temp >= minimumSeedCount)
  {
    setSeedCount(anotherCup.getSeedCount());
    return true;
  }
  else
    return false;
}
```

```
  public boolean
    exchangeSeedCounts(Cup6 anotherCup)
  {
    int tempthis = this.getSeedCount();
    int tempthat = anotherCup.getSeedCount();
    if(tempthis >= minimumSeedCount
      && tempthat >= minimumSeedCount)
    {
      this.setSeedCount(tempthat);
      anotherCup.setSeedCount(tempthis);
      return true;
    }
    else
      return false;
  }
}
```

8.6. Полная классификация переменных в Java

Вы уже знакомы с понятием типа переменных. Также возможно описать переменные, принимая во внимание их вид, а именно:

- ❒ локальные переменные;
- ❒ переменные экземпляра;
- ❒ переменные формального параметра;
- ❒ статические переменные.

В языке Java существуют только эти четыре вида переменных, и вы уже знакомы с каждым из них. Любая переменная, которая объявляется внутри скобок, является *локальной* для данных скобок. Этот термин обычно используется в программах в отношении переменных общего назначения. Возможно также объявить такие переменные внутри метода. Тогда они являются локальными для этого метода.

Мы также рассматривали объявление доступности переменных, включая общие (`public`), частные (`private`) и переменные без объявления. Локальные переменные просто объявляются по их

типу, и они доступны только внутри тех скобок, где они определяются. Параметры также не имеют объявления `public` или `private`. Они являются локальными по отношению к методу, которому они принадлежат. Объявления `public` или `private` могут быть применимы к статическим переменным и к переменным экземпляра (так же, как и к методам). По ошибке вы могли объявить переменные экземпляра, не используя ключевое слово `public` или `private`. Хотя такой код скомпилируется, вам следует придерживаться общепринятой практики, согласно которой переменные экземпляра должны быть частными (`private`), и объявлять их таким образом.

В дополнение к вопросу о том, объявлена ли переменная `public` или `private`, для каждой переменной можно ответить на вопросы, когда и где она входит и выходит из активного состояния, где она доступна внутри программы или класса, и есть ли у нее или нет инициализация по умолчанию. Далее даются ответы на эти вопросы для четырех видов переменных.

- *Локальные переменные.*
  - Локальные переменные возникают в том месте кода, где они объявлены. Их существование продолжается во времени и пространстве до конечной скобки блока, в котором они объявлены.
  - Пока этот блок кода выполняется, они существуют и могут использоваться в этом блоке. Как отмечено в *главе 7*, можно иметь больше одной переменной с тем же именем буквально внутри одного и того же блока, потому что блоки сами по себе являются вложенными. Система различает их следующим образом: во внутреннем вложении имя переменной относится к переменной, объявленной здесь. Вне такого блока, как до, так и после него, имя переменной относится к переменной, объявленной вне его.
  - Локальные переменные не имеют инициализации по умолчанию, и программисту следует инициализировать явно их.
- *Переменные экземпляра.*
  - Переменные экземпляра появляются тогда, когда создается объект, которому они принадлежат. Они прекращают свое

существование, когда этот объект больше не существует. С практической точки зрения, они больше не существуют, когда мы утрачиваем ссылку для объекта где-либо в нашей программе.

- Если эти переменные объявлены `private`, как и должно быть, они доступны только внутри класса, в котором они объявлены. Другими словами, где угодно — в конструкторах или методах этого класса — можно использовать переменные экземпляра без использования методов доступа. Когда переменные экземпляра используются без условий, объект, к которому они принадлежат, — это тот объект, который создается, или скрытый параметр кода, в котором они появляются. Вне кода их класса, например в программе, к ним можно получить доступ, только используя какой-либо метод.
- Переменные объекта инициализируются по умолчанию. Числовые переменные инициализируются 0, булевы переменные — `false`, объектные ссылки — `null`. Программист всегда может проигнорировать эти инициализации в конструкторах или написать конструкторы, которые явно производят такую инициализацию, чтобы не было сомнений в том, какие значения имеют переменные экземпляра во время выполнения кода.

❒ *Формальные параметры.*

- Формальные параметры начинают свое существование тогда, когда выполняется вызов метода или конструктора. Они прекращают существование, когда происходит возврат из такого метода или конструктора.
- Они доступны в любом месте внутри блока кода, принадлежащего методу или конструктору. Хотя код метода один и тот же для всех объектов, поскольку переменные прекращают существование в конце вызова, значения в одном вызове невозможно спутать со значениями в следующем вызове.
- Формальные параметры инициализируются теми значениями, которые передаются им в начале вызова от фактических параметров.

- *Статические переменные.*
  - Статические переменные начинают свое существование, когда программа, которая использует или импортирует их класс, начинает выполняться. Они прекращают существование, когда заканчивается программа.
  - Статические переменные могут быть либо общими (`public`), либо частными (`private`), в зависимости от их цели. Если они `public`, они доступны из любого места программы, которая импортировала класс, используя имя с точкой. Другими словами, их можно вызвать, используя `ClassName.VariableName`. Если они объявлены `private`, тогда они доступны из любого места кода класса, будь то конструкторы или методы. К ним можно получить доступ извне класса соответствующим методом.
  - Они имеют ту же инициализацию по умолчанию, как и переменные экземпляра, что может быть заменено программистом.

Теперь, когда мы знаем, что переменные экземпляра имеют инициализацию по умолчанию, мы можем освежить в памяти, какие типы конструкторов возможны. Фактически можно написать конструктор для любого класса, который не имеет никаких параметров и не содержит никакого кода. Например:

```
public MyCup6()
{
}
```

Это самый простой способ написать конструктор, в котором каждая переменная экземпляра будет инициализирована со значением по умолчанию. Такой код легко написать, но недостатком в данном случае является то, что, когда вы смотрите на код, вы не получаете сведений о том, чем являются ваши переменные или каковы значения по умолчанию для различных типов переменных. Это важно помнить.

Также можно написать код для класса вообще без конструкторов. Если это сделать, то система предоставит вам конструктор по умолчанию. Никакого кода вы не увидите, но у вас будет воз-

можность как обычно вызывать конструктор с тем же именем, что и класс, и не давать никаких параметров. Экземпляр будет построен со всеми переменными экземпляра.

Конечно, можно написать конструкторы, которые принимают параметры. Если вы написали код хотя бы для одного конструктора в вашем собственном классе, то система уже не предоставит вам конструктор по умолчанию. Если в таком случае вы захотите воспользоваться конструктором по умолчанию, вам придется самостоятельно его написать.

8.7. Еще о синтаксисе

Данный раздел описывает дополнительный синтаксис, который может быть использован в случае, когда ссылки передаются в качестве параметров. Получаемый в результате код может быть проще, чем рассмотренные выше примеры, но в крайних случаях он может стать непонятным. Примеры в этом разделе относятся к тем, что даны в *разд. 8.2*. Кроме нового синтаксиса, они изменены с тем, чтобы описывать последнюю версию класса чашек, `Cup6`.

В коде метода в *разд. 8.2* ключевое слово `this` было использовано для скрытого параметра. По правде говоря, это слово не является обязательным. Следующий вариант метода выполняет то же самое, как и первый пример в *разд. 8.2*. Обратите внимание, что этот метод содержит вызов метода `getSeedCount()`, но в вызове не показан никакой объект, к которому делается обращение. В подобной ситуации вызов к методу `getSeedCount()` делается на скрытый параметр, `this`, метода, в котором он содержится, `assignSeedCountToAnotherCup()`.

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  int temp = getSeedCount();
  anotherCup.setSeedCount(temp);
}
```

Кроме того, можно избавиться от временной переменной, сжав код до одной строки:

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  anotherCup.setSeedCount(getSeedCount());
}
```

Есть еще несколько синтаксических моментов, которые можно проиллюстрировать, когда параметр является объектной ссылкой. Посмотрите на следующую новую версию кода:

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  anotherCup.setSeedCount(this.seedCount);
}
```

Значение `seedCount` скрытого параметра получается напрямую, без использования метода доступа. В вызывающей программе такое было бы неправильным кодом. Но здесь это допустимо. Поскольку мы находимся в методе, `assignSeedCountToAnotherClass()`, который расположен внутри класса `Cup6`, объектная ссылка этого класса, такая как `this`, имеет прямой доступ к собственным переменным экземпляра. Кстати, метод также можно переписать следующим образом:

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  anotherCup.seedCount = this.seedCount;
}
```

Поскольку `anotherCup` также является ссылкой на объект класса `Cup6`, он имеет прямой доступ к его переменным экземпляра, точно так же, как и `this`.

Употребление `this` все еще не обязательно. Неуточненное имя переменной экземпляра рассматривается как переменная экземпляра, принадлежащая скрытому параметру. Код можно написать так:

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
```

```
{
  anotherCup.setSeedCount(seedCount);
}
```

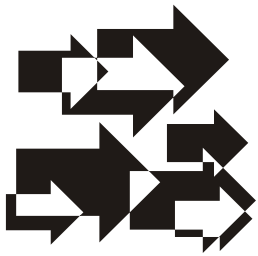
Можно написать код иначе, где вы будете полагаться на прямой доступ к переменным экземпляра как скрытых, так и явных параметров:

```
public void
  assignSeedCountToAnotherCup(Cup6 anotherCup)
{
  anotherCup.seedCount = seedCount;
}
```

Не рекомендуются более сложные версии кода. Всегда можно воспользоваться ключевым словом `this` и методами `set` и `get`. За программистом остается выбор синтаксиса, когда вы имеете дело с параметрами, которые являются объектными ссылками.

В завершение, напомним еще раз о том, что подчеркивалось на протяжении всех предыдущих глав. Переменные экземпляра должны быть объявлены частными (`private`) в определении классов. Когда это сделано, они не доступны извне класса. Ничто из описанного в данном разделе синтаксиса не может быть использовано для получения доступа к переменным экземпляра объекта в вызывающей программе. Этот синтаксис работает только тогда, когда вы пишете код для методов класса, и этот код использует ссылки на объекты данного класса.

ГЛАВА 9



Массивы, класс Vector и класс ArrayLists

9.1. Объявление и использование массивов

Массивы — это упорядоченные наборы однотипных данных. Используя подстрочные символы, массив a можно записать в следующем виде:

$(a_0, a_1, ..., a_i, ..., a_{n-1})$.

Каждый элемент массива имеет значение определенного типа, например `double`. Элементы массива определяются порядком, в котором они появляются в массиве. В программировании принято использовать индекс 0 для первого элемента массива. В приведенном далее примере массив содержит переменные типа `double`, при этом 13.2 занимает нулевую позицию, 5.5 занимает первую позицию и т. д.:

(13.2, 5.5, ..., 2.8, ..., 6.7).

В языке Java массивы являются объектами. Посмотрите на две строки кода, объявление и конструктор, для массива, в котором имеется набор элементов типа `double`.

```
double[] myarray;
myarray = new double[10];
```

Объявление `double` означает, что элементы массива будут числовыми значениями типа `double`. Квадратные скобки служат при-

знаком того, что это массив. Ссылка на массив получает имя myarray. Синтаксис для построения массива слегка отличается от синтаксиса для других объектов. Здесь все еще используется ключевое слово new, за которым следует имя типа. Затем указывается количество элементов в массиве, в квадратных скобках.

Как обычно, объявление и конструктор можно соединить в одной строке:

```
double[] myarray = new double[10];
```

Как только массив создан, доступ к отдельным элементам массива осуществляется за счет использования индекса значения в квадратных скобках. Переменная с индексом может быть использована, как и любая другая переменная этого типа. Например, так:

```
myarray[0] = 25.2;
double myvar = myarray[0];
myterminal.println(myarray[0]);
```

Если массиву назначен размер 10, то будут использованы индексы от 0 до 9. Распространенной ошибкой является предположение, что индексация начинается с 1. Игнорирование нулевого элемента допустимо. Однако грубой ошибкой будет пытаться обратиться к myarray[10], если такой элемент не существует.

Можно объявить массивы, содержащие простые типы, также как и массивы, содержащие объектные ссылки. Возможны такие массивы: int[], double[], boolean[], String[].

Несмотря на то, что, как правило, локальным переменным не присваивается инициализация по умолчанию, независимо от того, где они были объявлены или как они применяются, массивы обладают инициализацией по умолчанию. Числовые переменные инициализируются нулем по умолчанию, булевы — false, а ссылки объекта — null.

Рассмотрим следующую строку кода, которая присутствует в любой программе:

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

Теперь становится понятным, что означает такой синтаксис. Метод main() принимает массив String в качестве параметра. Если

команда выполняется из командной строки, то ввод можно осуществить в той же строке, после пробела. Такие элементы ввода известны как аргументы командной строки, и они сохраняются в выполняющейся программе в массивах `String`, получающих имя `args`.

Массив может быть объявлен для того, чтобы содержать элементы любого типа объекта. Используя класс из предыдущей главы, можно сделать такое объявление:

```
Cup6[] myCupArray = new Cup6[10];
```

Обратите внимание, что, как обычно, имя класса пишется с прописной (заглавной) буквы. Имя массива — это ссылка на построенный объект, и оно пишется со строчной (маленькой) буквы. Если элементы массива являются объектами, то можно использовать эти объекты, как и все остальные. Мы можем вызвать методы для них. Синтаксис в таком случае будет выглядеть необычно, но это абсолютно допустимо. Например, предположим, что массив `myCupArray` имеет ссылку на чашку, сохраненную в третьей позиции. Можно получить количество семян в этой чашке, используя метод `getSeedCount()`. Пусть вас не беспокоит тот факт, что точка идет сразу за закрывающей скобкой, указывая положение массива:

```
int mycount = myCupArray[3].getSeedCount();
```

Следующий пример иллюстрирует пользу циклов при работе с массивами. Так как элементы массива могут быть определены последовательностью целых чисел, то легко использовать индекс цикла как индекс массива:

```
for(int j = 0; j < 10; j++)
{
  myarray[j] = j * j;
}
```

Обратите внимание, что начальное значение `j` равно 0, это первый элемент массива, а последнее значение `j` равно 9. Условие остановки можно было также обозначить как `j <= 9`, но есть своя логика в использовании точных неравенств и размера массива.

Так как часто необходимо использовать размер массива во время работы с ним, язык Java обеспечен средствами по выявлению

размеров. Массивы являются объектами, а класс выполняется с переменной `length`, которая является `public` и `final`. Поскольку переменная является `public`, возможен прямой доступ к ней без вызова метода. Вот пример ее использования:

```
for(int j = 0; j < myarray.length; j++)
{
  myarray[j] = j * j;
}
```

Чтобы получить длину массива, вам необходима переменная `length`. Вспомните, что, для того чтобы получить длину `String`, вам необходим метод `length()`. Имейте в виду, что ни в том, ни в другом случае вам не нужен метод `getLength()`. Не совсем понятно, почему разработчики языка приняли такие правила. Удобно иметь в своем распоряжении длину как массивов, так и `String`, но важно помнить, как правильно получить эти данные.

Вы можете проходить элементы массива в обоих направлениях. Во время выполнения цикла от самых больших к самым маленьким индексам значения индексов для начала и остановки программы должны быть правильно установлены. Например:

```
for(int j = myarray.length – 1; j >= 0; j--)
{
  ...
}
```

Важно отметить, что, как и в случае с другими элементами данных, массив может быть объявлен только один раз. В этом объявлении устанавливается размер массива. Размер не может быть изменен позже во время работы программы. Это вызывает необходимость ожидать наибольшее количество элементов, которые могут когда-либо потребоваться, и объявлять соответствующий размер массива. Затем, во время работы, в большинстве случаев значительное количество элементов массива будет оставаться неиспользованным.

Массивы более чем одной размерности также могут быть объявлены. В качестве примера рассмотрим двумерные массивы. Так же, как и одномерные массивы, двумерные массивы существуют в математике. Ниже показан двумерный массив, или матрица, использующий математические символы. Индексация обоих раз-

меров начинается с 0. Матрица имеет размеры $m \times n$. Это означает, что индексация рядов ведется до $m-1$, а индексация колонок — до $n-1$.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,j} & \dots & a_{0,n} \\ a_{1,0} & a_{1,1} & \dots & a_{1,j} & \dots & a_{1,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i,0} & a_{i,1} & \dots & a_{i,j} & \dots & a_{i,n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{m,0} & a_{m,1} & \dots & a_{m,j} & \dots & a_{m,n} \end{pmatrix}.$$

Первый нижний индекс элемента представляет его ряд, а второй — столбец. В матрице имеется *m* рядов и *n* столбцов с произвольным элементом, определенным как a_{ij}. Объявление такого массива в Java, если он содержит элементы в виде целых чисел и имеет 4 ряда и 5 столбцов, примет такой вид:

```
int[][] myintarray = new int[4][5];
```

Объявление таких массивов частично аналогично объявлению одномерного массива. Различие состоит лишь в том, что используются два набора квадратных скобок, первый определяет ряды, а второй — столбцы. Доступ к отдельным элементам осуществляется за счет указания в квадратных скобках индекса для ряда и столбца. Например:

```
myintarray[2][4] = 95;
```

С двумерными массивами удобно работать с помощью вложенных циклов `for`. Так же, как и с одномерными массивами, индексация массива начинается с 0, и длина размеров может быть использована для установки пределов на индексы циклов. Это приводит к некоторым затруднениям. Рассмотрим пример:

```
for(int i = 0; i < myintarray.length; i++)
{
  for(int j = 0; j < myintarray[0].length; j++)
  {
    ...
  }
}
```

Здесь:

- `myintarray.length` — это количество рядов в `myintarray`;
- `myintarray[0].length` — это количество столбцов в нулевом ряду (или в любом ряду) `myintarray`.

Двумерный массив является, по сути, одномерным массивом ссылок к набору одномерных массивов. Значение длины, возвращаемое для названия самого массива, является числом рядов или числом одномерных массивов. Если имя двумерного массива дается с одним набором скобок, это представляет один ряд. Длина ряда является количеством столбцов в двумерном массиве. Могут быть многомерные массивы. Синтаксически, применение квадратных скобок для указания на размеры аналогично их использованию в двумерном массиве. Если речь идет о трехмерном массиве, то это массив из массива массивов, и то же самое относится и к любым многомерным массивам, которые будут в этом смысле аналогичны двумерному массиву.

9.2. Копирование массивов, передача массивов в качестве параметров и возвращение массивов методами

Копирование является обычной операцией с массивами. Она не сложна, но может быть источником путаницы. Массивы являются объектами. Это означает, что вам следует различать копирование ссылок и копирование содержания. Вот пример копирования ссылки:

```
double[] myarray = new double[10];
double[] yourarray;
yourarray = myarray;
```

В приведенном коде, `myarray` был создан; `yourarray` является просто ссылкой. Присваивание в конце является верным. В результате, будут существовать две ссылки к одному и тому же массиву. Если вы затем выполните следующее:

```
yourarray[0] = 3.0;
myterminal.println(myarray[0]);
```

то вы увидите на экране значение 3.0. Нет никакого существенного различия между `myarray[0]` и `yourarray[0]`. Они отсылают к одному и тому же сохраненному значению.

Конечно, также можно делать копии содержимого массивов. Это может быть выполнено следующим образом:

```
double[] myarray = new double[10];
double[] yourarray = new double[myarray.length];
for(j = 0; j < myarray.length; j++)
{
  yourarray[j] = myarray[j];
}
```

Копирование содержимого массива является настолько распространенной процедурой, что в Java имеется специальный статический метод для этого. Чтобы получить доступ к этому методу, вам необходимо следующее выражение:

```
import java.lang.System;
```

В документации Java API для класса `System` вы найдете описание этого метода:

```
static void arraycopy(Object src, int src_position,
                      Object dst, int dst_position,
                      int length)
```

Параметры объявленные как `Objects` — это массивы, *`src`* и *`dst`* обозначают источник (source) и приемник (destination) соответственно. Метод предназначен для того, чтобы выполнить частичное копирование. Вы определяете начальную позицию как в исходном, так и в принимающем массиве, плюс длину, или число элементов для копирования. Вам необходимо убедиться в том, что начальная позиция в сумме с длиной не превышает размер любого из двух массивов. Вот пример использования этого метода:

```
System.arraycopy(myarray, 0, yourarray, 0, myarray.length);
```

Метод, имеющийся в системе, принимает массивы в качестве явных параметров. Программисты также могут написать методы, где массивы передаются как параметры. При этом важно пом-

нить, что ссылка передается. Это означает, что любые изменения, сделанные в массиве в методе, отражаются в вызове программы. В примере выше массивы ожидаются как параметры, но формальные параметры метода объявляются как `Object`. Это позволяет массиву содержать любой тип элемента для передачи. Для программистов желательно объявлять параметр для определенного вида массива. Приведем использование простого статического метода:

```
public class MyMethodContainer1
{
  public static void squareArray(double[] anArray)
  {
    for(int j = 0; j < anArray.length; j++)
    anArray[j] = anArray[j] * anArray[j];
  }
}
```

Затем в вызывающей программе, где `myarray` был объявлен и создан, возможен следующий вызов:

```
MyMethodContainer1.squareArray(myarray);
```

Обратите внимание, что здесь не используются квадратные скобки. `myarray` — это одномерный массив, и имя само ссылается на массив.

Метод также может возвращать массив. Возвращенный массив создается с нуля внутри метода. Такой массив может быть полностью независим от любого другого массива. Вероятно, вы захотите написать метод, который принимает массив в качестве параметра и работает с его содержимым без изменения оригинала. В этом случае можно передать массив в качестве параметра, создать новый массив в методе, скопировать содержание параметра в новый массив, обработать новый массив и возвратить новый массив в качестве результата метода, оставив при этом оригинал без изменений. Сравните следующий статический метод с предыдущим:

```
public class MyMethodContainer2
{
  public static double[]
    squareArray(double[] anArray)
```

```
  {
    double[] returnArray =
      new double[anArray.length];
    for(int j = 0; j < returnArray.length; j++)
      returnArray[j] = anArray[j] * anArray[j];
    return returnArray;
  }
}
```

В этом случае вызов в программе примет такой вид:

```
double[] yourarray =
  MyMethodContainer2.squareArray(myarray);
```

Нет никакой необходимости отдельно создавать и указывать размер для `yourarray`. Это просто ссылка, которая получает созданный и имеющий размер массив элементов типа `double`, возвращенный методом.

Двумерные массивы также могут передаваться как параметры. В таком случае, весь массив передается в виде имени без квадратных скобок. Как было отмечено в предыдущем разделе, включение скобок с названиями многомерных массивов является способом обозначения отдельных рядов. Метод может выполнять алгоритм, где это требуется. Однако необходима осторожность, чтобы не включить по ошибке ненужные скобки во время объявления формальных параметров или в написании кода для вызова.

9.3. Примеры массивов

Рассмотрим пример, как если бы это было задачей на использование массивов.

Напишите класс, содержащий статический метод для нахождения среднего значения набора переменных типа, имеющихся в некотором множестве. Затем напишите программу, тестирующую данный метод. Вот математическая формула для нахождения среднего значения:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}.$$

Краткое описание метода для нахождения среднего значения:

1. Метод должен возвращать значение типа `double` и, кроме того, должен принимать в качестве параметров массив переменных типа `double` и одну переменную типа `integer` со значением того, сколько единиц информации сохранено в массиве. (Обратите внимание, что не годится использовать длину массива, поскольку это будет означать, что массив всегда совершенно заполнен, когда вы вызываете метод.)
2. В методе объявите одну переменную `double` для хранения среднего значения и другую переменную `double` для работы с суммой.
3. Теперь прогоните цикл через счетчик, суммируя значения в массиве.
4. Найдите среднее значение путем деления суммы на счетчик.
5. Верните среднее значение.

Краткое описание тестовой программы:

1. Начните, как обычно, с метода `main()`.
2. Постройте `myterminal`.
3. Объявите переменную `String` и инициализируйте ее на `yes`.
4. Объявите целочисленную переменную для счетчика и инициализируйте ее на 0.
5. Объявите переменную типа для хранения среднего значения.
6. Выполните цикл `while` до тех пор, пока ответом не будет `no`.
7. В цикл включите подсказку для пользователя о значении.
8. Возьмите `double` из `myterminal` и поместите ее в массив, используя текущее показание счетчика в качестве индекса.
9. Инкрементируйте счетчик.
10. Спросите пользователя, хочет ли он проделать все снова.
11. Возьмите ответ `String` из `myterminal`. Это конец цикла.
12. Затем вызовите метод для нахождения среднего значения.
13. Выведите среднее значение на экран.

Посмотрите на классы для этого решения. Они используют массивы, объясненные ранее в этой главе.

```
public class MeanClass
{
  public static double
    myMean(double[] inArray, int count)
  {
    double mean;
    double sum = 0.0;
    int i;
    for(i = 0; i < count; i++)
    {
      sum = sum + inArray[i];
    }
    mean = sum / count;
    return mean;
  }
}
public class MeanTestProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    double[] dataArray = new double[100];
    String answer = "yes";
    int dataCount = 0;
    double dataMean;
    myterminal.println("This program will handle a data set
                        of up to 100 values.");
    while(!answer.equals("no") && dataCount < 100)
    {
      myterminal.println("Enter a value.");
      dataArray[dataCount] =
        myterminal.getDouble();
      dataCount++;
      myterminal.println("Go again?  (yes/no)");
      answer = myterminal.getString();
    }
```

```
    dataMean =
      MeanClass.myMean(dataArray, dataCount);
    myterminal.println("The mean of the data set is:  " +
                       dataMean);
  }
}
```

Массивы могут быть использованы как переменные экземпляра. В синтаксисе для такой ситуации нет ничего нового. Посмотрите на пример далее, который иллюстрирует эту идею. Этот пример продолжает наше описание чашки с семенами. Здесь вводится отдельный класс семян (`Seed`). Класс чашек имеет переменную экземпляра, являющуюся массивом, которая содержит экземпляры класса `Seed`. Также приводится короткая программа, которая тестирует методы классов `Cup` и `Seed`.

Например, посмотрим на класс `Seed`. Хотя этот класс вводится впервые, его версии дан номер 7, так чтобы его имя соответствовало имени класса `Cup`. Класс `Seed` содержит только одну переменную экземпляра. Этот пример не подходит для графического изображения, так что цвет семян записан просто в виде строки, содержащей название цвета.

```
public class Seed7
{
  private String seedColorName;
  public Seed7(String seedColorNameIn)
  {
    seedColorName = seedColorNameIn;
  }
  public String getSeedColorName()
  {
    return seedColorName;
  }
}
```

Теперь посмотрим на класс `Cup`. Он имеет номер 7 в нашем ряду различных примеров с этим классом. Главное в данном случае — это массив `seedArray` в этом классе. Этот массив содержит ссылки на семена. Обратите внимание, что когда вы создаете чашку, необходимо указать максимальное количество семян, которое в ней

может содержаться. Этот вводный параметр становится длинной массива. При этом сохраняется `seedCount`, и эта переменная используется для того, чтобы убедиться, что пользователь не сможет ввести большее количество семян, чем может поместиться в чашку.

```
public class Cup7
{
  private int seedCount;
  private int maxSeedCount;
  private Seed7[] seedArray;
  public Cup7(int maxSeedCountIn)
  {
    seedCount = 0;
    maxSeedCount = maxSeedCountIn;
    seedArray = new Seed7[maxSeedCount];
  }
  public int getSeedCount()
  {
    return seedCount;
  }
  public boolean addSeed(Seed7 seedIn)
  {
    if(seedCount < maxSeedCount)
    {
      seedArray[seedCount] = seedIn;
      seedCount++;
      return true;
    }
    else
      return false;
  }
  public Seed7 removeSeed()
  {
    if(seedCount > 0)
    {
      seedCount--;
      Seed7 temp = seedArray[seedCount];
      seedArray[seedCount] = null;
```

```
      return temp;
    }
    else
      return null;
  }
}
```

Вот короткая программа, которая использует классы `Cup` и `Seed`. Она не делает ничего необычного. Она просто дает возможность повторить циклы `while` и `for`.

```
public class TestCup7AndSeed7
{
  public static void main(String[] args)
  {
    Cup7 myCup = new Cup7(5);
    Seed7 aSeed;
    int howManySeeds;
    boolean okToEnter = true;
    MyTerminalIO myterminal = new MyTerminalIO();
    while(okToEnter)
    {
      aSeed = new Seed7("blue");
      okToEnter = myCup.addSeed(aSeed);
    }
    howManySeeds = myCup.getSeedCount();
    for(int i = 0; i < howManySeeds; i++)
    {
      aSeed = myCup.removeSeed();
      myterminal.println(aSeed.getSeedColorName());
    }
  }
}
```

Должно быть очевидным, что представление о том, что в чашке находятся семена, более точно отражено в данном коде, чем ранее. Когда выполняется тестирующая программа и создается объект класса `Cup`, этот объект имеет экземпляры семян в нем, находящиеся в его массиве переменных экземпляра. Посмотрите на простую схему (рис. 9.1), иллюстрирующую эти отношения. Пояснения к схеме даются позднее.

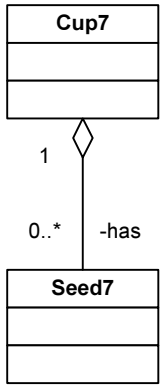


Рис. 9.1

9.4. Структура коллекций в языке Java

Структура коллекций в языке Java предоставляет собой последовательный набор стандартов для различных типов классов, экземпляры которых служат в качестве контейнеров для объектов. Структура коллекций имеет классы с особыми полезными характеристиками для работы с объектными ссылками. Некоторые из таких классов индексируют свое содержимое, как, например, массивы. Другие управляют содержимым без помощи индексов. Использование таких разработок при написании программ экономит время и дает возможность написать более эффективные программы. Составители Java API создали разработки, которые носят общий характер и имеют понятные рабочие характеристики. В документации Java API, если вы откроете раздел на классе `Collections`, то найдете ссылку на страницу с описанием структуры коллекций (***Java Collections Framework***), на которой будет раздел ***Overview*** (Общее описание). На главной странице Java документации компании Sun вы найдете ссылку на более подробные инструкции по структуре коллекций. Оба эти документа являются важными источниками информации, и эта информация во многом используется в настоящем разделе.

Цель данного и следующих разделов — дать представление о том, что такое структура коллекций, так чтобы программист имел

некоторую основу для понимания того, как они используются. Классы коллекций связаны друг с другом наследованием и интерфейсами. Ранее уже упоминалась идея наследования, а вот интерфейсы еще пока не обсуждались. Кратко говоря, и то и другое является способом иерархической категоризации классов. Эти категории связаны с чертами классов. Более широкие категории классов находятся выше в иерархии. Все нижестоящие классы имеют черты вышестоящей категории. В следующей главе наследование и интерфейсы будут объяснены более подробно. Пока же, общего представления об иерархической категоризации достаточно для понимания организации классов коллекций и их отношений друг с другом.

Подробные инструкции описывают две иерархии интерфейсов, как основу для всей структуры коллекций. Эти интерфейсы описывают характеристики, или черты, без указания на то, как может быть написан код, который их реализует. Например, термин "множество" (set) определен в математике, и множество имеет четко определенные свойства. Как реализовать эти свойства в коде, не имеет ничего общего с определением множества. Первая из таких иерархий происходит из интерфейса `Collections` (коллекции). Коллекции подразделяются на множества, списки и очереди, при этом множества могут быть расположены в некотором порядке. Вторая иерархия, от которой зависит структура коллекций, происходит из интерфейса `Map` (карта отображений). `Map` указывает отношения между одним набором значений и другим, и также может классифицироваться.

Общее описание структуры коллекций в отношении интерфейсов и их реализации дано в табл. 9.1. По вертикали указаны разные интерфейсы: множества, списки и отображения. По горизонтали даются основные виды реализации: хэш-таблица, изменяемые множества и т. д. Элементы таблицы являются классами в документации Java API. Не всякий интерфейс имеет каждый вид реализации, и наоборот, не любой вид реализации воплощает каждый интерфейс. С другой стороны, имеется, по крайней мере, один элемент для каждого ряда и каждого столбца. Различные классы в каждом ряду таблицы связаны между собой интерфейсом. Если вас интересует вопрос наследования иерархии для

различных классов коллекций, то они происходят из классов `AbstractCollection` и `AbtsractMap`. Не обязательно знать всю иерархию наследования для того, чтобы использовать эти классы. Однако важно импортировать эти классы в определенном порядке.

Таблица 9.1. *Структура коллекций интерфейсов*

Реализации / Интерфейсы	Хэш-таблица	Изменяемый массив	B-дерево	Связанный список	Хэш-таблица и связанный список
Множества	`HashSet`		`TreeSet`		`LinkedHashSet`
Списки		`ArrayList`		`LinkedList`	
Отображения	`HashMap`		`TreeMap`		`LinkedHashMap`

Эти иерархии классов чрезвычайно функциональны. Цель данного раздела — дать общее описание того, что имеется, и предоставить информацию, которая послужит отправной точкой для чтения документации Java API. Методы для добавления, удаления и изменения элементов коллекций представлены в этой документации с полными объяснениями. Каждый вид коллекции имеет свой порядок в отношении работы с элементами — добавляются ли элементы в начале или в конце, сортируются ли они по значениям, и т. д. Здесь мы не будем обсуждать эти детали. Заинтересованные читатели могут самостоятельно изучить документацию и решить, какой из классов может потребоваться для решения конкретной задачи.

9.5. Синтаксис коллекций

Данный раздел представляет синтаксис, связанный с коллекциями. Это делается с помощью класса `ArrayList` в качестве примера. Для того чтобы быть использованной в программе, коллекция должна быть импортирована. Полное описание пути для других классов коллекций можно найти в документации Java API.

Вот как будет выглядеть импорт для класса ArrayList:

```
import java.util.ArrayList;
```

В программе, которая использует класс коллекций, угловые скобки могут быть приведены в объявлениях для указания на то, какие элементы содержит коллекция. Например, следующая строка кода в программе объявит и создаст экземпляр класса ArrayList с именем mycollection, который содержит элементы, являющиеся ссылками на экземпляры класса Cup7:

```
ArrayList<Cup7> mycollection = new ArrayList<Cup7>();
```

Если желательно, чтобы ArrayList содержал простые типы, такие как int или double, значения должны находиться в экземплярах классов-оболочек (wrapper classes), Integer и Double, а ссылки на эти объекты могут быть элементами ArrayList. Например, следующая строка кода в программе объявляет ArrayList с именем yourcollection, который содержит элементы, являющиеся ссылками класса Double:

```
ArrayList<Double> yourcollection = new ArrayList<Double>();
```

Чтобы перебрать все элементы массива, обычно используют цикл, где индекс цикла проходит через значения индексов массива. Перебор всех элементов класса коллекций может быть проделан с помощью цикла for each. Вот пример, иллюстрирующий синтаксис для такого цикла. Обратите внимание, что в действительности слово each не появляется в нем. Это имя дается для того, чтобы отличить данный цикл от простого цикла for:

```
for(Cup7 someCup: mycollection)
{
  myterminal.println(someCup);
}
```

В операторе for объявляется локальная ссылка с именем someCup. Оператор for также содержит двоеточие, за ним следует имя коллекции, к которой необходим доступ, в данном примере она называется mycollection. Тип локальной ссылки согласуется с типом элементов коллекции. Цикл проходит через все элементы mycollection, успешно присваивая ссылки своих элементов локальной ссылке, someCup. Такой синтаксис работает для любого

класса структуры коллекций. `mycollection` может быть экземпляром любой из реализаций, перечисленных в табл. 9.1: `HashSet`, `TreeSet`, `LinkedHashSet` и т. д. Обратите внимание на то, что цикл обеспечивает доступ к последовательным элементам коллекции, но невозможно вызвать методы к самой коллекции. Это означает, что невозможно добавить элементы к коллекции или уменьшить количество элементов коллекции, используя данный синтаксис.

Прохождение элементов коллекции называется *итерацией*. Структура коллекции в языке Java также включает интерфейсы, называемые `Iterator` и `Iterable`. Они поддерживают прохождение коллекции там, где возможно изменить коллекцию во время обработки. Классы структуры коллекций реализуют такой вид итерации. Интерфейс `Iterable` содержит один метод с именем `iterator()`. Вызов этого метода к коллекции возвращает ссылку к итератору. При получении итератора, для того чтобы ваш код был последовательным, вам нужно использовать угловые скобки для указания на то, с какими видами элементов итератор имеет дело. Если `mycollection` содержит элементы `Cup7`, тогда объявление итератора для данной коллекции примет такой вид:

```
Iterator<Cup7> myiterator = mycollection.iterator();
```

Три разных метода могут быть вызваны к итератору. Вот они:

```
boolean hasNext()
```

Возвращает `true`, если в итерации имеется больше элементов.

```
E next()
```

Возвращает следующий элемент (типа `E`) в итерации. Многократный вызов этого метода до тех пор, пока `hasNext()` не вернет `false`, возвратит каждый элемент базовой коллекции точно один раз.

```
void remove()
```

Удаляет из базовой коллекции последний элемент, возвращенный итератором. Этот метод может быть вызван только один раз на вызов к `next`.

Эти три метода позволяют произвести итерацию по всей коллекции, проверяя, есть ли "следующий" элемент (next), получая этот

элемент, если он есть, и удаляя его при желании. Посмотрим на пример цикла `while`, основанного на коллекции `mycollection`, содержащей ссылки на класс `Cup7`. Итератор для коллекции достигает цикла, и тестом на продолжение цикла является вызов метода `hasNext()`. В теле цикла вызов `next()` возвращает ссылку на следующий элемент, а вызов `remove()` производит удаление такого элемента из коллекции. Данный пример убирает из `mycollection` все элементы, которые содержат 0 или менее семян:

```
import java.util.Iterator;
...
Iterator<Cup7> myiterator = mycollection.iterator();
Cup7 someCup;
while(myiterator.hasNext())
{
  someCup = myiterator.next();
  if(someCup.getSeedCount() <= 0)
  myiterator.remove();
}
```

Следующий пример делает то же самое, но он более компактен за счет использования цикла `for`. Первая часть оператора `for` получает итератор для коллекции. Вторая часть оператора `for` вызывает продолжение итерации до тех пор, пока находится следующий элемент в коллекции. Обратите внимание: цикл структурирован так, что он содержит три отдельных раздела, разделенных точкой с запятой. Тело цикла также более компактно, поскольку производится удаление элемента без использования временной ссылки:

```
import java.util.Iterator;
...
for(Iterator<Cup7> myiterator =
  mycollection.iterator(); myiterator.hasNext(); )
{
  if((myiterator.next().getSeedCount()) <= 0)
    myiterator.remove();
}
```

Этот небольшой пример иллюстрирует общее решение данной задачи с использованием класса коллекций. Содержание опера-

тора `for` было бы таким же, если бы коллекция `mycollection` содержала экземпляры класса `Double`, а не класса `Cup7`. Единственное различие было бы в теле цикла. Вызов метода `next()` возвращает ссылку на объект класса `Cup7`. Метод `getSeedCount()` принадлежит классу `Cup7`. Он может быть вызван по этой ссылке, но он не может быть вызван по ссылке к `Double`, если бы это было содержанием `mycollection`.

9.6. Примеры коллекций

Далее приводится решение задачи по нахождению среднего значения множества данных с использованием `ArrayList` вместо `Array`. Этот пример иллюстрирует четыре общие положения:

- нет предела количеству элементов в `ArrayList`, поэтому нет предела размеров множества;
- можно вести учет количества элементов в методе `myMean()`, а не в `main()` программы;
- программа иллюстрирует использование метода `add()` для добавления элементов к коллекции;
- Java обладает свойством, называемым *автоупаковкой* (autoboxing). В программе нет необходимости создавать экземпляры класса `Double` и присваивать значения типа `double` его переменным экземпляра. В тех случаях, когда необходима ссылка на экземпляр класса `Double`, например, в элементе `ArrayList`, можно предоставить простое значение `double`. Система создаст необходимый объект и присвоит значение автоматически.

Вот код решения:

```
import java.util.ArrayList;
public class CollectionMeanClass
{
  public static double
    myMean(ArrayList<Double> dataSetIn)
  {
    double mean;
    double sum = 0.0;
```

```
    int dataCount = 0;
    for(Double someDouble:  dataSetIn)
    {
      dataCount++;
      /*  Здесь происходит автоупаковка.  */
      sum = sum + someDouble;
    }
    mean = sum / dataCount;
    return mean;
  }
}
import java.util.ArrayList;
public class CollectionMeanTestProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    ArrayList<Double> dataSet =
      new ArrayList<Double>();
    String answer = "yes";
    double dataMean;
    while(!answer.equals("no"))
    {
      myterminal.println("Enter a value.");
      dataSet.add(myterminal.getDouble());
      myterminal.println("Go again?  (yes/no)");
      answer = myterminal.getString();
    }
    dataMean = CollectionMeanClass.myMean(dataSet);
    myterminal.println("The mean of the data set is:  "
                       + dataMean);
  }
}
```

Следующий пример показывает использование класса `ArrayList` при реализации класса `Cup`. Вместо того чтобы включать только `seedCount`, объект класса может содержать произвольное количество семян, где семена — это новый класс. Классы чашек и семян остаются абстракциями, но можно написать апплет, который придаст экземплярам этих классов графическое представление.

Далее показаны коды для классов `Seed8` и `Cup8` и апплет `TestCup8AndSeed8`. После кода даются необходимые пояснения к примеру:

Класс `Seed8`:

```
import java.awt.Color;
public class Seed8
{
  private Color seedColor;
  public Seed8(Color aColor)
  {
    seedColor = aColor;
  }
  public Color getColor()
  {
    return seedColor;
  }
}
```

Класс `Cup8`:

```
import java.util.ArrayList;
public class Cup8
{
  private ArrayList<Seed8> cupContents;
  public Cup8()
  {
    cupContents = new ArrayList<Seed8>();
  }
  /*  Класс Cup8 не имеет переменной экземпляра seedCount.
      Он использует метод size() класса ArrayList для
      определения количества элементов в нем.  */
  public int getSeedCount()
  {
    return cupContents.size();
  }
  /*  Этот метод возвращает ссылку на ArrayList,
      содержащую семена в чашке. Метод иллюстрирует
      синтаксис. Потом узнаем, почему такой метод
      доступа нежелателен.  */
```

```
  public ArrayList<Seed8> getContents()
  {
    return cupContents;
  }
  public void addSeed(Seed8 aseed)
  {
    cupContents.add(aseed);
  }
  /*  Обратите внимание, что метод remove() класса ArrayList
      отличается от метода remove() интерфейса Iterator.
      Объекты ArrayList индексируются как массивы, и при
      удалении элемента необходимо указать индекс.
      Этот метод удаляет последний элемент в ArrayList.
  */
  public Seed8 removeSeed()
  {
    return cupContents.remove(cupContents.size() - 1);
  }
}
```

Апплет `TestCup8AndSeed8`:

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.Ellipse2D;
import java.awt.Color;
import java.util.Random;
public class TestCup8AndSeed8 extends JApplet
{
  public void paint(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    Cup8 mycup = new Cup8();
    Random generator = new Random();
    Seed8 someSeed;
    ArrayList<Seed8> seedsInCup;
    double randX, randY, seedX, seedY;
    someSeed = new Seed8(Color.RED);
    mycup.addSeed(aseed);
    someSeed = new Seed8(Color.GREEN);
```

```
    mycup.addSeed(aseed);
    someSeed = new Seed8(Color.BLUE);
    mycup.addSeed(aseed);
    Rectangle2D.Double myrectangle =
      new Rectangle2D.Double(20, 20, 100, 100);
    g2.draw(myrectangle);
    seedsInCup = mycup.getContents();
    for(Seed8 someSeed:  seedsInCup)
    {
      g2.setColor(someSeed.getColor());
      randX = generator.nextDouble();
      randY = generator.nextDouble();
      seedX = 20 + randX * 90.0;
      seedY = 20 + randY * 90.0;
      Ellipse2D.Double mydot =
        new Ellipse2D.Double(seedX, seedY, 10, 10);
      g2.fill(mydot);
    }
  }
}
```

Среди прочих классов, использованных в этом примере, имеются два класса, написанные программистом (это классы `Cup8` и `Seed8`), а также системный класс, `ArrayList`. Этот пример иллюстрирует идею, которая пока не обсуждалась, а именно: отношения между различными классами в одном приложении или апплете. Если описать словами то, что должно быть понятно из кода, то объект класса `Cup` содержит экземпляр `ArrayList`, а объект класса `ArrayList` содержит объекты класса `Seed`. В общем, такого типа отношения называются *отношениями наследования*, т. е. отношения типа "содержит" (has-a). В некоторых случаях выражения "заключает в себе" или "состоит из" могут быть более точными. В действующей программе именно те объекты, которые существуют, участвуют в таких отношениях, но в целях обсуждения структуры приложения или апплетов в целом полезно говорить о классах как о вступающих в такие отношения, а не о конкретных объектах, привязанных к конкретному выполнению программы.

Понятие отношения "содержит" очень важно. Большинство приложений включают такого типа отношения, и эти отношения являются основополагающими для логики приложений. Иными

словами, если отношения понятны, то в принципе понятно и то, как работает программа. В предыдущих главах уже упоминался язык UML, как способ схематичного изображения отношений. Этот язык имеет специальные символы для отношений "содержит". Имена классов соединяются отрезками, которые заканчиваются ромбами или бубнами. Конец отрезка с ромбом направлен к классу, который "содержит", в то время как конец без ромба направлен к классу, который "содержится". Из примера с семенами и чашками видно, что один объект может содержать более одного экземпляра другого объекта. На схеме может быть указано конкретное число в конце отрезка или же может стоять звездочка для обозначения произвольного количества. На рис. 9.2 приведен пример такой схемы, основанный на чашках и семенах. Чашка содержит только один `ArrayList`, в то время как этот `ArrayList` может содержать от 0 до произвольного количества семян.

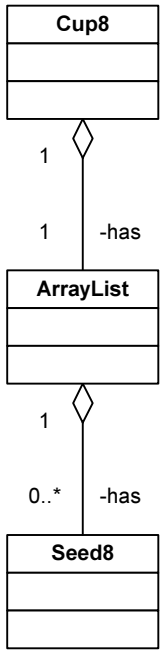
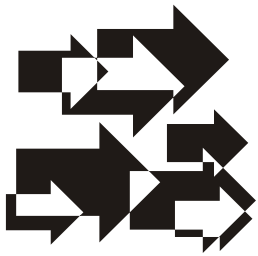


Рис. 9.2

ГЛАВА 10



Файл ввода/вывода

10.1. Вывод на экран и чтение из командной строки

В приводившихся до сих пор примерах использовался класс `MyTerminalIO` для ввода/вывода с клавиатуры. Это давало возможность простого, ориентированного на окна, интерфейса для начинающего программиста. В последующих главах будет затронута тема создания приложений с помощью интегрированного графического ввода/вывода и будет рассмотрен класс `Format`. Данная глава касается неграфического ввода/вывода, начиная с ввода/вывода по командной строке. Когда работает программа языка Java, в дополнение к окнам, которые могут открываться, открывается командная строка, в которой могут появляться сообщения системы. Также имеется возможность показывать результат работы программы в окне командной строки.

Существует класс `System`, содержащий переменную экземпляра с именем `out`, которая является стандартным потоком вывода. Переменная не объявляется частной, поэтому на нее можно ссылаться, используя запись с точкой, `System.out`, и вызвать методы непосредственно из программы пользователя. Методы `print()` и `println()` могут быть вызваны на `System.out`. Поскольку методы `print()` и `println()` класса `MyTerminalIO` были образованы по образцу этих системных методов, то если вы знакомы с ними, вам

будет понятно, как использовать их с переменной `System.out`. Посмотрите на два простых примера. Результат данных примеров будет выглядеть так же, если бы вызовы были сделаны на `myterminal`, но он появился бы в окне командной строки.

```
System.out.print("Hello, World!");
System.out.print(7 + 9);
```

Также можно использовать метод `printf()`, который поддерживает форматирование результата. Этот метод основан на указаниях по форматированию, которые состоят из знака `%`, символов конверсии и иных знаков. Символы конверсии соответствуют типам выводов. Вот неполный список этих символов с краткими пояснениями. Этот список включает только те типы, которые рассматривались в предыдущих главах:

- `d` — десятичный знак (для типа `int`);
- `f` — плавающая точка (для типа `double`);
- `s` — строка;
- `b` — булев.

Простейшие указания по форматированию состоят из знака `%`, за которым следует один из перечисленных выше символов, например, `%d`, `%s`, `%b`. Для типов с плавающей точкой также можно указать максимальную ширину числа, которое может быть принято, включая десятичный знак, и количество знаков стоит после него. Например, рассмотрим формат: `%7.2f`. Это означает, что будет максимум 7 позиций, включая десятичный знак, и до двух знаков справа от него. В указании формата можно использовать символ `n`. `%n` указывает на новую строку, когда используется метод `printf()`.

Список параметров для вызова `printf()` начинается со строки в двойных кавычках, которая может содержать буквальный текст для вывода на экран и указания по форматированию для различных значений, возможно, включенных в текст. За этой строкой идет список имен переменных, которые соответствуют требованиям форматирования и разделены запятыми. Обратите внимание на отличия данного метода от методов `print()` и `println()`. Параметры не соединяются вместе с помощью знака `+`, но отделены друг от друга запятыми.

Вот простой пример:

```
String measurementName = "Temperature";
double measurementValue = 32.9;
System.out.printf("Measurement:  %s%nValue: %6.2f",
  measurementName, measurementValue);
```

Результат этого фрагмента кода будет выглядеть так:

```
Measurement:  Temperature
Value:  32.90
```

Программа также может принимать ввод из командной строки. Это делается с использованием класса `Scanner`. Класс `Scanner` реализует часть интерфейса `Iterator`, упоминавшегося в предыдущей главе. Это означает, что в нем имеется метод `hasNext()`, который возвращает булево значение, сообщая, есть ли ввод из командной строки. Класс `Scanner` имеет набор методов `next()`, которые получают следующий элемент ввода. У этого класса нет метода `remove()`.

В своей простейшей форме методы для получения ввода аналогичны методам, используемым в `MyterminalIO`. Имеется возможность вызвать метод `nextInt()`, если ожидается тип `int`, `nextDouble()`, если ожидается `double`, и `nextLine()`, если ожидается строка, где пользователь прекращает ввод, нажав клавишу <Enter>. Для того чтобы использовать класс `Scanner`, необходимо импортировать его и создать его экземпляр. Параметром конструктора будет `System.in`, который является стандартным потоком ввода, равно как `System.out` является стандартным потоком вывода. Вот простой пример:

```
import java.util.Scanner;
...
Scanner myscanner = new Scanner(System.in);
System.out.println("Enter the name of the measurement.");
String measurementName = myscanner.nextLine();
System.out.println("Enter the value of the measurement.");
double measurementValue = myscanner.nextDouble();
```

В этом классе имеются методы и для других типов данных. В нем также есть простой метод `next()`, который дает пользователю возможность ввести в командной строке более одной единицы

данных одновременно. Специальным термином для различных данных является *лексема* (token), а разделителем различных лексем по умолчанию является *пробельный символ* (whitespace). К пробельным символам относятся пробел, табуляция и иные специальные символы, такие как символ перевода строки, которые не проявляются как печатные символы в командной строке. Метод `next()` подбирает следующую лексему и возвращает ее как экземпляр класса. Если лексема должна содержать целое число или число двойной точности, тогда необходимо использовать такие методы, как `parseInt()` и `parseDouble()`, чтобы выделить значение нужного типа. Можно переопределить разделитель между лексемами. Также можно определить производные модели, которым должен соответствовать ввод. Полное описание использования класса `Scanner` — это большая тема, и здесь мы ограничимся только этим введением.

10.2. Символы и Unicode

Знакомые нам символы, такие как буквы алфавита и иные знаки, представлены в компьютере с помощью числовых кодов. Система кодов, которая используется в языке Java, известна как Unicode. Этот код приписывает числовое значение каждому выводимому на экран символу. Unicode частично базируется на предыдущей системе кодов, которая называлась ASCII и, в свою очередь, основывалась на системе передачи текста по телетайпу. Unicode охватывает тысячи символов из самых разных языков и алфавитов. Наш краткий обзор касается только первых 128 кодов, которые соответствуют буквам английского алфавита, символам, обычно включаемым в клавиатуру пишущей машинки, а также некоторым кодам, которые не представляют выводимые на экран символы.

Первые 32 кода называются *контрольными кодами*. Они не печатаются. Их так называют потому, что они появились как инструкции для общения с некоторым устройством, таким как телетайп, получающим поток текста. Большинство этих кодов не нужно программисту. Однако некоторые из них, такие как перевод строки или возврат каретки, могут быть полезны в програм-

мах, где генерируется вывод на экран. Кстати, мы уже сталкивались со случаем, когда такие символы могут быть косвенно использованы в программе. В *разд. 2.2* было сказано, что подобная строка кода вызывает вывод на экран на новой строке:

```
myterminal.print("\n");
```

Escape-последовательность `\n` создает два контрольных кода: перевод строки и возврат каретки. Аналогичным образом `\t` создает горизонтальную табуляцию. Хотя сам символ не выводится на экран, если поместить его в строку, которая должна быть напечатана, то эти символы повлияют на то, как строка будет выглядеть. Контрольными символами также можно манипулировать напрямую, вместо того, чтобы использовать escape-последовательность. Как это делается, показано далее, где речь пойдет о печатаемых символах. Но сначала посмотрим на таблицу первых 32 кодов (табл. 10.1) в десятичной системе вместе с их названиями и описанием.

Таблица 10.1. *Unicode-значения от 0 до 31 (управляющие символы)*

Код	Название	Описание
0	NUL	Null, пустой
1	SOH	Начало заголовка
2	STX	Начало текста
3	ETX	Конец текста
4	EOT	Конец передачи
5	ENQ	Прошу подтверждения
6	ACK	Подтверждение
7	BEL	Звонок
8	BS	Возврат на один символ
9	HT	Горизонтальная табуляция
10	LF	Прогон строки
11	VT	Вертикальная табуляция
12	FF	Прогон страницы

Таблица 10.1 *(окончание)*

Код	Название	Описание
13	CR	Возврат каретки
14	SO	Смещение наружу
15	SI	Смещение вовнутрь
16	DLE	Следующие символы имеют специальный смысл
17	DC1	1-й символ управления устройством
18	DC2	2-й символ управления устройством
19	DC3	3-й символ управления устройством
20	DC4	4-й символ управления устройством
21	NAK	Не подтверждаю
22	SYN	Символ синхронизации
23	ETB	Конец блока передачи
24	CAN	Отмена
25	EM	Конец (перфоленты и т. д.)
26	SUB	Подставить
27	ESC	Escape. Следующие символы — специальные
28	FS	Разделитель файлов
29	GS	Разделитель групп
30	RS	Разделитель записей
31	US	Разделитель блоков

Коды от 32 по 127 являются печатаемыми символами, за исключением кода 127, который обозначает удаление. Порядок расположения букв алфавита и пунктуационных знаков сложился исторически. Порядок расположения символов или текстовых единиц основан на значении индексов Unicode. В табл. 10.2 приведены перечень этих кодов в десятичной системе счисления и обозначаемые ими символы. Как видно из таблицы, прописные буквы идут перед строчными, а различные пунктуационные знаки находятся до, между и после букв.

Таблица 10.2. *Unicode-значения от 32 до 127 (печатаемые символы)*

Код	Символ	Код	Символ	Код	Символ	Код	Символ
32	Пробел	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	‘	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	Удале-ние

Символьный тип данных, идентифицируемый ключевым словом `char`, уже упоминался в одной из предыдущих глав. Теперь мы можем остановиться на этом типе подробнее. Символьный тип является фактически целочисленным типом. Можно привести нечто, имеющее тип `char`, к типу `int`. Значение, лежащее в основе, всегда является целым числом. Тип определяет, выводится ли на экран значение целого числа или символ Unicode. Посмотрите на программу, которая иллюстрирует идею того, что целое число может быть приведено к символу и выведено в таком виде на экран:

```
public class UnicodeChars
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    char mychar;
    int i;
    for(i = 32; i < 128; i++)
    {
      mychar = (char) i;
      myterminal.print(mychar + "");
    }
  }
}
```

Вывод программы состоит из печатаемых символов в следующем порядке:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ...
```

Помните, что символьная константа отделяется парой одиночных кавычек. Следующая программа иллюстрирует перевод в обратном направлении, от символа к целому числу:

```
public class CharToInt
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    char mychar = 'A';
    int i = (int) mychar;
```

```
    myterminal.print(i);
  }
}
```

Выводом программы является целочисленное значение 65. Посмотрев в табл. 10.2, вы увидите, что этот код соответствует заглавной букве A.

10.3. Обработка исключений

Обработка исключений — это механизм языка Java для решения проблем или ошибок, возникающих во время выполнения. Этот аспект языка Java позволяет программам гибко реагировать на ситуацию, возникающую во время выполнения программы, которая привела бы к аварийному состоянию, если бы не использовалась обработка исключений. Эта тема затрагивается здесь потому, что при использовании файлов ввода/вывода необходимо обрабатывать исключения. В этом разделе мы рассмотрим основной синтаксис и логику обработки исключений, не касаясь специфики синтаксиса файла ввода/вывода. В последующих разделах будет дан синтаксис для файла ввода/вывода, включенный в соответствующие структуры для обработки исключений.

Исключение — это общий термин для какой-либо ошибки. Исключения бывают двух видов: обрабатываемые и необрабатываемые. *Необрабатываемые исключения* — это такие исключения, где ошибка находится внутри самой программы. Простым примером такого исключения будет деление на ноль. Это математически невозможно, и когда программа пытается это сделать, она останавливается. Программист отвечает за отсутствие в коде подобных проблем. Проблемы такого типа можно разрешить за счет проверки значений переменных перед выполнением деления.

В отличие от необрабатываемых исключений, *обрабатываемые* вызываются внешними, по отношению к программе, причинами и находятся вне контроля программиста. Когда система обнаруживает такого рода положение, она выбрасывает исключение, что означает, что объект исключения возвращен в выполняемую программу. Обрабатываемые исключения программист обязан пере-

хватить. Ввод/вывод охватывает такие действия, как открытие файла, запись в файл, чтение файла и закрытие файла. В программе может содержаться код, определяющий выполнение этих действий, но по какой-либо причине система может быть не в состоянии выполнить такие действия. Система выбросит исключение, а в программе должен присутствовать код для реагирования на исключение. Хотя может показаться, что иметь дело с исключениями неудобно, в действительности желательно их обрабатывать, поскольку программы смогут изящно среагировать на непредвиденные обстоятельства.

Ключевыми словами, связанными с исключениями, являются `try`, `catch` и `finally`. В коде за ними идет пара скобок, формирующих блоки, содержащие соответствующие части программ. Блок `try` должен содержать любые строки кода, которые могут вызвать выброс исключения. Блок `catch` построен так, чтобы перехватить любые исключения, которые могут быть выброшены. Блок `finally` содержит код, который должен быть выполнен независимо от того, было ли выброшено исключение или нет. Далее показана последовательность блоков `try`, `catch` и `finally` с комментариями, описывающими файл ввода/вывода и другие действия, которые произойдут в каждом блоке, если целью программы является файловый вывод:

```
try
{
  /*  Открыть интересующий файл. Система выбросит
      исключение, если не сможет открыть файл.  */
  /*  Записать в интересующий файл. Система выбросит
      исключение, если не сможет записать в файл.  */
}
```

Блок `catch` должен следовать сразу за блоком `try`. Блок `catch` принимает параметр, который является типом исключения. Этот механизм позволяет программе получить исключение, выброшенное системой. Класс `IOException` подходит как для открытия файла, так и записи исключения в файл. Иногда можно предпринять какие-либо действия в коде, чтобы разрешить проблему, но поскольку обрабатываемые исключения являются результатом ошибки, находящейся вне контроля программиста, это не всегда так.

В приведенном далее коде единственное предпринимаемое действие — это вывод на экран сообщения об ошибке и самого исключения. Как видно, вывод на экран производится с помощью вызова `System.out.println()`, а не `myterminal.println()`. Полезно разграничивать сообщения, являющиеся результатом исключений, от сообщений, являющихся стандартным выводом программы. Этот подход к выводу на экран будет использоваться и для остальных примеров в данной главе.

```
catch(IOException e)
{
  System.out.println("Caught an IOException during
    opening or writing:  " + e);
}
```

Сразу за блоком `catch` идет блок `finally`. Есть действия, которые необходимо предпринять независимо от того, продолжается ли исключение после того, как был успешно выполнен блок `try`, или после того, как был задействован блок `catch`. Если был открыт файл, то когда программа закончила свою работу с ним, следует закрыть такой файл. Это относится к любым ситуациям, независимо от того, была ли возможность записи в файл или нет.

Следующий пример иллюстрирует еще одну тонкость. Возможна ситуация, когда исключение было выброшено при попытке открыть файл, но сам файл открыт не был. Это означает, что код для закрытия файла должен быть помещен в собственный блок `try`, поскольку попытка закрыть файл, который не является открытым, также создаст исключение. За этим идет блок `catch`, но нет блока `finally`: файл либо требуется закрыть, либо не требуется. В данном случае, больше ничего делать не нужно. В целом, также не имеет смысла бесконечно использовать вложенные блоки `try`, `catch` и `finally`. В какой-то момент нужно просто попробовать `try`, `catch` и двигаться дальше. Вот пример блока `finally` для открытия и записи:

```
finally
{
  try
  {
    /*  Закрыть интересующий файл.  */
  }
```

```
  catch(IOException e)
  {
    System.out.println("Caught an IOException during closing"
                       + e);
  }
}
```

10.4. Запись символов в файл

В следующем примере файл сделан доступным для записи в него за счет создания объекта класса `FileWriter` и передачей имени текстового файла конструктору в качестве параметра. Имя файла может включать путь, но можно использовать и простое имя файла, если вы предпочитаете работать с файлами в том же каталоге, где находится сама программа. Если файла с таким именем не существует, он будет создан. Если файл уже существует, его содержимое будет заменено новым, которое будет в него записано. Когда файл открывается, запись начинается с начала. В течение выполнения программы каждый последующий символ присоединяется к тому содержимому, что уже было записано.

Создание объекта `FileWriter` открывает файл. Вы можете записывать по одному символу за один раз, вызвав метод `write()` к объекту `FileWriter`. Файл закрывается с помощью вызова метода `close()` к объекту `FileWriter`. Создание объекта `FileWriter` и вызов метода `write()` находятся в одном блоке, поскольку они оба вернут исключение `IOException`, если не будут выполнены успешно. Ссылка `FileWriter` инициализируется на `null`. Если объект не может быть создан, то ссылка остается равной `null`. В блоке `finally`, если файл был открыт, но при записи произошел сбой, файл будет закрыт. Однако если ссылка `FileWriter` равна `null`, то необязательно закрывать файл.

Эта программа использует последовательность блоков `try`, `catch` и `finally`, объясненную в предыдущем разделе. Она также представляет собой наглядный пример синтаксиса для открытия файла, записи в файл и закрытия файла. Она подобна примеру программы из *разд. 10.2*, которая генерировала печатаемые символы. Однако вместо того, чтобы выводить символы на экран, эта программа записывает их в файл. Часто доступ к файлу производит-

ся с использованием цикла. В данном случае подходит цикл `for`, т. к. количество символов и количество обращений к файлу заранее известно:

```
import java.io.FileWriter;
import java.io.IOException;
public class WriteAlphabet
{
  public static void main(String[] args)
  {
    int i;
    char mychar;
    FileWriter mywriter = null;
    try
    {
      mywriter = new FileWriter("alphabet.txt");
      for(i = 1; i <= 26; i++)
      {
        mychar = (char) (i + 64);
        mywriter.write(mychar);
      }
    }
    catch(IOException e)
    {
      System.out.println("Caught an IOException
        during opening or writing" + e);
    }
    finally
    {
      try
      {
        if(mywriter != null)
          mywriter.close();
      }
      catch(IOException e)
      {
        System.out.println("Caught an IOException
          during closing" + e);
      }
    }
  }
}
```

Данная программа показывает, как записать символы в файл. Это выполняется двумя строками кода:

```
mychar = (char) (i + 64);
mywriter.write(mychar);
```

Программы часто используют строки, состоящие из одного символа, а не просто символы, т. к. существует множество полезных методов, которые можно использовать для строк. Метод `write()` объекта `FileWriter` принимает строки в качестве параметров. Две строки кода, указанные выше, можно заменить тремя следующими. Для нашего примера такая замена нежелательна, поскольку программа станет длиннее, но эти строки показывают, как можно использовать строки вместо символов:

```
mychar = (char) (i + 64);
String charasstring = mychar + "";
mywriter.write(charAsString);
```

10.5. Чтение символов из файла

Чтение из файла аналогично записи в файл. Оно основано на той же самой структуре блоков `try`, `catch` и `finally`. Необходимо открыть файл до использования и желательно закрыть его после использования. Для чтения создается объект класса `FileReader`. Если файл еще не существует, это приведет к выбросу исключения. Чтение производится с помощью вызова метода `read()` к этому объекту. Во время выполнения программы каждый последующий вызов метода `read()` возвращает следующий символ в файл.

Существует несколько различий между чтением и записью. Когда происходит чтение текстового файла, система возвращает значение кода Unicode, которое соответствует некоторому символу. Необходимо преобразовать это значение обратно в символ. Поскольку существует множество полезных методов для строк, то удобно превратить этот символ в строку. Кроме того, как правило, вам неизвестно, какова длина файла, из которого вы читаете. Система возвращает значение –1, когда достигается конец файла. Для чтения символов из файла подходит цикл `while`, который останавливается тогда, когда считывается значение –1.

Рассмотрим простую программу, которая читает содержимое файла с тем же именем, которое было у файла, созданного в предыдущем примере, и выводит результат на экран:

```
import java.io.FileReader;
import java.io.IOException;
public class ReadFile
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    int charAsInt;
    char charAsChar;
    String charAsString;
    FileReader myreader = null;
    try
    {
      myreader = new FileReader("alphabet.txt");
      charAsInt = myreader.read();
      while(charAsInt != -1)
      {
        charAsChar = (char) charAsInt;
        charAsString = "" + charAsChar;
        myterminal.print(charAsString);
        charAsInt = myreader.read();
      }
    }
    catch(IOException e)
    {
      System.out.println("Caught an IOException
        during opening or reading" + e);
    }
    finally
    {
      try
      {
        if(myreader != null)
          myreader.close();
      }
```

```
      catch(IOException e)
      {
        System.out.println("Caught an IOException
          during closing" + e);
      }
    }
  }
}
```

В этой программе имеются две строки кода, которые преобразуют прочитанные символы в объект класса `String` и выводят результат на экран:

```
charAsString = "" + charAsChar;
myterminal.print(charAsString);
```

Символ преобразуется в объект класса `String`, поскольку класс может обращаться со строками, но не с символами. Последовательное считывание возвращает значения Unicode, которые могут или не могут быть преобразованы в символы или строки, в зависимости от требований приложения.

Посмотрите на пример программы, которая подсчитывает количество случаев употребления букв алфавита в файле и определяет их частоту. Тело программы содержит оператор `if`, который рассматривает каждую букву по отдельности. Для экономии места показаны только коды для букв `a` и `b`.

```
import java.io.FileReader;
import java.io.IOException;
public class LetterCount1
{
  public static void main(String[] args)
  {
    int charAsInt;
    char charAsChar;
    String charAsString;
    int totalCount = 0;
    double letterFrequency;
    int[] letterCount = new int[27];
    MyTerminalIO myterminal = new MyTerminalIO();
```

```
FileReader myreader = null;
myterminal.println("Enter the name of a text");
myterminal.println("file to count the letter");
myterminal.println("frequencies in.  Include");
myterminal.println("the .txt extension.");
String filename = myterminal.getString();
try
{
  myreader = new FileReader(filename);
  charAsInt = myreader.read();
  while(charAsInt != -1)
  {
    charAsChar = (char) charAsInt;
    charAsString = "" + charAsChar;
    if(charAsString.equalsIgnoreCase("a"))
    {
      totalCount++;
      letterCount[1]++;
    }
    else if(charAsString.equalsIgnoreCase("b"))
    {
      totalCount++;
      letterCount[2]++;
    }
    /* ... */
    else
    {
    }
    charAsInt = myreader.read();
  }
}
catch(IOException e)
{
  System.out.println("Caught an IOException
    during opening or reading" + e);
}
finally
{
  try
```

```
      {
        if(myreader != null)
          myreader.close();
      }
      catch(IOException e)
      {
        System.out.println("Caught an IOException
          during closing" + e);
      }
    }
    myterminal.println("Here are the letter counts
      and frequencies.");
    for(int i = 1; i <= 26; i++)
    {
      letterFrequency = (double) letterCount[i] /
        (double) totalCount;
      charAsChar = (char) (i + 64);
      myterminal.println(charAsChar + "  " +
        letterCount[i] + "  " + letterFrequency);
    }
  }
}
```

Тот же самый результат возможен с более коротким кодом, если оператор `if` в этом примере заменить на следующий код. Эта более краткая версия соотносит значения кодов Unicode для букв алфавита с индексами массива, в котором хранятся эти данные:

```
if(((charAsInt >= 65) && (charAsInt <= 90)) ||
   ((charAsInt >= 97) && (charAsInt <= 122)))
{
  totalCount++;
  if((charAsInt >= 65) && (charAsInt <= 90))
  {
    letterCount[charAsInt - 64]++;
  }
  else
  {
    letterCount[charAsInt - 96]++;
  }
}
```

10.6. Запись строк в файл

В языке Java имеется другой класс ввода/вывода, `PrintWriter`, который поддерживает методы `print()` и `println()` для присоединения текста к файлу. Для того чтобы использовать `PrintWriter`, необходимо, чтобы экземпляр `FileWriter` был соединен с файлом, а затем нужно построить класс `PrintWiter` с `FileWriter` в качестве его параметра. С этого момента все, что выводится на экран, будет записываться в `FileWriter`.

Рассмотрим простой пример:

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;
public class WriteStrings
{
  public static void main(String[] args)
  {
    String mystring;
    PrintWriter myprintwriter = null;
    FileWriter mywriter = null;
    MyTerminalIO myterminal = new MyTerminalIO();
    try
    {
      mywriter = new FileWriter("strings.txt");
      myprintwriter = new PrintWriter(mywriter);
      mystring = myterminal.getString("Enter the
        string you'd like added to the file.");
      myprintwriter.print(mystring);
    }
    catch(IOException e)
    {
      System.out.println("Caught an IOException during
                          opening or writing" + e);
    }
    finally
    {
      try
```

```
      {
        if(myprintwriter != null)
          myprintwriter.close();
        if(mywriter != null)
          mywriter.close();
      }
      catch(IOException e)
      {
        System.out.println("Caught an IOException
                            during closing" + e);
      }
    }
  }
}
```

Чтение строк из файла несколько сложнее, т. к. необходимо знать длину строки, которая будет считываться, или знать, какой символ в файле будет сигналом о конце строки. Эту тему мы не станем здесь рассматривать. Всегда можно написать код, который будет считывать из файла по одному символу за один раз и собирать строку из этих символов. Также можно читать и записывать содержимое объектов. Эта процедура называется *сериализацией* и будет рассмотрена в следующей главе.

10.7. Файл ввода/вывода с произвольным доступом

Этот раздел основан на классе `RandomAccessFile`. Рассмотрение обработки исключений и ввода/вывода символов в предыдущих разделах предоставляют необходимую информацию для использования класса `RandomAccessFile`. Подробное описание конструкторов и методов этого класса дается в документации Java API. Данный раздел иллюстрирует произвольный доступ в плане конкретной задачи, рассматривая возникающие для ее решения вопросы.

В случае символьного файла ввода/вывода, файл представляет собой последовательность отдельных символов. Структурированный файл содержит записи, состоящие из наборов полей с

фиксированной длиной. Поскольку эти поля имеют фиксированную длину и каждая запись состоит из фиксированного набора полей, то записи также имеют фиксированную длину. Класс `RandomAccessFile` может быть использован для поддержки структурированного файла и получения доступа к его содержимому. В качестве примера рассмотрим структурированный файл, содержащий информацию о работниках некоторой организации. Схема полей в базе данных о служащем такова:

- `deleteMarkField` — поле с целочисленными значениями. Пока предположим, что 0 означает активные данные;
- `empIdField` — текстовое поле, которое должно быть заполнено 9 цифрами без тире;
- `maritalStatusField` — текстовое поле длиной 8 знаков. Возможные значения: холост, женат, разведен, вдовец;
- `allowancesField` — поле с целочисленными значениями;
- `additionalAmtField` — поле для значений типа `double`.

В целом, такие данные о работнике могут быть полезны при высчитывании налогов. Поле `deleteMarkField` будет использовано для некоторых процедур по обработке файла. Поле `empIdField` является полем идентификации работника. Поле `maritalStatusField` содержит одно из 4-х возможных семейных положений. Поле `allowancesField` указывает на количество иждивенцев плюс 1 (сам работник). Поле `additionalAmtField` содержит сумму, которую необходимо удержать из заработной платы работника, только с учетом количества иждивенцев. Изначальной целью является запись таких сведений в файл и чтение подобных сведений из файла.

Файл произвольного доступа в языке Java имеет системный указатель, и программист может разместить указатель по своему усмотрению и писать или читать с определенного места, *смещения* (offset), в файле. Записи в файле организованы в некоторой последовательности, согласно так называемому *относительному номеру записи* (relative record number, RRN). Смещение начального символа любой записи равно размеру записи, умноженному на относительный номер требуемой записи. Если запись сделана в

определенном смещении, то она перекрывает все предыдущие записи в этом месте. Файлы произвольного доступа также поддерживают чтение и запись с начала файла (смещение равно 0) или присоединение записи в конец файла.

Смещения в файлах высчитываются в байтах. Бит, или бинарная единица, содержит 0 или 1. Байт является последовательностью 8 битов и является мельчайшей единицей для записи информации в компьютере. В языке Java имеется тип `byte`, который может использоваться для действий низкого уровня, но он не требуется, когда целью является просто вычисление количества байтов, занимаемых каким-либо полем. Кстати, еще один тип полезен, когда мы считаем смещения. Файлы могут содержать так много байтов, что тип `int` может не вместить их. Предпочтительнее пользоваться типом `long` для размещения подобных величин.

В нашем примере данные о работнике включают поля, содержащие типы `int` и `double`. Класс `RandomAccessFile` имеет методы для чтения и записи разного типа значений. Например, такие методы, как `readInt()`, `writeInt()`, `readDouble()` и `writeDouble()`. Учет места в файле, т. е. высчитывание смещения для чтения или письма не слишком сложно, когда мы имеем дело с числовыми полями. Тип `int` принимает до 4 байтов, а тип `double` — до 8 байтов.

Символы Unicode содержат до 2 байтов. Если каждое поле состоит из одного символа, то с ними проблем не возникнет. Текстовые поля обычно определяются как строки, и их длина в байтах будет равна длине в символах, умноженной на 2. Однако ситуация со строками более сложна. Не существует методов чтения или письма, которые напрямую поддерживают строки. Есть методы `writeChar()`, `writeChars()` и `readChar()`, которые имеют дело с отдельными символами. Не существует метода `readChars()`.

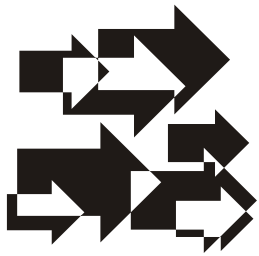
Причина, по которой разработчики языка Java не предусмотрели прямую поддержку строковых полей, заключается в том, что когда пользователь взаимодействует с программой, содержание строки может не совпадать с фиксированным размером поля. Предположим, вы хотите разместить строку, содержащую значение "холост" в поле, длина которого определена 8-ю знаками.

В таком случае программист должен заполнить лишние знаки в поле пробелами. Аналогичным образом пользователь может по ошибке попытаться заполнить значение, которое длиннее 8 знаков. В таком случае программист должен проследить за тем, чтобы только 8 знаков были введены в поле. Те же проблемы возникают и при чтении.

В простом случае управления файлом поле `deleteMarkFiedl` может быть использовано следующим образом. Если запись стирается, то `deleteMarkFiedl` присваивается значение, допустим, 1, которое обозначает стирание. Если это значение присутствует, то данное место доступно для записи сверху. Когда добавляется новая запись, то файл последовательно просматривается в происках первого стертого места, и новая запись прикрепляется к концу файла.

Пространством файла можно распоряжаться и без того, чтобы искать первую стертую запись. Для получения доступа к содержанию файла можно использовать поле `deleteMarkFiedl`, где можно легко посчитать смещение для любой из стертых записей, доступных для новой записи. Концептуально такая схема организует записи в файле так же, как организованы узлы в стеке.

ГЛАВА 11



Рекурсия

11.1. Описание рекурсии

Когда метод содержит вызов самого себя, это называется *рекурсией.*

Рассмотрим класс, содержащий основные компоненты статического метода:

```
public class MyClass
{
  public static int mymethod(int parameterOne)
  {
    int parameterTwo;
    ...
    return mymethod(parameterTwo);
  }
}
```

Язык Java поддерживает рекурсию, и показанная схема иллюстрирует синтаксис для рекурсии. Метод содержит вызов самого себя. Обратите внимание, что синтаксически не обязательно писать вызов в виде `MyClass.mymethod(...)`, поскольку вызов происходит внутри кода, принадлежащего `MyClass`.

Рассмотрим рис. 11.1 набора шаров для бильярда. Он будет использован для иллюстрации того, как рекурсивное определение

может быть использовано для суммирования положительных целых чисел.

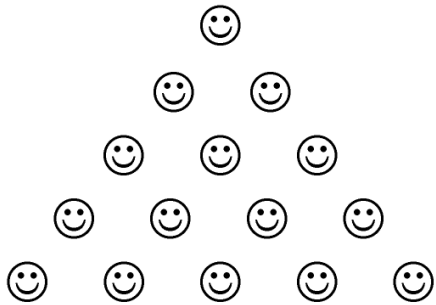


Рис. 11.1

При передвижении из верхнего ряда вниз может возникнуть вопрос: как общее число шаров зависит от числа рядов? Для пяти рядов это достаточно легко вычислить и свести информацию в маленькую таблицу.

Ряды	Сумма шаров в рядах	Общее количество шаров
1	1	1
2	1+2	3
3	1+2+3	6
4	1+2+3+4	10
5	1+2+3+4+5	15

Используя математическую запись, эти суммы можно представить в виде функции:

$$f(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n\,.$$

Теперь посмотрим на иное определение той же самой функции. Оно является рекурсивным:

- рекурсия для $n > 1$

$$f(n) = n + f(n-1) = n + \sum_{i=1}^{n-1} i\,;$$

❒ основание $n = 1$

$$f(1) = 1.$$

Идея состоит в том, что для любых чисел n, больше чем 1, функцию можно определить в терминах n и применить функцию к $n-1$. Для $n = 1$ значение функции просто определяется равным 1. Это основание, где значение функции уже не определяется в терминах ее самой.

Рассмотрим определение другой математической функции. Если x является действительным числом и n каким-либо неотрицательным целым числом, то x в n-й степени будет произведением n-го количества множителей x:

$$x^n = x \cdot x \cdot ... \cdot x.$$

Если вы откроете учебник математики, то найдете следующее рекурсивное определение этого:

❒ рекурсия для $n > 0$

$$f(n) = x + f(n-1) = x \cdot x^{n-1};$$

❒ основание $n = 0$

$$f(0) = x^0 = 1.$$

Можно привести много примеров. В каждом из них определение состоит из двух частей. В первой, рекурсивной части, функция определяется в терминах самой себя. Во второй части определения, основании, функция определяется относительно какого-либо конкретного значения. Во втором примере основание определяется для $n = 0$, а не для $n = 1$.

В приведенных выше примерах n являлось целым положительным числом. В рекурсивной части определения $f(n)$ определяется в терминах n и $f(n-1)$. Иными словами, $f(n)$ определялось в терминах применения функции к значению на 1 меньше. Основание определяет $f(n)$ для $n = 0$ или $n = 1$. Эти примеры иллюстрируют общую структуру рекурсии: n уменьшается на 1 в каждом последующем применении рекурсивной функции. Важно,

чтобы n достигло значения основания, так чтобы рекурсия остановилась.

11.2. Составление и выполнение рекурсивного кода

Рекурсивное определение может быть выполнено в коде для того, чтобы вычислить значение функции. Та часть определения, где функция определяется в терминах самой себя, выполняется как метод, вызывающий сам себя. Исходный случай также является необходимой частью выполнения. Обе эти части будут проиллюстрированы на примерах.

Рассмотрим тестовую программу с классом, содержащим статический метод с рекурсией, которая находит число бильярдных шаров. Все элементы правильного выполнения представлены в примере. Существует определенная избыточность в использовании переменной `returnValue` и в возвращаемых выражениях метода. Это сделано специально для того, чтобы легче было подчеркнуть некоторые действия кода в последующем пояснении. В определениях, приведенных ранее, рекурсивная часть была показана первой для того, чтобы подчеркнуть рекурсию. В этом и последующих примерах оператор `if` проверяет основание и рассматривается первым.

```
public class TestPool
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    int myrows, myresult;
    myterminal.print("How many rows of balls?  ");
    myrows = myterminal.getInt();
    myresult = PoolClass.poolRack(myrows);
    myterminal.println("This is the total number of balls:  "
                       + myresult);
  }
}
```

```
public class PoolClass
{
  public static int poolRack(int numOfRows)
  {
    int returnValue;
    if(numOfRows == 1)
    {
      returnValue = 1;
      return returnValue;
    }
    else
    {
      returnValue = numOfRows + poolRack(numOfRows - 1);
      return returnValue;
    }
  }
}
```

Обратите особое внимание на то, что тело метода построено как оператор `if`. Если выполняется основание, то возвращается простое значение. Если исходное условие не выполняется, происходит еще один вызов метода. В этом вызове значение параметра уменьшается на 1. Последовательные вызовы приближаются к значению, данному в основании, и в конечном итоге достигают его, так что рекурсия успешно остановится.

Просто глядя на код, все еще может быть непонятно, как работает рекурсия. Существует общий способ понимания кода, который может быть применен к этому примеру. Этот метод включает отслеживание значений параметра и возвращаемых значений через последовательность рекурсивных вызовов. Можно нарисовать схему с использованием квадратов для изображения вызывающей программы и отдельных выполнений рекурсивного метода. При написании кода и проверке его на ошибки это можно сделать вручную, отмечая значения параметра и возвращаемые значения на схеме.

На рис. 11.2 приведена подобная схема для метода с бильярдными шарами, где вызывающая программа посылает первоначальный параметр со значением 3. Эта схема была выполнена аппле-

том, который был создан для анимации рекурсии. Путь исполнения идет вниз по левой стороне через последовательность активаций рекурсивного метода и снова вверх по правой стороне, по мере того как каждая активация завершается и возвращает значение.

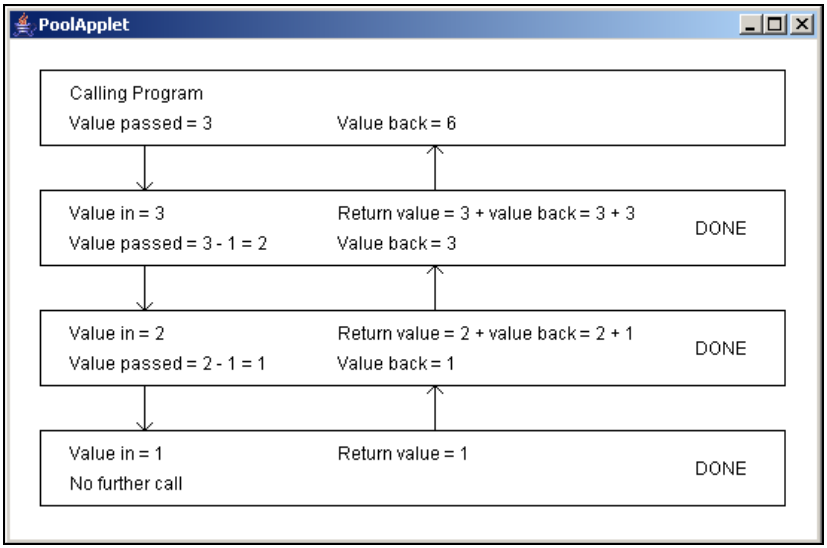


Рис. 11.2

Точная связь между различными вызовами может быть все еще непонятной. Фактически, рекурсивные вызовы являются отложенными вызовами. Каждая активация метода работает до тех пор, пока он не создаст рекурсивный вызов, затем ждет, пока тот возвратит значение, прежде чем продолжать до завершения вычисления. На схеме метод `main()` и две активации метода находятся в ожидании, пока завершится третья активация метода. Третья активация достигает исходного условия, завершается и возвращается во вторую, которая тоже завершается. Когда вторая завершается, она возвращается в первую, которая завершается и возвращается в `main()`.

Известно, что в данном классе существует только одна копия метода, которая является общей для всех его вызовов. Объекты не

получают своих копий методов. Однако каждый вызов метода является уникальным и может быть идентифицирован. Любая активация метода также имеет собственные копии значений любых переменных. Хотя копируются только переменные, в целях объяснения полезно показать процесс вызова так, как если бы существовали многие копии кода, одна вызывающая другую.

Рассмотрим пример. Здесь в методе `main()` производится вызов метода со значением параметра 3:

```
int myresult = PoolClass.poolRack(3);
```

Это начало первого вызова метода. Значение входящего параметра — 3. Метод работает до строки, выделенной жирным шрифтом. Однако эта строка не завершена. Вызов `poolRack()` создан, но получить результат счета нельзя до тех пор, пока не будет сделан возврат значения из вызова. Первый вызов в данный момент просто ждет.

```
public static int poolRack(int numOfRows)
{
  int returnValue;
  if(numOfRows == 1)
  {
    returnValue = 1;
    return returnValue;
  }
  else
  {
    returnValue = numOfRows + poolRack(numOfRows - 1);
```

Вот начало второго вызова метода. Значение входящего параметра — 2. Метод работает до строки, выделенной жирным шрифтом. Однако эта строка не завершена. Вызов `poolRack()` создан, но получить результат счета нельзя до тех пор, пока значение не будет возвращено. Второй вызов в данный момент просто ждет.

```
public static int poolRack(int numOfRows)
{
  int returnValue;
  if(numOfRows == 1)
  {
      returnValue = 1;
```

```
    return returnValue;
  }
  else
  {
    returnValue = numOfRows + poolRack(numOfRows - 1);
```

А вот третий вызов метода. Значение входящего параметра — 1. Метод работает до строки, выделенной жирным шрифтом. Это основание, и в этот момент третий вызов метода завершается. Значение 1 возвращается во второй вызов метода.

```
public static int poolRack(int numOfRows)
{
  int returnValue;
  if(numOfRows == 1)
  {
    returnValue = 1;
    return returnValue;
```

Второй вызов метода теперь возобновляет исполнение. Значение 1 возвращается вызовом к `poolRack()`. Оно прибавляется к переменной `numOfRows`, которая имеет значение 2, и присваивается переменной `returnValue`. Метод работает до завершения, возвращающего переменную в строке, выделенной жирным шрифтом. Значение 3 возвращается первому вызову метода.

```
returnValue = numOfRows + poolRack(numOfRows - 1);
return returnValue;
```

Первый вызов теперь продолжает выполнение. Значение 3 возвращается вызовом `poolRack()`. Оно прибавляется к переменной `numOfRows`, которая имеет значение 3, и присваивается переменной `returnValue`. Метод завершает работу возвращением объявления, показанного в строке, выделенной жирным шрифтом. Значение 6 возвращается вызывающей программе.

```
returnValue = numOfRows + poolRack(numOfRows - 1);
return returnValue;
```

Это выражение возвращает значение 6 методу `main()`. Все вызовы `poolRack()` завершили свою работу, и вызывающая программа может завершить выполнение с помощью объявления вызова, показанного ниже. Возвращенное значение присваивается пере-

менной myresult, и вызывающая программа может продолжить исполнение.

```
int myresult = PoolClass.poolRack(3);
```

Модель, показанная выше с фрагментами кода, может быть описана так:

1. Вызов в главной программе.
2. Первый вызов метода, выполнение до рекурсивного вызова.
3. Второй вызов метода, выполнение до рекурсивного вызова.
4. Третий вызов метода, базовый случай возвращает значение.
5. Возобновление второго вызова, работает до завершения и возвращает значение.
6. Возобновление первого вызова, работает до завершения и возвращает значение.
7. Вызывающая программа возобновляет выполнение.

11.3. Особенности рекурсии

Некоторые задачи, которые могут быть решены с использованием рекурсии, достаточно сложны. Здесь мы рассматриваем только относительно простые примеры. Вы могли заметить, что алгоритм для нахождения количества бильярдных шаров можно было бы выполнить, используя цикл, а не рекурсию. Фрагмент ниже иллюстрирует, как просто это может быть:

```
int sum = 0;
for(int i = 1; i < n; i++)
  sum = sum + 1;
```

Это проще, чем рекурсия, и вообще, любые примеры, представленные в этой главе с помощью рекурсии, могли быть выполнены посредством циклов. Однако вам необходимо иметь представление о рекурсии, как минимум, по двум причинам:

- синтаксически возможно вызвать метод внутри него самого; даже если вы никогда не будете этого делать, вам необходимо знать, что происходит, если вы случайно сделаете такое. Иными словами, существование этой возможности в синтаксисе

ведет к вероятному появлению новых логических ошибок, о которых вам необходимо знать;

- некоторые более сложные понятия информатики проще объяснить в терминах рекурсии, чем в циклах. Поэтому вы должны иметь некоторое представление о рекурсии, даже если никогда не собираетесь использовать ее в своих программах.

Как только начинающий программист научится писать программы, используя циклы, он сталкивается с возможностью нового вида ошибки: бесконечный цикл, или зацикливание программы. Аналогичным образом, как только приобретаются навыки программирования с использованием рекурсии, возникает вероятность появления нового вида ошибки: бесконечная рекурсия. Это особенно неприятная ошибка. Обычно можно прервать бесконечный цикл, закрыв окно программы или используя сочетание клавиш <Ctrl>+<Alt>+<Del>. Однако бесконечная рекурсия может заблокировать систему или процесс, и вам придется перезагрузить компьютер, чтобы продолжить работу.

Почему это случается? Причина заключается в том, что каждый вызов метода занимает определенное пространство в памяти системы для хранения копий параметров, локальных переменных и других вещей, которые относятся к вызову. Когда возникает бесконечная рекурсия, память для вызовов метода выделяется до тех пор, пока она не закончится. И тогда процесс или система не имеют больше свободной памяти — они зависают. С этим можно справиться с помощью нажатия комбинации клавиш <Ctrl>+ +<Alt>+<Del>, но в зависимости от конфигурации системы это может не сработать, и тогда придется перезагрузить систему иным способом, чтобы избавиться от бесконечной рекурсии. Проверка рекурсии на ошибки требует внимательного чтения кода, поскольку компилятор может не обнаружить ошибки, а ошибки, возникающие во время работы программы, трудно установить.

Существует множество причин, вызывающих ошибку, и много необычных результатов, которые можно получить из неправильного рекурсивного кода. Но самая неприятная ошибка, которую можно сделать, — это ошибка в основании. Либо тело метода не

построено как `if` с двумя альтернативами, одна из которых является основанием, либо неправильно написано условие проверки основания, либо последовательность значений, посланных в последующих вызовах к методу, не достигает в конечном счете значения, проверенного в исходном условии. Такие ошибки могут привести к бесконечной рекурсии и иным проблемам.

11.4. Рекурсия и циклы, рекурсия с двумя параметрами

Пример с бильярдными шарами может быть выполнен с использованием цикла `for`. Можно также написать рекурсивные методы для решения тех же задач, которые решаются с помощью цикла `while`. Деление на некоторый делитель является удобной отправной точкой для иллюстрации того, как определить такую функцию рекурсивно. Пусть требуется узнать, сколько раз константа со значением 2 помещается целое число раз в другом числе. Значение 1, которое добавляется к рекурсивному случаю, подсчитывает, сколько раз произошла рекурсия. Этот счет дает результат деления. Обратите внимание, что в рекурсивном случае n уже не уменьшается на 1. Оно уменьшается на 2. Тестом для исходного случая является не равенство, а неравенство, т. к. значения, которые принимает n, варьируются в зависимости от того, является ли оно изначально четным или нечетным числом:

- рекурсия для $n \geq 2$

$$f(n) = 1 + f(n-2);$$

- основание $n < 2$

$$f(n) = 0.$$

Вот простая программа. Рекурсивный метод `recursiveDivisionByTwo()` имеет тип `int` и принимает параметр с именем `dividend`.

```
public static int
  recursiveDivisionByTwo(int dividend)
{
  if(dividend < 2)
```

```
    return 0;
  else
    return 1 + recursiveDivisionByTwo(dividend – 2);
}
```

На основании изложенного выше, можно написать рекурсивное определение для функции деления в целом. Функция имеет два параметра. Добавление 1 к рекурсивному случаю снова будет подсчитывать, сколько раз происходит рекурсия. В этом определении первый параметр для рекурсивного случая — это *делимое – делитель*. Значение, на которое происходит уменьшение каждый раз, остается тем же самым, но *делитель* может быть любым (положительным целым) числом. Второй параметр — это *делитель*, который передается без изменения через все вызовы.

Итак:

- рекурсия: *делимое* >= *делитель*

$$f(\textit{делимое}, \textit{делитель}) = 1 + f(\textit{делимое} - \textit{делитель}, \textit{делитель});$$

- основание: *делимое* < *делитель*

$$f(\textit{делимое}, \textit{делитель}) = 0.$$

Вот код для полной тестовой программы, за которым идет код для класса, реализующего данную рекурсивную функцию. Как обычно, это не общая реализация, но она работает для соответствующих исходных значений и иллюстрирует основные моменты. Обратите внимание, что в примере использована переменная `retVal`:

```
public class GenRecDivProg
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    int retVal;
    int dividend, divisor;
    myterminal.print("Enter the dividend:  ");
    dividend = myterminal.getInt();
    myterminal.print("Enter the divisor:  ");
    divisor = myterminal.getInt();
```

```
    retVal = GenRecDivClass.genRecDiv(dividend, divisor);
    myterminal.println("The integer quotient is:  "
      + retVal);}
  }
}
public class GenRecDivClass
{
  public static int
    genRecDiv(int dividend, int divisor)
  {
    int retVal = 0;
    if(dividend < divisor)
      return retVal;
    else
    {
      retVal = 1 + genRecDiv(dividend - divisor, divisor);
      return retVal;
    }
  }
}
```

Если деление можно представить как последовательное вычитание, то нахождение логарифма можно описать как последовательное деление. Для того чтобы найти логарифм данного числа, которое будет представлено переменной с именем `toFindLogOf`, с некоторым целым основанием, рассмотрим следующее простое рекурсивное определение логарифмической функции:

- рекурсия: `toFindLogOf` >= *основание*

$$f(\texttt{toFindLogOf}, \textit{основание}) = 1 + f(\texttt{toFindLogOf}/\textit{основание}, \textit{основание});$$

- основание: `toFindLogOf` < *основание*

$$f(\texttt{toFindLogOf}, \textit{основание}) = 0.$$

Следующий пример содержит два статических метода, которые находят простой логарифм за счет последовательного деления. Один делает это с помощью цикла, а другой — с помощью рекурсии. Для положительных исходных данных, состоящих из данного числа `toFindLogOf` типа `double` и данного целого числа в

виде основания, код дает на выходе наибольшее целое значение, которое будет меньше или равно логарифму для этого числа и этого основания.

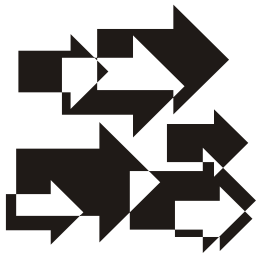
```
public class TestRecAndLoop
{
  public static void main(String[] args)
  {
    MyTerminalIO myterminal = new MyTerminalIO();
    double toFindLogOf;
    int base;
    double answer;
    myterminal.println("What number would you like
      to find the log of?");
    toFindLogOf = myterminal.getDouble();
    myterminal.println("What should the base of the
      logarithm be?");
    base = myterminal.getInt();
    answer =
      RecursionAndLooping.myLogLooping(toFindLogOf, base);
    myterminal.println("The looping answer is:  " +
      answer);
    answer =
      RecursionAndLooping.myLogRec(toFindLogOf, base);
    myterminal.println("The recursive answer is:  "
      + answer);
  }
}
public class RecursionAndLooping
{
  public static int
    myLogLooping(double toFindLogOf, int base)
  {
    int retVal = 0;
    while(toFindLogOf >= base)
    {
      toFindLogOf = toFindLogOf / base;
      retVal++;
    }
```

```
    return retVal;
  }
  public static int
    myLogRec(double toFindLogOf, int base)
  {
    int retVal = 0;
    if(toFindLogOf < base)
      return retVal;
    else
    {
      toFindLogOf = toFindLogOf / base;
      retVal = 1 + myLogRec(toFindLogOf, base);
      return retVal;
    }
  }
}
```

Обратите внимание, что следующая версия рекурсивного метода будет более экономной и компактной, но одновременно и более непонятной:

```
public static int
  myLogRec(double toFindLogOf, int base)
{
  int retVal = 0;
  if(toFindLogOf > base)
  {
    toFindLogOf = toFindLogOf / base;
    retVal = 1 + myLogRec(toFindLogOf, base);
  }
  return retVal;
}
```

ГЛАВА 12



Подклассы и наследование

12.1. Что такое наследование?

Все классы в Java располагаются в иерархическом порядке относительно друг друга, за исключением класса на самом верху иерархии, над которым ничего нет. Непосредственно над каждым классом находится другой класс. Над этим классом может быть еще один и т. д. В общем, любой данный класс может иметь более одного класса над собой, на различных уровнях иерархии. Класс, расположенный сразу же над данным классом, называется *родительским классом* или *суперклассом*. Вообще, все классы над данным классом называются суперклассами. Класс также может иметь больше одного класса ниже себя по иерархии. Ниже могут быть и другие классы и т. д. В общем, любой данный класс может иметь больше одного класса ниже себя, на различных уровнях иерархии. Класс, следующий сразу же за данным классом, называется *дочерним классом* или *потомком*. Все классы ниже данного называются *подклассы*.

Вид иерархии, описанный выше, представляет собой своего рода дерево, хотя принято изображать его "вверх ногами", с корнем наверху и ветвями внизу. В Java на вершине иерархии, т. е. у корней дерева, находится единственный класс, и этот класс называется `Object`. Все классы, поддерживаемые системой, организованы в иерархию ниже этого класса. Все классы, созданные и ис-

пользованные программистами в Java, также входят в эту иерархию. В предыдущих главах был приведен код для нескольких различных классов, таких как класс `Cup` или класс `Seed`. Когда родительский класс не определен, такие классы становятся прямыми потомками, или подклассами, класса `Object`. Существует специальный синтаксис, позволяющий программистам создать класс, который будет подклассом другого класса. Это дает программистам возможность создавать собственные иерархии. Такой синтаксис также делает возможным для программистов ввод своих классов и иерархий в существующую иерархию на любом уровне ниже класса `Object`.

Классы в иерархии связаны *наследованием*. Любой класс, который имеет классы над собой в иерархии, может унаследовать характеристики этого класса, включая переменные и методы. Конструкторы классов не наследуются. Естественно, имеется возможность для подклассов добавлять атрибуты, которые не существуют в суперклассах. Это основной способ, которым подклассы отличаются от своих суперклассов. Также могут быть ситуации, когда нежелательно наследовать характеристики. В таких случаях возможно переопределить или заменить что-то в подклассах, что было бы унаследовано от суперклассов. Наследование — сложная тема для обсуждения из-за гибкости Java в использовании своих различных возможностей.

Иерархия наследования может сначала показаться незнакомым понятием. Фактически это общепринятый способ классификации вещей. Отношения между компонентами в такой иерархии могут быть описаны как отношения "является одним из" или "является видом". Другими словами, все, что появляется ниже в иерархии, должно быть примером или более специфичным видом какого-либо класса, который появился выше него в иерархии. В сфере биологии классификация всех живых существ основана на этой идее. Это называется таксономией, и таксономия человеческих существ показана на рис. 12.1. Это упрощенная классификация, взятая из следующего источника: **http://anthro.palomar.edu/animal/table_humans.htm**.

Тип хордовые является подразделением мира животных. Есть животные со спинным мозгом и есть животные без спинного моз-

га. Однако, если какое-либо существо классифицируется как член типа хордовых, оно, разумеется, является животным и имеет все характеристики мира животных. Подобные отношения существуют по всей иерархии. Если некое существо является млекопитающим, оно обладает всеми характеристиками типа хордовых и мира животных. Каждая ступенька вниз по иерархии ведет к более узкому подразделению.

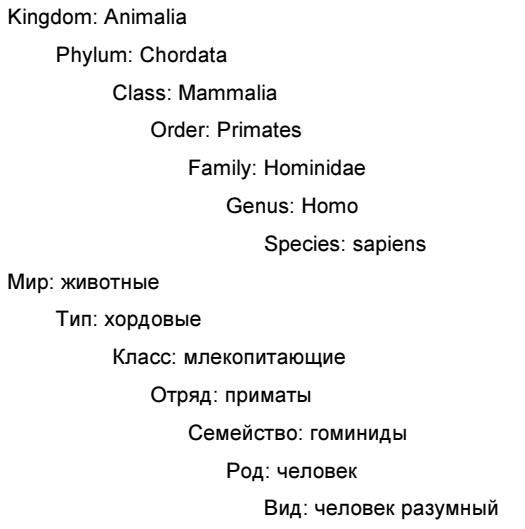


Рис. 12.1

Различные использования термина "наследование" могут потенциально привести к путанице. Например, мы говорим о наследовании, когда речь идет о чертах, которые ребенок получает от своего родителя. Таксономический пример основан на характеристиках категорий, а не индивидуумов. Существуют экземпляры человека разумного, т. е. отдельные люди. Человек будет родительским классом для человека разумного в иерархии наследования, но он не будет являться биологическим родителем конкретных людей.

Примеры иерархий наследования существуют по всей документации Java API. Если вы найдете класс `Ellipse2D`, вверху

страницы документации будет представлен фрагмент иерархии (рис. 12.2).

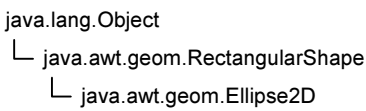


Рис. 12.2

Как обычно, первые буквы имени класса — прописные. Пока не будем обращать внимание на имена пакетов, `java.lang` и `java.awt.geom`; остальная информация означает следующее: класс `Object` является родительским классом для `RectangularShape`, а `RectangularShape` является родительским классом для `Ellipse2D`. Может показаться немного странным то, что эллипсы происходят из прямоугольных форм, но это основано на том, что место, размер и форма эллипса определены параметрами ограничивающего прямоугольника. Эллиптическая форма может унаследовать переменные экземпляра для этих параметров из прямоугольной формы. Класс также наследует методы от своего суперкласса. В документации для класса `Ellipse2D` после описания переменных, конструкторов и методов данного класса вы найдете раздел с методами, унаследованными от класса `java.awt.geom.RectangularShape`.

Среди других методов, перечисленных здесь, вы найдете такие примеры: `getCenterX()`, `getCenterY()`. Центр эллипса может быть найден через центр прямоугольника, ограничивающего его. Поскольку такие методы уже существуют для класса `RectangularShape`, нет необходимости записывать их снова для класса `Ellipse2D`. Они могут наследоваться. Одним из преимуществ объектно-ориентированного языка с наследованием является то, что код может таким образом использоваться вновь.

В документации класса `Ellipse2D` вы также найдете раздел, описывающий методы, унаследованные из класса `java.lang.Object`. Среди перечисленных здесь методов есть метод `equals()`. Если класс не выполняет метод `equals()` сам, он наследует этот метод из класса `Object`. Этот наследованный метод проверяет равенство

ссылки, а не равенство содержимого. Если требуется проверить равенство содержимого, то метод с таким же именем должен быть выполнен в подклассе.

В этом разделе документации вы также увидите принятые в Java полные имена пакетов, такие как `java.lang.Object` и `java.awt.geom.RectangularShape`. Классы в Java группируются вместе и хранятся в разных пакетах. Если вы посмотрите гипертекстовую версию документации, то в левом верхнем углу вы увидите перечень классов, сгруппированных по пакетам. В общем, классы, которые тесно соотносятся каким-то образом, упаковываются вместе. Однако важно понять, что эти пакеты используются просто для удобства и не представляют подклассовые или суперклассовые отношения между классами.

12.2. Написание кода для подклассов

Легче всего объяснить процесс написания подклассов на примере. Не все аспекты наследования можно проиллюстрировать, используя чашки и семена; поэтому будет введен новый класс. Этот класс представляет различные наименования продуктов с точки зрения магазина, где они продаются. Переменными экземпляра для этого класса являются `name` (название), `wholesaleCost` (оптовая цена) и `markup` (наценка). Например, объект этого класса может иметь значение переменной экземпляра Del Monte 12 oz. canned corn, .99, 05 (консервированная кукуруза Дель Монте, 12 унций). Имя класса — `FoodV1.V1` в названии класса означает первую версию класса. Этот пример будет продолжен и далее, и имена последующих версий будут `V2` и `V3`. Данный класс не использует ключевое слово `extends`. Это означает, что по умолчанию он является подклассом класса `Object`. В первом примере для нас наиболее важны переменные экземпляра и методы. Конструкторы будут подробно обсуждаться позднее. Рассмотрим код для этого класса:

```
public class FoodV1
{
  private String name;
  private double wholesaleCost;
```

```
  private double markup;
  public FoodV1()
  {
  }
  public void setName(String nameIn)
  {
    name = nameIn;
  }
  public String getName()
  {
    return name;
  }
  public void
    setWholesaleCost(double wholesaleCostIn)
  {
    wholesaleCost = wholesaleCostIn;
  }
  public double getWholesaleCost()
  {
    return wholesaleCost;
  }
  public void setMarkup(double markupIn)
  {
    markup = markupIn;
  }
  public double getMarkup()
  {
    return markup;
  }
  public double getPrice()
  {
    return (wholesaleCost + wholesaleCost * markup);
  }
}
```

Можно создать подкласс этого класса, например `TaxedFoodV1`. Рассмотрим схему (рис. 12.3), иллюстрирующую отношения наследования между этими классами и классом `Object`.

Ключевым словом при определении подкласса является `extends`. Удобно синтаксически зафиксировать отношения между класса-

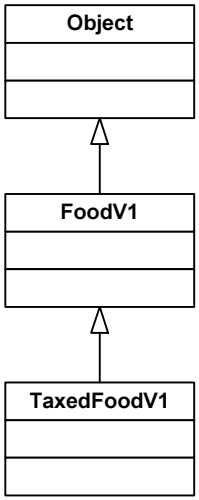


Рис. 12.3

ми за счет того, чтобы потомок указывал своего родителя, а не наоборот, родители указывали бы своих потенциальных детей. Этим также объясняется направление стрелок на схеме. Подкласс "знает" своего родителя. Родительский класс "не знает", кто его дети. Рассмотрим код для класса `TaxedFoodV1`, который является потомком класса `FoodV1`:

```
public class TaxedFoodV1 extends FoodV1
{
  private double taxRate;
  public TaxedFoodV1()
  {
  }
  public void setTaxRate(double taxRateIn)
  {
    taxRate = taxRateIn;
  }
  public double getTaxRate()
  {
    return taxRate;
  }
}
```

Первая строка объявления класса показывает использование ключевого слова `extends` в синтаксисе для создания подкласса:

```
public class TaxedFoodV1 extends FoodV1
```

Класс `TaxedFoodV1` достаточно прост. У него одна собственная переменная экземпляра. Он также наследует все переменные экземпляра класса `FoodV1`. Это значит, что объект данного класса имеет следующие переменные экземпляра:

- `name` — унаследовано;
- `wholesaleCost` — унаследовано;
- `markup` — унаследовано;
- `taxRate` — не унаследовано.

Класс `TaxedFoodV1` имеет конструктор по умолчанию. Создание экземпляра этого класса ничем не отличается от создания экземпляра любого другого класса:

```
TaxedFoodV1 mytaxedFood = new TaxedFoodV1();
```

Класс `TaxedFoodV1` имеет методы `set` и `get` для переменной `taxRate`, определяемой в классе. Он также наследует все методы класса `FoodV1`. Пусть объект `mytaxedfood` был построен, тогда можно свободно осуществлять вызовы к этим унаследованным методам:

```
mytaxedFood.setname("Peas");
double retrievedCost =  mytaxedFood.getWholesaleCost();
double retrievedPrice = mytaxedFood.getPrice();
```

Конструкторы в обоих классах полагаются на инициализацию переменных экземпляра по умолчанию. Это значит, что строки инициализируются `null`, числовые переменные — 0, а булевы переменные — `false`. Попытка сделать иначе приведет к сложностям, которые лучше отложить на потом. Поскольку имеются методы `set` для всех этих переменных, объекты могут свободно использоваться, и их переменным могут присваиваться значения, хотя классы и включают только конструкторы по умолчанию.

12.3. Доступ к переменным экземпляра и переопределение методов

Как переменные экземпляра, так и методы наследуются. В программе, где используются экземпляры подклассов, можно работать с переменными экземпляра объекта путем использования методов доступа (`get`) и модифицирующих методов (`set`), которые также унаследованы от суперкласса. В коде, написанном в самом подклассе, необходимо использовать наследованные методы `get` и `set` для доступа к наследованным переменным экземпляра. Эти наследованные переменные не являются непосредственно доступными в подклассе, поскольку они объявлены частными (`private`) в суперклассе.

По определению подклассы более подробны по своей природе, чем их суперклассы. Они могут включать новые, дополнительные переменные экземпляра и методы. Так как подклассы становятся более специфичными, возможно, что унаследованный метод не полностью выполняет то, что должно быть выполнено. Мы видели, что в одном классе можно иметь более одного метода с одним и тем же именем, и эти методы различаются перечнями своих параметров. Это называется *перегрузкой*. Также возможно, чтобы в подклассе был метод с тем же именем, что и метод в одном из его суперклассов, с тем же перечнем параметров. Это называется *переопределением* унаследованного метода.

В случае переопределения метода в подклассе, если сделан вызов объекту подкласса с помощью имени метода, используется версия метода, определенного в подклассе, а не версия суперкласса. Иначе говоря, в этом случае наследование не происходит. Также можно написать метод в подклассе с тем же именем, которое имеет метод в суперклассе, но с иным списком параметров. В таком случае, не происходит переопределение унаследованного метода. Это перегрузка. Список параметров, используемых в вызове, определяет, будет ли использована версия метода, определенного в суперклассе, или версия, определенная в подклассе. Обратите внимание на то, что можно спутать перегрузку и переопределение. При перегрузке у вас есть разные методы с одинаковым именем, которые различаются списком параметров. При

переопределении у вас есть разные методы с тем же именем и одинаковым списком параметров, которые различаются местом в иерархии наследования.

Рассмотрим пример, который иллюстрирует и переопределение метода, и тот факт, что внутри подкласса нет прямого доступа к переменным, унаследованным от суперкласса. Метод `getPrice()`, как он определен в классе `FoodV1`, подсчитывает цену для покупателей, основанную на оптовой цене наименования и наценке магазина. Однако цена для покупателей на продукты, облагаемые налогом, должна включать налог на этот продукт. Следующее — это не код Java, это псевдокод, который иллюстрирует общее вычисление, необходимое, чтобы найти цену наименования после того, как в нее включены наценка магазина и налог:

```
costPlusMarkup = wholesaleCost + wholesaleCost * markup;
price = costPlusMarkup + costPlusMarkup * taxRate;
```

Вот пример того, как метод может быть правильно применен в классе `TaxedFoodV1`:

```
public double getPrice()
{
  double retrievedCost;
  double retrievedMarkup;
  double costPlusMarkup;
  double price;
  retrievedCost = getWholesaleCost();
  retrievedMarkup = getMarkup();
  costPlusMarkup = retrievedCost + retrievedCost *
    retrievedMarkup;
  price = costPlusMarkup + costPlusMarkup * taxRate;
  return price;
}
```

Обратите внимание на эти строки кода в применении метода:

```
retrievedCost = getWholesaleCost();
retrievedMarkup = getMarkup();
```

Вызовы могут быть написаны с использованием скрытого параметра. В таком случае, код может быть понятнее. Полученные значения исходят от того объекта, к которому был вызван метод `getPrice()`:

```
retrievedCost = this.getWholesaleCost();
retrievedMarkup = this.getMarkup();
```

Эти методы объявляются общими (`public`) в суперклассе, поэтому они свободно могут быть вызваны в коде подкласса. В этом примере важно видеть следующее:

- скрытый параметр вызова `getPrice()`, объект `TaxedFoodV1`, наследует переменные `wholesaleCost` и `markup`. Однако доступ к ним возможен только с помощью унаследованных `get`-методов;
- иногда желательно записать реализацию данного метода в подклассе, отличающуюся от той, которая была бы унаследована. В нашем примере метод `getPrice()` класса `FoodV1` переопределен в подклассе.

Пусть построены следующие объекты:

```
FoodV1 myfood = new FoodV1();
TaxedFoodV1 yourfood = new TaxedFoodV1();
```

Также предположим, что `set`-методы были вызваны, так что переменные `price`, `markup` и `taxRate` были установлены. Теперь рассмотрим следующие два вызова:

```
double x = myfood.getPrice();
double y = yourfood.getPrice();
```

Идея переопределения заключается в том, что два случая метода `getPrice()` в вызовах выше являются двумя разными версиями метода. Какая версия вызывается, зависит от того, каков тип объекта, к которому вызывается метод: необлагаемые или облагаемые налогом товары.

Возникает еще один вопрос, касающийся наследования и переопределения методов. Рассмотрим общую иерархию наследования, представленную на рис. 12.4.

Если `methodX()` реализован в классе `Top` и не реализуется в других двух классах, тогда он наследуется как классом `Middle`, так и классом `Bottom`. Если `methodX()` реализован в `Middle`, тогда он просто не будет существовать для объектов класса `Top`. Наследование не идет вверх. Однако этот метод наследуется классом

Bottom. Теперь предположим, что methodX() реализуется в классе Top, переопределяется в классе Middle и наследуется классом Bottom. Какая из версий метода наследуется? Ответ: версия, самая близкая к классу Bottom, когда вы идете вверх по иерархии. Иными словами, в этом случае наследуется версия, выполненная в классе Middle. Наследование идет только вниз. Когда система ищет методы, которые были унаследованы, она просматривает вверх и останавливает просмотр после того, как находит близлежащий родительский класс, содержащий эти методы.

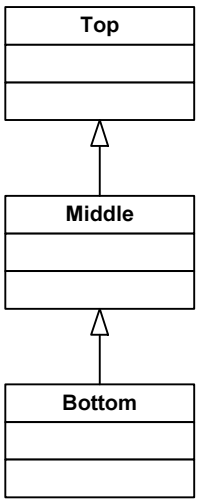


Рис. 12.4

Последний вопрос возникает относительно переопределения. Понятно, что унаследованные методы могут быть переопределены. Можно ли как-нибудь переопределить переменные экземпляра? Ответ прост: нет. Синтаксически можно объявить переменные экземпляра в подклассах с тем же названием, что и переменные в суперклассах. Однако переопределение не происходит. Мы просто получаем различные переменные с теми же названиями, определенные в различных местах. Поступить так будет ошибкой. Не только будет чрезвычайно трудно разобраться в коде; настоящей причиной ошибки является то, что такой подход концепту-

ально неверен: идея наследования заключается в том, что подклассы — это разновидность суперклассов. Методы могут переопределяться, но переменные экземпляра должны представлять более неотъемлемые характеристики классов. Если возникает необходимость изменить переменные экземпляра в подклассе, то это говорит о том, что данный класс не является настоящим подклассом того суперкласса, ниже которого он находится в иерархии.

12.4. Ключевое слово *super*, методы и конструкторы

Вы могли заметить в предыдущем разделе, что при переопределении метода `getPrice()` большинство строк кода просто дублировали последовательность переопределяемого метода. Необходимо было получить цену, включающую наценку, и затем применить к ней налоговый коэффициент. Но нахождение цены с включенной наценкой — это именно то, что делает метод `getPrice()` в суперклассе. Пример далее показывает, как можно использовать метод, определенный выше в иерархии, для того чтобы выполнить переопределение метода, находящегося ниже в иерархии:

```
public double getPrice()
{
  double price;
  price = super.getPrice();
  price = price + price * taxRate;
  return price;
}
```

Во время исполнения ключевое слово `super` сигнализирует о том, что вызов `getPrice()` не является рекурсивным вызовом. Система ищет в иерархии самый близлежащий класс выше текущего класса, в котором выполняется `getPrice()`, и использует версию метода, найденную там.

Ключевое слово `super` также необходимо при написании конструкторов подкласса, которые принимают параметры. Причина, по

которой эта тема сложна, заключается в том, что конструкторы в подклассе также не имеют прямого доступа к переменным экземпляра, унаследованным от суперкласса. Без нового синтаксиса невозможно написать конструктор для класса `TaxedFoodV1`, который должен иметь параметры для переменных `wholesaleCost` и `markup`, и успешно применять эти параметры для инициализации наследованных переменных.

Можно написать классы вовсе без конструкторов. Если это сделано, то система автоматически выдает невидимые конструкторы по умолчанию, которые не принимают параметров и выполняют инициализацию по умолчанию любых переменных экземпляра. Так же можно поступить, как и в предыдущих примерах, а именно написать видимый конструктор по умолчанию, который не имеет параметров и не содержит никакого кода и, таким образом, полагается на инициализацию по умолчанию. Результат будет тот же самый, как и в случае написания класса вовсе без конструкторов, но эта процедура может быть более понятной. Наконец, можно написать конструктор, который не принимает никаких параметров, но содержит код для инициализации переменных. Важно помнить, что, как только программист создаст какой-нибудь конструктор для класса, система не предоставит конструктор по умолчанию. В такой ситуации (когда уже создан некий конструктор), если программисту затем потребуется конструктор, который не принимает параметров, ему придется создать его.

Пока мы использовали только конструкторы по умолчанию. Конструкторы по умолчанию подкласса предполагают существование конструктора по умолчанию в суперклассе. Инициализация переменных, в том числе унаследованных, производится автоматически. После этого, при желании, можно использовать `set`-методы для сохранения значений переменных, отличающихся от значений по умолчанию. Теперь изменим пример так, чтобы использовать конструктор с параметрами. Изменения будут достаточно большими, так что классы в данном разделе будут названы `V2` (т. е. вторая версия). Примеры будут иллюстрировать различные участки кода для данной версии (с целью экономии места, мы пока не будем приводить полный вариант кода). В следующем разделе будет показан полный вариант кода с внесенными

дополнительными изменениями. Вот как будет выглядеть конструктор для класса `FoodV2`, который принимает полный набор параметров для своих переменных:

```
public FoodV2(String nameIn,
              double wholesaleCostIn, double markupIn)
{
  name = nameIn;
  wholesaleCost = wholesaleCostIn;
  markup = markupIn;
}
```

Следовательно, вот как будет выглядеть правильный вид конструктора следующего класса, который принимает полный набор параметров, включая параметры для унаследованных переменных:

```
public TaxedFoodV2(String nameIn,
                   double wholesaleCostIn, double markupIn,
                   double taxRateIn)
{
  super(nameIn, wholesaleCostIn, markupIn);
  taxRate = taxRateIn;
}
```

Когда система встречает вызов к `super()` в конструкторе, она начинает искать конструктор выше его в данной иерархии. Она будет использовать ближайший конструктор в иерархии выше его, который имеет список параметров, совпадающий с параметрами в вызове. Если вы явно вызываете метод `super()` в конструкторе подкласса, этот вызов должен быть первой строкой кода. Если вы явно не вызываете `super()`, тогда полагайтесь на конструктор суперкласса по умолчанию. Однако не забывайте, что если суперкласс содержит какой-либо другой конструктор, то тогда задачей программиста будет предоставление нужного конструктора по умолчанию. Иными словами, вы получите ошибку компилятора со следующим сценарием: суперкласс содержит конструктор, который принимает параметры, но не явный конструктор по умолчанию; подкласс содержит конструктор, который не вызывает `super()`.

12.5. Ссылки на классы и подклассы в программах и методах

Использование и тестирование подклассов не сильно отличается от использования и проверки любых классов, написанных программистами. Создавая иерархию классов, вы должны идти сверху вниз. Подклассы могут быть скомпилированы только при условии, что существуют их суперклассы. Программа тестирования на любом уровне должна включать код, который используется всеми конструкторами и методами в классе для того, чтобы быть уверенным, что они функционируют правильно. Прежде чем писать подкласс, необходимо проверить его суперкласс. Если это не сделано, то можно встретить проблемы при написании и тестировании подкласса, которые фактически вызваны ошибками в суперклассе.

При работе с иерархией классов объектные ссылки получают новый аспект. Ссылка на объект подкласса всегда может быть присвоена ссылке, объявленной тем типом, который имеет любой из суперклассов подкласса. Это основано на общей идее наследования: экземпляры классов ниже по иерархии являются разновидностью вышестоящих классов. Аналогичным образом, люди являются членами мира животных, облагаемые налогом продукты являются видом продуктов. Синтаксически эту идею можно проиллюстрировать следующим образом:

```
FoodV2 myfood;
TaxedFoodV2 mytaxedFood = new TaxedFoodV2("bread",
  .50, .05, .07);
myfood = mytaxedFood;
```

Объект подкласса может быть присвоен ссылке суперкласса без какого-то особого синтаксиса. Это и логически, и синтаксически правильно. В данном примере объектом являются продукты питания, облагаемые налогом, но это не мешает им принадлежать к продуктам питания в целом, и это присвоение разрешено сделать без преобразования. Важно отметить, однако, что ссылка суперкласса на объект подкласса имеет четкие ограничения. Можно вызывать только методы, определенные в суперклассе, или мето-

ды, наследованные в суперклассе по этой ссылке. Невозможно вызвать методы, определенные в данном подклассе, по ссылке суперкласса.

Хотя можно выполнить присваивание ссылки подкласса ссылке суперкласса без преобразования, идти в другом направлении не всегда возможно. Преобразование может быть использовано для того, чтобы восстановить данный класс, к которому относится ссылка суперкласса. Вот приводившийся ранее код с добавлением одной строки:

```
FoodV2 myfood;
TaxedFoodV2 mytaxedFood = new TaxedFoodV2("bread",
  .50, .05, .07);
myfood = mytaxedFood;
TaxedFoodV2 newtaxedFood = (TaxedFoodV2) myfood;
```

В последней строке, справа, `myfood` является ссылкой суперкласса к объекту подкласса. Система знает о том, на какой вид объекта указывает ссылка, и можно преобразовать эту ссылку назад к первоначальному типу и восстановить ее в новой ссылке этого типа.

Такой тип преобразования не станет работать, если тип слева не будет являться тем же типом, что и существующий объект справа. Вот грубый пример того, что невозможно:

```
 /* НЕТ! НЕТ! НЕТ! */
Ellipse2D.Double myellipse =
  new Ellipse2D.Double(10, 20, 30, 40);
TaxedFoodV2 mytaxedfood = (TaxedFoodV2) myellipse;
```

Должно быть очевидным, что было бы нелогичным пытаться привести эллипс к наименованию продукта. Следующее также невозможно:

```
/*  НЕТ!  НЕТ!  НЕТ!  */
FoodV2 myfood = new FoodV2("cheese", 1.25, .06);
TaxedFoodV2 mytaxedFood = (TaxedFoodV2) myfood;
```

Даже хотя `myfood` и `mytaxedFood` находятся в той же наследственной иерархии, в данном примере `myfood` никогда не было продуктом, облагаемым налогом (taxed food). Невозможно использовать преобразование для того, чтобы восстановить вид ссылки, кото-

рый никогда не существовал. В общем, нельзя преобразовывать суперклассы в подклассы, кроме тех случаев, где вы восстанавливаете тип первоначальной ссылки.

Это приводит к использованию другого ключевого слова, которое позволяет проверить, принадлежит ли ссылка к какому-либо классу, перед тем как пытаться преобразовать его. Ключевое слово `instanceof` функционирует как оператор, который возвращает булево значение. Оно сравнивает существующий тип ссылки с данным классом. Фрагмент далее иллюстрирует его использование для разрешения допустимого преобразования и предотвращения ошибочного преобразования:

```
if(myfood instanceof TaxedFoodV2)
  TaxedFoodV2 newtaxedFood = (TaxedFoodV2) myfood;
```

Вопрос о типе ссылки становится решающим во время передачи объектных ссылок в качестве параметров в наследственной иерархии. Объекту подкласса всегда может быть присвоена ссылка суперкласса. Отсюда следует, что когда устанавливается ссылка суперкласса, возможно использование объекта или ссылки подкласса вместо нее. Рассмотрим новый метод для класса `Food`. Пусть это будет следующим изменением нашего примера с этим классом, так что названия классов теперь включают `V3` (версия 3) в конце. Можно установить наценку одного из наименований продуктов по наценке, уже включенной в другом продукте. Для этого передается объект, содержащий наценку, в качестве явного параметра к методу. Параметр определен как `FoodV3`, так же как и класс, в котором появляется метод.

```
public void
  setMarkupToThisMarkup(FoodV3 anotherFood)
{
  double tempMarkup = anotherFood.getMarkup();
  setMarkup(tempMarkup);
}
```

Теперь предположим, что существует следующий код в программе, которая использует этот набор классов:

```
TaxedFoodV3 mytaxedFood = new TaxedFoodV3("bread",
  .50, .05, .07);
```

```
TaxedFoodV3 yourtaxedFood = new TaxedFoodV3("milk",
  .75, .08, .07);
mytaxedFood.setMarkupToThisMarkup(yourtaxedFood);
```

Верен ли вызов в третьей строке? Метод наследуется подклассом. Таким образом, можно выполнить вызов в `mytaxedFood`. Очевидная проблема заключается в том, что типом явного параметра `yourtaxedFood` является `TaxedFoodV3`, в то время как в определении метода тип ожидаемого параметра — `FoodV3`. В других ситуациях, когда фактический параметр не согласовывался с типом формального параметра, компилятор обнаруживал ошибку. Однако в этом случае проблем не возникает. При передаче параметра фактический параметр придается формальному параметру. Допускается придать объект подкласса ссылке суперкласса без преобразования. Система знает, что `TaxedFoodV3` является подклассом `FoodV3`, так что когда действительный параметр, облагаемый налогом продукт питания, передается, эта ссылка автоматически преобразуется в ссылку `FoodV3`.

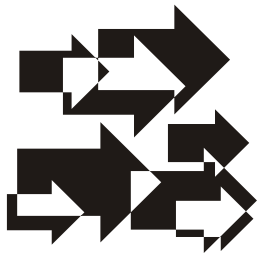
Рассмотрим пример классов со всеми изменениями, обсуждавшимися ранее:

```
public class FoodV3
{
  private String name;
  private double wholesaleCost;
  private double markup;
  public FoodV3(String nameIn,
                double wholesaleCostIn, double markupIn)
  {
    name = nameIn;
    wholesaleCost = wholesaleCostIn;
    markup = markupIn;
  }
  public void setName(String nameIn)
  {
    name = nameIn;
  }
  public String getName()
  {
    return name;
  }
```

```
  public void
    setWholesaleCost(double wholesaleCostIn)
  {
    wholesaleCost = wholesaleCostIn;
  }
  public double getWholesaleCost()
  {
    return wholesaleCost;
  }
  public void setMarkup(double markupIn)
  {
    markup = markupIn;
  }
  public double getMarkup()
  {
    return markup;
  }
  public double getPrice()
  {
    return (wholesaleCost + wholesaleCost * markup);
  }
  public void
    setMarkupToThisMarkup(FoodV3 anotherfood)
  {
    double tempMarkup = anotherfood.getMarkup();
    setMarkup(tempMarkup);
  }
}
public class TaxedFoodV3 extends FoodV3
{
  private double taxRate;
  public TaxedFoodV3(String nameIn,
                     double wholesaleCostIn, double markupIn,
                     double taxRateIn)
  {
    super(nameIn, wholesaleCostIn, markupIn);
    taxRate = taxRateIn;
  }
```

```
  public void setTaxRate(double taxRateIn)
  {
    taxRate = taxRateIn;
  }
  public double getTaxRate()
  {
    return taxRate;
  }
  public double getPrice()
  {
    double price;
    price = super.getPrice();
    price = price + price * taxRate;
    return price;
  }
}
```

ГЛАВА 13



Абстрактность, полиморфизм и интерфейсы

13.1. Абстрактность в конструировании классов

Отношения между классами в иерархии наследования являются важной темой. Создание иерархии включает в себя определение того, на каком уровне должны реализовываться конкретные характеристики, и должно или не должно быть разрешено создание экземпляров определенных классов. Общим принципом при создании наследственной иерархии является то, что атрибуты и операции должны быть реализованы на самом высоком возможном уровне.

Еще одним аспектом абстракции является вопрос о том, должны ли некоторые классы иметь экземпляры или нет. Давайте снова рассмотрим таксономическую иерархию (см. рис. 12.1).

В животном мире каждое животное может быть в конечном счете рассмотрено как отдельный вид. Пользуясь терминологией экземпляров в Java, можно сказать, что существуют только экземпляры видов, таких как человеческие существа, собаки, божьи коровки и т. д. Не существует экземпляров класса `Animalia`, потому что нет такого вида, как животное вообще: каждое животное принадлежит определенному виду.

Снова рассмотрим иерархию классов из предыдущей главы. Имеет ли смысл создавать объекты класса `Food` или только объекты

классов `PackagedFood` и `BulkFood`? Рассуждая логически, можно предположить, что ответ на этот вопрос зависит от ответа на другой. Когда вы идете в магазин покупать продукты, возможно ли купить их ни упакованными, ни продаваемыми вразвес, без упаковки? Если нельзя, тогда не существует и такого, как экземпляр общего класса `Food`. Существуют либо экземпляры класса `PackagedFood`, либо экземпляры класса `Bulkfood`.

Идея о том, что некоторые классы не должны иметь экземпляров, заложена непосредственно в коде языка Java. Предположим, что вы хотите, чтобы все классы в иерархии "продукты" использовали метод `getUnitCost()`. В принципе было бы желательно реализовать метод, общий для всех классов в иерархии класса `Food`, на вершине этой иерархии. Однако поскольку цена за единицу товара вычисляется по-разному для двух подклассов, то в каждом из них метод будет переопределяться. Поэтому возникает вопрос, как указать, что такой метод существует во всех классах, и при этом иметь отдельную реализацию этого метода в каждом классе?

Возможно несколько вариантов ответа на такой вопрос, за счет реализации метода `getUnitCost()` в суперклассе. Такие решения далеко не идеальны, но синтаксически верны. Например, вы можете реализовать там одну из версий подкласса, и тогда нужно будет переопределить метод только во втором подклассе. Или можно объявить метод с пустым телом, не содержащим никакого кода. Эти решения нежелательны по той причине, что такой метод, реализованный в суперклассе, не может быть успешно или правильно вызван к объектам суперкласса.

Решение данной проблемы заключается в том, чтобы сделать метод в суперклассе абстрактным методом. Это аналогично написанию пустого метода. В нашем примере метод появится в классе `Food` в следующем виде:

```
public abstract double getUnitCost();
```

Запись абстрактного метода не имеет ни фигурных скобок, ни тела и не содержит никакого кода, но указывает возвращаемый тип и список параметров. Объявление абстрактного метода имеет три главных последствия:

- это требует переопределения метода в каждом из непосредственных подклассов;

- если класс содержит абстрактный метод, класс также должен быть объявлен абстрактным;
- невозможно построить экземпляры абстрактного класса.

В нашем примере класс должен быть объявлен следующим образом:

```
public abstract class Food
```

В иерархии наследования может появиться класс, единственной целью которого является объединение набора атрибутов и операций, которые должны быть общими для подклассов. Если нет оснований создавать экземпляры такого класса, то концептуально он является абстрактным. Данный пример показал, что если метод должен быть определен в классе, тогда и метод, и класс должны быть объявлены абстрактными. Невозможно вызвать конструкторы абстрактного класса в программе, написанной пользователем. Однако абстрактный класс может содержать конструкторы, которые используются конструкторами подклассов.

Далее приводится полная иерархия класса `Food`, переписанная так, чтобы использовать идею абстрактных методов и абстрактных классов, описанных выше:

```
public abstract class FoodV5
{
  private String brand;
  private String productName;
  private String units;
  public FoodV5(String brandIn,
                String productNameIn, String unitsIn)
  {
    brand = brandIn;
    productName = productNameIn;
    units = unitsIn;
  }
  public void setBrand(String brandIn)
  {
    brand = brandIn;
  }
```

```
  public String getBrand()
  {
    return brand;
  }
  public void setProductName(String productNameIn)
  {
    productName = productNameIn;
  }
  public String getProductName()
  {
    return productName;
  }
  public void setUnits(String unitsIn)
  {
    units = unitsIn;
  }
  public String getUnits()
  {
    return units;
  }
  public void
    setBrandToThisBrand(FoodV5 anotherFood)
  {
    String tempBrand = anotherFood.getBrand();
    setBrand(tempBrand);
  }
  public abstract double getUnitCost();
}
public class PackagedFoodV5 extends FoodV5
{
  private double size;
  private double itemCost;
  public PackagedFoodV5(String brandIn,
                        String productNameIn, String unitsIn,
                        double sizeIn, double itemCostIn)
  {
    super(brandIn, productNameIn, unitsIn);
    size = sizeIn;
    itemCost = itemCostIn;
  }
```

```
  public void setSize(double sizeIn)
  {
    size = sizeIn;
  }
  public double getSize()
  {
    return size;
  }
  public void setItemCost(double itemCostIn)
  {
    itemCost = itemCostIn;
  }
  public double getItemCost()
  {
    return itemCost;
  }
  public double getUnitCost()
  {
    return itemCost / size;
  }
}
public class BulkFoodV5 extends FoodV5
{
  private double unitCost;
  public BulkFoodV5(String brandIn,
                    String productNameIn, String unitsIn,
                    double unitCostIn)
  {
    super(brandIn, productNameIn, unitsIn);
    unitCost = unitCostIn;
  }
  public void setUnitCost(double unitCostIn)
  {
    unitCost = unitCostIn;
  }
  public double getUnitCost()
  {
    return unitCost;
  }
```

```
  public double findCost(double amount)
  {
  return amount * unitCost;
  }
}
```

13.2. Полиморфизм и динамическая диспетчеризация

Можно иметь ссылку суперкласса на объект подкласса. Любая данная ссылка может относиться к объекту своего класса или к объектам любого из его подклассов. Тип ссылки не указывает, на какой тип объекта она ссылается. Термин *"полиморфизм"* происходит из двух греческих корней, "много" и "форма". Этот термин используется для того, чтобы обобщить идею о том, что ссылка может указывать на различные виды объектов.

Предположим, что каждый класс в иерархии наследования переопределяет данный метод. Если метод вызывается ссылкой суперкласса, то какую версию метода будет использовать система? Термин *"динамическая диспетчеризация"* означает, что решение о том, какую версию использовать, будет основываться на фактическом типе объекта, на который указывает ссылка во время выполнения, а не на типе ссылки, заявленной в коде. Если метод вызван к ссылке суперкласса на объект подкласса, то будет использоваться та версия метода, которая действительна для этого подкласса. Если метод переопределяется в подклассе, тогда вызов делается к той версии метода, которая определена в подклассе. Если метод не переопределен в подклассе, то используется наследованная им версия.

Иерархии наследования создаются за счет добавления новых переменных экземпляра и методов к подклассам. Ссылки суперклассов имеют ограниченный доступ к объектам подкласса. Ссылка суперкласса не может "знать" о тех переменных или методах, которые определены ниже в иерархии. Если метод не определен в суперклассе, а определен только в подклассе, то невозможно вызвать этот метод по ссылке суперкласса к объекту подкласса.

Снова посмотрим на классы из предыдущего раздела, но с пропусками тех частей, которые не нужны для обсуждения данного вопроса. После кода полиморфизм и динамическая диспетчеризация иллюстрируются на примерах этих классов.

```
public abstract class FoodV5
{
  private String brand;
  private String productName;
  private String units;
  ...
  public void setBrand(String brandIn)
  {
    brand = brandIn;
  }
  public String getBrand()
  {
    return brand;
  }
  ...
  public void
    setBrandToThisBrand(FoodV5 anotherFood)
  {
    String tempBrand = anotherFood.getBrand();
    setBrand(tempBrand);
  }
  ...
  public abstract double getUnitCost();
}
public class PackagedFoodV5 extends FoodV5
{
  private double size;
  private double itemCost;
  ...
  public void setSize(double sizeIn)
  {
    size = sizeIn;
  }
```

```
  public double getSize()
  {
    return size;
  }
  ...
  public double getUnitCost()
  {
    return itemCost / size;
  }
}
public class BulkFoodV5 extends FoodV5
{
  private double unitCost;
  ...
  public void setUnitCost(double unitCostIn)
  {
    unitCost = unitCostIn;
  }
  public double getUnitCost()
  {
    return unitCost;
  }
  ...
}
```

Сначала рассмотрим вызов метода подкласса по ссылке суперкласса. Предположим, у вас есть ссылка класса `FoodV5` на объект `PackageFoodV5`, как показано ниже:

```
FoodV5 foodReference;
PackagedFoodV5 packagedFoodReference =
  new PackagedFoodV5(...);
foodReference = packagedFoodReference;
```

Следующие вызовы невозможны, т. к. методы не определены в суперклассе:

```
/*  НЕТ!  НЕТ!  НЕТ!  */
foodReference.setSize(...);
...foodReference.getSize();
```

То же самое справедливо и для ссылки класса `FoodV5` на объект `BulkFoodV5` и методы `setUnitCost()` и `getUnitCost()`. Если ссылка

подкласса восстанавливается, то такие вызовы становятся возможными:

```
PackagedFoodV5 recovered =
  (PackagedFoodV5) foodReference;
recovered.setSize(...);
```

Теперь предположим, что у вас есть ссылка суперкласса на объект подкласса, и вы хотите вызвать метод `setBrandToThisBrand()` к нему:

```
FoodV5 foodReference;
PackagedFoodV5 packagedFoodReference =
  new PackagedFoodV5(...);
foodReference = packagedFoodReference;
foodReference.setBrandToThisBrand(...);
```

Вызов по ссылке суперкласса разрешен, поскольку данный метод определен для суперкласса. Метод не переопределяется в подклассе, поэтому в подклассе используется унаследованная от суперкласса версия.

Посмотрим на вызов этого метода по ссылке подкласса в качестве явного параметра:

```
PackagedFoodV5 packagedFoodReference =
  new PackagedFoodV5(...);
...setBrandToThisBrand(packagedFoodReference);
```

Ссылка подкласса может быть передана, когда формальным параметром является ссылка суперкласса. Происходит автоматическое преобразование. Посмотрите еще раз на код для метода, где `anotherFood` становится ссылкой суперкласса к явному параметру, `packagedFoodReefernce`:

```
public void setBrandToThisBrand(FoodV5 anotherFood)
{
  String tempBrand = anotherFood.getBrand();
  setBrand(tempBrand);
}
```

В данном методе явный параметр используется в этой строке кода:

```
String tempBrand = anotherFood.getBrand();
```

Поскольку метод `getBrand()` определен в суперклассе и только в суперклассе, вызов его по ссылке суперкласса к объекту подкласса просто использует метод в том виде, как он определен в суперклассе.

Предположим, вы хотите использовать метод `setUnitCostToThisUnitCost()`. Класс `BulkFoodV5` имеет переменную экземпляра `UnitCost`, которой может быть приписано значение, в то время как класс `PackagedFoodV5` только возвращает вычисленное значение для стоимости единицы товара. Это значит, что метод может быть вызван только к объекту класса `BulkFoodV5`.

Явным параметром метода может быть или одна из ссылок `PackagedFoodV5`, или одна из ссылок `BulkFoodV5`. Можно установить стоимость единицы товара, равной либо вычисленной стоимости единицы товара, возвращенной каким-либо элементом класса `PackagedFoodV5`, либо стоимости единицы товара класса `BulkFoodV5`. Для того чтобы допустить любой из этих классов в качестве явного параметра, формальный параметр должен иметь тип суперкласса. Реализация этого метода может выглядеть, например, так:

```
public void
  setUnitCostToThisUnitCost(FoodV5 anotherFood)
{
  double tempUnitCost = anotherFood.getUnitCost();
  setUnitCost(tempUnitCost);
}
```

Ключевым элементом в данном примере является следующая строка кода:

```
double tempUnitCost = anotherFood.getUnitCost();
```

Когда вызов метода `setUnitCostToThisUnitCost()` производится, объект подкласса принимается в качестве явного параметра. Параметр автоматически преобразуется в ссылку суперкласса. Метод `getUnitCost()` существует в суперклассе, поэтому он может быть вызван по ссылкам суперкласса. Однако метод объявлен абстрактным, что означает, что его реализация существует в подклассах. Если во время выполнения формальный параметр `anotherFood` относится к объекту класса `PackagedFoodV5`, то систе-

ма выявит это и использует ту реализацию метода `getUnitCost()`, которая находится в классе `PackagedFoodV5`. Если во время выполнения формальный параметр `anotherFood` относится к объекту класса `BulkFoodV5`, то система это выявит и использует ту реализацию метода `getUnitCost()`, которая находится в классе `BulkFoodV5`. Благодаря полиморфизму, формальный параметр метода `setUnitCostToThisUnit()` может ссылаться более чем на один объект во время разных вызовов. Благодаря динамической диспетчеризации, когда система выполняет этот метод, она определяет фактический тип параметра и вызывает к нему соответствующую версию метода `getUnitCost()`.

Подведем итоги относительно полиморфизма и динамической диспетчеризации.

- Методы, определенные в суперклассе, могут быть вызваны по ссылке суперкласса. Ссылка суперкласса не имеет доступа к методам, которые определены только в подклассе.
- Полиморфизм означает, что возможны ссылки суперкласса к объектам подкласса.
- Динамическая диспетчеризация означает, что во время выполнения система определяет фактический тип объекта, на который ссылается ссылка суперкласса. Если метод вызывается по ссылке суперкласса и если метод был переопределен в подклассе, то система использует ту версию метода, которая существует в подклассе, а не версию суперкласса.
- Можно передавать объекты подкласса в качестве параметров к методам, у которых формальный, явный параметр имеет тип суперкласса. В таком случае происходит автоматическое преобразование, и формальный параметр становится ссылкой суперкласса. Правила динамической диспетчеризации относятся к таким ссылкам суперкласса.

13.3. Множественное наследование и интерфейсы

Возможно, вы задумались о том, может ли отдельный класс быть частью более чем одной иерархии наследования одновременно. Вот простой пример (рис. 13.1), иллюстрирующий эту идею.

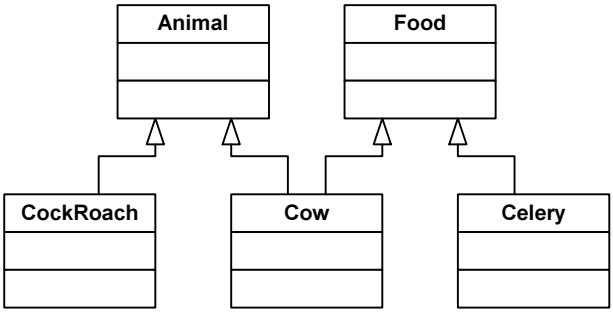


Рис. 13.1

Может ли класс `Cow` (корова) быть одновременно подклассом и класса `Animal` (животные), и класса `Food` (продукты)? Это определяется как *множественное наследование*, которое не поддерживается в языке Java. Класс может быть продолжением только одного другого класса. Класс может иметь только одного прямого родителя.

Однако Java поддерживает нечто подобное, что известно как интерфейс. *Интерфейс* — это спецификация набора методов. Интерфейс можно сравнить с абсолютно абстрактным классом, поскольку он не содержит реализации для этих методов. Он также не содержит переменных экземпляра. Интерфейс — это не класс, и в нем не может быть экземпляров. Однако класс, расположенный в произвольной иерархии наследования, может быть рассчитан на применение интерфейса. Применение интерфейса означает реализацию всех методов, перечисленных в описании интерфейса, и предоставление всех переменных экземпляра, которые могут потребоваться для поддержки этих методов.

На рис. 13.2 представлена еще одна схема, иллюстрирующая идею интерфейса, после которой мы рассмотрим пример. Нет необходимости менять что-либо в классах `FoodV5`, `PackagedFoodV5` и `BulkFoodV5`, так что их имена остаются прежними. Будет введен интерфейс `Taxable` (облагаемый налогом), и в дальнейшем мы встретимся с другими его версиями, так что пока мы не даем ему никакого номера. Также будут введены два новых класса. И хотя это будут первые случаи применения этих классов, для того что-

бы их имена соответствовали именам других классов, мы добавим к их названиям версию номер 5.

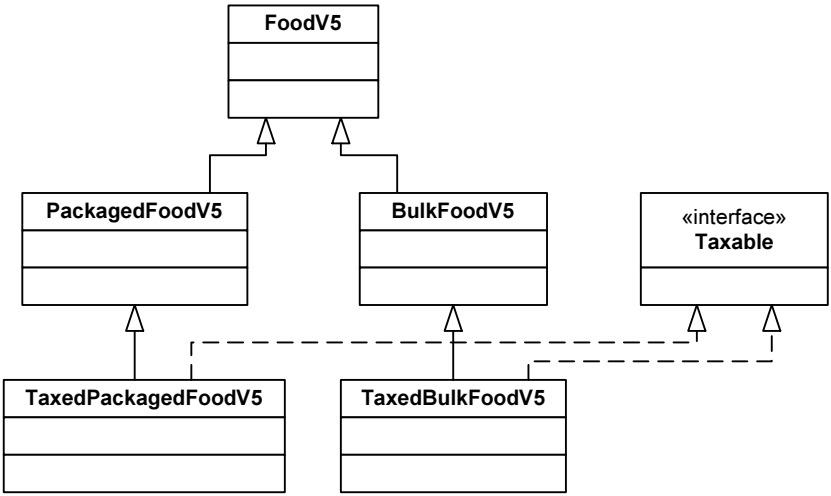


Рис. 13.2

Идея заключается в том, что мы хотим поддерживать одновременно и облагаемые (taxed), и необлагаемые (untaxed) налогом продукты. Нам нужно, чтобы обработка облагаемых налогом продуктов была одинаковой во всех случаях. Если описан интерфейс и назначены классы для реализации данного интерфейса, тогда реализация налогообложения будет одинаковой во всех этих классах.

Пусть интерфейс `Taxable` будет определен следующим образом: любой класс, который выполняет этот интерфейс, должен иметь переменную типа `double`, называемую `taxRate`. Он также должен иметь два метода: `setTaxRate()` и `getTaxRate()`. Наконец, он должен иметь метод `findTaxValuePerUnit()`. В случае `BulkFood` (нерасфасованные товары) третий метод будет зависеть от значения, сохраненного в объектной переменной `unitCost`. В случае `PackagedFood` (расфасованные товары) он будет зависеть от вычисляемого значения `unitCost`. Это означает различную реализацию метода для разных классов.

Так будет выглядеть код интерфейса:

```
public interface Taxable
{
  void setTaxRate(double TaxRateIn);
  double getTaxRate();
  double getPriceWithTax();
}
```

Интерфейс должен быть сохранен в файле с тем же именем, что и название интерфейса, и с расширением java, так же как и файл класса. Интерфейс объявлен `public` (общим), однако спецификации метода не были объявлены `public`. Они являются `public` по умолчанию.

Вот один из классов, реализующих интерфейс `Taxable`, который соответствует приведенной выше иерархии:

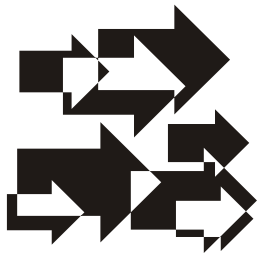
```
public class TaxedPackagedFoodV5 extends
  PackagedFoodV5 implements Taxable
{
  private double taxRate;
  public TaxedPackagedFoodV5(String brandIn,
    String productNameIn, String unitsIn,
    double sizeIn, double itemCostIn,
    double taxRateIn)
  {
    super(brandIn, productNameIn,
          unitsIn, sizeIn, itemCostIn);
    taxRate = taxRateIn;
  }
  public void setTaxRate(double taxRateIn)
  {
    taxRate = taxRateIn;
  }
  public double getTaxRate()
  {
    return taxRate;
  }
  public double findTaxValuePerUnit()
```

```
  {
    return (getItemCost() / getSize()) * taxRate;
  }
}
```

Он является одновременно и подклассом класса, и реализацией интерфейса. Синтаксис показывает, что сначала вы расширяете класс, а затем реализуете интерфейс. Объявление необходимой переменной экземпляра и ее обработка в конструкторе не является чем-то новым. Нет ничего нового и в методах `get` и `set`. Уникальным в данном классе является реализация метода `findTaxValuePerUnit()`.

Идея интерфейса может быть обобщена очень кратко: вы ничего не наследуете от интерфейса, ни переменные экземпляра, ни методы. Однако вы получаете нечто от интерфейса. Если вы решили использовать интерфейс, то получите набор требований, которые должны быть выполнены. Определенные методы должны быть реализованы, а также все, что необходимо в классе для поддержания этих методов, например переменные, должно быть предоставлено. Интерфейсы — это самое близкое, что существует в языке Java для поддержания идеи множественного наследования. Они предоставляют некоторые из аспектов множественного наследования без тех осложнений, которые связаны с ним.

ГЛАВА 14



Методы *toString()*, *equals()* и *clone()*

14.1. Метод *toString()*

Классы, написанные программистом, наследуют три важных метода от класса `Object`: `toString()`, `equals()` и `clone()`. Тема этой главы: нужно ли переопределять эти методы, и если нужно, то как сделать это правильно. Мы рассмотрим последовательные стадии этой процедуры, и на каждом ее этапе, если пример будет основан на иерархии классов продуктов питания, классу, отражающему текущие изменения, будет присвоена версия номер 6. В конце главы наименования с версией 6 будут использованы для иерархии классов продуктов питания, содержащей последнюю, наиболее желательную реализацию решений тех вопросов, которые рассматриваются в данной главе.

Метод `toString()` возвращает строковую константу, содержащую информацию об объекте. Версия метода, унаследованная от класса `Object`, дает не очень полезный результат. Метод переопределяется в системных классах, и, как правило, желательно сделать то же самое и в классах, написанных программистом. Посмотрите на пример кода, использующего метод `toString()`, как для объекта, предоставленного системой, так и для объекта класса, определенного в предыдущей главе, где метод не переопределялся:

```
Point mypoint = new Point(10, 20);
String mypointString = mypoint.toString();
```

```
myterminal.println(mypointString);
PackagedFoodV6 myfood =
  new PackagedFoodV6("Ace", "Peas", "grams", 250.0, 1.59);
String myfoodString = myfood.toString();
myterminal.println(myfoodString);
```

Если вы запустите программу, содержащую этот код, то получите такой результат:

```
Java.awt.Point[x=10,y=20]
PackagedFood@720eeb
```

Для класса, где метод был переопределен, вы получите имя класса, за которым в квадратных скобках идут имена и значения переменных экземпляра объекта. Для класса, созданного программистом, где применение метода, унаследованного от класса `Object`, не переопределено, дано название класса, за которым следует знак `@`, а далее идут цифры и буквы. Цифры и буквы образуют шестнадцатеричное целое число, которое называется *хэш-кодом*. Фактически, перед нами предоставленный системой уникальный идентификатор для данного объекта во время любого выполнения программы. Значение хэш-кода может отличаться от одного выполнения программы к другому. Хотя в Java нет прямого доступа к памяти, система может извлечь хэш-код из адреса объекта, поскольку нахождение объекта не изменяется в течение одного выполнения программы.

Можно получить тот же самый вывод без того, чтобы специально вызывать метод `toString()`. Посмотрите на следующий код:

```
Point mypoint = new Point(10, 20);
myterminal.println(mypoint);
PackagedFoodV6 myfood =
  new PackagedFoodV6("Ace", "Peas", "grams", 250.0, 1.59);
myterminal.println(myfood);
```

А вот результат:

```
Java.awt.Point[x=10,y=20]
PackagedFood@720eeb
```

Это происходит потому, что реализация метода `println()` зависит от `toString()`. Он вызывает этот метод, когда выводит объект на экран, независимо от того, был ли метод переопределен или нет.

Метод `toString()` также используется системой, чтобы поддержать конкатенацию строк и объектов. Рассмотрим следующие две строки кода, где вторая строка иллюстрирует конкатенацию:

```
String mystring = "xyz";
String astring = mystring + myfood;
```

В результате `astring` содержит конкатенацию содержимого `mystring` и того, что возвращается вызовом метода `toString()` по ссылке `myfood`.

Из этих примеров можно сделать несколько выводов. Обычно полезнее, чтобы метод `toString()` вернул имя класса, за которым следуют переменные экземпляра и их значения, а не хэш-код. Даже если для вашего кода не нужно вызывать `toString()`, его реализация важна, в силу его использования методом `println()`, а также для конкатенации. Это значит, что каждый класс, написанный программистом, должен переопределять метод `toString()`.

Относительно реализации метода `toString()` можно сделать еще несколько наблюдений. Обратите внимание, что хотя и `Point`, и `PackagedFoodV6` являются подклассами в иерархии, их классы-родители не показаны как часть возвращаемого значения из `toString()`. Для класса, предоставленного системой, показан его пакет, но это для нас неважно, поскольку мы не храним наши классы в пакетах. Другим предметом рассмотрения являются переменные экземпляра, которые показаны внутри квадратных скобок. Показаны все переменные, включая унаследованные.

Посмотрим на полную реализацию метода `toString()` для класса `PackagedFoodV6`. Обратите внимание на то, что должен быть вызван `get`-метод, чтобы получить значения унаследованных переменных экземпляра:

```
public String toString()
{
  String returnString =
    "PackagedFoodV6[brand=" + getBrand()
    + ", name=" + getProductName()
    + ", units=" + getUnits()
    + ", size=" + size
    + ", itemCost=" + itemCost + "]";
```

```
  return returnString;
}
```

Для большинства классов в иерархии требуется полная реализация метода `toString()`. Синтаксически было бы возможно упростить реализацию, используя вызовы `super.toString()`. Это помогло бы избежать необходимости выводить каждый раз названия `get`-методов и использовать эти методы, чтобы показывать значения унаследованных переменных. Однако по ряду причин это нежелательно. Это привело бы к вложению имен класса и переменных. В данном примере это означало бы реализацию метода `toString()` в родительском классе, `FoodV6`, что было бы непоследовательным. В своем конечном состоянии класс `Food` был абстрактным. Если невозможно создать объекты для класса, тогда реализация метода `toString()` для этого класса не нужна. Имейте в виду, что когда объект выводится на экран с использованием метода `println()`, то результат зависит от метода `toString()` данного объекта. Все, что желательно иметь в выводе, — это имя текущего класса и значения переменных экземпляра данного объекта, а не иерархический список всех его родительских классов, с переменными, данными вместе с классом, в котором они объявлены.

14.2. Метод *equals()*

Мы уже рассмотрели метод `equals()`, сравнивая `if`-выражения и циклы. Метод `equals()` предоставляется в классе `Object`. Он выполняет проверку равенства ссылок и наследуется в классах, написанных программистом. Он переопределяется в классах, предоставляемых системой, путем реализации, которая проверяет равенство содержимого. Метод `equals()` действует в классе `Object`, используя хэш-коды, упомянутые в предыдущем разделе. *Хэш-код* — это уникальный, предоставляемый системой идентификатор объекта, который не меняется в ходе одного выполнения программы. Система проверяет равенство ссылок путем проверки равенства хэш-кодов. Если хэш-коды одинаковы, тогда и ссылки должны быть на один и тот же объект.

Теперь посмотрим, как переопределить метод `equals()` так, чтобы он проверял равенство содержимого, а не равенство ссылок в

классах, написанных программистом. С достаточным объемом знаний о наследовании это нетрудно сделать. При переопределении метода тип метода, имя метода и перечень параметров новой реализации должны быть точно такими же, как в оригинале. В API-документации вы найдете следующую информацию для метода `equals()`:

```
public boolean equals(Object obj)
```

Это значит, что, за исключением имени формального параметра, *obj*, который может быть изменен, объявление в классе, где происходит переопределение, должно быть точно таким же. Поскольку класс `FoodV6` является абстрактным и не может иметь экземпляров, применение теста на равенство здесь не имеет большого смысла. Предположим, что вместо этого мы заинтересованы в переопределении метода в классе `PackagedFoodV6`. Следующее объявление иллюстрирует нужный нам случай, который мы рассмотрим:

```
public boolean equals(Object anotherPackagedFood)
```

Скрытым параметром вызова этого метода будет объект класса `PackagedFoodV6`. Явный параметр назван здесь таким образом, чтобы показать, что мы надеемся сравнивать с другим экземпляром класса `PackagedFoodV6`. Однако явный параметр должен иметь тип ссылки класса `Object`, для того чтобы согласовываться с методом, который мы переопределяем. Формальный параметр является ссылкой суперкласса, которая может принимать объекты подкласса в качестве фактических параметров.

Равенство имеет смысл только тогда, когда сравниваются объекты одного класса. Если это так, тогда их содержание можно с уверенностью сравнить. Чтобы сделать это, необходимо получить значения переменных экземпляра. Поскольку ссылка суперкласса не имеет прямого доступа к переменным или методам в его подклассах, то необходимо восстановить ссылку подкласса для явного параметра.

Поскольку явный параметр имеет тип `Object` и класс `Object` находится на вершине иерархии наследования, синтаксически ничто не мешает пользователю сделать вызов с явным параметром любого класса. Ссылка подкласса может быть восстановлена за счет

переопределения, но в результате вы получите ошибку выполнения, если переопределение произведено на ссылке, которая не принадлежит данному классу. Следовательно, правильное применение метода `equals()` начинается с проверки, принадлежит ли явный параметр требуемому классу. Как только ссылка подкласса успешно восстановлена, может быть применена проверка равенства содержимого. Вот пример использования этой процедуры для класса `PackagedFoodV6`:

```
public boolean equals(Object obj)
{
  if(obj instanceof PackagedFoodV6)
  {
    PackagedFoodV6 that = (PackagedFoodV6) obj;
    if(this.getBrand().equals(that.getBrand())
      && this.getProductName().equals
      (that.getProductName())
      && this.getUnits().equals(that.getUnits())
      && this.size == that.size
      && this.itemCost == that.itemCost)
        return true;
    else
      return false;
  }
  else
    return false;
}
```

В этом примере использован новый синтаксис. Также данный пример можно было написать множеством других способов — главное, чтобы результат был тем же самым.

14.3. Клонирование

Клонирование касается общей идеи копирования объекта. Мы уже увидели, что можно сделать копию ссылки к объекту, используя простой оператор присваивания, и что такая скопированная ссылка относится точно к тому же объекту, что и оригинал. Таким образом, есть один объект и две ссылки к нему. Термин

"клон" употребляется для того, чтобы ясно показать, что вы хотите создать второй, отдельный объект, но этот новый объект должен иметь то же содержимое, что и первый. В этом разделе мы рассмотрим, что необходимо сделать, если вы хотите использовать метод `clone()` в классе `Object`. В следующем разделе объясняется, как переопределить метод `clone()`.

Клонирование является сложным вопросом как синтаксически, так и концептуально. Синтаксис, необходимый для использования метода `clone()` в классе `Object`, является уникальным. В отличие от других методов, метод `clone()` свободно не наследуется подклассами. Он может наследоваться только теми классами, которые реализуют интерфейс `Cloneable`. В отличие от других интерфейсов, данный интерфейс не указывает методы, которые должен иметь реализующий класс. Он просто предоставляет доступ к методу `clone()` в классе `Object`. Это обстоятельство и другие проблемы будут объяснены подробнее далее.

Концептуальная сложность клонирования объясняется тем, что объекты могут содержать переменные экземпляра, которые являются ссылками на другие объекты. Объект класса `PackagedFoodV6` содержит переменную типа `String` с именем `brand`. Это не приводит к проблемам, т. к. содержание `String` неизменяемо. Однако можно написать класс `ShippingBox`, который содержит переменную экземпляра, являющуюся ссылкой на объект класса `PackagedFoodV6`. Объекты класса `PackagedFoodV6` являются изменяемыми. На клонировании экземпляров класса `ShippingBox` можно продемонстрировать сложности клонирования. Вот код для класса `ShippingBox`:

```
public class ShippingBox
{
  private PackagedFoodV6 aFood;
  private int numberOfPackages;
  public ShippingBox(PackagedFoodV6 aFoodIn,
                     int numberOfPackagesIn)
  {
    aFood = aFoodIn;
    numberOfPackages = numberOfPackagesIn;
  }
}
```

Вот несколько строк кода, которые демонстрируют, как создается экземпляр этого нового класса:

```
PackagedFoodV6 myfood =
  new PackagedFoodV6("Ace", "Peas", "grams", 250.0, 1.59);
ShippingBox mybox = new ShippingBox(myfood, 12);
```

Копии объектов можно создать и без использования клонирования. Пусть указанный выше код был выполнен и был создан объект `mybox`; тогда следующий код создает новый объект, `yourbox`, который является копией `mybox`:

```
ShippingBox yourbox = new ShippingBox(myfood, 12);
```

Два объекта, `mybox` и `yourbox`, разные, но они имеют общую ссылку на один и тот же объект, `myfood`. Они не являются независимыми друг от друга, потому что у них общая внутренняя ссылка. В такой ситуации `yourbox` называется *поверхностной копией* `mybox`.

Рассмотрим следующую альтернативу. Вновь допустив, что `myfood` и `mybox` созданы, как показано выше, сделаем следующее:

```
PackagedFood yourfood =
  new PackagedFood("Ace", "Peas", "grams", 250.0, 1.59);
ShippingBox yourbox =
  new ShippingBox(yourfood, 12);
```

`yourbox` снова является копией `mybox`. Однако на этот раз эти два объекта не имеют общей ссылки. Ссылка `yourfood`, находящаяся в `yourbox`, содержит те же значения, что и `myfood`, но это не тот же самый объект. Таким образом, внутренняя ссылка теперь также является копией. Данная ситуация — это случай, когда `yourbox` называется *полной копией* `mybox`. В общем, именно это имеется в виду, когда вы говорите о клоне объекта, который содержит объектные ссылки. Сложности клонирования исходят из необходимости различать полные и поверхностные копии объектов.

Еще одна предварительная деталь, прежде чем мы продолжим рассмотрение метода `clone()`. В API-документации для класса `Object` вы найдете следующий метод:

```
getClass
public final Class getClass()
Returns the runtime class of an object.
```

Последняя строка переводится так: возвращает класс объекта во время выполнения программы.

Совсем как оператор `instanceof`, который помогал проверить, принадлежала ли объектная ссылка данному классу, этот метод позволяет выяснить во время выполнения, какому классу принадлежит ссылка. Данный метод упоминается сейчас потому, что он встречается в выдержке из документации, приводимой далее.

Если вы посмотрите API-документацию, раздел для класса `Object`, вы найдете следующую запись:

```
clone
protected Object clone()
                throws CloneNotSupportedException
Creates and returns a copy of this object. The precise meaning
of "copy" may depend on the class of the object. The general
intent is that, for any object x, the expression:
    x.clone() != x
will be true, and that the expression:
    x.clone().getClass() == x.getClass()
will be true, but these are not absolute requirements. While
it is typically the case that:
    x.clone().equals(x)
will be true, this is not an absolute requirement.

clone
protected Object clone()
                throws CloneNotSupportedException
```

Это описание можно перевести так.

Создает и возвращает копию объекта. Точное значение "копии" может зависеть от класса объекта. В общем, для любого объекта `x` выражение:

```
x.clone() != x
```

будет верно, и выражение:

```
x.clone().getClass() == x.getClass()
```

будет верно, но это не абсолютные требования. Хотя, как правило:

```
x.clone().equals(x)
```

будет верно, это не абсолютное требование.

Хотя многое здесь кажется непонятным, если вы проигнорируете случаи "не абсолютных требований" и "не абсолютного требования", то получите довольно хорошее представление о том, что делает метод `clone()`. Его задача — возвращение объекта, который отличается от оригинала, но который имеет с ним равное содержимое. Суперкласс не может "знать" заранее, какие виды ссылок могут содержать объекты подкласса, поэтому на уровне класса `Object` метод `clone()` действует, создавая простые копии значений переменных экземпляра, содержащихся в объектах. Если переменные экземпляра оказываются объектными ссылками, то он не может создать их клоны. Иначе говоря, метод `clone()` в классе `Object` создает только поверхностные копии, а не полные копии.

Обратите внимание, что в документации метод объявлен `protected` (защищенным). Мы видели объявления `public` и `private`, так же как объявление методов, где доступ не определен. Объявление метода `clone()` защищенным имеет такие последствия: можно вызвать данный метод изнутри любого другого метода, определенного в том же классе. Также можно вызвать данный метод изнутри методов, которые принадлежат классам, наследующим данный класс.

Поскольку пока мы рассматривали только метод `clone()` для класса `Object`, а все остальные классы находятся ниже класса `Object` в иерархии наследования, то объявление метода `protected` не имеет какого-либо практического эффекта. Однако скоро мы рассмотрим, как реализовать метод `clone()` в классах, написанных пользователем. Предположим, что метод `clone()` реализован в классе `ShippingBox` и объявлен `protected`. Тогда будет возможным вызвать метод `clone()` изнутри остальных методов данного класса или его подклассов. Однако программа, которая не является частью иерархии наследования данного класса, не сможет вызвать этот метод.

Класс может наследовать метод `clone()` от класса `Object` только в том случае, если он реализует интерфейс `Cloneable`. В документации Java API интерфейсы, предоставляемые системой, перечислены наряду с классами. Вот описание интерфейса `Cloneable`:

```
public interface Cloneable
A class implements the Cloneable interface to indicate to the
Object.clone() method that it is legal for that method to make
a field-for-field copy of instances of that class.
Invoking Object's clone method on an instance that does not
implement the Cloneable interface results in the exception
CloneNotSupportedException being thrown.
...Note that this interface does not contain the clone
method...
```

Вот перевод этого фрагмента документации.

Общий интерфейс `Cloneable`.

Класс реализует `Cloneable`-интерфейс, чтобы указать методу `Object.clone()`, что для него допустимо делать поверхностные копии экземпляров этого класса.

Попытки клонировать переменные, которые не реализуют `Cloneable`-интерфейс, заканчиваются выбросом исключения `CloneNotSupportedException`.

Интерфейс `Cloneable` не объявляет методов.

Как видно, этот интерфейс отличается от интерфейсов, объясненных в предыдущей главе, двумя особенностями. Он сам не объявляет никаких методов, но при этом вызывает наследование метода `clone()` из класса `Object`. Классы, написанные программистом, могут наследовать метод `clone()`, определенный в классе `Object`, только в том случае, если они реализуют интерфейс `Cloneable`. Попытка вызвать метод `clone()` к объекту, класс которого не реализует интерфейс `Cloneable`, вызовет выброс исключения.

Если класс реализует интерфейс `Cloneable`, наследованный метод `clone()` защищен в отношении данного класса. Метод может быть использован только изнутри самого кода класса и не может быть использован в других программах. Наследованный метод делает поверхностные копии, хотя во многих ситуациях желательна полная копия. Требование о том, чтобы класс реализовывал интерфейс `Cloneable` для наследования метода, является тем механизмом, который заставляет программиста следовать этим ограничениям. Таким образом, одним из вариантов использования клонирования в классах, написанных программистом, является

реализация интерфейса `Cloneable`. Этот и некоторые другие варианты мы и рассмотрим далее.

14.4. Реализация метода *clone()*

У программиста, в целом, есть 3 варианта действий в отношении клонирования:

1. Не использовать метод `clone()`. Если же копия объекта когда-либо потребуется, создать ее с нуля, используя конструкторы и методы.
2. Решить, что поверхностная копия достаточна и необходима для использования в коде самого класса или подклассов, и реализовать интерфейс `Cloneable`.
3. Переопределить метод `clone()`, что позволит его вызывать вне кода класса, и потенциально реализовать полную копию.

Вариант 1 не требует дальнейшего обсуждения. Вариант 2 сравнительно прост, и с него мы и начнем. Хотя описание в предыдущем разделе включало класс, содержащий изменяемую объектную ссылку, более простой класс может быть использован для иллюстрации синтаксиса. Снова возьмем для примера класс `PackagedFoodV6`. Его объявление может быть изменено следующим образом:

```
public class PackagedFoodV6
  extends FoodV6 implements Cloneable
```

Этот пример доказывает, что можно просто использовать унаследованный метод `clone()`, чтобы сделать копии объекта данного класса. Поскольку класс не содержит непостоянных ссылок, это совершенно приемлемо. Наследованный метод объявлен защищенным, поэтому `clone()` может использоваться только внутри класса `PackagedFoodV6` или его подклассов. Вы можете включить следующий метод в класс `PackagedFoodV6` просто для того, чтобы проверить этот подход:

```
public void justChecking()
{
  PackagedFoodV6 myfood =
    new PackagedFoodV6("jkl", "uio", "hjk", 5, 4);
```

```
  try
  {
    PackagedFoodV6 newFood =
      (PackagedFoodV6) myfood.clone();
  }
  catch(CloneNotSupportedException exception)
  {
    System.out.println("CloneNotSupportedException caught.");
  }
}
```

Этот пример иллюстрирует два аспекта использования метода `clone()`, которые требуют дальнейшего объяснения. Рассмотрим следующую строку кода:

```
PackagedFoodV6 newfood =
  (PackagedFoodV6) myfood.clone();
```

Ссылка, возвращенная из вызова `clone()`, приводится к классу `PackagedFoodV6`. Это необходимо потому, что метод `clone()` в классе `Object`, который наследуется, возвращает ссылку класса `Object`. Объявление метода `clone()` дано в документации API следующим образом:

```
protected Object clone()
```

Другой аспект данного примера, который должен сразу привлечь ваше внимание, — это блоки `try` и `catch`:

```
try
{
  ...
}
catch(ClassName parameter)
{
  ...
}
```

Вызов метода `clone()` выбрасывает исключение, если класс не реализует интерфейс `Cloneable`.

Теперь рассмотрим вариант 3 из перечня, приведенного в начале данного раздела, где целью является переопределение метода `clone()` так, чтобы его можно было использовать вне его класса

или подклассов. И в этом случае необходимо объявить, что класс реализует интерфейс `Cloneable`:

```
public class PackagedFoodV6
  extends FoodV6 implements Cloneable
```

Посмотрим на метод `clone()` класса `PackagedFoodV6`. Этот пример является просто иллюстрацией переопределения метода в случае, когда класс не содержит никаких объектных ссылок и полная копия не требуется:

```
public Object clone()
{
  try
  {
    PackagedFoodV6 returnFood =
      (PackagedFoodV6) super.clone();
    return returnFood;
  }
  catch(CloneNotSupportedException exception)
  {
    return null;
  }
}
```

Эта реализация метода `clone()` объявлена не защищенной (`protected`), а общей (`public`). При переопределении наследованного метода модификатор доступа — `public`, `private` или `protected` — является той частью объявления, которая может быть изменена. Поскольку объявление является общим, то метод может быть использован программой. Так как в классе нет объектных ссылок, то достаточно сделать поверхностную копию. Это можно сделать за счет вызова метода `clone()` в суперклассе, `Object`. Этот вызов должен быть заключен в блоках `try` и `catch`, поскольку реализация метода в классе `Object` выбрасывает исключение, если вызов к нему является неуспешным. Внутри переопределенного метода возвращаемое значение от вызова к `super.clone()` должно быть приведено к фактическому типу, `PackagedFoodV6`. Однако метод все равно возвращает ссылку класса `Object` в соответствии с тем методом, который он переопределяет. Синтаксически метод должен вернуть какую-либо ссылку.

Ссылка на `null` возвращается из блока `catch`, если никакой объект не создается.

Наконец, рассмотрим вариант 3 снова, но на этот раз с условием, что мы хотим переопределить метод `clone()`, чтобы сделать полные копии. Класс `ShippingBox` содержит ссылку, так что он может быть использован в качестве примера. Ссылка, которую он содержит, относится к классу `PackagedFoodV6`. Предположим, что метод `clone()` был переопределен в классе `PackagedFoodV6`, как это описано ранее. При переопределении метода `clone()` класс должен по-прежнему реализовать интерфейс `Cloneable`. Посмотрим на код для класса `ShippingBox`, с методом `clone()`, который может быть вызван программой из-за пределов иерархии наследования `ShippingBox` и который делает полные копии:

```
public class ShippingBox implements Cloneable
{
  private PackagedFoodV6 aFood;
  private int numberOfPackages;
  public ShippingBox(PackagedFoodV6 aFoodIn,
                     int numberOfPackagesIn)
  {
    aFood = aFoodIn;
    numberOfPackages = numberOfPackagesIn;
  }
  public Object clone()
  {
    try
    {
      ShippingBox returnBox = (ShippingBox) super.clone();
      returnBox.aFood = (PackagedFoodV6) this.aFood.clone();
      return returnBox;
    }
    catch(CloneNotSupportedException exception)
    {
      return null;
    }
  }
}
```

В методе `clone()` вызов `super.clone()` создает новый объект и занимается копированием значения переменной `numberOfPackages` из оригинала в клон. Затем метод `clone()` класса `PackagedFoodV6` используется, чтобы создать новую и отдельную копию переменной экземпляра `aFood`. При неудаче любого из этих вызовов метода `clone()` вернулся бы тот же вид исключения, поэтому они могут содержать общие блоки `try` и `catch`. Метод возвращает ссылку на экземпляр класса `Object`. Эта ссылка будет ссылкой на объект класса `ShippingBox`. Если блок `catch` вызван, возвращается `null`.

Когда метод `clone()` переопределяется и объявляется общим, он может использоваться для общей программы за пределами того класса в иерархии наследования, в котором он был определен. Важно помнить, когда делается вызов, что возвращенное значение выводится с типом `Object`, и оно должно быть приведено назад к первоначальному типу объекта. Вот простой пример:

```
PackagedFoodV6 myfood =
  new PackagedFoodV6("jkl", "uio", "hjk", 5, 4);
PackagedFoodV6 yourfood =
  (PackagedFoodV6) myfood.clone();
```

Неверный вызов переопределенной версии метода `clone()` выбросит непроверяемое исключение. Это значит, что вызовы метода `clone()` внутри программы не должны быть заключены в блоки `try` и `catch`. При написании метода `clone()` для иерархии классов программист должен контролировать все возможные исключения внутри кода метода.

Последнее замечание, касающееся написания методов `clone()`. В классе типа `PackagedFoodV6` было бы возможно записать объявленный метод так:

```
public PackagedFoodV6 clone()
{
  ...
}
```

Метод возвращает ссылку на экземпляр класса `PackagedFoodV6`, поэтому не требуется приведение типа возвращенного значения. Класс также не объявлен реализующим интерфейс `Cloneable`, и

содержимое метода может быть относительно простым. Однако этот метод не переопределяет метод `clone()` в классе `Object`. Для того чтобы переопределить метод, имя, перечень параметров и возвращаемый тип переопределяющего метода должны быть те же самыми. Нет ничего плохого в идее написания такого нового метода `clone()`, но важно понимать разницу между ним и переопределением унаследованного метода `clone()`.

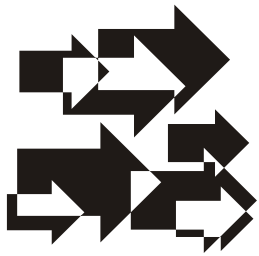
14.5. Важное использование клонирования

Класс `ShippingBox` не имеет `get`-метода для ссылки на объект класса `PackagedFoodV6`, который он содержит. Используя предыдущие примеры на метод `get` в качестве образцов, программист может реализовать такой метод для возвращения ссылки на объект класса `PackagedFoodV6`, содержащейся в объекте класса `ShippingBox`. Это будет неверным, т. к. нарушает принцип инкапсуляции. Если возвращается ссылка на переменную экземпляра, то будет возможным вызывать методы непосредственно по этой ссылке. Доступ к переменной более не ограничен вызовами методов того класса, в котором она находится.

Решение этой проблемы заключается в том, чтобы методы `get` для ссылок вернули клоны объектов. Посмотрите на правильную реализацию метода `get` для переменной `aFood` экземпляра класса `PackagedFoodV6` в классе `ShippingBox`:

```
public PackagedFoodV6 getAFood()
{
  return aFood.clone();
}
```

ГЛАВА 15



Проекты игровых программ

Wari и *Togiz Kumalak* — это две версии одной и той же досочной игры. В этом разделе дается некоторая историческая и культурная информация об этих играх. При реализации надо будет принимать ряд решений, и они обсуждаются в последнем разделе главы.

15.1. Исторический обзор

Wari и иные аналогичные игры хорошо изучены. В данном разделе приведены выдержки из опубликованных работ[1] об этой игре.

Wari — это одна из многих досочных игр, в которую играют в различных частях света. Они общеизвестны как игры *mancala*, и в них играли тысячи лет тому назад в Египте, где были найдены игровые доски, высеченные в камнях пирамиды Хеопса и в храмах Люксора и Карнака. Позже игра распространилась в Азию и Африку, где арабы усовершенствовали несколько вариантов. Та-

[1] Games of the World: How to Make Them, How to Play Them, How They Came to Be, edited by Frederic V. Grunfeld, Introduction by R. C. Bell, Design, Illustration, and Game Models by Pieter van Delft and Jack Botermans, Holt, Rinehart, and Winston, 1975; R. C. Bell.
Board and Table Games From Many Civilizations. Dover Publications, 1979; David Parlett.
The Oxford History of Board Games. Oxford University Press, 1999.

ким образом, она пережила все эпохи египетской истории. Европейские путешественники познакомились с ней в кофейнях Каира в XIX веке, где было принято, чтобы проигравший платил за кофе, выпитый во время игры.

Африканские рабы привезли игры mancala в Суринам и Вест-Индию, где они до сих пор существуют в первоначальном виде. В некоторых сельских районах современной Африки дети играют в эти старинные игры на "досках", выкопанных в земле.

Профессор Флиндерс Петри (британский археолог) обнаружил кусок известняка в Мемфисе, в котором были три ряда по 14 лунок в каждом, что похоже на раннюю форму игры mankala. Судя по камню, фигуры для игры захватывались игроками, и лунки были настолько малы, что фигурами были, вероятно, бобы или семена.

Несколько наборов глубоко вырезанных отверстий были найдены в плитах, образующих крышу храма Курна, недалеко от Фив (около XIV в. до н. э.). Другие наборы досок вырезаны в вершине разрушенной части огромной колонны, построенной во времена Птоломея при входе в храм Карна, а также в храме в Люксоре. Доски состоят из двух рядов по шесть, семь и восемь отверстий, имеющих форму блюдца, самое крупное из них имеет 3,5 дюйма (около 9 см) в ширину и 1 дюйм (2,54 см) в глубину.

Доски были обнаружены в Аравии, относящиеся к временам до Мухаммеда, и последователи его учений принесли эту игру в те страны, куда они несли свою культуру. Mancala используется в качестве общего названия для всех игр этого типа.

В Новом Свете африканские рабы играли в свои игры и учили им своих детей. До сих пор можно проследить родословную некоторых негров из Вест-Индии по тому виду mancala, в который они играют. Четыре варианта существуют среди негров Гвианы и Карибского моря, которые совпадают с играми, существующими в Бенине, Того и Нигерии. У этих игр разные названия, например, awari. По интеллектуальному уровню эти игры находятся на уровне шахмат. Негры играют в эти игры ради развлечения и престижа, который приобретает хороший игрок. Они не будут играть в нее на деньги.

Игра mancala, вероятно, имеет происхождение среди негров Африки, и Стюард Кулин (американский антрополог) ее справедливо называет африканской национальной игрой. Однако не существует общепринятой официальной или стандартной версии. Наоборот, это типично народная игра — наподобие диалектов, носители которых понимают друг друга, но каждый из них существует только на своей территории. Каждая разновидность может иметь несколько названий в одном и том же месте, и, наоборот, то же самое название часто обозначает разные, хотя и похожие игры. Игра имеет столь же широкое распространение, как и карты.

Mancala — это игра, для которой характерны полная информация, полное равенство и свобода выбора, а следовательно, она требует большого мастерства. Поначалу представители западной культуры могут подумать, что тот, кто начинает игру, имеет явный шанс выиграть или, по крайней мере, свести вничью, и что игра будет медленной и вымученной, когда каждый из игроков просчитывает все возможные ходы на столько шагов вперед, насколько возможно. Такая игра идеальна для компьютерного анализа, и исследователи полагают, что окончательное решение этой игры неизбежно будет найдено. Некоторые могут сторониться этой игры на том основании, что она лишена, с одной стороны, случайностей бриджа и триктрака, что делает эти игры подходящими для игры по интуиции, а с другой стороны, лишена двумерной структуры шахмат, которые представляют собой более сложную задачу для интеллектуального анализа. Сложность шахмат лежит в их глубине, а игры mancala — в ее продолжительности. Однако представители западной культуры не видят, что ценность игры mancala заключается в том, что она объединяет общество и представляет собой интеллектуальное развлечение. Любители этой игры среди местного населения достигают поразительной скорости, а специалисты играют с убийственной точностью. Даже не подсчитывая содержимое наиболее заполненных лунок, многие обладают навыком безошибочно выбирать наилучшую стратегию игры, исходя из своего опыта. Играть в эту игру с медлительностью шахмат — значит, не понимать саму идею игры. Игра mancala, по крайней мере, в Африке, является

общественной, и наблюдающие за игрой участвуют в ней настолько, что иногда трудно отличить игроков от непрошеных советчиков. Некоторые из наиболее крупных и сложных игр действительно играются двумя командами по обе стороны доски больших размеров. На этом фоне некоторые исследователи с сожалением отмечают, что "окультуренные" жители Западной Африки в наши дни смотрят на mancala, как на игру для крестьян и детей, и что некоторые доски для игры, продающиеся в крупных городах, имеют неправильные размеры или текстуру, что говорит о том, что те, кто их производит, потеряли связь со своими культурными корнями. Это явление хорошо известно в любой стране, в которой доминирует одна (в данном случае, "западная") культура, считающаяся более полезной и потому более достойной подражания.

15.2. Wari

Теперь рассмотрим правила игры Wari, описание которых дано в нашем первом источнике. Доска для игры состоит из двух рядов по шесть чашек в каждой, один ряд для каждого из игроков. Первоначально, в каждой чашке находятся четыре семени. Кроме того, у каждого игрока есть отдельная чашка для хранения выигранных семян; в начале игры эти чашки пусты. На рис. 15.1 показано состояние доски в начале игры.

Приведем простую версию правил для игры. За ней следуют более подробные правила, которые можно включить в реализацию программы.

Итак, основные правила игры следующие. Игрок берет содержимое одной из чашек на своей стороне доски. Затем, двигаясь против часовой стрелки и начиная с чашки, следующей за той, из который он взял семена, опускает по одному семени последовательно в каждую чашку, пока он не разложит все. Если последнее семя попадает в чашку противника, где находится только одно или два семени, это является выигрышем. Игрок получает право забрать два или три семени, оказавшиеся в данной чашке, и положить их в свою чашку для выигрышей. После завершения, неважно, выиграл ли игрок что-либо или нет, настает очередь иг-

рать второму игроку. Так игра продолжается до тех пор, пока у одного из игроков не остается семян для игры. Тот игрок, у которого остаются семена на его стороне доски, получает право добавить их к своему выигрышу. Побеждает тот, у кого оказывается наибольшее количество семян.

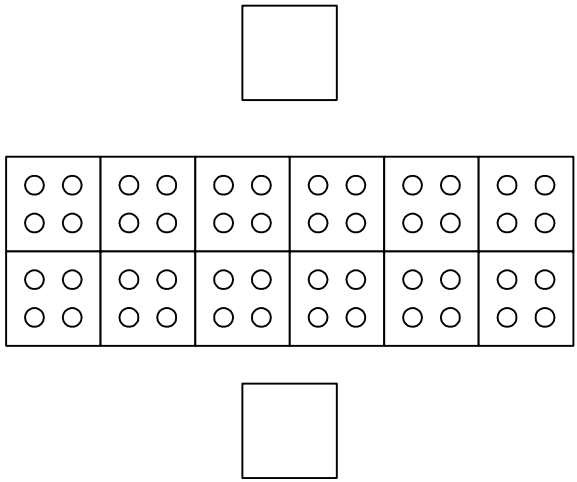


Рис. 15.1

Реализация уже одних этих правил представляет собой интересную задачу для программирования. В процессе написания кода вам нужно будет думать о различных ситуациях, которые могут возникнуть, и о типах ходов, которые могут быть разрешены. С точки зрения выполнения такого задания, вы можете решать эти вопросы так, как посчитаете уместным, без обращения к дополнительным правилам. Однако существуют и дополнительные правила, и если вы заинтересованы в реализации игры в более полном виде, мы их приводим далее.

Эти правила касаются некоторых ходов игры и выигрышей семян из чашек. Чашка может так наполниться, что семена могут выпасть и рассыпаться по всей доске. Если такое происходит, то в такую чашку нельзя класть больше семян. Кроме того, если вы выигрываете содержимое какой-либо чашки, то получаете право забрать семена и из предыдущей чашки на стороне противника,

если в ней два или три семени. Таким образом, после выигрыша чашки, в которую вы положили свое последнее семя, вы двигаетесь в обратном направлении по чашкам противника и забираете все семена до тех пор, пока вам не попадается чашка, число зерен в которой не составляет два или три.

Кроме того, существуют дополнительные правила относительно окончания игры. Если вы опорожнили все чашки на своей стороне доски, и остается еще ход, в котором ваш противник может переместить семя на вашу сторону, то он обязан сделать такой ход, тем самым игра продолжается. Кроме того, не разрешается выиграть за один раз все семена, оставшиеся на стороне противника за исключением ситуации, когда это ваш последний ход. Общая цель этих правил — сделать так, чтобы игру нельзя было прекратить, если на доске остается много семян, но случайно они оказались на одной стороне. С другой стороны, игра может достичь такого момента, когда на доске остается очень мало семян. В таком случае, игрокам разрешается согласиться окончить игру, при этом каждый забирает те семена, которые остались на его стороне доски.

15.3. Togiz Kumalak

Эта игра относится к тому же семейству игр, что и Wari. В нее играют в Казахстане, и описание, которое дается далее, получено от жителя Казахстана, играющего в эту игру.

Ислам, распространяясь по всему Востоку из места своего происхождения, Аравийского полуострова, принес с собой и такие культурные артефакты, как игры. По Великому шелковому пути ислам достиг Центральной Азии, где позднее возник Казахстан. На казахском языке Togiz Kumalak означает "девять шаров". В этой игре используется 9 шаров для каждой чашки, вместо 4 семян в Wari. Игры, подобные Wari, традиционно использовали семена или гальку в качестве фигур. Такие фигуры легко было найти в Африке. Но Центральная Азия — это край степей и пустынь. На больших территориях только земля и песок, где не растет ничего, кроме травы. Традиционная казахская культура основана на кочевом скотоводстве, таком как овцеводство. Эти жи-

вотные дают сравнительно безобидные катышки. При отсутствии дерева для изготовления досок, их можно было выкопать в земле. А вместо гальки или семян в качестве фигур можно было использовать легкодоступные катышки от животных.

В игре Wari не принято, чтобы игроки подсчитывали количество семян в чашках прежде, чем делать ход, и в целом чашки расставлены так, что с ходу невозможно сказать, сколько в них семян. Как вы увидите далее, правила Togiz Kumalak требуют, чтобы игроки знали, четное или нечетное количество фигур в чашке. На доске соответствующей формы и размера, форма и размер катышков позволяют сразу увидеть, имеется ли четное или нечетное количество, и игра продолжается без необходимости подсчитывать перед каждым ходом. Материал игры, т. е. доска, вырытая в земле, и катышки животных, вероятно, не давали возможность играть в быстрые игры. Кроме того, игроки, будучи кочевыми скотоводами, которым нужно было чем-то заполнить время, предпочитали более медленные игры.

На рис. 15.2 показана доска для игры. Сверху — сторона игрока 1, а внизу — сторона игрока 2. У каждого игрока по 9 чашек, и в начале игры в каждой чашке 9 шаров.

Правила игры.

Игроки играют по очереди, и при каждом ходе вы можете брать шары из любой чашки на своей стороне, в которой еще есть шары. Если в чашке только один шар, вы берете один. Если шаров больше одного, вы берете все, кроме одного. Затем вы опускаете шары по одному в чашки, двигаясь против часовой стрелки.

Если в чашке имеется четное число шаров, она считается закрытой. Если в чашке нечетное число, она считается открытой. Если последний шар, бросаемый вами, попадает в чашку противника, которая открыта, вы выигрываете все шары в этой чашке. Общая стратегия игры — открыть все чашки противника, делая количество шаров в них нечетным, и закрыть свои чашки, делая количество шаров в них четным.

Если последний шар бросается в одну из чашек на стороне противника и в этой чашке 2 шара, то эта чашка становится тем, что называется вашим "домом" на стороне противника. Вы сразу же

выигрываете три шара, которые находятся в этой чашке, и в течение всей последующей игры все шары, которые попадают в эту чашку, неважно, брошены они вами или противником, становятся вашим выигрышем. У игрока может быть только один такой дом, поэтому после того, как вы его получаете, стандартные правила применяются к другим чашкам с двумя шарами — это чашки с четным числом шаров. Дом может быть у обоих игроков. Тот факт, что дом есть у одного игрока, не мешает получить его и второму игроку. На самой доске чашка, которая становится домом, обычно помечается специальной фигурой, которая отличается от обычных шаров.

Рис. 15.2

Игра заканчивается, когда у одного из игроков заканчиваются шары на его стороне доски. Когда это происходит, то игрок, у которого еще остались шары, прибавляет их к своему выигрышу. Выигрывает тот игрок, у которого оказывается наибольшее количество шаров.

15.4. Варианты реализации

Каждый программист может предложить свою реализацию игры Togiz Kumalak. В этой главе дан обзор трех вариантов реализации[1] игры Wari, с кратким описанием. Совсем необязательно реализовывать игру так, как это показано в примерах, но эти примеры могут служить руководством для некоторых из возможных вариантов.

`WariV1` является простой реализацией, которая находится в одном файле программы с методом `main()`. Эта программа не зависит ни от каких других классов. Чашки моделируются в виде множества, и подсчет семян представлен целочисленным значением, хранящемся в массиве. На экране доска представлена символами Unicode и числами. Поскольку такая реализация может быть сделана с помощью одной-двух страниц кода, такое решение представляется совершенно разумным. Если выбран такой подход, то задание по составлению такой программы можно выполнить, не затрагивая содержимого *глав 12—14*.

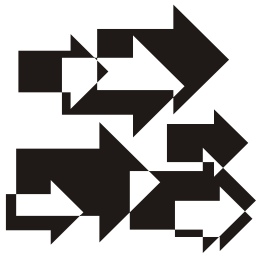
Версия игры `WariV21` основана на реализации классов для доски и чашек. Разные программисты могут получить различный набор классов для реализации игры. Хотя не обязательно создавать игру с выводом с помощью Unicode, такой подход будет первым шагом в создании приложения, которое впоследствии можно сделать целиком графическим. Данный код был создан на основании кода предыдущей версии. Здесь также имеется массив чашек, но доступ к чашкам уже более не осуществляется с помощью индексов массива. Чашки связаны между собой ссылками друг на друга. Движение по доске во время игры моделируется за счет следования ссылкам, а не с помощью следования подстрочным индексам массива. В данной реализации не используется материал *глав 12—14*.

Версия `WariV22` — это еще один шаг в развитии реализации игры. Чашки, в которых находится выигрыш игроков, не нужно соеди-

[1] Напомним читателям, что задания к этому и иным разделам находятся на сайте издания *(см. введение)* и на компакт-диске.

нять между собой так же, как остальные чашки на доске. Это различие дает возможность выполнить реализацию, которая иллюстрирует иерархию классов с наследованием, за счет того, что имеются разные классы чашек. Такое различие не является принципиально важным для правильной реализации программы. Это может быть полезным для того, чтобы использовать идеи, рассмотренные в *главе 12*.

ГЛАВА 16



Простые схемы UML с помощью программы Visio

В *главе 1* нам уже встречался язык UML, и простые схемы с применением его символов были использованы для иллюстрации некоторых примеров в предыдущих главах. В дальнейшем такие схемы будут использованы для иллюстрации все более сложных по структуре программ. В этой главе будет дана дополнительная информация о языке UML вместе с простыми инструкциями по использованию программы Microsoft Visio для создания подобных рисунков.

16.1. Схемы статической структуры для классов и объектов

В разных источниках аббревиатуру UML расшифровывают поразному: "единый язык моделирования" (unified modeling language) или "универсальный язык моделирования" (universal modeling language). Как бы то ни было, UML — это набор визуальных символов и условных обозначений, который можно использовать для схематического изображения объектно-ориентированных программ. Можно графически изобразить такие вещи, как классы, переменные экземпляра, методы, экземпляры классов, отношения между классами и т. д. Язык UML также имеет свой словарь. Термины *"класс"* и *"объект"* имеют такие же значения, как и в языке Java. Однако UML использует термин *"am-*

рибут" там, где Java применяет *"поле"*. UML использует термин *"операция"* для того, что в Java называется *методом*. Язык UML, в том числе его графика и терминология, не связан с каким-либо конкретным языком программирования. Программа, написанная на любом объектно-ориентированном языке, может быть схематически изображена с помощью языка UML.

Классы и объекты изображаются с помощью прямоугольников. Отношения указаны с помощью пометок, линий, стрелок и иных символов. В своей простейшей форме класс может быть представлен в виде прямоугольника, содержащего имя класса. Имя пишется с большой буквы жирным шрифтом. Некоторые примеры в дальнейшем тексте основаны на идее о чашках, в которых находятся семена. Два разделенных прямоугольника (рис. 16.1) будут представлять эти два класса в их простейшем виде.

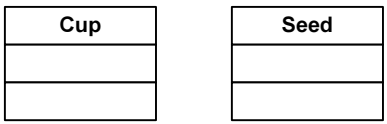


Рис. 16.1

Чтобы нарисовать подобную схему с помощью программы MS Visio, нужно сделать следующее:

1. В меню выберите **File | New | Software | UML Model Diagram**.
2. В палитре **Shapes** выберите раздел **UML Static Structure**.
3. В разделе **UML Static Structure** перетащите пиктограмму класса в рабочую область.
4. В рабочей области два раза щелкните кнопкой мыши и введите имя класса.

На схеме может быть дана более подробная информация о классе, включая его атрибуты и операции. В общем, подобная схема может выглядеть так, как показано на рис. 16.2.

Для нашего примера, пусть в классе `Cup` будет одна переменная экземпляра, с именем `seedCount`. Пусть у класса будут два метода: `setSeedCount()` и `getSeedCount()`. Переменная `seedCount` будет це-

лым числом, и когда создается новый экземпляр класса, пусть его переменная `seedCount` инициализируется со значением 0. Метод `setSeedCount()` присваивает значение переменной `seedCount` объекта. Метод `getSeedCount()` не требует параметров. Он просто возвращает текущее значение, которое хранится в переменной `seedCount`. Эту информацию можно обобщить на схеме (рис. 16.3).

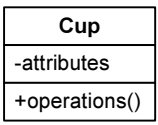


Рис. 16.2

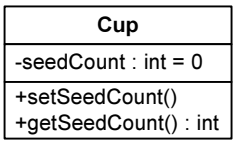


Рис. 16.3

Для того чтобы создать схему, представленную на рис. 16.3, нужно сделать следующее в программе MS Visio:

1. В рабочей области два раза щелкните кнопкой мыши на пиктограмме класса.
2. В разделе **Categories** выберите вкладку **Attributes**. Введите имена, типы и другие характеристики атрибутов.
3. В разделе **Categories** выберите вкладку **Operations**. Введите имена операций, возвращаемые типы и т. д.

В языке UML объекты представлены прямоугольниками. В приложении объект некоторого класса может иметь свое имя. Предположим, экземпляр класса `Cup` известен как `mycup`. В языке UML имя этого объекта подчеркивается, и за ним следует двоеточие и имя класса, экземпляром которого он является (рис. 16.4).

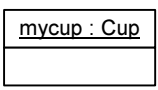


Рис. 16.4

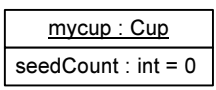


Рис. 16.5

Если класс `Cup` был определен с атрибутом `seedCount`, тогда можно добавить информацию об экземпляре, например, текущее значение для семян (рис. 16.5).

Чтобы получить схему, представленную на рис. 16.5, нужно сделать следующее в программе MS Visio:

1. Обратите внимание, что класс `Cup` уже должен существовать в документе Visio, для того чтобы указать, что объект является экземпляром этого класса.
2. В разделе **UML Static structure** перетащите пиктограмму объекта в рабочую область.
3. В рабочей области два раза щелкните кнопкой мыши по пиктограмме, введите имя для объекта и выберите класс, экземпляром которого он является, из списка классов в выпадающем меню.
4. В разделе **Categories**, если класс `Cup` имеет атрибут `seedCount`, выберите опцию **Attributes Value**. Введите значение.

Отношения между классом и объектом можно графически показать с помощью стрелки. В языке UML разные типы отношений показываются разными видами стрелок, так что форма стрелки имеет значение. В данном случае отношение изображается с помощью пунктирной стрелки с открытым, перовидным концом. Концептуально, объект относится к своему классу. В символах UML это показано с помощью стрелки, направленной от объекта к классу (рис. 16.6).

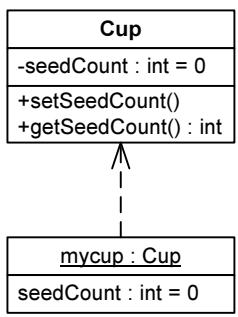


Рис. 16.6

Для этого нужно сделать следующее в программе MS Visio:

1. В разделе **UML Static Structure** перетащите пиктограмму зависимости в рабочую область.

2. В рабочей области проверьте, чтобы стрелка указывала от объекта к классу, щелкнув кнопкой мыши и перетащив ее острие и хвост.

16.2. Ассоциации, агрегации и композиции

Ассоциация — это тип отношения между объектами, которые могут входить в разные классы. Ассоциация изображается на схемах дугой, соединяющей два прямоугольника. Дуга может иметь стрелки в конце. Термин "ассоциация" очень широк по значению. Любые отношения между объектами можно назвать ассоциацией, и не существует никаких требований, как такие отношения реализуются в коде. В силу этого, такое отношение должно получить описательное имя, и на схеме оно должно иметь соответствующую пометку. Желательно сделать пометку на обоих концах отношения, так чтобы не было сомнений во взаимном характере отношения. Также такими отношениями могут охватываться объекты более чем двух классов, но мы не будем касаться таких случаев в нашем обзоре.

Хотя речь идет об отношениях между объектами, схема, показывающая ассоциацию, основана на классах. Статическая структура приложения изображена на схеме с использованием именно классов, а не конкретных объектов, которые могут существовать во время выполнения. Ассоциации имеют две важные характеристики, которые могут быть указаны на схемах. Одна из них — это *множественность* (cardinality). На каждом конце ассоциации может быть более одного объекта. Сколько именно объектов участвует в отношении, указывается с помощью чисел на схеме. Вторая характеристика называется *доступностью* (navigability). Речь идет о том, что в коде, реализующем отношения, один объект может иметь доступ к другому. Если доступ имеется, то это может быть указано на схеме помещением стрелки на той стороне, которая доступна.

Рассмотрим простую ситуацию, которая иллюстрирует идею ассоциации. Пусть имеется класс `SeedDistributor`, содержащий ме-

тод, который вызывает методы `getSeedCount()` и `setSeedCount()` класса `Cup` и использует их некоторым образом для добавления семян в чашки. Пусть в данном приложении имеется один экземпляр класса `SeedDistributor` с именем `distributor`, и он может влиять на нулевое или большее количество экземпляров класса `Cup`. Пусть далее экземпляры класса `Cup` доступны `distributor`, но `distributor` не доступен экземплярам класса `Cup`. Отношения между объектами классов `SeedDistributor` и `Cup` могут быть отражены на рис. 16.7.

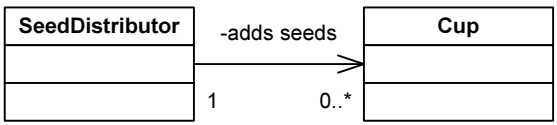


Рис. 16.7

Схему, представленную на рис. 16.7, можно построить в программе MS Visio следующим образом:

1. В разделе **UML Static Structure** перетащите пиктограмму **Binary Association** в рабочую область.
2. В рабочей области два раза щелкните кнопкой мыши по пиктограмме. Дайте ей имя. Присвойте имена концам ассоциации, агрегации, множественности, управляемости и т. д. Кнопка **Properties** дает подробный доступ к характеристикам на концах ассоциации.

Вероятно, наиболее важным или распространенным видом отношений между классами является ситуация, когда экземпляр одного класса имеет экземпляры другого класса. Такого типа отношения называют иногда *отношениями "имеет"* ("has a" relationship). В языке Java такое отношение называется *агрегацией*. В контексте конкретного приложения оно может быть описано как "содержит" или "состоит из", или может быть использован какой-либо иной более узкоспециализированный термин. Например, в университете *находятся* студенты, или университет *принимает* студентов; в магазине *есть* товары, или магазин *имеет в наличии* товары, и т. д. В общем, речь идет об отношениях между целым и его частями.

В языке UML есть специальные символы для обозначения агрегации. Имена классов соединяются дугой с ромбовидными стрелками. Конец дуги со стрелкой направлен к классу, который "имеет", т. е. к целому. Конец без стрелки идет к классу, который "имеется", т. е. к части. Класс `ArrayList` может быть использован для продолжения примера с семенами и чашками. Пусть объект `cup` содержит один `ArrayList`, который может содержать произвольное число объектов `seed`. Схема, иллюстрирующая эти две агрегации, представлена на рис. 16.8. Обратите внимание, что возможно и желательно как пометить агрегацию, так и указать на множественность отношений на каждом конце дуги.

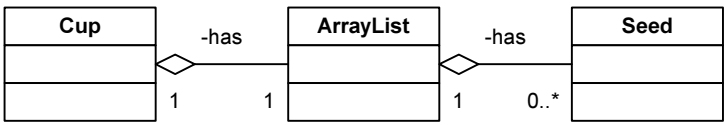


Рис. 16.8

Чтобы построить такую схему, нужно сделать следующее в программе MS Visio:

1. В разделе **UML Static Structure** перетащите пиктограмму **Composition** в рабочую область.
2. В рабочей области проделайте то же самое, что вы делали для бинарных ассоциаций.

Существует разновидность агрегации, которая достаточно важна, поэтому получила собственное название. Она известна как *композиция* и представлена ромбовидными стрелками с закрашенными черными наконечниками. Идея композиции заключается в том, что целое безраздельно владеет частями, и части не могут существовать независимо от целого. Предыдущая диаграмма повторяется на рис. 16.9 с одним изменением: отношения между классом `Cup` и классом `ArrayList` представляют собой композицию. Это значит, что класс `ArrayList` не может существовать без `Cup`, в котором он находится.

Помните, что не существует каких-либо общих требований относительно того, как те или иные отношения должны быть реализо-

ваны в коде. Однако если объект одного класса является переменной экземпляра другого класса, синтаксически это приводит к композиции. Переменная экземпляра существует только внутри экземпляра класса, который ее содержит. В более сложных разработках программы различие между агрегацией и композицией может быть значительным. И хотя мы говорим об этом различии сейчас, в последующих схемах, если это различие не существенно для функционирования программы, мы будем показывать обычную агрегацию на схеме, даже если представление отношения в виде композиции может быть более точным.

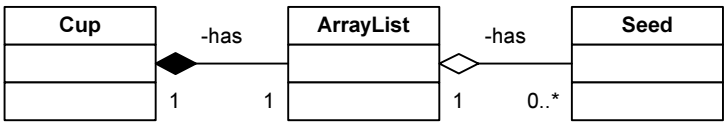


Рис. 16.9

Для того чтобы создать схему, представленную на рис. 16.9, в программе MS Visio сделайте то же, что вы делали для агрегации.

16.3. Наследование, интерфейсы и внутренние классы

В *главе 5* мы познакомились с графическими классами. Предоставляемые системой классы `Point` могут быть использованы для иллюстрации идеи наследования. Если вы обратитесь к документации Java API, то увидите четыре класса, иерархически связанных друг с другом. Классы `Point`, `Point2D.Double` и `Point2D.Float` являются подклассами класса `Point2D`. Такого типа иерархические отношения называются *иерархией наследования*. Подклассы наследуют переменные экземпляра и методы от своих суперклассов. Подклассы также могут содержать иные компоненты, которые делают их отличными от других.

Полезно обобщить все это с помощью схем UML. Отношения наследования между классами отображаются с помощью стрелок от подклассов к их суперклассам. Стрелки изображают сплошной

линией с замкнутым наконечником. Хотя наследование идет сверху вниз в иерархии, стрелки указывают от более частного к более общему классу. Суперкласс может иметь несколько подклассов, и может оказаться непрактичной попытка показать все подклассы какого-либо суперкласса. Однако подкласс имеет только один непосредственный суперкласс. Это значит, что при записи отношений более практично, чтобы каждый подкласс соотносился со своим суперклассом. Рассмотрим такую схему для классов `Point` (рис.16.10).

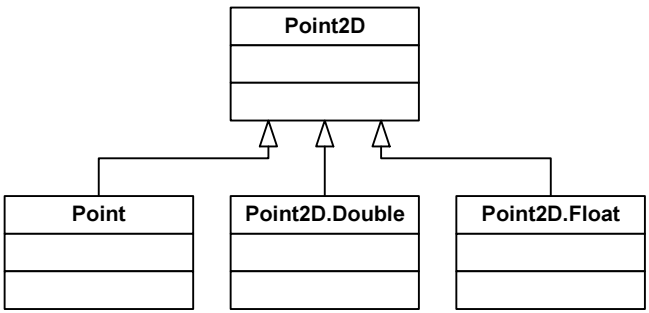


Рис. 16.10

Чтобы создать схему, представленную на рис. 16.10, нужно сделать следующее в программе MS Visio:

1. В разделе **UML Static Structure** перетащите пиктограмму **Generalization** в рабочую область.
2. В рабочей области убедитесь, что стрелки направлены от подкласса к суперклассу.

Схема на рис. 16.11 первоначально была показана в *главе 13* (см. рис. 13.2). Обратите внимание, что прямоугольник, обозначающий интерфейс, отличается от прямоугольника, обозначающего класс. Он имеет помету «`interface`» (в угловых кавычках). Стрелки, обозначенные пунктирной линией, используются для указания реализации интерфейса.

Построить схему, представленную на рис. 16.11, можно, выполнив в программе MS Visio такие действия:

1. В разделе **UML Static Structure** перетащите пиктограмму **Interface** в рабочую область.

2. В рабочей области два раза щелкните кнопкой мыши на пиктограмме и введите имя интерфейса.
3. В разделе **UML Static Structure** перетащите пиктограмму **Generalization** в рабочую область.
4. В рабочей области выделите стрелку и щелкните правой кнопкой мыши для вызова контекстного меню. Выберите команду **Format | Line**. Выберите пунктирную линию из списка в диалоговом окне.

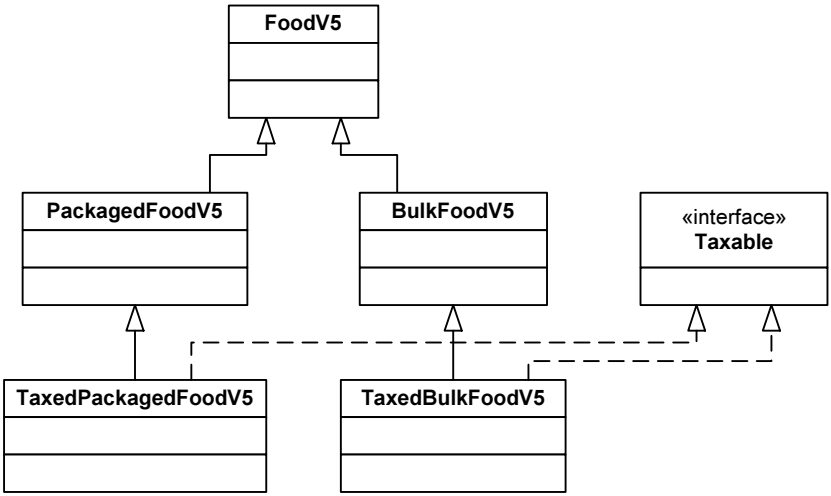


Рис. 16.11

Вложенные классы — это такие классы, определения которых полностью содержатся внутри определений других классов. Внутренние классы являются нестатическими вложенными классами. Эти термины будут использоваться как взаимозаменяющие, т. к. ни один из примеров не связан со статическими классами. Классы группы `Point`, предоставляемые системой, иллюстрируют идею вложенности. Определение класса `Point2D.Double` содержится в определении класса `Point2D`. *Внутренние классы* — это не то же самое, что и подклассы, но внутренний класс может быть подклассом внешнего класса. Примером такого является класс `Point2D.Double`.

К сожалению, в символах UML имеются разные варианты, и программа MS Visio включает не все из них. Одним из способов обозначить отношения между внутренними и внешними классами будет поместить небольшую окружность со знаком плюс рядом с внешним классом и соединить ее с внутренним классом сплошной линией. Посмотрите эту ссылку о символах UML: **www.holub.com/goodies/uml**. Если вы откроете эту страницу, то внизу увидите схему, как на рис. 16.12, показывающую обозначение внутренних классов.

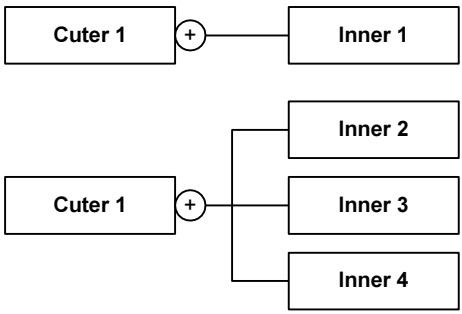


Рис. 16.12

В зависимости от того, какая версия программы MS Visio установлена на вашем компьютере, самое близкое, что вы сможете изобразить, — это поставить знак "плюс" рядом с маленькой окружностью. Это показано на рис. 16.13 и будет использовано в дальнейшем тексте.

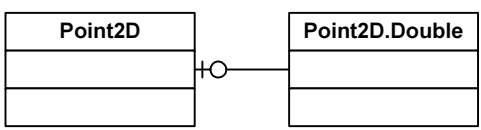


Рис. 16.13

Для того чтобы построить линию, показанную на рис. 16.13, нужно сделать следующее в программе MS Visio:

1. В разделе **UML Static Structure** перетащите пиктограмму **Generalization** в рабочую область.

2. В рабочей области выделите стрелку и щелкните правой кнопкой мыши для вызова контекстного меню. Выберите команду **Format | Line**. Выберите требуемый символ из списка **Line Ends/Begin** в диалоговом окне.

16.4. Схемы последовательностей

Кроме обозначения статической структуры, язык UML имеет символы для схематического изображения последовательности действий, предпринимаемых выполняемым кодом. Это можно проиллюстрировать, еще раз используя пример с `SeedDistributor` и `Cup`. Пусть у нас есть метод `disrtribute()` в классе `SeedDistributor`. Пусть он действует с помощью вызова метода `addSeeds()` класса `Cup`. В выполняемом коде это происходит между объектами, а не классами. Вверху диаграммы прямоугольники обозначают участвующие объекты. Вертикальные пунктирные линии жизни (lifelines) идут от объектов, где время движется сверху вниз. На эти линии накладываются прямоугольники активации, представляющие выполнение кода метода. Горизонтальные стрелки между активациями представляют передачу сообщений между методами, т. е. вызовы и возвращения методов. Пример, приводимый на рис. 16.14, является простым, так что там нет ни параметров, ни возвращаемых значений. В более сложном случае их можно было бы легко добавить; параметры записываются в скобках метода, а возвращаемые значения изображаются как пометки на возвращаемых стрелках.

Для того чтобы построить схему, представленную на рис. 16.14, нужно сделать следующее в программе MS Visio:

1. В разделе **Sequence** перетащите пиктограммы **Object Lifeline**, **Activation**, **Message (call)** и **Message (return)** в рабочую область и расположите их так, как требуется.
2. В рабочей области два раза щелкните кнопкой мыши на связях объекта для наименования объекта; два раза щелкните кнопкой мыши на стрелках сообщений для наименований их, укажите их характеристики и т. д.

Еще одним аспектом схем для изображения последовательности действия является следующее. Если метод данного объекта вы-

зывает другой метод к тому же объекту, активации объекта складываются вместе, при этом стрелки показывают вызов и возвращение (рис. 16.15).

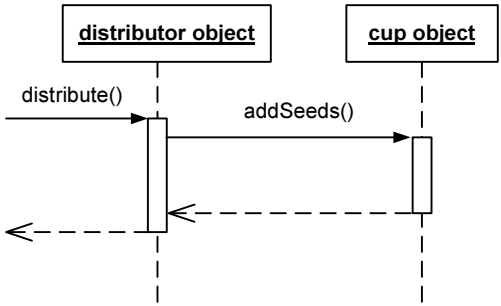


Рис. 16.14

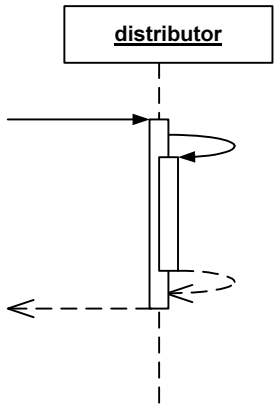
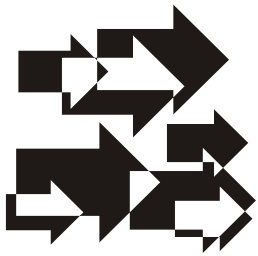


Рис. 16.15

ГЛАВА 17



Контейнеры, рисование и диалоговые окна

Данная глава представляет собой введение к материалу о том, как написать код с графическим интерфейсом пользователя. Первые два раздела содержат вводные сведения и служат отправной точкой для чтения документации Java API, если вам потребуется более детальное описание. Третий раздел содержит первые примеры кодов.

17.1. Контейнеры и компоненты

Посмотрим на фрагмент иерархии системных классов, где представлены некоторые классы, необходимые для создания графических интерфейсов пользователя (рис. 17.1).

Хотя классы `JFrame`, `JDialog` и `JApplet` находятся внизу иерархии, они известны как контейнеры верхнего уровня. Они являются основным строительным материалом для графических приложений. Класс `JPanel` является важным подклассом класса `JComponent`. Другие классы, имеющиеся в системе, находятся на этом уровне иерархии, как это показано в прямоугольнике "другие классы `JComponent`" на схеме. Компонент, который включает такой текст, как `JTextBox`, будет находиться под классом `JTextComponent` в иерархии. Примеры программ в данной и следующих главах покажут использование этих и иных компонентов.

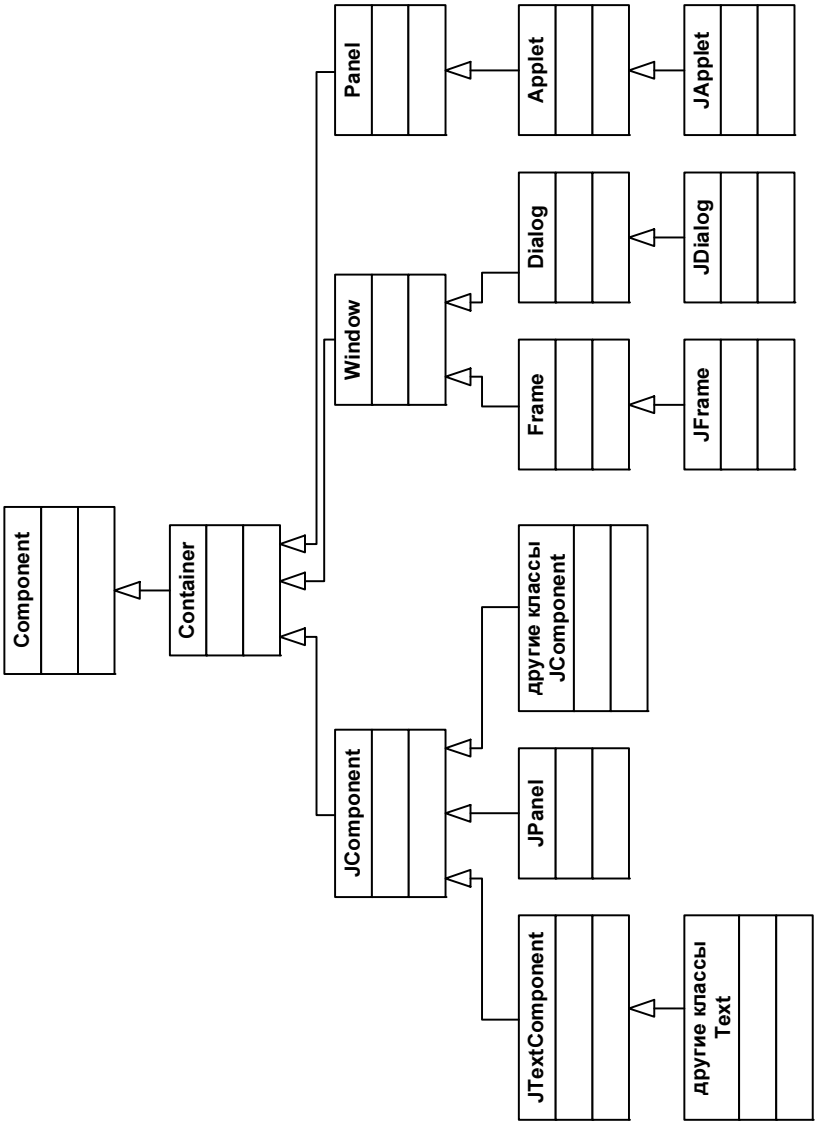

Component
Container
Panel
Window
Applet
JApplet
Dialog
JDialog
Frame
JFrame
JComponent
другие классы JComponent
JPanel
JTextComponent
другие классы Text

Рис. 17.1

Все компоненты графического приложения или апплета должны содержаться в контейнере верхнего уровня. Для апплетов таким контейнером верхнего уровня является `JApplet`. Для приложений, описываемых в этой и следующей главах, таким контейнером верхнего уровня будет `JFrame`. Неформально, `JFrame` — это класс, предоставляемый системой, который создает окно на экране, и графические элементы могут быть помещены во фрейм. Класс `JPanel` полезен в качестве субконтейнера для организации элементов во фрейме.

Контейнеры верхнего уровня имеют структуру, которая позволяет организовать графические компоненты приложений. Вот краткое описание этой структуры. Контейнер верхнего уровня имеет корневое подокно. Корневое подокно управляет еще одним подокном (layered pane), которое функционально связано с упорядочением графических компонентов при их изображении. В свою очередь, это подокно включает подокно содержания и строку меню, если она есть. Корневое подокно также управляет прозрачным подокном. Прозрачное подокно "покрывает" весь контейнер верхнего уровня. Функционально оно связано с получением щелчков от мыши и рисованием поверх сложных компонентов.

Код для простых графических приложений напрямую не затрагивает большинство из указанных выше элементов структуры. Простые приложения используют фрейм и его подокно содержания, как это описано в следующих четырех пунктах.

- Каждый контейнер верхнего уровня имеет подокно содержания, которое является контейнером для видимых компонентов. В коде для экземпляра контейнера верхнего уровня можно получить доступ к подокну содержания вызовом метода `getContentPane()`. Подокно содержания является экземпляром класса `JComponent`, но метод возвращает ссылку суперкласса в класс `Container`.
- Каждый видимый компонент приложения должен находиться в иерархии содержания, исходящей от подокна содержания. Каждый компонент должен содержаться внутри другого компонента. Как правило, одна или две панели находятся на вершине иерархии и присутствуют внутри подокна содержания фрейма. Другие элементы могут быть добавлены к панелям.

- Компонент помещается в контейнер вызовом метода `add()` к контейнеру и передачей компонента в качестве параметра. Каждый компонент может быть добавлен только к одному контейнеру. Таким образом и строится иерархия содержания в коде.
- Меню тоже являются важной частью многих графических приложений. Они добавляются к фрейму приложения, а не к подокну содержания.

Графическое приложение может использовать классы, находящиеся ниже класса `JComponent` в схеме наследования, приведенной на рис. 17.1. Оно также может содержать классы, написанные программистом, которые являются продолжением классов, имеющихся в системе. Наконец, оно также может иметь классы, написанные пользователем, которые реализуют функциональность приложения. Классы и подклассы, находящиеся в системе, имеют модель, которая связана с ними. Например, экземпляр класса `JButton` имеет объект `ButtonModel`. Модель хранит состояние компонента. Когда состояние изменяется, система обновляет внешний вид компонента. Разделение компонентов и моделей желательно по двум причинам: данные программы не копируются в сами компоненты; при необходимости особая функциональность может быть реализована в специально созданной модели компонента. Система обращается с обновлениями за счет оповещения *блока прослушивания событий* (listener), связанного с компонентом. Блок прослушивания событий — это часть кода внутри класса, который может обнаружить и среагировать на событие, например, щелчок мыши.

Простые программы не работают напрямую с моделями системы. Система обрабатывает обновления автоматически. Более сложные примеры, которые приводятся в последующих главах, иллюстрируют использование моделей и видов. Классы будут содержать данные в переменных экземпляра, равно как и атрибуты, определяющие их внешний вид в приложении. Программы также будут содержать блоки прослушивания событий, написанные программистом, которые реагируют на графические события.

17.2. Рисование

В языке Java существуют два вида компонентов и два вида рисования. Вы работаете с компонентами, которые либо создали сами, либо предоставляют классы, имеющиеся в системе. Рисование может быть запущено или приложением, или системой. Рисование, запущенное приложением, обновляет внешний вид компонента, когда его состояние или содержание изменились. Для тех компонентов, которые вы создаете сами, необходимо написать специальный код и явно сделать вызовы, которые позаботятся о рисовании для данного случая. Для компонентов, предоставляемых системой, большинство требований в отношении рисования, даже если оно запущено приложением, будет выполнено самой системой.

Рисование, запущенное системой, происходит при следующих трех условиях:

- когда компонент впервые появляется на экране;
- когда меняются размеры компонента;
- когда компонент, который был полностью или частично закрыт другим компонентом, открывается и требуется его рисование заново.

Класс `JComponent` имеет несколько методов рисования, которые включают `paint()` и `paintComponent()`. Они наследуются подклассами данного класса, в том числе подклассом `JApplet`. В *главе 5* были показаны примеры простых апплетов, которые были реализованы с помощью переопределения метода `paint()` в классе `JApplet`. В данной главе метод `paintComponent()` будет переопределен в компонентах некоторых классов, и это будет основой для графического изображения приложений.

Рисование основано на том, что называется механизмом *обратного вызова* (callback). Метод рисования переопределяется, но рисование не выполняется прямыми вызовами этого метода в коде. Программа вызывает другой метод, предоставляемый системой, чем вызывается последовательность вызовов. Система делает вызовы нескольких методов и затем вызывает переопределенный метод рисования. В *главе 5* именно система в конечном итоге

делала вызов метода `paint()`, который вызывал появление апплета, точно так же, как система вызывает метод `main()` для простых приложений.

В примерах программ в данной главе класс `JPanel` расширяется. Иными словами, определяется специальный подкласс класса `JPanel`. В этом подклассе метод `paintComponent()` переопределяется. В тех методах приложения, где содержимое панели меняется, имеется вызов к наследованному методу `repaint()` без параметров. Этот вызов инициирует последовательность обратных вызовов. В методе `repaint()` вызываются другие методы, и затем делается вызов метода `paintComponent()` на панели, что приводит к рисованию всей панели. В более сложных приложениях может быть необходимым перерисовать только часть компонента или его содержания, и версии метода `repaint()`, которые принимают параметры, существуют для этого.

Рассмотрим перечень событий в процедуре обратного вызова. Обратите внимание, что наведение порядка включает вызов метода `update()`, который обновляет базовую модель компонента. В простых приложениях программисту не требуется непосредственно делать такой вызов — он совершается методом `repaint()`.

1. Программа определяет, что нечто требует обновления на экране.
2. Программа делает вызов метода `repaint()`.
3. Метод `repaint()` вызывает метод `update()`. Этот метод очищает фон компонента. После вызова метода `update()` делается вызов метода `paintComponent()` компонента, который инициировал процедуру обратного вызова.

Приложение должно вызвать метод `repaint()`, а не метод `paintComponent()`. Этим гарантируется, что вызов метода `update()` будет произведен. В более сложных приложениях можно переопределить метод `update()` для того, чтобы выполнить постепенное рисование, что может быть более эффективным. Приложение также может содержать контейнеры внутри других контейнеров. В такой ситуации предоставляемый системой метод `update()` не выполняет автоматическую реализацию рекурсивного рисования

компонентов. Для выполнения этого надо будет переопределить метод `update()`.

Есть еще один важный аспект правильной реализации рисования в графических приложениях. Когда метод `paintComponent()` переопределяется, он должен включать вызов метода `super.paintComponent()`. Этим вызывается метод `paintComponent()` класса `Container`, который отвечает за перерисовывание любых других компонентов контейнера. Он координирует процедуру перерисовывания, если они пересекаются. В приведенных примерах кода компоненты не перекрываются. Если такое происходит, то тот элемент, состояние которого было изменено (что и инициировало вызов), будет обновлен. Другие компоненты также будут заново воспроизведены, но без изменений.

В целом, полезно полагаться на функциональность системы для обновления внешнего вида. Однако причины плохой работы крупных графических приложений часто заключаются в рисовании. Неэффективность и большие затраты времени могут быть результатом того, что код построен так, чтобы полностью обновлять внешний вид, в то время как изменилось только ограниченное количество компонентов. В таком случае имеет смысл написать код для постепенного рисования, для чего может потребоваться переопределение классов, имеющихся в системе.

Итак, необходимо сделать три действия для поддержки рисования в простых графических приложениях вроде тех, что будут показаны в примерах этой и следующих глав:

- реализовать метод `paintComponent()` в подходящем классе приложения, например, в специальном подклассе класса `JPanel`;
- убедиться, что реализация метода `paintComponent()` включает вызов метода `super.paintComponent()`;
- разместить вызовы метода `repaint()` в приложении в тех местах, где меняется состояние приложения. Это приведет к тому, что измененное состояние будет отражено в графическом представлении.

17.3. Диалоговые окна и строковый вывод на панель

Последний раздел этой главы посвящен примерам трех программ: Echo1.java, Echo2.java и Echo3.java. Это начало серии из девяти приложений, называемых Echo, которая продолжается в следующей главе. Большинство примеров принимают ввод и показывают его в виде графического вывода. Некоторые являются просто иллюстрациями синтаксиса и не создают эхо (отображение на экране нажимаемых на клавиатуре символов). В этой главе появятся первые примеры использования классов `JFrame` и `JPanel`. Эти примеры показывают, шаг за шагом, как создать приложение, которое имеет интегрированный ввод/вывод и не зависит от командной строки или класса `MyTerminalIO`.

Информация об этих примерах в этой и следующей главе организована таким образом:

- краткое введение, сжато излагающее, что выполняет данное приложение;
- скриншоты, иллюстрирующие, что видно на экране, когда выполняется приложение. Эти снимки окон приводятся не для всех примеров;
- схемы UML, показывающие компоненты и структуру приложения;
- комментарии к содержанию кода; приводятся пояснения, но не даются построчные комментарии;
- код Java для приложения без комментариев.

Echo1

Эта программа показывает использование имеющегося в системе диалогового окна ввода. Вывод осуществляется с помощью объекта `MyterminalIO`. Поскольку методы ввода класса `MyTerminalIO` используют диалоговые окна, интерфейс пользователя, представляемый программой во время выполнения, не отличается от интерфейса, который был бы представлен программой, которая использует только `MyTerminalIO` (рис. 17.2 и 17.3).

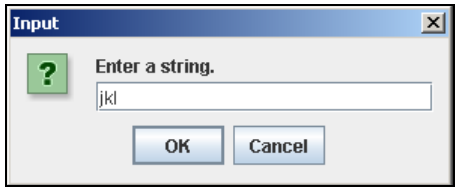


Рис. 17.2

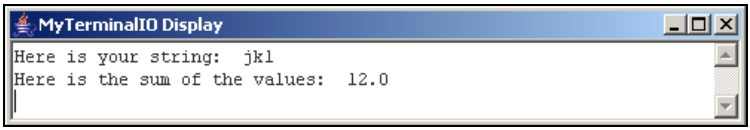


Рис. 17.3

Поскольку в этом приложении создается только один объект, `myterminal`, то схема UML очень проста (рис. 17.4).

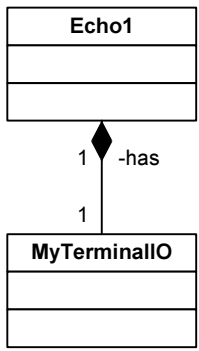


Рис. 17.4

При использовании диалоговых окон не создается никакого объекта. Вызывается статический метод к предоставляемому системой классу, `JOptionPane`. Метод `showInputDialog()` всегда привносит экземпляр класса `String`, так что если требуется числовой тип, то необходим анализ содержания этой переменной. Методы, которые проводят такой анализ, называются `parseInt()` и `parseDouble()`. Вызов метода `exit()` не связан с использованием диалоговых окон. Он включен для иллюстрации кодировки явно-

го конца программы. Когда метод `exit()` используется таким образом, последнее сообщение "Enter anything to quit" ("Нажмите любую клавишу для завершения") необходимо для того, чтобы сохранить существование объекта `MyTerminalIO`, в котором находится вывод. Как только вызывается метод `exit()`, все объекты, связанные с программой, исчезают. Вот этот код:

```
import javax.swing.*;
public class Echo1
{
  public static void main(String[] args)
  {
    String inputString;
    int inputInt;
    double inputDouble;
    double result;
    MyTerminalIO myterminal = new MyTerminalIO();
    inputString =
      JOptionPane.showInputDialog("Enter a string.");
    myterminal.println("Here is your string:  " +
      inputString);
    inputString = JOptionPane.showInputDialog
      ("This time enter an integer.");
    inputInt = Integer.parseInt(inputString);
    inputString = JOptionPane.showInputDialog
      ("This time enter a double.");
    inputDouble = Double.parseDouble(inputString);
    result = inputInt + inputDouble;
    myterminal.println("Here is the sum of the values:  " +
                       result);
    inputString =
      JOptionPane.showInputDialog("Enter anything to quit.");
    System.exit(0);
  }
}
```

Echo2

Эта программа показывает, как осуществить текстовый вывод на панель, принадлежащую фрейму в приложении. Она является

одной из программ в серии Echo, но в данном случае название не совсем точно. Программа не осуществляет ввод или эхо. Она просто производит заданный вывод (рис. 17.5).

Рис. 17.5

Схема UML для этого приложения хорошо подытоживает структуру программы. Приложение имеет специальный фрейм, `Echo2Frame`, который является подклассом класса `JFrame`. `Echo2Frame` имеет подокно содержания. Приложение также имеет специальную панель, `Echo2Panel`, которая является подклассом класса `JPanel`. Экземпляр этой панели добавлен к подокну содержания фрейма (рис. 17.6).

Поскольку это первый полномасштабный пример, рассмотрим его структуру подробнее:

1. В методе `main()`:
   - создается экземпляр класса `Echo2Frame`;
   - вызывается метод, который закрывает приложение, когда закрывается его фрейм;
   - видимые свойства фрейма устанавливаются на `true`.
2. В классе `Echo2Frame`:
   - фрейму дается название;
   - фрейму дается размер;
   - создается экземпляр класса `Echo2Panel`;
   - устанавливается ссылка на подокно содержания фрейма;
   - экземпляр класса `Echo2Panel` добавляется к подокну содержания.

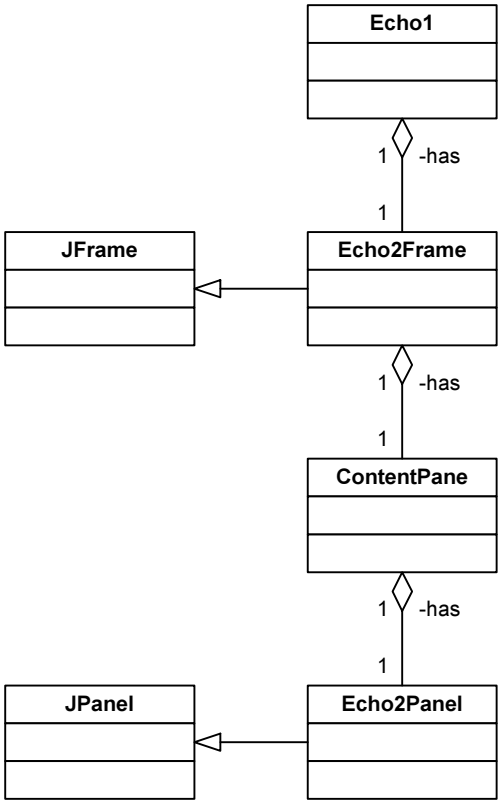


Рис. 17.6

3. В классе `Echo2Panel`:

   - метод `paintComponent()` имеет графический параметр, и этот параметр приведен к `Graphics2D`;
   - делается вызов метода `super.paintComponent()`.

Имеется одна строка кода, использующая графический метод, наподобие того, как это было изложено в *главе 5*, в результате графический вывод появляется в определенном месте на панели.

Поскольку данное приложение выполняет только одну операцию, его состояние не может поменяться, соответственно метод `repaint()` не вызывается в коде:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo2
{
  public static void main(String[] args)
  {
    Echo2Frame myframe = new Echo2Frame();
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myframe.setVisible(true);
  }
}
class Echo2Frame extends JFrame
{
  public Echo2Frame()
  {
    setTitle("Echo2 Frame");
    setSize(360, 180);
    Echo2Panel mypanel = new Echo2Panel();
    Container frameContentPane = getContentPane();
    frameContentPane.add(mypanel);
  }
}
class Echo2Panel extends JPanel
{
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    g2.drawString("This is hardcoded, not echoed.", 80, 80);
  }
}
```

Echo3

Это настоящая эхо-программа. Вместо использования `MyTerminalIO` она объединяет диалоговое окно Echo1 с подокном вывода Echo2. Пользователь может неоднократно вводить строки в диалоговое окно, и они будут отражаться на панели (рис. 17.7 и 17.8).

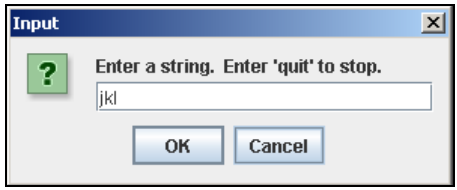


Рис. 17.7

Рис. 17.8

Общая структура данного приложения не сильно отличается от Echo2, как это видно на схеме UML (рис. 17.9). В целях иллюстрации схема показывает более детально атрибуты и операции. На ней указано, что `Echo3Panel` имеет объект класса `Graphics`, что было пропущено на предыдущей схеме. Она также показывает, что класс `Echo3Panel` имеет переменную экземпляра класса `String`. Эта переменная содержит ввод и позволяет перевести его в вывод.

Программа имеет циклический характер, чтобы продолжать принимать ввод. Большая часть данного приложения аналогична Echo2. Класс `Echo3Frame` имеет метод `setPanelString()`, что позволяет инициировать изменения строки панели внутри метода `main()`. Класс `Panel` имеет метод `setStringToShow()`, который завершает процесс. Этот метод делает две вещи: сначала он приписывает явный параметр переменной экземпляра панели, а затем он вызывает метод `repaint()`. Установка значения строки является изменением состояния приложения, поэтому необходимо обновить его внешний вид для отражения нового состояния.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo3
```

```
{
  public static void main(String[] args)
  {
    String localString = "go";
    Echo3Frame myframe = new Echo3Frame();
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    myframe.setVisible(true);
    while(!(localString.equals("quit")))
    {
      localString = JOptionPane.showInputDialog
        ("Enter a string. Enter 'quit' to stop.");
      myframe.setPanelString(localString);
      myframe.setVisible(true);
    }
  }
}
class Echo3Frame extends JFrame
{
  private Echo3Panel mypanel;
  public Echo3Frame()
  {
    setTitle("Echo3 Frame");
    setSize(360, 180);
    mypanel = new Echo3Panel();
    Container frameContentPane = getContentPane();
    frameContentPane.add(mypanel);
  }
  public void setPanelString(String stringIn)
  {
    mypanel.setStringToShow(stringIn);
  }
}
class Echo3Panel extends JPanel
{
  private String stringToShow;
  public void setStringToShow(String stringIn)
  {
    stringToShow = stringIn;
    repaint();
  }
```

```
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    g2.drawString(stringToShow, 80, 80);
  }
}
```

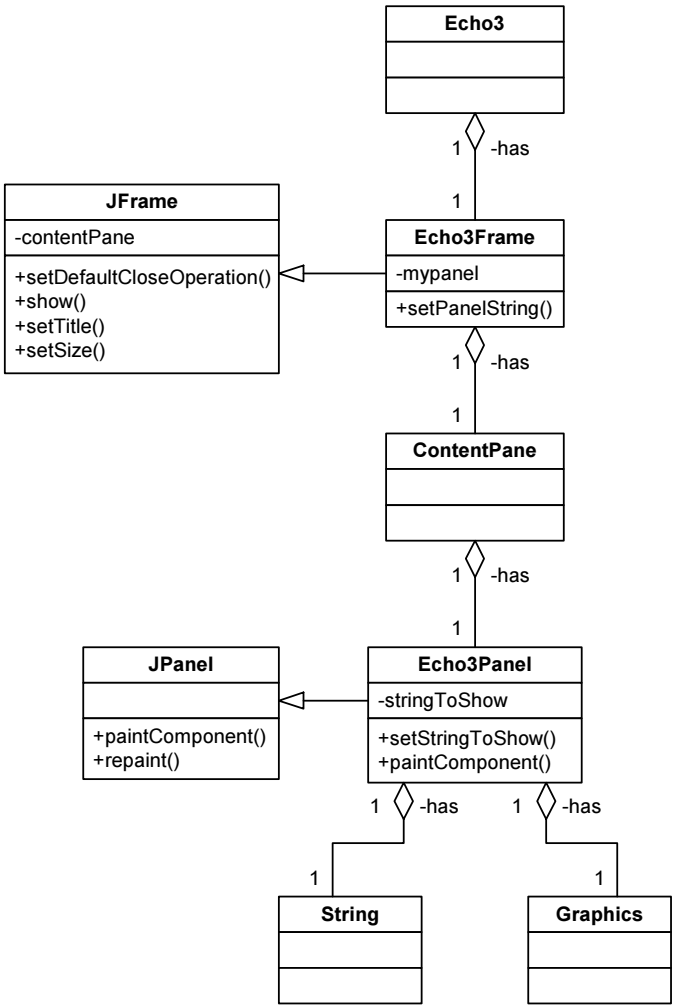
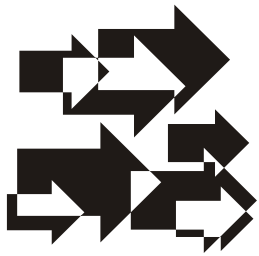


Рис. 17.9

ГЛАВА 18



Внутренние классы, блоки прослушивания и классы-адаптеры

В данной главе, которая является продолжением предыдущей, примеры программ объясняются с помощью тех же средств: скриншоты (screenshots), схемы UML, текст и код. Примеры развиваются до приложения с интегрированным текстовым окном для ввода, которое показывает вывод с помощью графического представления объекта в окне.

18.1. Внутренние классы, блоки прослушивания и интегрированные внутренние поля

Внутренний класс — это такой класс, определение которого находится внутри определения другого класса. *Блок прослушивания* — это фрагмент кода, который может обнаружить и среагировать на графические события, такие как щелчок мыши, например. Если блок прослушивания реализован как внутренний класс, его методы будут иметь прямой доступ к переменным того класса, в котором он находится. Это может быть удобно при создании приложений с целью уменьшения количества необходимых вызовов метода, для того чтобы вызвать изменения в состоянии приложения. Примеры программ с Echo4 по Echo6 предназначены для демонстрации внутренних классов и блоков прослушивания и их использования.

Echo4

Данная программа не совершает ничего полезного, и для нее не приводятся скриншоты. Она иллюстрирует синтаксис внутренних классов и показывает, как внутренний класс осуществляет прямой доступ к переменным экземпляра внешнего класса, в котором он находится. Схема UML (рис. 18.1) иллюстрирует использование знака "плюс" и маленькой окружности для внутренних классов. В то же время внешний класс может иметь экземпляр внутреннего класса в отношениях агрегации или композиции.

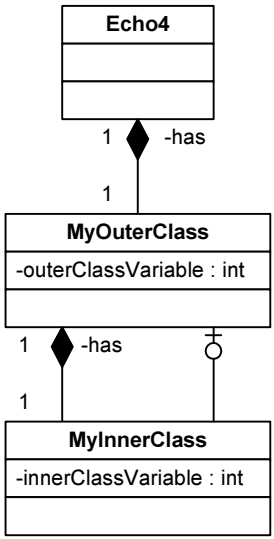


Рис. 18.1

В этом коде экземпляр внешнего класса создается в методе `main()`. Далее следует вызов метода `makeInnerObject()`. Внутри кода для этого метода вызывается конструктор внутреннего класса. Ссылка на объект внутреннего класса, который строится, не хранится в переменной экземпляра. Она хранится в переменной, местной для метода, в котором она создается. Схема UML указывает, что между внешним классом и экземпляром внутреннего класса существуют отношения композиции.

Основной момент кода показан в конструкторе для внутреннего класса. Значение переменной экземпляра внешнего класса присваивается переменной экземпляра внутреннего класса. Не ставится точка при определении имени переменной внешнего класса, и не существует вызова метода `get` для того, чтобы получить значение переменной внешнего класса. Эта переменная напрямую доступна внутри всех методов внутреннего класса. Именно это и делает внутренние классы удобными. Это также означает, что программисту не следует использовать одинаковые имена для переменных внутреннего и внешнего классов, потому что может возникнуть путаница.

```
public class Echo4
{
  public static void main(String[] args)
  {
    MyOuterClass myOuterObject =
      new MyOuterClass(5);
    myOuterObject.makeInnerObject();
  }
}
class MyOuterClass
{
  private int outerClassVariable;
  public MyOuterClass(int inval)
  {
    outerClassVariable = inval;
  }
  public void makeInnerObject()
  {
    MyInnerClass innerObject = new MyInnerClass();
  }
  private class MyInnerClass
  {
    private int innerClassVariable;
    public MyInnerClass()
    {
      innerClassVariable = outerClassVariable;
    }
  }
}
```

Echo5

Эта программа принимает переменную экземпляра класса `String` через интегрированное текстовое поле и отражает его в панели во фрейме приложения. Она принимает новый ввод каждый раз, когда клавиша <Enter> нажимается в текстовом поле, и будет работать до тех пор, пока фрейм не закроют (рис. 18.2).

Рис. 18.2

Схема UML (рис. 18.3) для этого приложения сочетает элементы нескольких предшествующих программ. Это приложение имеет класс `Echo5Frame` с переменной экземпляра класса `String`. Эта переменная содержит введенные данные, отражаемые как вывод. Класс `Echo5Frame` имеет подокно содержания с добавленным к нему классом `Echo5Panel`. Класс `Echo5Panel` показывает вывод и реализуется как внутренний класс класса `Echo5Frame`. Это значит, что методы класса `Echo5Panel` имеют прямой доступ к переменным экземпляра класса `Echo5Frame`. К подокну содержания также добавлена переменная экземпляра класса `JTextField`. В ней, в свою очередь, имеется блок прослушивания, который реализован как внутренний класс класса `Echo5Frame`. Это значит, что когда приложение обнаруживает ввод в `JTextField`, методы блока прослушивания совершают вызовы, которые влияют на то, что изображено в `Echo5Panel`. Последнее, что следует из схемы UML — это то, что блок прослушивания для `JTextField` реализует интерфейс блока прослушивания действий.

Этот пример показывает механизм обработки событий в языке Java. Приложение будет многократно принимать новый ввод через текстовое поле и отражать его на панели, но код приложения не содержит цикл. Система Java постоянно проверяет, не проис-

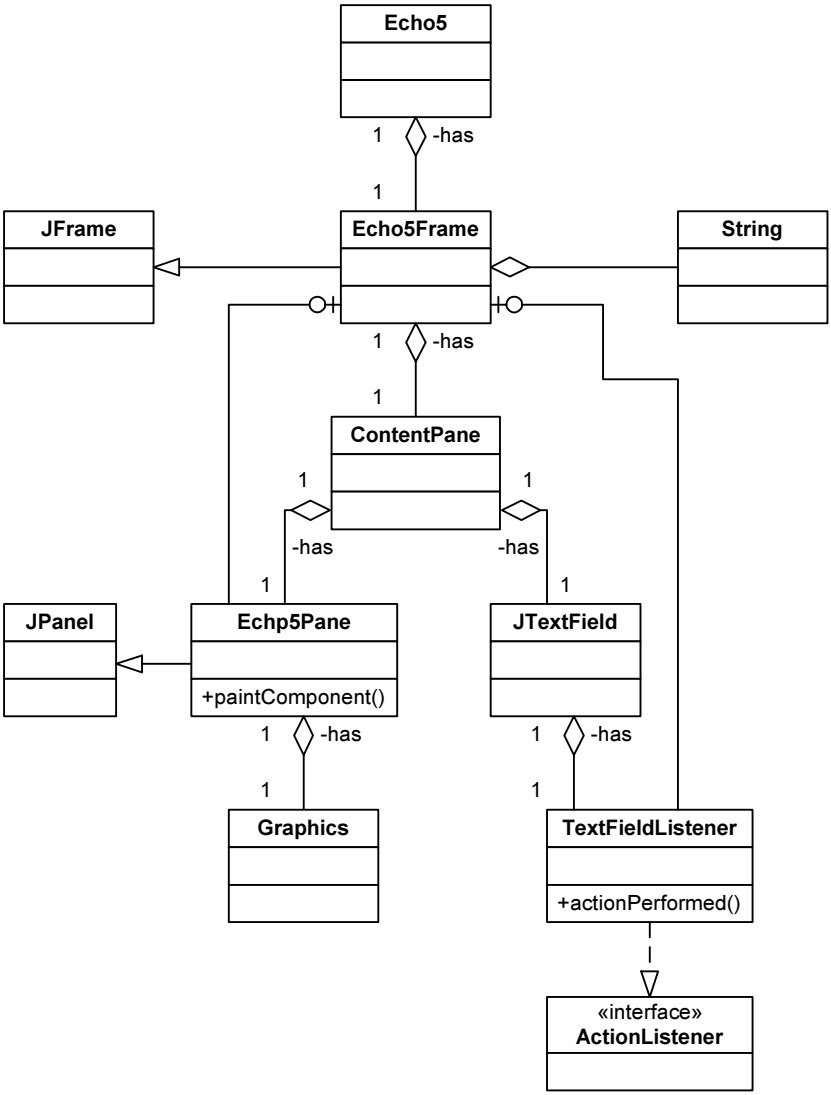


Рис. 18.3

ходят ли графические события, такие как движение указателя или щелчки мыши, нажатие клавиш и т. д. Если в приложении имеется блок прослушивания, который соответствует определенному

типу событий, система вызывает этот блок и передает ему это событие, когда оно происходит. Блоки прослушивания имеют методы, которые предпринимают определенные действия, когда их извещают, что произошло некоторое событие.

Обратите внимание, что есть некоторое сходство между обработкой событий и рисованием. Вы пишете код для методов, но нигде в самом приложении вы не вызываете эти методы. Вы полагаетесь на механизм самой системы, который и делает такой вызов. В случае блоков прослушивания таким скрытым механизмом является механизм обработки событий. Упрощенно, вы можете представить для себя функционирование системы в этой ситуации в виде цикла. Система совершает цикл, проверяя наличие событий. Когда событие происходит, система вызывает соответствующий системный метод, если в приложении имеется блок прослушивания для такого события. Когда метод действия завершается, если система обнаруживает другое событие, то процесс повторяется.

В отличие от программы Echo3, в данном приложении переменная класса `String` является переменной экземпляра класса `Echo5Frame`. Блок прослушивания и панель являются внутренними классами класса `Echo5Frame`. Это дает обоим прямой доступ к переменной экземпляра класса `String`. Блок прослушивания может установить значение переменной экземпляра, а панель может получить это значение, чтобы вывести его на экран. В данном приложении вызов метода `repaint()` происходит в методе действия блока прослушивания. Именно здесь инициируются изменения в состоянии приложения.

При добавлении нескольких компонентов к панели содержания можно использовать такие строки, как "Center", "North", "East", "South" и "West", для указания относительного места. Система позаботится о точном расположении компонентов. Числовые значения для отправной точки строки были уже использованы и продолжают использоваться. Соотношение между числовыми значениями и действительным положением во фрейме может быть непросто установить. В этом и иных примерах расположение элементов делается "на глаз".

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo5
{
  public static void main(String[] args)
  {
    Echo5Frame myframe = new Echo5Frame();
    myframe.setVisible(true);
  }
}
class Echo5Frame extends JFrame
{
  private String stringInQuestion = "";
  private JTextField myField;
  private Echo5Panel myPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  private final int STRINGX = 80;
  private final int STRINGY = 80;
  public Echo5Frame()
  {
    setTitle("Echo5 Frame");
    setSize(FRAMEW, FRAMEH);
    myField = new JTextField();
    TextFieldListener myListener = new TextFieldListener();
    myField.addActionListener(myListener);
    myPanel = new Echo5Panel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
    contentPane.add(myField, "North");
  }
  private class TextFieldListener
    implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    {
      stringInQuestion = myField.getText();
      myField.setText("");
```

```
      myPanel.repaint();
    }
  }
  private class Echo5Panel extends JPanel
  {
    public Echo5Panel()
    {
    }
    public void paintComponent(Graphics g)
    {
      Graphics2D g2 = (Graphics2D) g;
      super.paintComponent(g2);
      g2.drawString(stringInQuestion, STRINGX, STRINGY);
    }
  }
}
```

Echo6

Данный пример отражает ввод, как и предыдущий, и на экране это выглядит так, как показано на рис. 18.4.

Рис. 18.4

По сравнению с предыдущим приложением внутренняя структура изменилась. Класс `Echo6Panel` уже не является внутренним классом класса `Echo6Frame` и переменная экземпляра `String` снова является частью `Echo6Panel`, как в `Echo3`. В этом отличие от программы Echo5. Блок прослушивания по-прежнему имеет прямой доступ к `Echo6Panel` в классе `Echo6Frame`, но он не имеет прямого доступа к переменным экземпляра `Echo6Panel`. Такая структура легче поддается обобщению, но за это надо платить: классу

`Echo6Panel` требуется свой метод `set` для переменной экземпляра `String` (рис. 18.5).

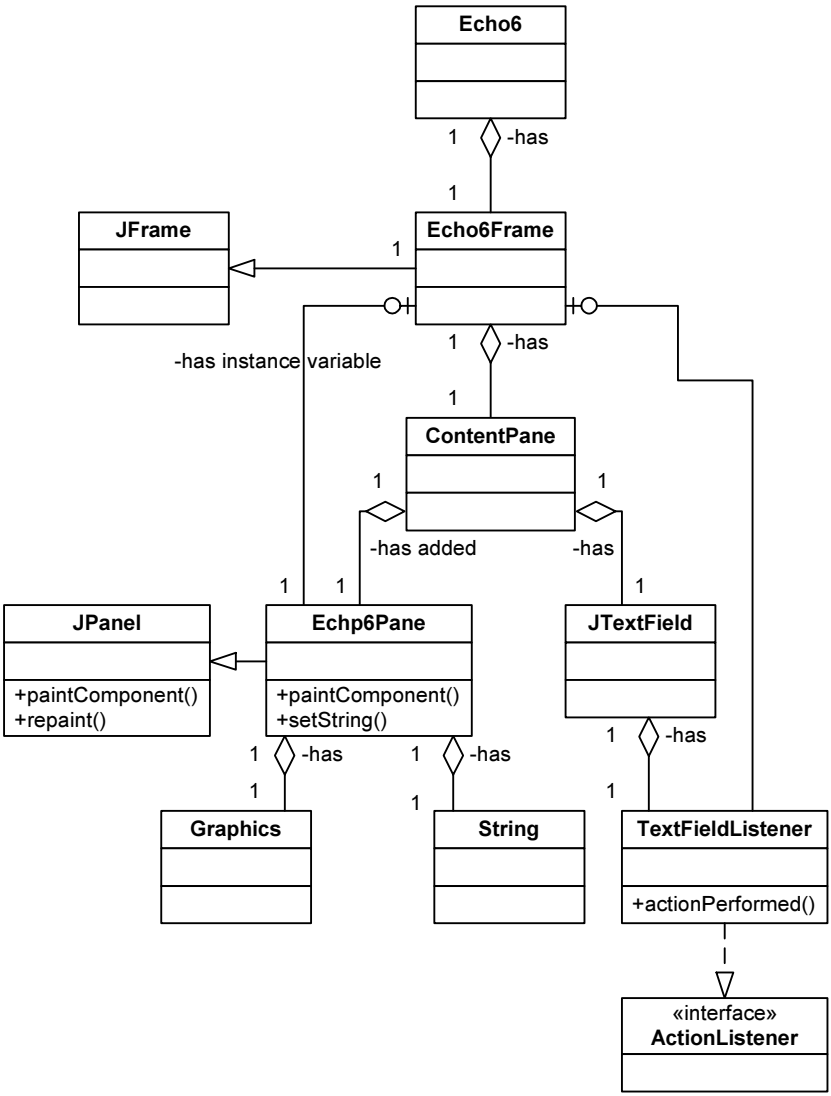


Рис. 18.5

Изменения в коде показывают: определение класса `Echo6Panel` вне класса `Echo6Frame`; объявление переменной `stringInQuestion` в классе `Echo6Panel`; вызов метода `set` к `Echo6Panel` в блоке прослушивания; определение метода `set` в классе `Echo6Panel`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo6
{
  public static void main(String[] args)
  {
    Echo6Frame myframe = new Echo6Frame();
    myframe.setVisible(true);
  }
}
class Echo6Frame extends JFrame
{
  private JTextField myField;
  private Echo6Panel myPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public Echo6Frame()
  {
    setTitle("Echo6 Frame");
    setSize(FRAMEW, FRAMEH);
    myField = new JTextField();
    TextFieldListener myListener =
      new TextFieldListener();
    myField.addActionListener(myListener);
    myPanel = new Echo6Panel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
    contentPane.add(myField, "North");
  }
  private class TextFieldListener
    implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
```

```
    {
      String inputString = myField.getText();
      myPanel.setString(inputString);
      myField.setText("");
      myPanel.repaint();
    }
  }
}
class Echo6Panel extends JPanel
{
  private String stringInQuestion = "";
  private final int STRINGX = 80;
  private final int STRINGY = 80;
  public Echo6Panel()
  {
  }
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    g2.drawString(stringInQuestion, STRINGX, STRINGY);
  }
  public void setString(String stringIn)
  {
    stringInQuestion = stringIn;
  }
}
```

18.2. Панели, классы-адаптеры и анонимные внутренние классы

Echo7

Эта программа обладает такой же функциональностью, что и предыдущая. Заметным различием является то, что текстовое поле имеет фиксированную ширину. Как выглядит приложение Echo7, показано на рис. 18.6.

У программы есть несколько структурных изменений по сравнению с предыдущей. Класс `JTextField` имеет текст по умолчанию,

у него есть размер, и он помещен в его собственный класс `JPanel` в подокне содержания. Если текстовое поле не поместить в его собственную панель, то назначение ему размера не будет иметь никакого эффекта. Единицы, в которых измеряется ширина поля, не соответствуют количеству символов в шрифте по умолчанию, который отображается в поле. Как и в случае с другими визуальными атрибутами, нужно просто выбрать то значение, которое хорошо выглядит на экране, чтобы не копаться в деталях того, как связаны размеры поля и шрифта.

Рис. 18.6

В примерах программ Echo1 и Echo2 было показано использование метода `setDefaultCloseOperation()`. В примерах Echo5 и Echo6 этот метод отсутствует. Закрытие программы было оставлено для выполнения системой по умолчанию. В данном примере в классе `Echo7Frame` имеется класс `WindowCloser`, который явно определяет, что нужно сделать, когда приложение закрывается. Если вы посмотрите на схему UML (рис. 18.7), то увидите, что закрывающий класс окна является подклассом класса `WindowAdapter`, а не реализатором интерфейса.

Класс `WindowCloser` является еще одним примером элемента, аналогичного блоку прослушивания. Он интересен тем, что демонстрирует концепцию класса-адаптера. Вспомните, что интерфейсы не содержат определения методов. Все, что у них есть, — это спецификации для всех методов, которые должны быть определены в классе, реализующем интерфейс. Потенциально, в интерфейсе может быть много методов. И хотя реализующий класс в конкретном приложении может задействовать только один или несколько из этих методов, он должен будет включать реализации всех из них.

Для таких интерфейсов, как интерфейсы блока прослушивания, система включает соответствующий класс-адаптер. Различие между интерфейсом и адаптером можно обобщить следующим образом. Пусть интерфейс содержит такую сигнатуру для метода:

```
public void someMethod(parameter);
```

Адаптер будет содержать следующее определение метода:

```
public void someMethod(parameter)
{
}
```

Если вы расширяете класс-адаптер, то наследуете пустые реализации всех остальных методов. Вам только нужно переопределить методы, которые потребуются вашему классу. В отличие от интерфейса, вам не надо включать реализации ненужных методов. В нашем примере `TextFieldListener` реализует интерфейс `ActionListener`, у которого только один метод, и наличие класса-адаптера не даст нам ничего в плане удобства. Класс `WindowAdapter` имеет методы, например, для активации и открытия, так же как и для закрытия. Это приложение создано так, чтобы прекращать работу при закрытии окна. Оно не рассчитано на то, чтобы реагировать на любые иные события в окне. В результате, закрывающий окно класс расширяет класс `WindowAdapter` и только переопределяет в нем метод `windowClosing()` (рис. 18.7).

Код отражает те отличия, которые были указаны ранее. Текстовое поле получает размеры, когда создается. Имеется новая панель, к которой добавляется текстовое поле, и эта панель добавляется к подокну содержания во фрейме. Имеется экземпляр класса `WindowCloser`, и он добавляется непосредственно к `Echo7Frame`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo7
{
  public static void main(String[] args)
  {
    Echo7Frame myframe = new Echo7Frame();
```

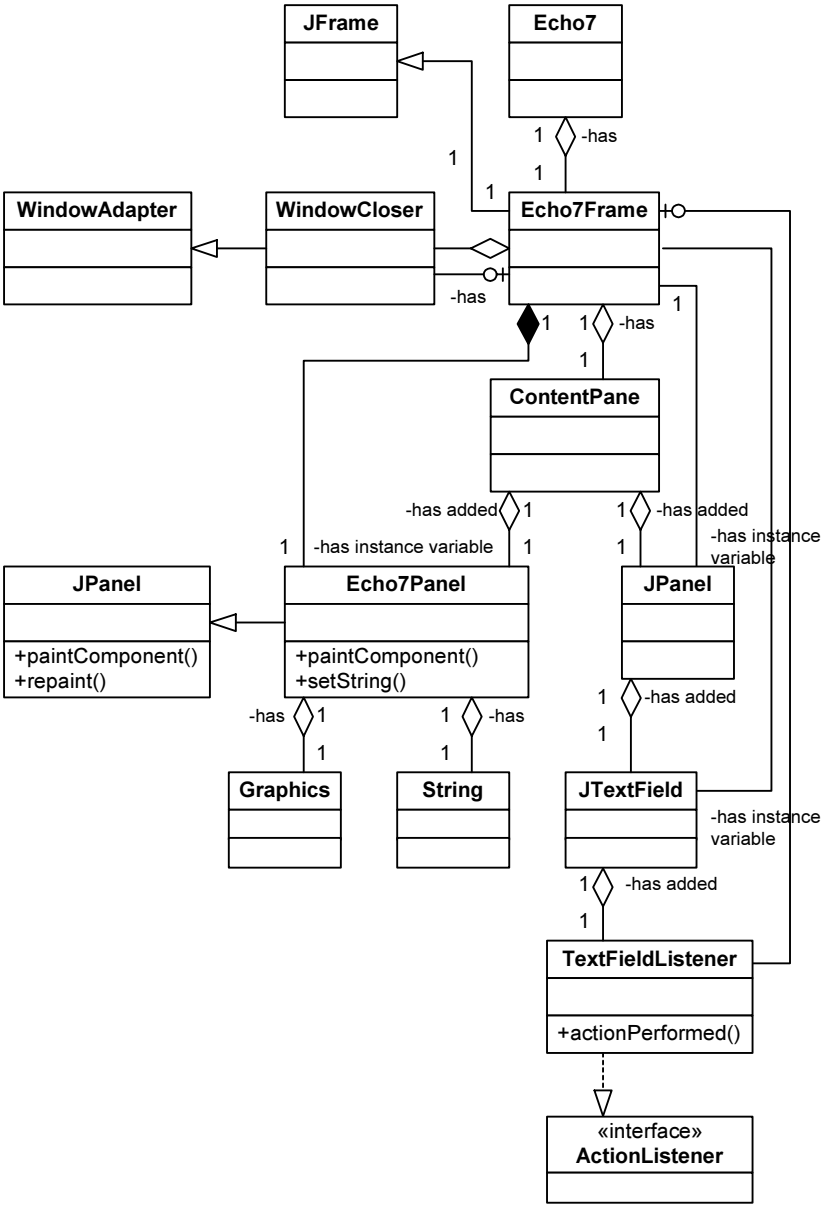

JFrame
Echo7
1
-has
1
1
WindowAdapter
WindowCloser
Echo7Frame
-has
1
1
-has
1
1
ContentPane
-has added
1
1
-has added
1
-has instance variable
1
1
-has instance variable
JPanel
+paintComponent()
+repaint()
Echo7Panel
+paintComponent()
+setString()
JPanel
-has
1
1
-has
1
1
-has added
1
1
Graphics
String
JTextField
-has instance variable
1
-has added
1
TextFieldListener
+actionPerformed()
«interface»
ActionListener

Рис. 18.7

```
    myframe.setVisible(true);
  }
}
class Echo7Frame extends JFrame
{
  private JTextField myField;
  private Echo7Panel myOutputPanel;
  private JPanel myInputPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public Echo7Frame()
  {
    setTitle("Echo7 Frame");
    setSize(FRAMEW, FRAMEH);
    myOutputPanel = new Echo7Panel();
    myField = new JTextField("xyz", 12);
    TextFieldListener myListener = new TextFieldListener();
    myField.addActionListener(myListener);
    myInputPanel = new JPanel();
    myInputPanel.add(myField);
    Container contentPane = getContentPane();
    contentPane.add(myOutputPanel, "Center");
    contentPane.add(myInputPanel, "North");
    addWindowListener(new WindowCloser());
  }
  private class TextFieldListener
    implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    {
      String inputString = myField.getText();
      myOutputPanel.setString(inputString);
      myField.setText("");
      myOutputPanel.repaint();
    }
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
```

```
    }
  }
}
class Echo7Panel extends JPanel
{
  private String stringInQuestion = "";
  private final int STRINGX = 80;
  private final int STRINGY = 80;
  public Echo7Panel()
  {
  }
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    g2.drawString(stringInQuestion, STRINGX, STRINGY);
  }
  public void setString(String stringIn)
  {
  stringInQuestion = stringIn;
  }
}
```

Схема UML также приводится для этого примера. Она показывает полную последовательность вызовов, которая запускается вызовом метода `actionPerformed()` — метода, находящегося в блоке прослушивания текстового поля. Здесь демонстрируется, что происходит с точки зрения объектно-ориентированного подхода, когда пользователь набирает на клавиатуре несколько символов в поле `JTextField` приложения и нажимает клавишу <Enter>.

Когда последовательность вызовов заканчивается, графическое представление приложения обновляется, но на этом приложение не завершается. Система продолжает проверять события, и если снова нажимается клавиша <Enter> в `JTextField`, то последовательность повторяется. Приложение будет выполняться, таким образом, до тех пор, пока не будет закрыто. Как уже говорилось, механизм обработки событий в языке Java напоминает циклы. Схема последовательности показывает ту часть приложения, которая концептуально представляет собой тело цикла (рис. 18.8).

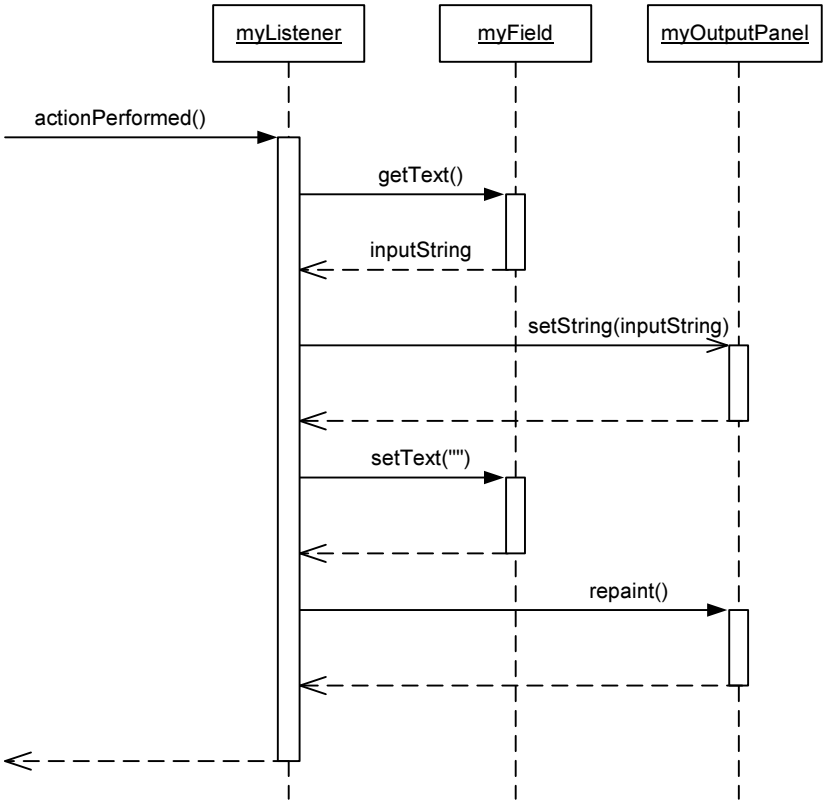


Рис. 18.8

Echo8

Эта программа иллюстрирует употребление анонимного внутреннего класса. Для тех случаев, когда имеется единственный экземпляр класса, нет оснований давать классу имя. Некоторые программисты могут посчитать такой вариант удобным. В более сложных случаях, чем наш пример, могут присутствовать и другие синтаксические детали. По определению, анонимный класс является достаточно сложным, и использование такого синтаксиса не обязательно. Этот класс не будет использоваться в дальнейших примерах программ, но программисту следует знать о его существовании, в случае если ему попадется такой синтаксис.

Для этого примера нет рисунка с видом окна приложения и схемы UML.

Код показывает синтаксис для создания блока прослушивания событий и анонимного внутреннего класса. При создании блока прослушивания сразу за ключевым словом `new` идет полное определение класса. Класс блока прослушивания не имеет конструктора, поэтому вам придется рассчитывать на конструктор, предоставляемый системой по умолчанию. В классе имеется определение метода `actionPerformed()`, который является важнейшим элементом. Обратите внимание, что за закрывающей скобкой определения следует точка с запятой, которая заканчивает строку кода, создающую блок прослушивания. Частой ошибкой является пропуск точки с запятой в этом месте.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Echo8
{
  public static void main(String[] args)
  {
    Echo8Frame myframe = new Echo8Frame();
    myframe.setVisible(true);
  }
}
class Echo8Frame extends JFrame
{
  private JTextField myField;
  private Echo8Panel myOutputPanel;
  private JPanel myInputPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public Echo8Frame()
  {
    setTitle("Echo8 Frame");
    setSize(FRAMEW, FRAMEH);
    myOutputPanel = new Echo8Panel();
    myField = new JTextField("xyz", 12);
    ActionListener myListener = new ActionListener()
    {
      public void actionPerformed(ActionEvent event)
```

```
      {
        String inputString = myField.getText();
        myOutputPanel.setString(inputString);
        myField.setText("");
        myOutputPanel.repaint();
      }
    };
    myField.addActionListener(myListener);
    JPanel myInputPanel = new JPanel();
    myInputPanel.add(myField);
    Container contentPane = getContentPane();
    contentPane.add(myOutputPanel, "Center");
    contentPane.add(myInputPanel, "North");
    addWindowListener(new WindowCloser());
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
    }
  }
}
class Echo8Panel extends JPanel
{
  private String stringInQuestion = "";
  private final int STRINGX = 80;
  private final int STRINGY = 80;
  public Echo8Panel()
  {
  }
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    g2.drawString(stringInQuestion, STRINGX, STRINGY);
  }
  public void setString(String stringIn)
  {
```

```
    stringInQuestion = stringIn;
  }
}
```

Echo9

Этот пример содержит последний элемент, необходимый для выполнения задания по программированию на тему игр Wari и Togiz Kumalak. Он также является подготовкой к тем примерам, которые будут приведены в последующих главах. В данном примере вводится класс `Echo9Cup`. Этот класс представляет чашку, которая содержит счетчик семян, и также имеет атрибуты, которые описывают его как прямоугольник и дают его координаты *x* и *y* во фрейме. Приложение принимает целочисленное значение в текстовом поле, которое используется для установки счетчика семян чашки. Скриншот для данного примера (рис. 18.9) отличается от предыдущего только тем, что имеется прямоугольник, изображающий чашку.

Рис. 18.9

Структура схемы UML отличается от предыдущей только тем, что здесь присутствует экземпляр класса `Cup` вместо переменной класса `String` в классе `Echo9Panel` (рис. 18.10).

Для того чтобы выполнить требуемые действия, должны существовать различные методы, которые должны вызываться в определенной последовательности. Блок прослушивания текстового поля — это внутренний класс фрейма, поэтому он имеет доступ к панели вывода, которая является переменной экземпляра фрейма.

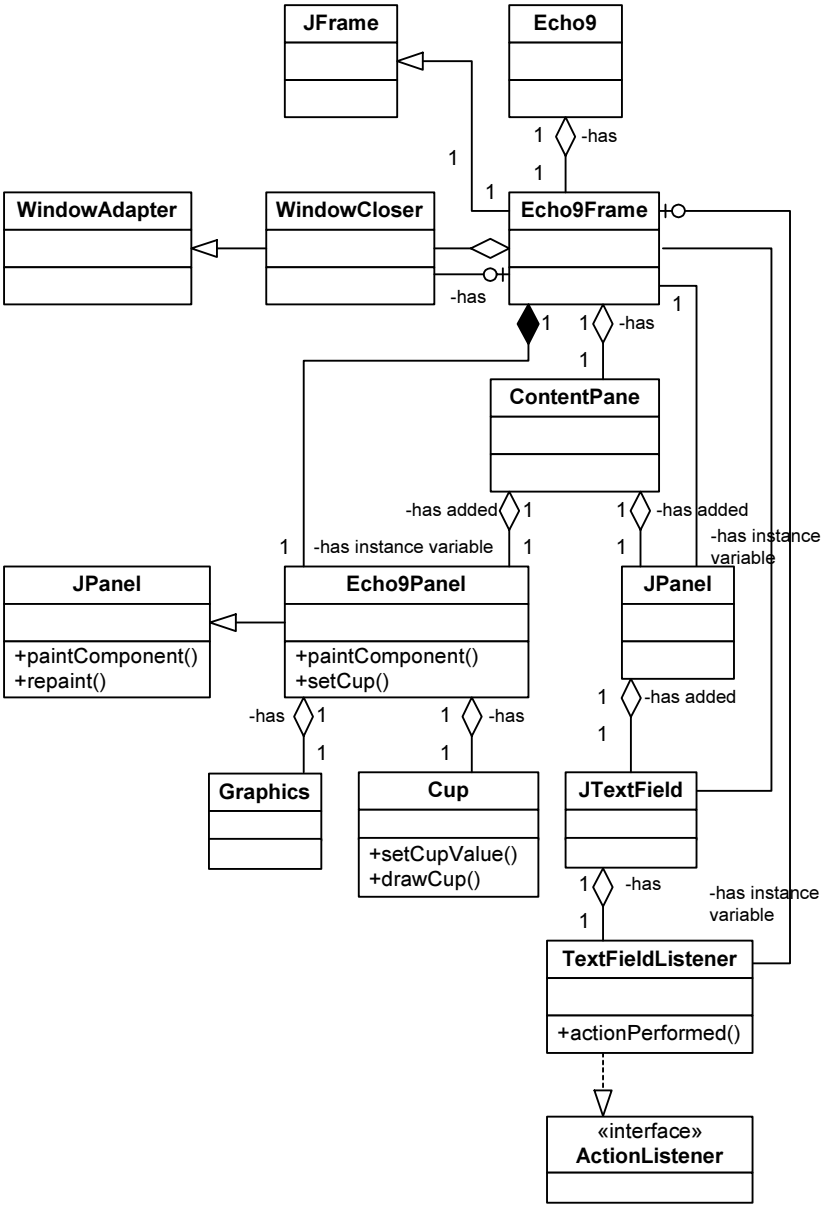

JFrame
Echo9
1
-has
1
1
1
WindowAdapter
WindowCloser
Echo9Frame
-has
1
1
-has
1
1
ContentPane
-has added
1
1
-has added
1
-has instance variable
1
1
-has instance variable
JPanel
+paintComponent()
+repaint()
Echo9Panel
+paintComponent()
+setCup()
JPanel
-has
1
1
-has
1
1
1
-has added
1
Graphics
Cup
+setCupValue()
+drawCup()
JTextField
1
-has
1
-has instance variable
TextFieldListener
+actionPerformed()
«interface»
ActionListener

Рис. 18.10

Когда счетчик семян вводится в текстовое поле, блок прослушивания вызывает метод `setCup()` на панель, передавая количество семян. Метод `setCup()` вызывает метод `setSeedCount()` на экземпляр класса `Echo9Cup`, принадлежащий панели, передавая количество семян. Затем блок прослушивания вызывает метод `repaint()` на панель. Это инициирует вызов метода `paintComponent()`, который вызывает метод `drawCup()` на экземпляр класса `Echo9Cup` панели. Графический параметр `g2` передается между методами, так что метод `drawCup()` может нарисовать прямоугольник класса и его значение на панели. В итоге, приложение принимает новое значение для `seedCount`, изменяет соответствующим образом статус класса `Echo9Cup` и затем выводится на экран с обновленным значением.

Обратите внимание на следующие два момента. Вспомните, что, начиная с *главы 5*, в графических примерах использовались геометрические классы, такие как `Rectangle2D`, а классы типа `Rectangle` не использовались. В данном примере используется простой класс `Rectangle`. Это сделано для того, чтобы не было противоречий с примерами в следующей главе. Кроме того, прямоугольник, который представляет чашку, создается в то время, когда эта чашка выводится на экран. Он не сохраняется в качестве переменной экземпляра класса `Echo9Cup`. Это обстоятельство изменится в последующих примерах.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.Graphics2D;
import java.awt.Rectangle;
public class Echo9
{
  public static void main(String[] args)
  {
    Echo9Frame myframe = new Echo9Frame();
    myframe.setVisible(true);
  }
}
class Echo9Frame extends JFrame
{
```

```
  private JTextField myField;
  private Echo9Panel myOutputPanel;
  private JPanel myInputPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public Echo9Frame()
  {
    setTitle("Echo9 Frame");
    setSize(FRAMEW, FRAMEH);
    myOutputPanel = new Echo9Panel();
    myField = new JTextField("4", 12);
    TextFieldListener myListener = new TextFieldListener();
    myField.addActionListener(myListener);
    JPanel myInputPanel = new JPanel();
    myInputPanel.add(myField);
    Container contentPane = getContentPane();
    contentPane.add(myOutputPanel, "Center");
    contentPane.add(myInputPanel, "North");
    addWindowListener(new WindowCloser());
  }
  private class TextFieldListener
    implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    {
      String inputString = myField.getText();
      int temp = Integer.parseInt(inputString);
      myOutputPanel.setCup(temp);
      myField.setText("");
      myOutputPanel.repaint();
  }
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
    }
  }
}
```

```
class Echo9Panel extends JPanel
{
  private Echo9Cup myCup;
  public Echo9Panel()
  {
    myCup = new Echo9Cup(0, 150, 40, 40, 40);
  }
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    myCup.drawCup(g2);
  }
  public void setCup(int inValue)
  {
    myCup.setSeedCount(inValue);
  }
}
class Echo9Cup
{
  private int seedCount;
  private int x;
  private int y;
  private int w;
  private int h;
  private int textX;
  private int textY;
  public Echo9Cup(int inValue,
                  int cupX, int cupY,
                  int cupW, int cupH)
  {
    seedCount = inValue;
    x = cupX;
    y = cupY;
    w = cupW;
    h = cupH;
    textX = cupX + (int) (.25 * cupW);
    textY = cupY + (int) (.6 * cupH);
  }
  public void setSeedCount(int inValue)
```

```
  {
    seedCount = inValue;
  }
  public void drawCup(Graphics2D g2)
  {
    g2.draw(new Rectangle(x, y, w, h));
    g2.drawString(Integer.toString(seedCount),
                  textX, textY);
  }
}
```

Для этого примера приведем две схемы последовательности событий. Они графически показывают то, что было описано в пояснении, предшествующем коду. Первая схема демонстрирует, что

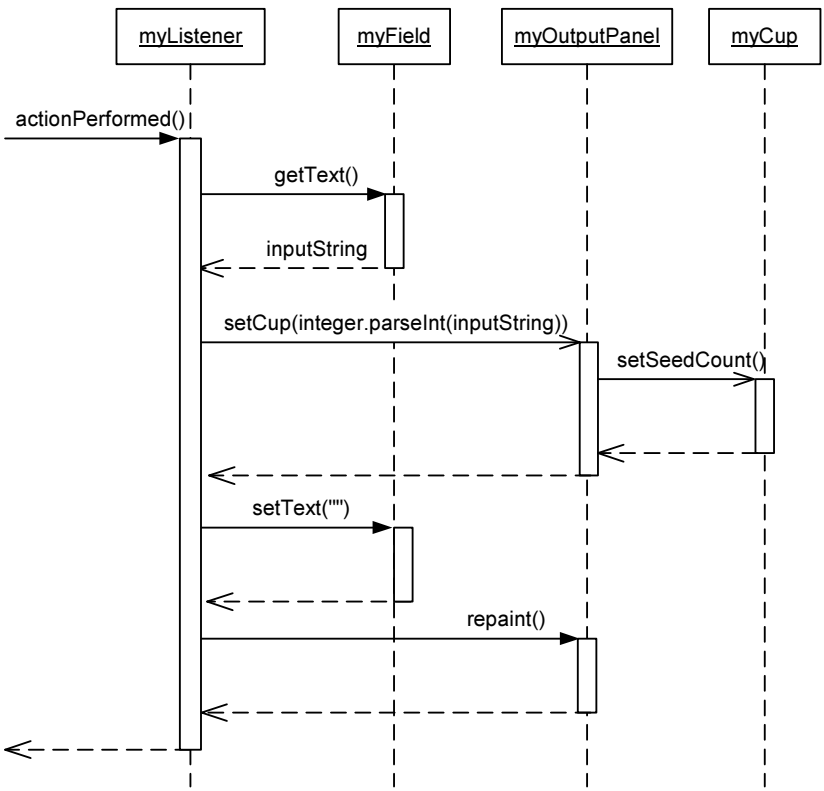


Рис. 18.11

происходит в приложении на стадии ввода. Она показывает последовательность методов, начиная с регистрации события блоком прослушивания текстового поля и до вызова метода `repaint()` (рис. 18.11).

Вторая схема показывает, что происходит в приложении на стадии вывода. Она демонстрирует последовательность методов, начиная с вызова метода `repaint()` и до изображения `myCup` и его счетчика семян, за тем следует завершение метода `repaint()` (рис. 18.12).

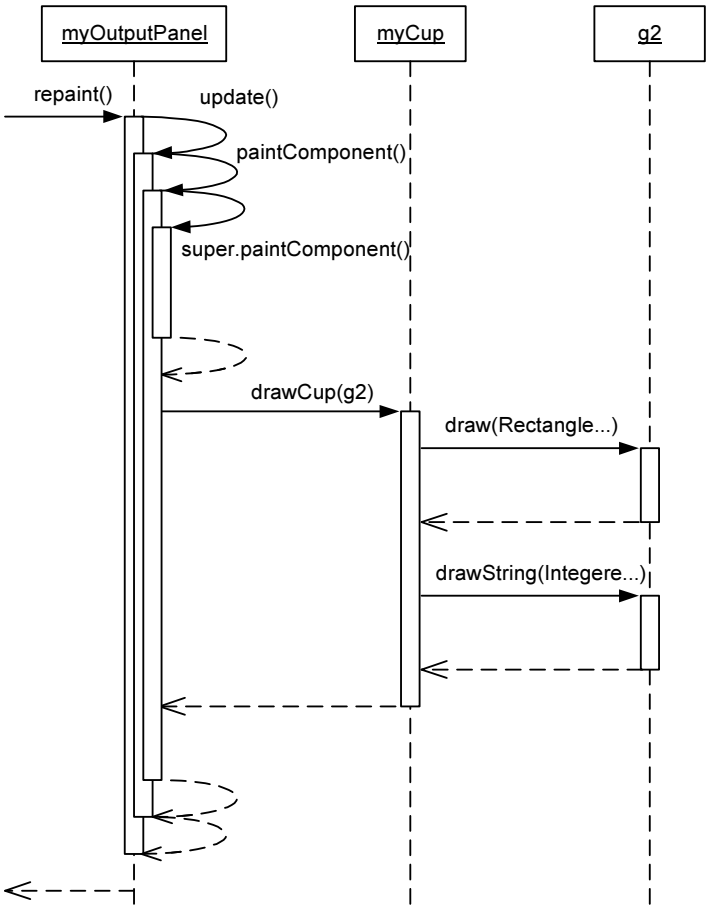
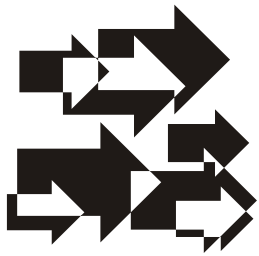


Рис. 18.12

ГЛАВА 19



Объекты, реагирующие на щелчок мыши, апплеты и jar-файл

В этой главе мы продолжим рассмотрение графических интерфейсов пользователя на новых примерах. Примеры сопровождаются иллюстрациями положения на экране (скриншотами) и схемами UML. Для первого примера показан полный код, а затем приводятся только те части кода, которые изменены. Примеры в этой главе основаны на ситуации с чашками и семенами, и они расположены в том порядке, который является логической последовательностью в развитии задания для игры Togiz Kumalak. Это значит, что иногда более сложные графические черты могут быть представлены раньше, чем более простые.

19.1. Объекты, реагирующие на щелчок мыши

ClickCup

В этой программе на панели находятся два квадратика, имитирующие чашки. Щелчок мыши по одной чашке перемещает ее содержимое в другую. Как это выгляди на экране, показано на рис. 19.1.

Структурная схема для этой программы аналогична схеме для приложения Echo9. Запись показывает, что в `ClickCupPanel`

имеются два экземпляра класса `ClickCupCup`. Обратите внимание, что `ClickCupCup` теперь имеет экземпляр класса `Rectangle`. В `ClickCupPanel` также имеется экземпляр класса `MouseHandler`, который расширяет класс `MouseAdapter` (рис. 19.2).

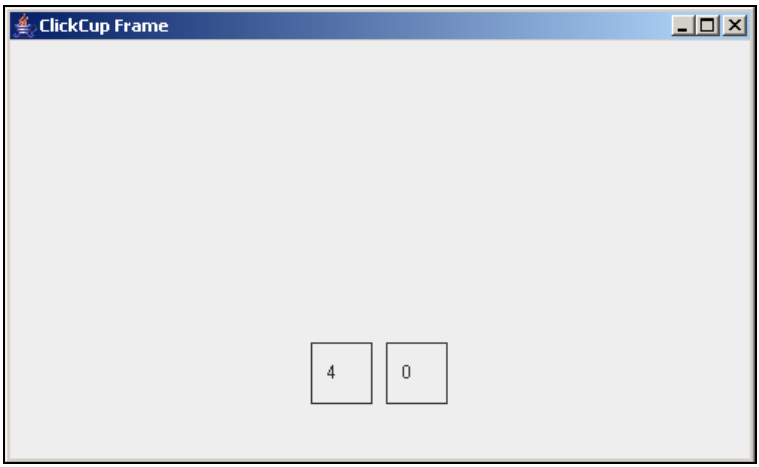


Рис. 19.1

Далее приводится полный код для этого примера, поскольку он является основой для последующих примеров. В этом коде класс `ClickCupPanel` также имеет новую переменную `whichCupIsActive`. Логика программы находится в методе `mouseClicked()` класса `MouseHandler`. Он проверяет, произошел ли щелчок мыши по чашке, и является ли эта чашка активной. Методы `contains()` и `getPoint()` определяют, где произошел щелчок и был ли он на данной чашке. Метод `repaint()` вызывается здесь для обновления внешнего вида приложения в зависимости от изменений в состоянии. Обратите внимание, что класс `ClickCupCup` имеет как целочисленное `seedCount`, так и экземпляр `cupRectangle` класса `Rectangle`. Прямоугольник, называемый `cupRectangle`, является внутренним атрибутом класса, а не просто преходящим визуальным элементом, сконструированным тогда, когда класс выводится на экран. Проверка, пришелся ли щелчок мыши по прямоугольнику, является основной для создания объекта `cup`, опре-

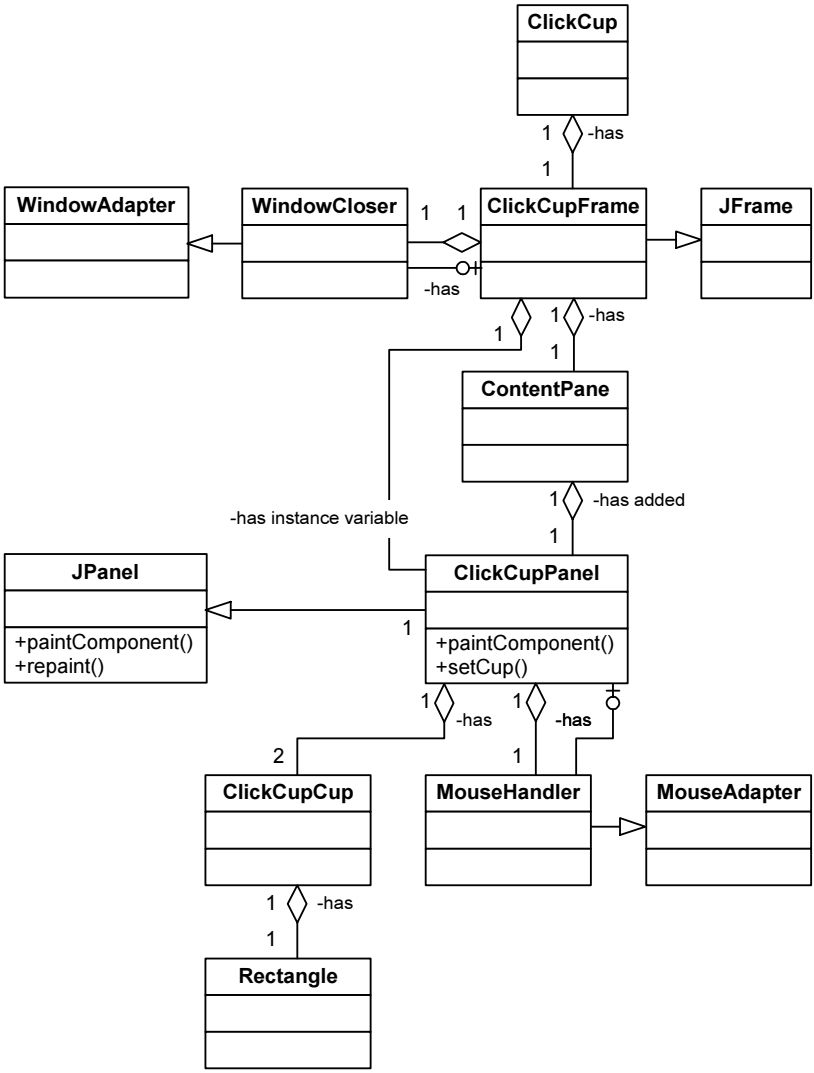


Рис. 19.2

деленного пользователем, — объекта, реагирующего на щелчок мыши.

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.lang.*;
public class ClickCup
{
  public static void main(String[] args)
  {
    ClickCupFrame myframe = new ClickCupFrame();
    myframe.setVisible(true);
  }
}
class ClickCupFrame extends JFrame
{
  private ClickCupPanel myPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public ClickCupFrame()
  {
    setTitle("ClickCup Frame");
    setSize(FRAMEW, FRAMEH);
    myPanel = new ClickCupPanel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
    addWindowListener(new WindowCloser());
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
    }
  }
}
class ClickCupPanel extends JPanel
{
  private ClickCupCup myCupA;
  private ClickCupCup myCupB;
  private ClickCupCup whichCupIsActive;
```

```
  public ClickCupPanel()
  {
    myCupA = new ClickCupCup(4, 130, 60, 40, 40);
    myCupB = new ClickCupCup(0, 180, 60, 40, 40);
    whichCupIsActive = myCupA;
    addMouseListener(new MouseHandler());
  }
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
    super.paintComponent(g2);
    myCupA.drawCup(g2);
    myCupB.drawCup(g2);
  }
  private class MouseHandler extends MouseAdapter
  {
    public void mouseClicked(MouseEvent event)
    {
      Point thePoint = event.getPoint();
      if(myCupA.getRectangle().contains(thePoint))
        && whichCupIsActive == myCupA)
      {
        myCupA.moveCupValueTo(myCupB);
        whichCupIsActive = myCupB;
        repaint();
      }
      else
        if(myCupB.getRectangle().contains(thePoint)
          && whichCupIsActive == myCupB)
      {
        myCupB.moveCupValueTo(myCupA);
        whichCupIsActive = myCupA;
        repaint();
      }
      else
      {
      }
    }
  }
}
```

```
class ClickCupCup
{
  private int seedCount;
  private Rectangle cupRectangle;
  private int textX;
  private int textY;
  public ClickCupCup(int seedCountIn, int cupX,
                     int cupY, int cupW, int cupH)
  {
    seedCount = seedCountIn;
    cupRectangle = new Rectangle(cupX, cupY, cupW, cupH);
    textX = cupX + (int) (.25 * cupW);
    textY = cupY + (int) (.6 * cupH);
  }
  public void
    moveCupValueTo(ClickCupCup destination)
  {
    destination.seedCount = this.seedCount;
    this.seedCount = 0;
  }
  public Rectangle getRectangle()
  {
    return cupRectangle;
  }
  public void drawCup(Graphics2D g2)
  {
    g2.draw(cupRectangle);
    g2.drawString(Integer.toString(seedCount), textX, textY);
  }
}
```

ClickDot

Эта программа содержит новый класс — `Seed`. Семена, находящиеся в чашках, являются экземплярами класса `Seed` и изображаются в виде точек на экране. Щелчок по чашке вызывает перемещение семян из одной чашки в другую, а также изменение изображения на экране, соответствующее произошедшей перемене. Как это выглядит, показано на рис. 19.3.

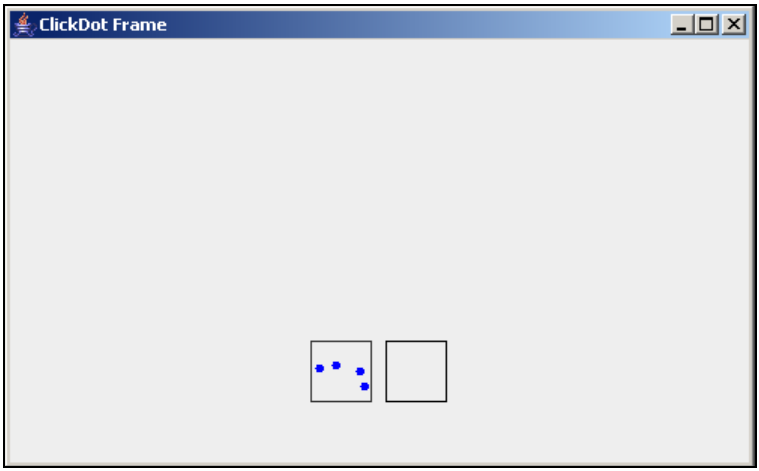


Рис. 19.3

Вместо того чтобы иметь простой счетчик целых чисел для подсчета количества семян в чашках, экземпляры класса `Seed` содержатся в экземплярах класса `ArrayList`, которые являются переменными экземпляра класса `SeedCup` (рис. 19.4).

Код для классов `SeedCup` и `Seed` приведен далее. У семян нет определенного местонахождения, поэтому у них нет переменных экземпляра, которые определяли бы их расположение. Их графическое представление, которое называется точкой, создается по мере необходимости при выводе на экран состояния программы. Расположение точек является произвольным в рамках прямоугольника того класса, в котором они находятся. В этом отличие от класса `Cup`, у которого есть переменная экземпляра `cupRectangle` с фиксированным расположением. Как для чашек, так и для семян объект и его расположение не являются идентичными, хотя в случае чашек сходство между ними больше.

```
class SeedCup
{
  private Rectangle cupRectangle;
  private ArrayList<Seed> seedList;
  public SeedCup(int seedCountIn, int cupX,
                 int cupY, int cupW, int cupH)
```

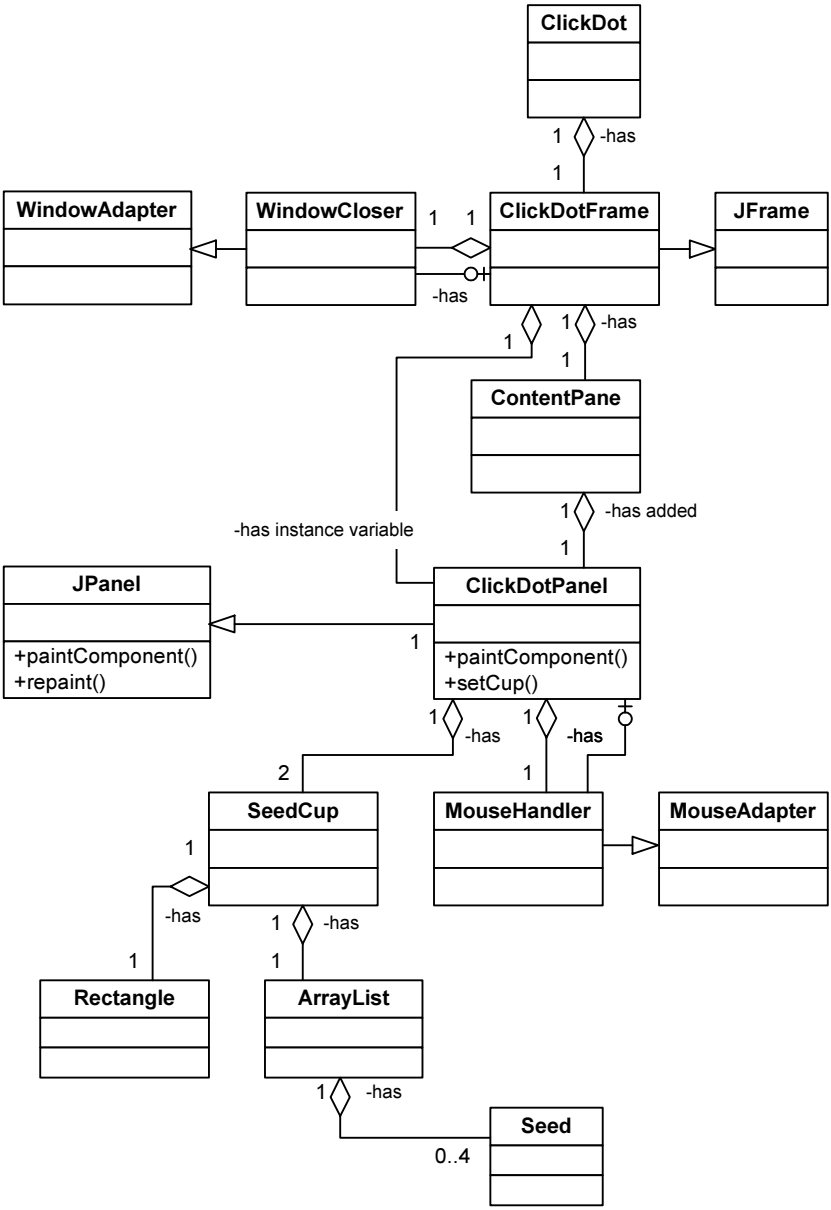

ClickDot
1
-has
1
WindowAdapter
WindowCloser
1
1
ClickDotFrame
JFrame
-has
1
1
-has
1
1
ContentPane
1
-has added
-has instance variable
1
JPanel
+paintComponent()
+repaint()
ClickDotPanel
1
+paintComponent()
+setCup()
1
-has
1
-has
2
1
SeedCup
MouseHandler
MouseAdapter
1
-has
1
-has
1
1
Rectangle
ArrayList
1
-has
Seed
0..4

Рис. 19.4

```
{
  int i;
  cupRectangle = new Rectangle(cupX, cupY, cupW, cupH);
  seedList = new ArrayList<Seed>();
  for(i = 0; i < seedCountIn; i++)
  {
    Seed aSeed = new Seed();
    seedList.add(aSeed);
  }
}
public void moveCupValueTo(SeedCup destination)
{
  int i;
  for(Iterator myiterator = seedList.iterator();
    myiterator.hasNext(); )
  {
    destination.seedList.add((Seed) myiterator.next());
    myiterator.remove();
  }
}
public Rectangle getRectangle()
{
  return cupRectangle;
}
public void drawCup(Graphics2D g2)
{
  int i;
  int seedX, seedY;
  Random generator = new Random();
  g2.draw(cupRectangle);
  for(Seed aSeed:  seedList)
  {
    if((aSeed.getDiameter() <= cupRectangle.getWidth())
    && (aSeed.getDiameter() <= cupRectangle.getHeight()))
    {
      seedX = (int) cupRectangle.getX() +
        generator.nextInt((int)
        cupRectangle.getWidth() -
        aSeed.getDiameter());
```

```
        seedY = (int) cupRectangle.getY() +
          generator.nextInt((int)
          cupRectangle.getHeight() -
          aSeed.getDiameter());
        aSeed.drawSeed(g2, seedX, seedY);
      }
    }
  }
}
class Seed
{
  private final Color seedColor = Color.BLUE;
  private final int diameter = 6;
  public Seed()
  {
  }
  public int getDiameter()
  {
    return diameter;
  }
  public void drawSeed(Graphics2D g2,
                       int xCoord,
                       int yCoord)
  {
    Ellipse2D.Double dot =
      new Ellipse2D.Double(xCoord, yCoord,
                           diameter, diameter);
    g2.setColor(seedColor);
    g2.fill(dot);
    g2.setColor(Color.BLACK);
  }
}
```

ClickHand

В этой программе курсор мыши изображается в виде руки, появляющейся над той чашкой, которая является активной в данный момент. Изображения на экране не дается, поскольку система не показывает, какой вид принимает курсор в скриншоте. Измене-

ние вида курсора требует нового блока прослушивания, а именно блока прослушивания движения мыши. Он реализуется в виде экземпляра класса `MouseMotionHandler`, который является расширением класса `MouseMotionAdapter`. Это показано на схеме UML (рис. 19.5).

`ClickHandPanel` содержит код для `MouseMotionHandler` в качестве внутреннего класса. Когда создается `ClickHandPanel`, к нему должен быть добавлен экземпляр класса `MouseMotionHandler`. Также имеются небольшие изменения в методе `mouseClicked()`:

```
class ClickHandPanel extends JPanel
{
  private SeedCup myCupA;
  private SeedCup myCupB;
  private SeedCup whichCupIsActive;
  public ClickHandPanel()
  {
    myCupA = new SeedCup(4, 130, 60, 40, 40);
    myCupB = new SeedCup(0, 180, 80, 40, 40);
    whichCupIsActive = myCupA;
    addMouseListener(new MouseHandler());
    addMouseMotionListener(new MouseMotionHandler());
  }
```

Последняя строка кода в методе `mouseClicked()` устанавливает курсор мыши на его значение по умолчанию, после того как было произведено действие:

```
private class MouseHandler extends MouseAdapter
{
  public void mouseClicked(MouseEvent event)
  {
    Point thePoint = event.getPoint();
    if(myCupA.getRectangle().contains(thePoint))
        && whichCupIsActive == myCupA)
    {
      myCupA.moveCupValueTo(myCupB);
      whichCupIsActive = myCupB;
      repaint();
    }
```

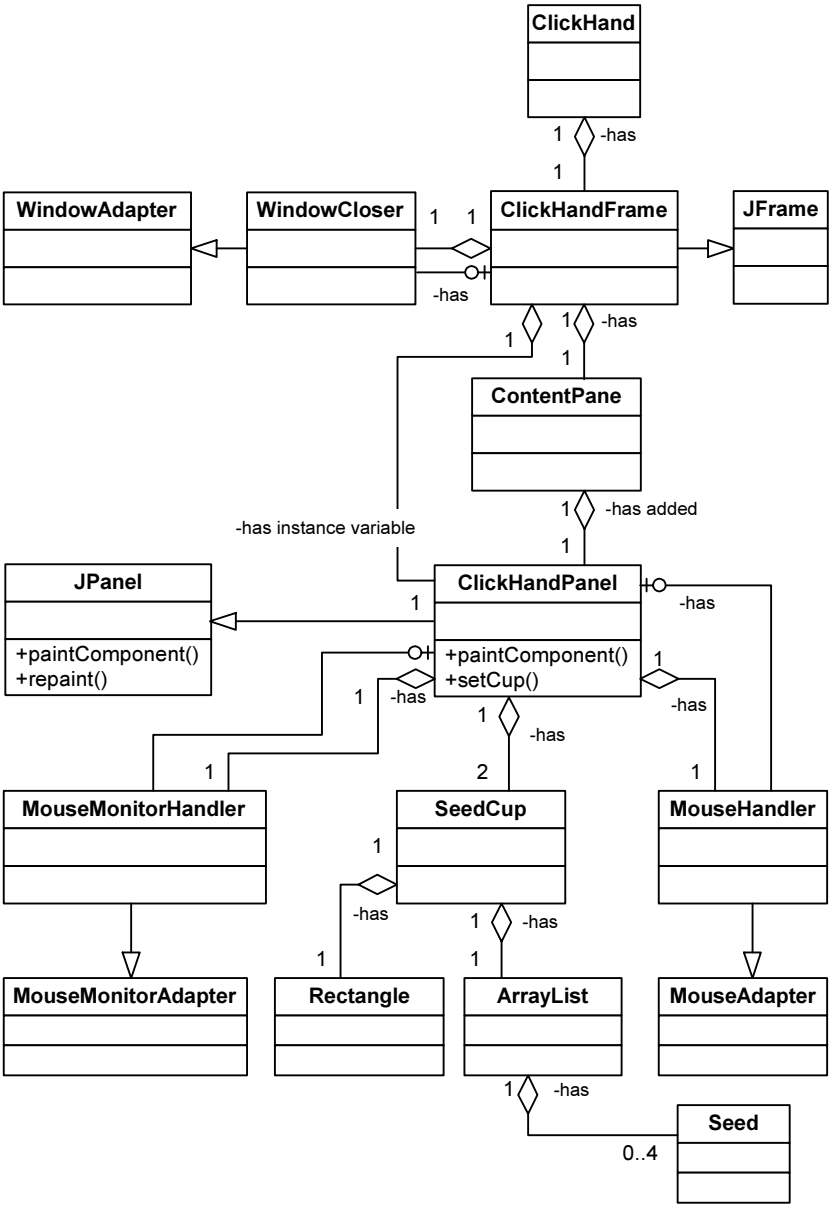

ClickHand
1
-has
1
WindowAdapter
WindowCloser
1
1
ClickHandFrame
JFrame
-has
1
-has
1
1
ContentPane
1
-has added
-has instance variable
1
JPanel
+paintComponent()
+repaint()
1
ClickHandPanel
+paintComponent()
+setCup()
-has
1
-has
1
-has
1
-has
1
2
1
MouseMonitorHandler
SeedCup
MouseHandler
1
-has
1
-has
1
1
MouseMonitorAdapter
Rectangle
ArrayList
MouseAdapter
1
-has
Seed
0..4

Рис. 19.5

```
    else
      if(myCupB.getRectangle().contains(thePoint))
        && whichCupIsActive == myCupB)
      {
        myCupB.moveCupValueTo(myCupA);
        whichCupIsActive = myCupA;
        repaint();
      }
    else
    {
    }
    setCursor(Cursor.getDefaultCursor());
  }
}
```

Обработчик действий мыши постоянно отслеживает положение курсора мыши и изменяет его внешний вид на изображение руки, если он находится над активной чашкой. Функционально это основано на методе `getPoint()`, подобно тому, как функционирует обычный обработчик действий мыши:

```
private class MouseMotionHandler
  extends MouseMotionAdapter
{
  public void mouseMoved(MouseEvent event)
  {
    Point thePoint = event.getPoint();
    if(whichCupIsActive.getRectangle()
       .contains(thePoint))
       setCursor(Cursor.getPredefinedCursor
         (Cursor.HAND_CURSOR));
    else
      setCursor(Cursor.getDefaultCursor());
  }
}
```

19.2. Преобразование приложения в апплет и размещение его на Web-странице

Простые апплеты были описаны в *главе 5*. Теперь рассмотрим, как превратить в апплет графическое приложение. Эта процедура

иллюстрируется на примере приложения ClickHand. Для того чтобы преобразовать приложение в апплет, необходимо сделать следующее:

1. Импортировать пакеты `java.awt*` и `java.swing*`.
2. Изменить класс на апплет за счет расширения его с помощью `JApplet`.
3. Заменить метод `main()` методом `unit()`. Метод `unit()` должен содержать код из конструктора класса `Frame`, который создает содержимое фрейма. Он не может содержать никакого кода, относящегося к фрейму, например, дающего фрейму название или размер.
4. Уберите из кода класс `Frame` и все его ссылки.
5. Посмотрите пример кода. Здесь приведены метод `main()` и класс `ClickHandFrame` приложения ClickHand до того, как оно преобразовано в апплет. Остальную часть кода мы не показываем, т. к. она не изменяется при преобразовании:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Point2D;
import java.lang.*;
import java.util.*;
public class ClickHandBefore
{
  public static void main(String[] args)
  {
    ClickHandFrame myframe = new ClickHandFrame();
    myframe.setVisible(true);
  }
}
class ClickHandFrame extends JFrame
{
  private ClickHandPanel myPanel;
  private final int FRAMEW = 360;
```

```
  private final int FRAMEH = 180;
  public ClickHandFrame()
  {
    setTitle("ClickHand Frame");
    setSize(FRAMEW, FRAMEH);
    myPanel = new ClickHandPanel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
    addWindowListener(new WindowCloser());
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
    }
  }
}
```

А вот как выглядит код после преобразования. Фрагменты кода комментируются, но не убираются полностью:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Point2D;
import java.lang.*;
import java.util.*;
public class ClickHandAfter extends JApplet
{
/*
  public static void main(String[] args)
  {
    ClickHandFrame myframe = new ClickHandFrame();
    myframe.setVisible(true);
  }
}
```

```
class ClickHandFrame extends JFrame
{
  private ClickHandPanel myPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public ClickHandFrame()
  {
    setTitle("ClickHand Frame");
    setSize(FRAMEW, FRAMEH);
*/
  public void init()
  {
    ClickHandPanel myPanel = new ClickHandPanel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
  }
}
/*
    addWindowListener(new WindowCloser());
  }
  private class WindowCloser extends WindowAdapter
  {
    public void windowClosing(WindowEvent event)
    {
      System.exit(0);
    }
  }
}
*/
```

Для размещения апплета на Web-странице нужен файл HTML с правильным кодом. Посмотрите на простой пример для ClickHand. Для HTML требуется существование скомпилированного class-файла апплета. Код HTML, показанный далее, написан с допущением, что class-файл находится в той же папке, что и файл HTML. Обратите внимание, что определение размеров апплета на Web-странице делается в HTML, т. к. у апплета нет фрейма.

```
<html>
  <title>ClickHand as an Applet</title>
  <body>
    Here is ClickHandApplet.
    <applet code="ClickHandApplet.class" width="360"
            height="180">
    </applet>
  </body>
</html>
```

19.3. Преобразование приложения в самовыполняющийся jar-файл

Удобно иметь возможность сгруппировать компоненты какого-либо приложения в одном файле, который можно выполнить без необходимости открытия его в качестве исходного кода и сначала компилировать. Файлы jar могут быть использованы для того, чтобы создать такие самовыполняющиеся файлы в языке Java. Далее приведены указания, как это сделать. Кроме того, можно создать такие файлы и для других целей. При попытке создать самовыполняющийся файл вы можете по ошибке создать один из других видов файла jar. Поэтому важно проверять результат, чтобы убедиться, что вы получили именно то, что хотели.

Для того чтобы преобразовать приложение в самовыполняющийся файл jar, нужно сделать следующее:

1. Скомпилируйте исходный код приложения, в результате чего создаются class-файлы. Компиляция данного набора исходных файлов может привести к большему количеству файлов типа class, в зависимости от того, какое количество классов было определено в исходных файлах. Все эти получившиеся файлы классов потребуются позже.
2. Напишите *файл описания* (manifest file). Он должен иметь расширение mf; имя его может быть любым. Например, в нашем случае он имеет имя mainclass.mf. В файле описания вы указываете, какой класс в приложении является главным. В нашем примере в этом файле содержится:

   ```
   Main-Class: ClickHand
   ```

Здесь важно отметить три момента:

- имя файла ClickHand не принимает расширение;
- за именем файла ClickHand идет новая строка; файл описания не будет работать правильно, если за его содержанием не следует новая строка;
- имеется только один пробел между двоеточием и именем файла ClickHand.

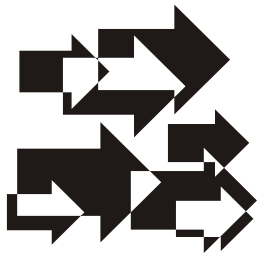
3. Если все файлы классов, являющиеся результатом компиляции, находятся в одной папке, и если можно выполнить exe-файлы подпапки bin (из папки Java), из этого же места, то вы можете создать самовыполняющийся jar-файл вводом следующей команды по подсказке командной строки для этой папки:

```
jar cvfm ClickHand.jar mainclass.mf *.class
```

При желании можно перечислить все файлы классов, вместо того чтобы использовать выражение со звездочкой (*.class). Результатом успешной команды будет появление нового файла, ClickHand.jar, в том же каталоге.

У вас могут возникнуть проблемы при выполнении команды `jar`, в зависимости от того, как была установлена система Java на вашем компьютере. Может случиться так, что она не будет запускаться из того каталога, где вы скомпилировали свое приложение. Возможным решением будет уточнить путь к каталогу. Кроме того, есть быстрое и "грязное" решение — скопировать все требуемые файлы в каталог bin, где была установлена система Java на вашем компьютере. Именно здесь находится утилита jar, которая будет запускаться из командной строки в каталоге bin. Недостатком такого решения является то, что вы засоряете каталог с программой Java своими файлами, и вам следует быть внимательным при очистке, чтобы не стереть какой-либо из системных файлов Java.

ГЛАВА 20



Меню, множественные фреймы, сериализация и текстовые поля

Примеры, которые используются в данной главе, — это последние примеры с играми Wari и Togiz Kumalak. Объяснения будут менее подробными, чем в предыдущих главах. За исключением одного примера, к программам не даются иллюстрации в виде схем UML. Кроме того, для этой главы не существует образцов выполнения программ на примере игры Wari.

20.1. Ввод с клавиатуры, меню и множественные фреймы

ClickKey

Эта программа демонстрирует возможность использования ввода с клавиатуры, а также щелчков мышью для того, чтобы выполнить некоторые действия. Использование клавиатуры для ввода не несет никаких изменений в интерфейсе программы при ее выполнении, поэтому мы не приводим вид ее окна.

Представленный далее код демонстрирует изменения в классе `ClickKeyPanel`. В дополнение к построению и добавлению блока прослушивания клавиатуры, необходимо также сделать экземпляр класса `ClickKeyPanel` фокусируемым, т. е. способным полу-

чать ввод самостоятельно. Тема фокуса будет рассмотрена более подробно на ряде примеров, начиная со следующей главы.

```
class ClickKeyPanel extends JPanel
{
  private SeedCup myCupA;
  private SeedCup myCupB;
  private SeedCup whichCupIsActive;
  public ClickKeyPanel()
  {
    myCupA = new SeedCup(4, 130, 60, 40, 40);
    myCupB = new SeedCup(0, 180, 60, 40, 40);
    whichCupIsActive = myCupA;
    addMouseListener(new MouseHandler());
    addMouseMotionListener(new MouseMotionHandler());
    MyKeyListener mylistener = new MyKeyListener();
    addKeyListener(mylistener);
    setFocusable(true);
  }
```

Рассмотрим код для блока прослушивания клавиатуры. Его основная логика такая же, как и у блока прослушивания мыши. Разница заключается в том, что событие, которое он обнаруживает, является нажатием клавиши, и это нажатие не является вводом символа в текстовое поле. Точно так же, как можно определить место щелчка мышью, можно определить, какая клавиша была нажата, когда приложение имело фокус. В приложении чашки показаны, как `myCupA` и `myCupB`. Нажатием одной из клавиш — <A> или <B> — совершается выбор соответствующей чашки, если она активна. Других изменений в коде нет.

```
private class MyKeyListener extends KeyAdapter
{
  public void keyTyped(KeyEvent event)
  {
    char keyChar = event.getKeyChar();
    if(keyChar == 'a' && whichCupIsActive == myCupA)
    {
      myCupA.moveCupValueTo(myCupB);
      whichCupIsActive = myCupB;
```

```
        repaint();
      }
      else if(keyChar == 'b' && whichCupIsActive == myCupB)
      {
        myCupB.moveCupValueTo(myCupA);
        whichCupIsActive = myCupA;
        repaint();
      }
      else
      {
      }
    }
}
```

ClickMenu

Эта программа демонстрирует простое меню с двумя опциями. Одна опция вызывает завершение работы программы. Другая возвращает к началу программы, т. е. переводит ее в первоначальное состояние (рис. 20.1).

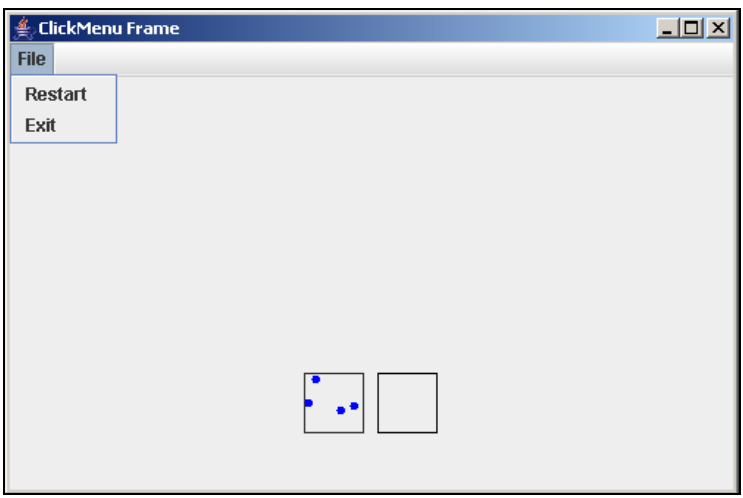
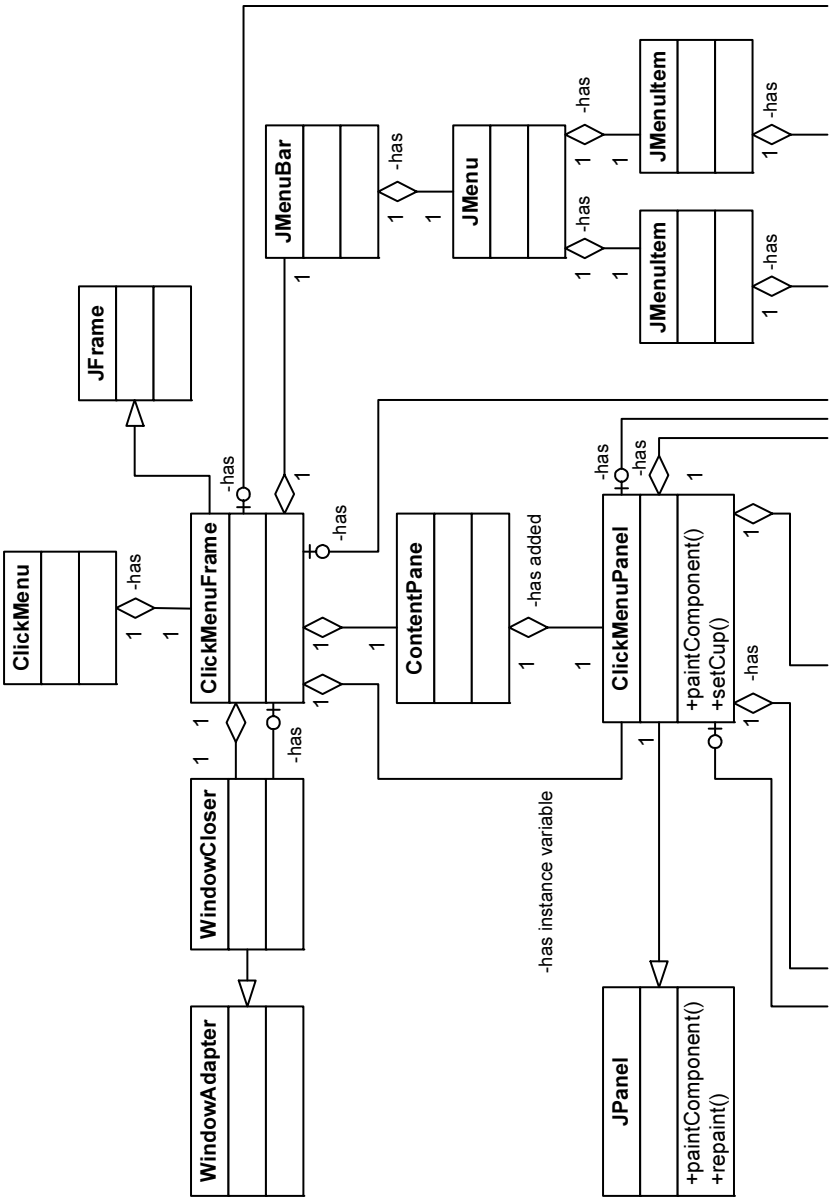


Рис. 20.1

Для этого примера приведем схему UML (рис. 20.2), поскольку полезно посмотреть структуры разных меню, компоненты меню и


JFrame
JMenuBar
JMenu
JMenuItem
JMenuItem
-has
ClickMenu
ClickMenuFrame
ContentPane
-has added
ClickMenuPanel
+paintComponent()
+setCup()
WindowCloser
WindowAdapter
-has instance variable
JPanel
+paintComponent()
+repaint()

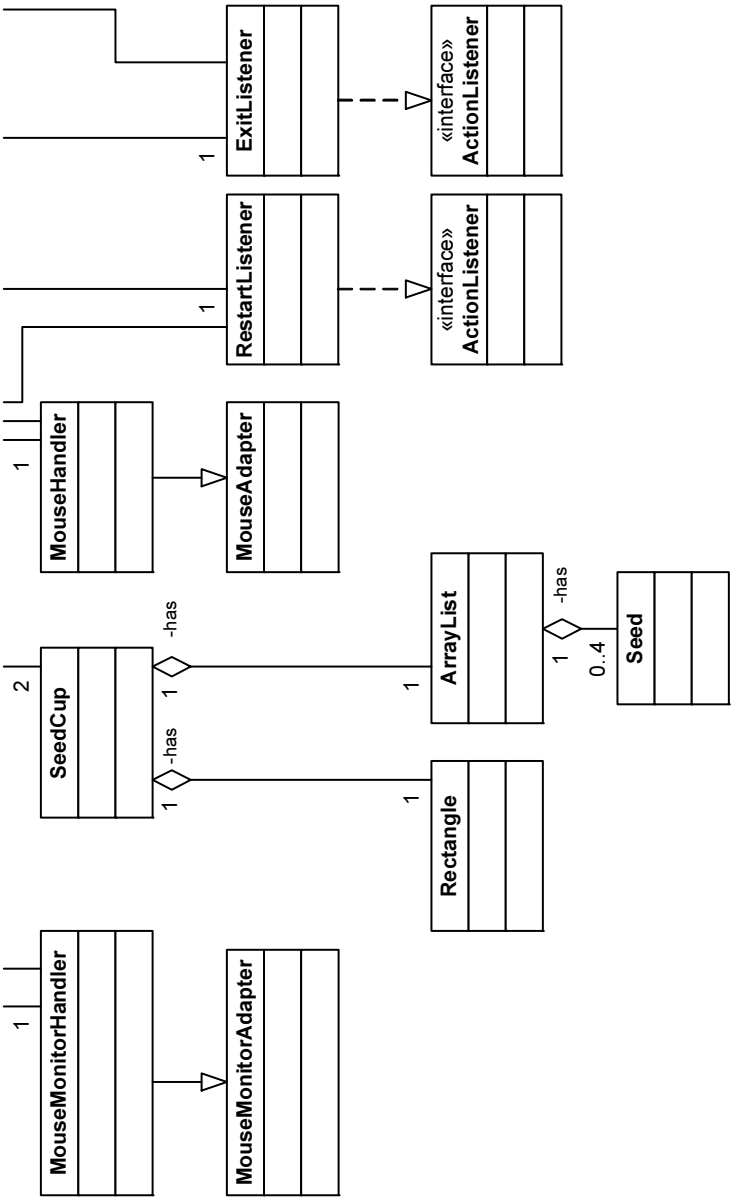

ExitListener
«interface»
ActionListener
RestartListener
«interface»
ActionListener
MouseHandler
MouseAdapter
SeedCup
ArrayList
Seed
Rectangle
MouseMonitorHandler
MouseMonitorAdapter
1
2
-has
0..4

Рис. 20.2

блоки прослушивания, которые сопровождают их. Меню и блоки прослушивания для двух опций принадлежат `ClickMenuFrame`, а не `ClickMenuPanel`. Блок прослушивания для первой опции дублирует функцию `WindowCloser`. Блок прослушивания во второй опции содержит код, который избавляется от старого экземпляра `ClickMenuPanel` и создает новый. Рассмотрим класс `ClickMenuFrame`, показывающий, как создаются элементы меню:

```
class ClickMenuFrame extends JFrame
{
  private ClickMenuPanel myPanel;
  private final int FRAMEW = 360;
  private final int FRAMEH = 180;
  public ClickMenuFrame()
  {
    setTitle("ClickMenu Frame");
    setSize(FRAMEW, FRAMEH);
    myPanel = new ClickMenuPanel();
    Container contentPane = getContentPane();
    contentPane.add(myPanel, "Center");
    addWindowListener(new WindowCloser());
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);
    JMenu fileMenu = new JMenu("File");
    menuBar.add(fileMenu);
    JMenuItem restartItem = new JMenuItem("Restart");
    fileMenu.add(restartItem);
    RestartListener myRestartListener = new RestartListener();
    restartItem.addActionListener(myRestartListener);
    JMenuItem exitItem = new JMenuItem("Exit");
    fileMenu.add(exitItem);
    ExitListener myExitListener = new ExitListener();
    exitItem.addActionListener(myExitListener);
  }
```

А вот блоки прослушивания для опций меню. Больше в этой программе никаких изменений нет.

```
  private class RestartListener
    implements ActionListener
```

```
  {
    public void actionPerformed(ActionEvent event)
    {
      Container contentPane = getContentPane();
      contentPane.remove(myPanel);
      myPanel = new ClickMenuPanel();
      contentPane.add(myPanel, "Center");
      setVisible(true);
    }
  }
  private class ExitListener
    implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    {
      System.exit(0);
    }
  }
}
```

ClickMany

Эта программа демонстрирует открытие более чем одного фрейма с чашками. Меню для главного фрейма позволяет открывать новые фреймы или закончить работу приложения. Меню для производного фрейма предназначено для повторного запуска или закрытия данного фрейма. На рис. 20.3 изображен один главный фрейм, а на рис. 20.4 — один производный.

Код для этого приложения содержит два фреймовых класса, `ClickManyFrame` и `ClickSubFrame`. Класс `ClickManyFrame` имеет блок прослушивания к опции меню для создания экземпляра класса `ClickSubFrame`:

```
private class MakeSubFrameListener
  implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    ClickSubFrame myframe = new ClickSubFrame();
```

```
    myframe.setLocation(subFrameLocation,
                        subFrameLocation);
    if(subFrameLocation >= 500)
    {
      subFrameLocation = 50;
    }
    else
    {
      subFrameLocation += 50;
    }
    myframe.setVisible(true);
  }
}
```

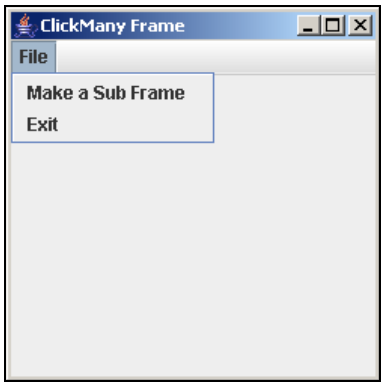


Рис. 20.3

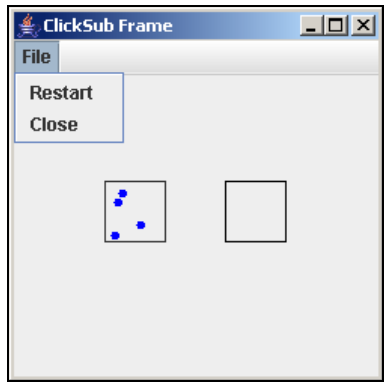


Рис. 20.4

Меню и содержание класса `ClickSubFrame` похожи на фреймовый класс из предыдущего примера. Здесь вводится еще один фрагмент синтаксиса. Вместо того чтобы использовать `WindowCloser`, можно применить следующий синтаксис:

```
setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

20.2. Сериализация, выборщики файлов, текстовые поля и полосы прокрутки

ClickSave

Эта программа добавляет две опции меню в предыдущий пример: одна опция служит для того, чтобы сохранить текущее состояние чашек в приложении, а вторая — чтобы перезагрузить их. Вид окна программы представлен на рис. 20.5.

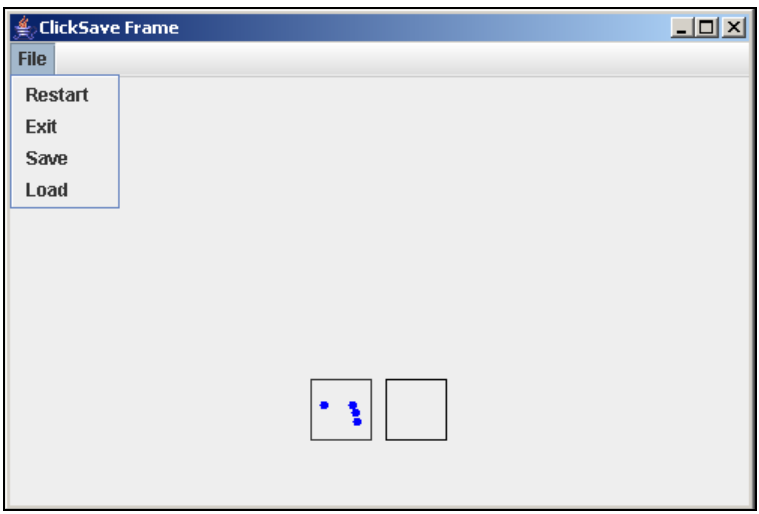


Рис. 20.5

Можно написать специальный файл ввода/вывода для сохранения состояния объектов в приложении. При перезагрузке приложения новые объекты того же типа могут быть сконструированы, и их переменным экземпляра могут быть присвоены сохраненные значения. Кроме того, в языке Java имеется интерфейс `Serializable` для выполнения этой задачи. Это особый вид интерфейса, который не требует реализации каких-либо методов. Если класс реализует данный интерфейс, то система поддерживает возможность сохранения и перезагрузки объектов напрямую. Сериализация

имеет определенные требования и ограничения, которые необходимо кратко обсудить, прежде чем рассмотреть пример кода.

Чтобы иметь способность к сериализации, класс должен реализовать интерфейс `Serializable`. Каждый объект, который содержится в классе, должен реализовать данный интерфейс. Некоторые классы, перечисленные в документации Java API, реализуют такой интерфейс, а некоторые — нет. В примере программы встречаются классы `SeedCup` и `Seed`. Класс `SeedCup` содержит прямоугольник для графического представления класса. Начиная с примера Echo9, использовался простой класс `Rectangle`, а не класс `Rectangle2D.Double`. Причина такого выбора заключается в том, что класс `Rectangle` реализует интерфейс `Serializable`, а класс `Rectangle2D.Double` — нет. Если бы использовался класс `Rectangle2D.Double`, то класс `SeedCup` не смог бы реализовать интерфейс `Serializable`.

В чашках также находятся семена, поэтому класс `Seed` также должен иметь способность к сериализации. Точки, которые использовались для визуального представления семян, являются экземплярами класса `Ellipse2D.Double`. Этот класс не реализует интерфейс `Serializable`. Однако точки не сохраняются как переменные экземпляра класса `Seed`, а создаются, когда приложение обновляется. Это означает, что класс `Seed` может реализовать интерфейс `Serializable`, даже несмотря на то, что класс `Ellipse2D.Double` не может этого. В результате, способность к сериализации класса `SeedCup` не испытывает отрицательного воздействия со стороны класса `Seed`.

Классы эллипсов иллюстрируют ограничения сериализации. Не существует простого класса `Ellipse`, и ни один из существующих классов этой группы не реализует интерфейс `Serializable`. Наш пример был организован таким образом, чтобы это не составляло проблему, поскольку точки, представляющие семена, не являются постоянными. Но предположим, ради обсуждения, что вы хотите, чтобы точки были постоянными, и желаете использовать сериализацию. Можно расширить один из классов эллипсов, имеющихся в системе, ничего не переопределяя, и просто объявить ваш подкласс реализующим интерфейс `Serializable`. К сожалению, это не сработает. Вы можете также написать собствен-

ный класс эллипсов, пытаясь реализовать все графические методы самостоятельно, вместе с сериализацией. Это потребует большой работы. Еще один вариант — написать специальный файл ввода/вывода для сохранения и перезагрузки состояния вашего приложения.

Добавление пунктов в меню для сохранения и перезагрузки означает добавление блоков прослушивания к каждому из пунктов. Это делается так же, как и в случае с другими блоками прослушивания, поэтому и не демонстрируется в книге. Блоки прослушивания показаны далее. Обратите внимание на прямолинейную логику сериализации. При сохранении объекты в приложении записываются по порядку, один за другим. При перезагрузке это будет равносильно чтению файла с самого начала. Объекты считываются заново в том же порядке, в каком они были сохранены.

```
private class SaveListener
  implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    String inputString;
    inputString =
      JOptionPane.showInputDialog
      ("Enter file path name for saving.");
    try
    {
      ObjectOutputStream objOut = new ObjectOutputStream
        (new FileOutputStream(inputString));
      objOut.writeObject(myPanel.getMyCupA());
      objOut.writeObject(myPanel.getMyCupB());
      objOut.writeObject(myPanel.getWhichCupActive());
    }
    catch(IOException e)
    {
      System.out.println("Problem making or
        writing to an output stream.");
      System.out.println(e);
    }
  }
}
```

```
private class LoadListener
  implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    String inputString;
    inputString =
      JOptionPane.showInputDialog
      ("Enter file path name for loading.");
    try
    {
      Container contentPane = getContentPane();
      contentPane.remove(myPanel);
      myPanel = new ClickSavePanel();
      contentPane.add(myPanel, "Center");
      ObjectInputStream objIn =
        new ObjectInputStream
        (new FileInputStream(inputString));
      myPanel.setMyCupA((SeedCup) objIn.readObject());
      myPanel.setMyCupB((SeedCup) objIn.readObject());
      myPanel.setWhichCupActive
        ((SeedCup) objIn.readObject());
      setVisible(true);
    }
    catch(IOException e)
    {
      System.out.println("Problem making
                          an input stream.");
    }
    catch(ClassNotFoundException e)
    {
      System.out.println("Class not found problem
        when reading objects.");
    }
  }
}
```

Строки, где объявляются классы `SeedCup` и `Seed`, также должны быть изменены, как показано далее. До тех пор, пока все, что в

них содержится, обладает свойством сериализации, для реализации сериализации никаких других изменений не требуется; все, что нужно, — это объявить ее.

```
class SeedCup implements Serializable
class Seed implements Serializable
```

ClickChooser

Эта программа демонстрирует использование класса `JFileChooser` для того, чтобы сделать процедуры сохранения и открытия файлов более удобными. Рассмотрим рис. 20.6 с видом окна приложения для интерфейса, представленного `JFileChooser`.

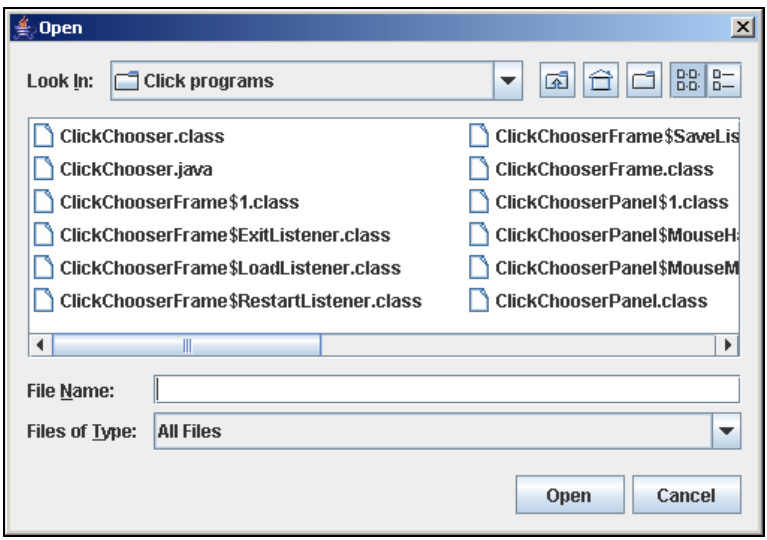


Рис. 20.6

Ссылка `JFileChooser` объявляется в качестве переменной экземпляра в классе `ClickChooserFrame`, и экземпляр `JFileChooser` создается в конструкторе класса `ClickChooserFrame`. Эти изменения в коде не показаны. Блоки прослушивания для сохранения и загрузки изменены с тем, чтобы использовать `JFileChooser`, а не диалоговое окно. Согласно представленному коду, когда на экране изображается экземпляр класса `JFileChooser`, открывается

домашняя папка выполняемого приложения, в которой он находится.

```
private class SaveListener
  implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    String fileName;
    myChooser.setCurrentDirectory(new File("."));
    myChooser.showSaveDialog(ClickChooserFrame.this);
    fileName = myChooser.getSelectedFile().getPath();
    try
    {
      ObjectOutputStream objOut =
        new ObjectOutputStream
        (new FileOutputStream(fileName));
      objOut.writeObject(myPanel.getMyCupA());
      objOut.writeObject(myPanel.getMyCupB());
      objOut.writeObject
        (myPanel.getWhichCupActive());
    }
    catch(IOException e)
    {
      System.out.println("Problem making or
        writing to an output stream.");
      System.out.println(e);
    }
  }
}
private class LoadListener
  implements ActionListener
{
  public void actionPerformed(ActionEvent event)
  {
    String fileName;
    myChooser.setCurrentDirectory(new File("."));
    myChooser.showOpenDialog(ClickChooserFrame.this);
    fileName = myChooser.getSelectedFile().getPath();
    try
```

```
    {
      Container contentPane = getContentPane();
      contentPane.remove(myPanel);
      myPanel = new ClickChooserPanel();
      contentPane.add(myPanel, "Center");
      ObjectInputStream objIn =
        new ObjectInputStream(
          new FileInputStream(fileName));
      myPanel.setMyCupA((SeedCup) objIn.readObject());
      myPanel.setMyCupB((SeedCup) objIn.readObject());
      myPanel.setWhichCupActive((SeedCup) objIn.readObject());
      setVisible(true);
    }
    catch(IOException e)
    {
      System.out.println("Problem making an
        input stream.");
    }
    catch(ClassNotFoundException e)
    {
      System.out.println("Class not found problem
        when reading objects.");
    }
  }
}
```

ClickTextScroll

В этом примере используются классы `JTextArea` и `JScrollPane` для того, чтобы сохранять текстовую запись о последовательности действий в приложении. На рис. 20.7 представлено окно приложения после четырех действий.

Для того чтобы каждый пример программы иллюстрировал что-то одно, здесь класс `ClickTextScroll` не включает пункты меню для сохранения и перезагрузки. Класс `JTextArea` реализует интерфейс `Serializable`, что позволяет сохранять весь `JTextArea` одновременно. Можно также написать файл ввода/вывода и сохранять содержание текста в текстовом файле. В некоторых случаях, для более широких приложений, это решение может быть

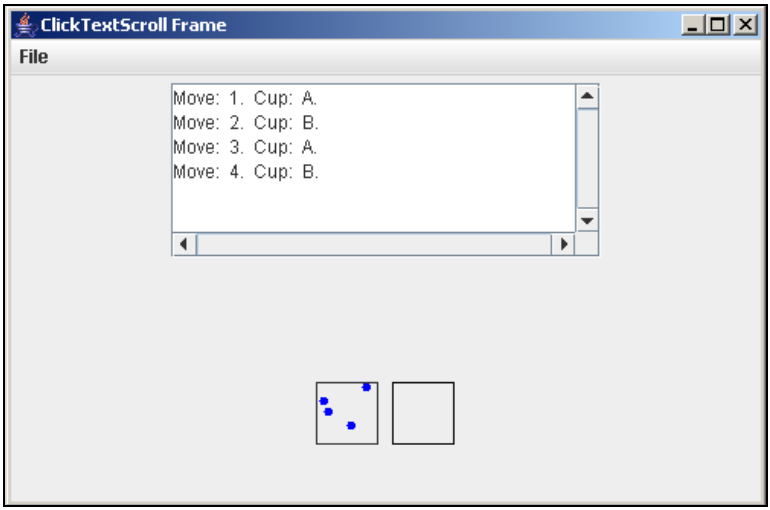


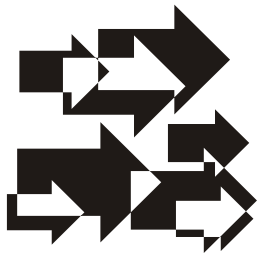
Рис. 20.7

лучше. Все дополнения к коду в данном примере находятся в классе `ClickTextScrollPanel` и его конструкторе. Они и показаны далее. Обратите внимание, что переменная экземпляра `actionRecordArea` создана там, где она объявлена в классе `ClickTextScrollPanel`. Она также могла бы быть создана в конструкторе этого класса.

```
class ClickTextScrollPanel extends JPanel
{
  private JTextArea actionRecordArea = new JTextArea(6, 24);
  private JScrollPane actionScrollPane;
  private SeedCup myCupA;
  private SeedCup myCupB;
  private SeedCup whichCupIsActive;
  private int moveCount = 1;
  public ClickTextScrollPanel()
  {
    myCupA = new SeedCup(4, 130, 155, 40, 40);
    myCupB = new SeedCup(0, 180, 155, 40, 40);
    whichCupIsActive = myCupA;
    actionRecordArea.setEditable(false);
```

```
    actionScrollPane =
      new JScrollPane(actionRecordArea,
          JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
          JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
    add(actionScrollPane, "North");
    addMouseListener(new MouseHandler());
    addMouseMotionListener(new MouseMotionHandler());
  }
```

ГЛАВА 21



Задание для итогового проекта

В этой главе начинается новая серия примеров, которые позволяют ввести новые графические свойства и понятия. Здесь не будет отдельного задания для выполнения. Примеры составляют необходимую основу, на которой строится итоговый проект по программированию, описываемый в следующей главе. Порядок обсуждения примеров в данной главе определяется логикой выработки решения для задания в следующей. Первые три примера иллюстрируют функции кнопок с использованием класса `JButton`. Эти примеры основаны на примерах с чашками и до некоторой степени повторяют идеи, рассмотренные в предыдущей главе.

Остальные примеры основаны на понятиях регистра и памяти. Это элементы архитектуры компьютерного чипа. Итоговый пример основан на моделировании подобного чипа, включающего регистры и память. Регистры и память в данной главе не будут настолько функциональны, как это требуется для итогового проекта. Однако для объяснения примеров полезно сначала объяснить, что такое регистры и память и как они используются.

21.1. Регистры и память: общие сведения

Регистр — это небольшой блок для хранения информации на компьютерном чипе, которая может быть использована для запо-

минания значения одной переменной, например, адреса или инструкции. Значения в регистре записываются в двоичной форме. В настоящих чипах регистры могут быть до 32 или 64 битов в длину, и эти биты хранят электронный эквивалент 1 или 0 в двоичной системе счисления. В примерах программ регистры будут иметь длину 8 битов, и их содержимое будет представлено экземпляром класса `String` с длиной 8 и содержащем знаки 1 и 0.

Компьютерные чипы "понимают" язык ассемблера и машинный язык. Если программист пишет код на этих языках, то возможен непосредственный доступ к регистрам общего назначения в чипе. Предположим, что данный чип имеет регистры общего назначения с именами от `A` до `D`. Тогда объявление на языке ассемблера для хранения значения 1 в регистре `D` будет иметь следующий вид:

```
MOVE D, X01
```

Сначала пишется имя команды, затем операнд назначения, а потом — операнд источника. `X` в `X01` обозначает, что значение дано в шестнадцатеричной системе счисления, и `01` просто обозначает две шестнадцатеричные цифры.

Объявления на языке ассемблера могут быть переведены в объявления на машинном языке, которые чип может выполнить. Приведенная выше инструкция `MOVE` может быть переведена так:

```
10000101    00000100    00000001
```

Первые восемь битов составляют машинный код для команды `MOVE`. Вторые восемь битов — это код для регистра `D`. Значение, которое дано, равно 4 в десятичной системе, что было бы подходящим, если бы регистры нумеровались от 1 до 4. Последние 8 битов составляют значение 1 в двоичной системе.

Вы можете представить результат выполнения этой команды внутри чипа так, как показано на рис. 21.1.

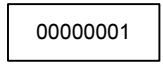


Рис. 21.1

Предположим, что десятичное значение 10 перемещено в регистр C. Результаты двух перемещений вместе можно представить следующим образом (рис. 21.2).

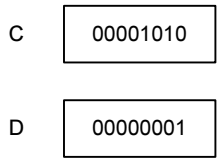


Рис. 21.2

Набор команд для чипа будет также включать арифметические операции, такие как сложение и вычитание. На языке ассемблера вычитание содержимого регистра D из содержимого регистра C может выглядеть так:

```
SUB C, D
```

На машинный язык это может быть переведено так:

```
10001011    00000011    00000100
```

Первые восемь битов — это код для команды вычитания, вторые восемь — код регистра C, а третьи восемь битов — это код для регистра D. Содержимое регистра D не изменится в этой операции, но из значения в регистре C, равное 10 в десятичной системе, будет вычтена 1. Результаты этой операции можно представить так, как показано на рис. 21.3, где двоичное число 00001001 равно 9 в десятичной системе.

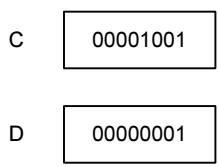


Рис. 21.3

Примеры в этой главе касаются только графических аспектов перемещения значений из одного регистра в другой. В следующей главе коды в примерах также будут включать арифметические функции.

Последовательность примеров также демонстрирует работу с памятью. В компьютере *память* является тем местом, где программы и данные хранятся для использования чипом. Память измеряется в байтах, а не в битах. Она также может быть измерена в "словах", где слово будет некоторым постоянным количеством байтов. В следующей главе будет показано, что машинное моделирование основано на блоках памяти по 4 байта. Первый байт предназначен для команды (например, `MOVE`), второй — для операнда цели, третий — для операнда источника, а последний байт остается не использованным. Там, где хранятся не программы, а просто данные, для последовательности также используются четырехбайтовые блоки, хотя значение не может превышать 8 битов, или одного байта.

В памяти можно указывать адреса. Это значит, что слова в памяти нумеруются по порядку, начиная с 0. Это дает машинному языковому коду возможность указывать места в программах или места, где хранятся значения программных переменных. В простых примерах программ свойство адресации не является очевидным. Оно является имплицитным и основано на изложенных допущениях моделирования. Память представлена в примерах, использующих текстовые области. В таких примерах двоичные значения перемещаются между регистрами и памятью. В следующей главе назначение памяти станет понятнее, и оно будет важно для функционирования программ, моделирующих компьютерный чип.

21.2. Примеры программ

Этот раздел начинает серию новых примеров. Дается краткое описание примеров вместе с изображением на экране.

MiscButton1

Эта программа иллюстрирует использование кнопки для вызова действия, а не чашки, на которой можно щелкнуть мышью. Содержимое чашки было снова изменено с графического изображения на текст, поскольку следующие примеры будут демонстриро-

вать работу с регистрами, содержащими строки символов, вместо чашек с семенами (рис. 21.4).

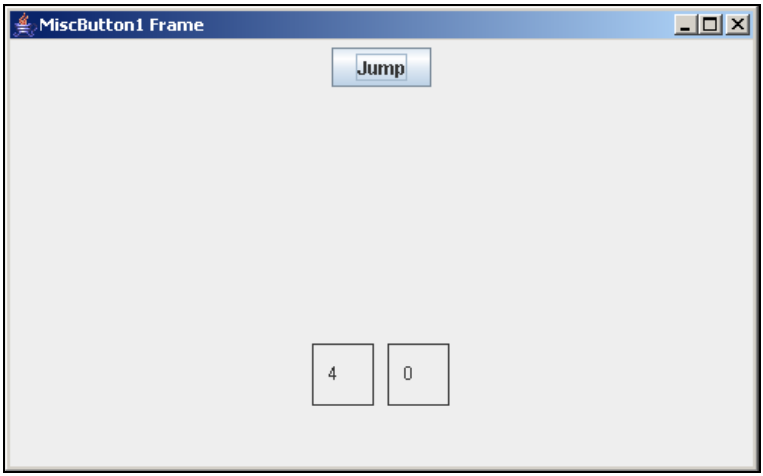


Рис. 21.4

MiscButton2

В этой версии приложения кнопка сдвинута вниз панели. Недостатком такого варианта является то, что ширина кнопки равна ширине панели (рис. 21.5).

MiscButton3

Эта версия приложения создает новую панель, на которой помещается кнопка. Когда кнопка помещается на панели, то изображение кнопки становится лучше. В целом, размещение компонентов на панели является простым способом компоновки приложения (рис. 21.6).

MiscRegister

Эта версия приложения переключается с чашек на регистры, которые содержат строки символов, представляющих двоичные числа. Код реализует обмен содержимого регистров. Нас сейчас

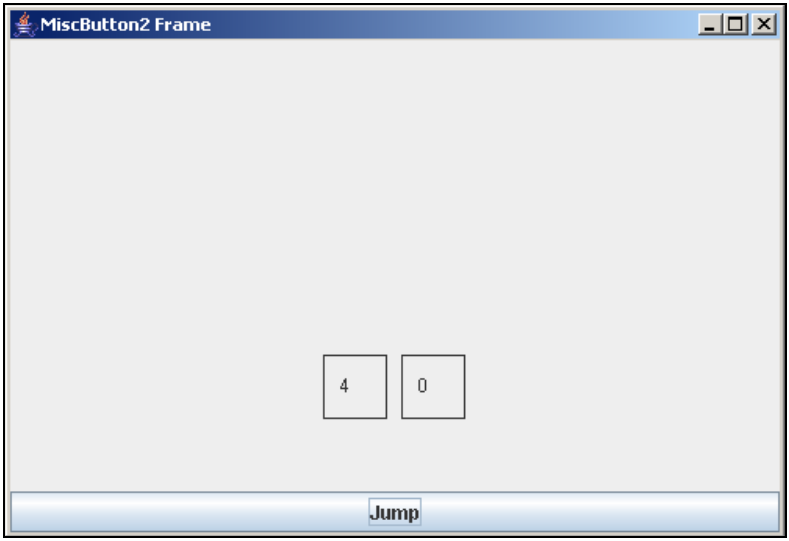


Рис. 21.5

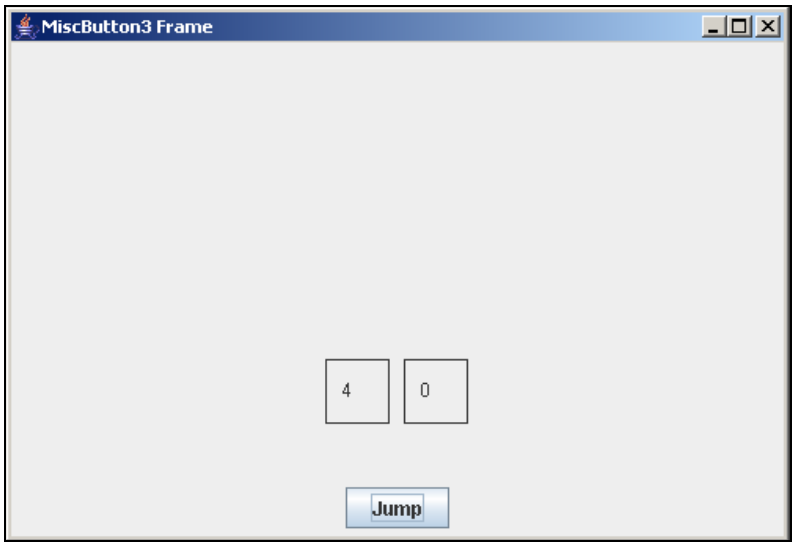


Рис. 21.6

не интересует, какой из регистров является активным. С этого момента в примерах больше не применяется та логика, на которой строились примеры с чашками (рис. 21.7).

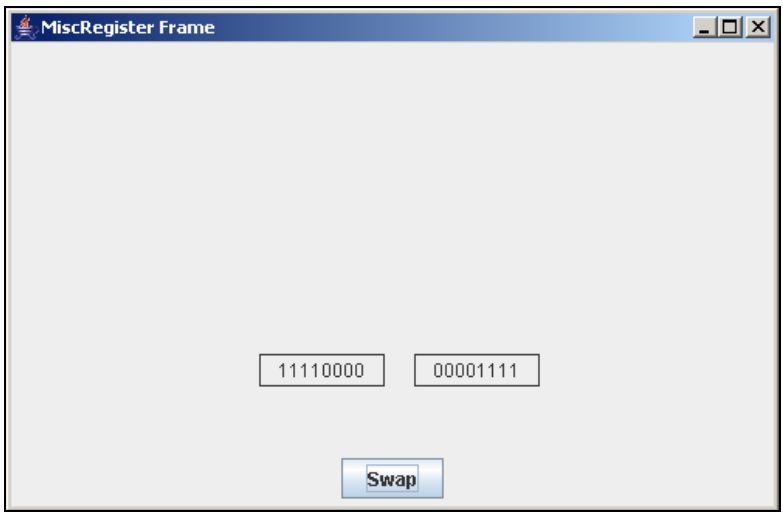


Рис. 21.7

MiscText

В этой программе каждый регистр представлен текстовым полем. Текстовые поля имеют собственные блоки прослушивания, так что можно вводить новые значения в регистры. Также имеется кнопка с блоком прослушивания. Кнопка вызывает обмен значениями между регистрами. С этого момента в наших примерах компоненты приложения уже не являются объектами класса, например `Cup`, которые пишутся с нуля. Компоненты предоставляются системой, и система следит за их обновлением, когда это необходимо. В коде для данного примера вызовы метода `repaint()` включены там, где их можно ожидать, но они представлены в виде комментариев. Время появления приложения на экране не зависит от их наличия или отсутствия. Во всех последующих примерах в коде отсутствует вызов метода `repaint()` (рис. 21.8).

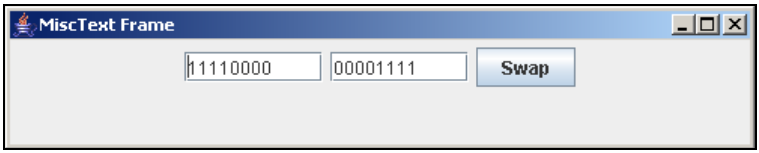


Рис. 21.8

MiscLabel

Эта программа содержит метки к текстовым полям. Она также помещает каждый визуальный компонент в его собственную панель и затем располагает эти подпанели в виде таблицы в классе `MiscLabelPanel` для того, чтобы получить желаемый внешний вид приложения. Использование компоновки, таким образом, оказывается очень удобным. Если вы выполните программу и затем измените размер ее окна на экране, вы увидите, что система автоматически расположит компоненты в таблице так, чтобы они умещались в новый размер окна (рис. 21.9).

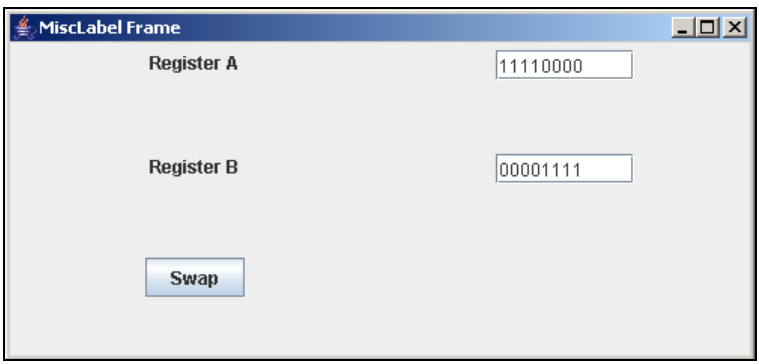


Рис. 21.9

21.3. Фокус

MiscFocus

Эта программа дает начальное представление об использовании *фокуса*. Когда в одно из текстовых полей вводится новое значе-

ние, курсор движется к следующему текстовому полю. Каждый регистр имеет переменную, содержащую ссылку на компонент, который должен получить фокус следующим. Эта переменная установлена на следующий регистр, когда создается очередной регистр. Код, который передает фокус, находится в блоке прослушивания текстового поля. Если код не будет явно управлять фокусом в графическом приложении, использование клавиши табуляции заставляет фокус перемещаться от одного компонента к следующему. Нет радикальных изменений во внешнем виде приложения, хотя рис. 21.10 показывает, что курсор находится в том текстовом поле, на котором в данный момент установлен фокус.

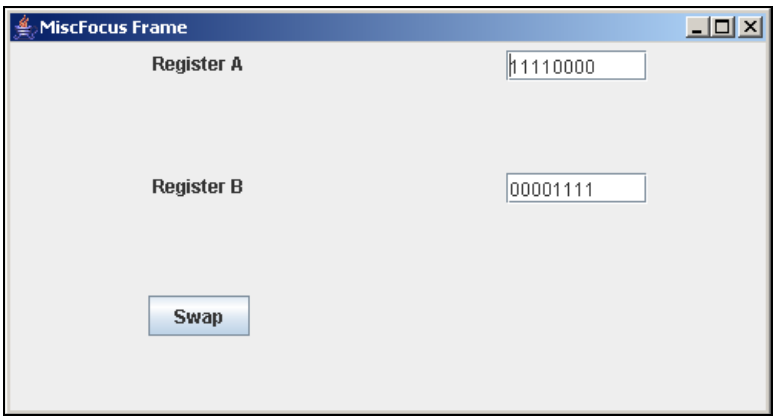


Рис. 21.10

Тема фокуса оказывается очень важной в графических приложениях. Остальная часть этого раздела посвящена краткому введению к методу, используемому в примере программы MiscFocus.java. Затем следует более глубокий анализ. Вы можете найти подробную информацию в учебной программе по языку Java или в описаниях подсистемы Java Focus. В документации Java API, в разделе о классе `Component`, имеется ссылка на оба эти источника.

Вспомните, как работает мышь. Если вы используете компоненты, имеющиеся в системе, и если курсор мыши находится на компоненте и вы производите щелчок, то этот щелчок будет ас-

социироваться с этим компонентом. Начиная с примера ClickCup *(см. разд. 19.1)*, было показано, что вы можете сделать так, чтобы ваш собственный компонент приложения работал таким же образом; достигается это за счет написания блока прослушивания мыши, получении местонахождения щелчка и сравнении этого местонахождения с местом вашего компонента. Таким образом, использование мыши достаточно просто. Имеются все необходимые функции, и принцип действия очевиден.

Понятие фокуса связано с обработкой клавишных событий и ассоциирования их с компонентами в приложении. Пример ClickKey *(см. разд. 20.1)* продемонстрировал простое использование клавишных событий. Все окно приложения реагировало на нажатие клавиш. Там имелся блок прослушивания, который предпринимал определенные действия в зависимости от того, какая из чашек была активной и какая клавиша была нажата. Однако фокус приложения оставался неизменным. Щелчки мыши воспринимало всегда окно целиком.

Начиная с примера MiscText *(см. разд. 21.2)*, приложения содержат два текстовых поля и кнопку. Поскольку это компоненты, имеющиеся в системе, то они обладают способностью получать фокус. Используя мышь, можно поместить в фокус одно из двух текстовых полей. Когда они находятся в фокусе, курсор будет располагаться в них, и они будут воспринимать нажатие клавиш и показывать соответствующие символы. Щелчок мыши по кнопке также помещает кнопку в фокус, что сигнализируется за счет выделения кнопки. Этим также вызывается действие кнопки.

Только один из компонентов приложения может иметь фокус в некоторый момент времени. Можно перемещать фокус с одного компонента на другой за счет клавиши табуляции, или щелчком мыши по компоненту. Можно перемещать фокус в обратном порядке, используя комбинацию клавиш <Shift>+<Tab>. Кроме того, если кнопка получила фокус, нажатие клавиши <Пробел> эквивалентно щелчку. Кстати, текстовое поле воспринимает нажатие клавиши табуляции как текстовый символ. Если в приложении есть текстовая область, которая находится в фокусе, то фокус может быть перемещен на следующий компонент за счет комбинации клавиш <Ctrl>+<Tab>.

В дополнение к управлению фокусом, имеющемуся в системе по умолчанию, существует возможность написания специального кода для управления фокусом в вашем приложении. Пример MiscFocus иллюстрирует такой код *(см. ранее в этом разделе)*. Когда клавиша <Enter> нажимается в одном поле, фокус перемещается в другое поле. Это указывается в коде для блока прослушивания текстового поля. Кроме того, что этот блок отыскивает содержимое поля для использования в приложении, он также содержит строку кода, которая запрашивает фокус для другого поля. Это достигается за счет следующего вызова:

```
component.requestFocusInWindow();
```

Тема фокуса очень обширна. Существует множество различных ситуаций, когда может быть желательным написать специальный код для управления фокусом в приложении. Иногда необходимо создать несистемные компоненты, которые могут получать фокус. Может потребоваться переопределить действия по умолчанию, например, порядок перемещения фокуса от одного компонента к другому. Может быть желательным ввести новое поведение фокуса, в дополнение к имеющемуся по умолчанию. Может потребоваться, чтобы приложение реагировало на нажатие других клавиш, а не тех, что предусмотрены по умолчанию. Иногда желательно написать специальный код, чтобы приложение действовало одинаково на разных платформах. В учебной программе по языку Java, например, упомянута следующая ситуация: в Windows то окно, которое располагается сверху, находится в фокусе; в Solaris фокусом обладает то окно, на котором находится курсор мыши.

Более полное понимание фокуса зависит от понятий *контейнеров* и *компоновки*. Любой компонент Java, который сам является контейнером, может иметь свою заданную компоновку, и она служит в качестве основы для цикла перемещения фокуса между всеми компонентами, которые могут быть в фокусе. Приложение может иметь более одного контейнера, поэтому на верхнем уровне возможно перемещение фокуса между контейнерами. Каждый контейнер имеет свою компоновку. Порядок перемещения фокуса по умолчанию в рамках контейнера соответствует тому порядку, в котором компоненты были добавлены в контейнер. В примере

MiscFocus имеется всего один общий контейнер — фрейм, где есть подокно содержания, которое включает общую панель приложения. Эта панель содержит многочисленные компоненты, каждый в собственной панели. Перемещение фокуса происходит между этими компонентами в их индивидуальных панелях.

Для того чтобы получать фокус, компонент, написанный программистом, должен иметь визуальное представление и должен быть сделан вызов метода `setFocusable()`:

```
component.setFocusable(true);
```

Вернемся к примеру ClickKey: в том примере только окно воспринимало нажатие клавиш. Можно написать такой код, чтобы каждая чашка имела свой блок прослушивания и, когда она находится в фокусе, реагировала бы по-разному на нажатие различных клавиш. Эта идея рассматривается в одном из примеров далее, MiscAction *(см. разд. 21.4)*.

Упрощенный, но логичный подход к изменению порядка перемещения фокуса заключается в изменении порядка, в котором компоненты добавляются к контейнеру. Можно также изъять какой-либо компонент из этой цепочки, сделав его неспособным воспринимать фокус. Это достигается с помощью следующего вызова:

```
component.setFocusable(false);
```

В целом, перемещение фокуса зависит от класса `LayoutFocusTraversalPolicy`. Как видно из названия, действия по перемещению фокуса зависят от компоновки (layout). Для того чтобы произвести крупные изменения в принципе перемещения фокуса, необходимо расширить класс `LayoutFocusTraversalPolicy`, создать экземпляр этого класса и затем добавить его к контейнеру со следующим вызовом:

```
container.setFocusTraversalPolicy(instance of policy);
```

Точно так же, как у контейнера есть компоновка, у него может быть и специальный порядок перемещения фокуса, связанный с такой компоновкой, который переопределяет описанный выше порядок, заданный по умолчанию. Также можно задать иные клавиши, которые будут вызывать перемещение фокуса.

Понятие фокуса касается ряда различных аспектов клавиш и приложений. Например, в Java имеется класс `InputVerifier`. Он может быть использован для того, чтобы убедиться, что текст, вводимый в поле, согласуется с заранее определенным форматом. Связь с фокусом заключается в следующем. Неформально, вы можете представлять себе перемещение фокуса с данного компонента в тот момент, когда текст воспринимается за счет нажатия клавиши <Enter>. Запускается проверка ввода за счет попытки изменить фокус, и если ввод не согласуется с заданными характеристиками, то фокус не перемещается. Он остается в том же поле, чтобы пользователь мог сделать новую попытку.

Как уже отмечалось, фокус представляет собой обширную тему; с ним связано несколько классов, имеющихся в системе Java. Класс `KeyBoardFocusManager` является главным классом по управлению фокусом, связанным с клавиатурой. Классы блоков прослушивания, имеющие отношение к фокусу, — это классы `FocusListener` и `PropertyChangeListener`. Более подробную информацию о фокусе можно прочитать в документации Java API.

В документации Java перечисляются следующие восемь понятий, связанных с классом `KeyBoardFocusManager` (общее представление о большинстве из них мы рассмотрели в нашем обсуждении):

- обладатель фокуса (компонент);
- постоянный обладатель фокуса (компонент);
- окно, находящееся в фокусе (фреймовое или диалоговое окно, которое содержит компонент, обладающий фокусом);
- активное окно (фреймовое или диалоговое, которое или имеет фокус, или обладает фреймовым или диалоговым окном с фокусом);
- перемещение фокуса;
- цикл перемещения фокуса;
- источник цикла перемещения фокуса;
- порядок перемещения фокуса.

Общая модель по перемещению фокуса основана на ряде событий, которые определяются в классах `WindowEvent` и `FocusEvent`.

Эта модель может быть объяснена на простом примере. Пусть в приложении имеются два окна, каждое содержит по компоненту. Пусть в окне A содержится компонент A, а окно B содержит компонент B. Пусть в данный момент фокус находится на компоненте A и пользователь щелкает мышью по компоненту B. Вот полная цепочка событий с фокусом, которая создается и происходит после совершения данного действия:

1. Компонент A получит `FocusEvent`: `FOCUS_LOST`.
2. Окно A получит `WindowEvent`: `WINDOW_LOST_FOCUS`.
3. Окно A получит `WindowEvent`: `WINDOW_DEACTIVATED`.
4. Окно B получит `WindowEvent`: `WINDOW_ACTIVATED`.
5. Окно B получит `WindowEvent`: `WINDOW_GAINED_FOCUS`.
6. Компонент B получит `FocusEvent`: `FOCUS_GAINED`.

Обратите внимание на то, что схема симметрична. При написании собственного кода для фокуса важно это помнить и использовать подобную схему. Пропуск какого-либо из шагов приведет к проблемам. При выполнении кода вы можете также встретить неожиданные задержки в перемещении фокуса с одного компонента на другой. Такие проблемы могут быть вызваны следующим. Компоненту дается команда получить фокус, но тот компонент, который в данный момент обладает фокусом, еще не отдал его. Иногда такие проблемы можно решить просто за счет изменения порядка команд в вашем коде. В сложных приложениях решение может быть не таким простым.

Все сказанное является кратким введением в тему фокуса; мы рассмотрели достаточно информации для того, чтобы в целом объяснить, что происходит в примере программы MiscFocus. Эти объяснения также должны служить вам отправной точкой при чтении документации Java API и просмотре демонстрационной программы Java, где можно найти более подробную информацию, если она вам потребуется при написании более сложных приложений.

21.4. Новые примеры программ

MiscAction

Эта программа показывает, как установить действие таким образом, чтобы сочетание клавиш <Ctrl>+<S> имело такой же эффект, как и щелчок мыши по кнопке **Swap**. Это достигается за счет создания экземпляра класса `AbstractAction` и присоединения его как к кнопке, так и к сочетанию клавиш. Это дает возможность того, чтобы более чем одно событие, например, щелчок мыши и сочетание клавиш, запускали один и тот же блок прослушивания. Для достижения этого необходимо только внести изменения в класс панели. Такое решение по управлению вводом с клавиатуры является более общим, чем пример ClickKey *(см. разд. 20.1)*. Внешне этот пример ничем не отличается от предыдущего (рис. 21.11).

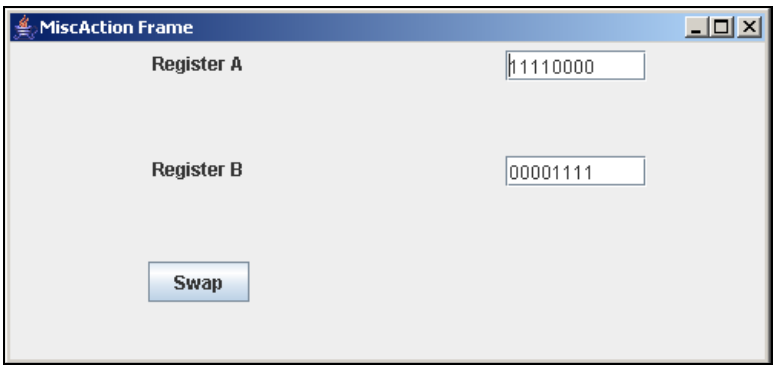


Рис. 21.11

MiscMulti

Этот пример иллюстрирует идею групповой передачи (multicasting). В данном случае каждое текстовое поле имеет собственный блок прослушивания, который отмечает единственное событие — нажатие кнопки **Clear**. Это процесс, обратный предыдущему примеру. Там разные события инициировали один и тот же блок прослушивания. В этом примере одно событие инициирует действие нескольких блоков. Это называется *групповой переда-*

чей. Изображение на экране (рис. 21.12) показывает результат щелчка по кнопке **Clear**. Оба текстовых поля установлены на значение 00000000.

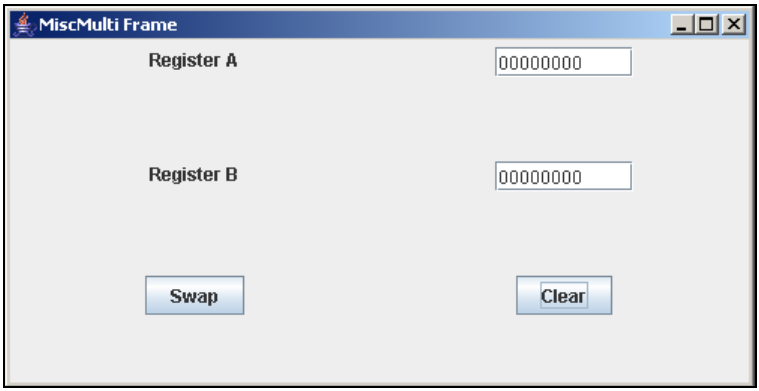


Рис. 21.12

MiscMemory

Эта программа иллюстрирует реализацию машинной памяти с использованием класса и визуальным представлением его с помощью текстовой области с полосами прокрутки. Функция загрузки файла в память реализуется с помощью специального пункта меню. В этом примере нет явного управления фокусом. Действует автоматическая смена компонентов с помощью клавиши табуляции. Однако, как только пользователь щелкает мышью на текстовом поле и помещает туда курсор, фокус перемещается туда. До тех пор, пока эта область не утрачивает фокус, все нажатия клавиш будут интерпретироваться как текст, вводимый в текстовую область. Они не будут восприниматься как команды. Хотя для данного приложения это не существенно, но блок прослушивания клавиатуры может быть реализован в качестве внутреннего класса панели и может быть добавлен к текстовой области панели. Текстовая область будет автоматически получать нажатие клавиш в качестве ввода, но также можно одновременно определить действия для этих клавиш в коде блока прослушивания.

Рис. 21.13 демонстрирует приложение с некоторыми данными, введенными в текстовую область. До тех пор, пока не будет введено достаточно данных, так, чтобы они выходили за границы этой области, полосы прокрутки не появятся.

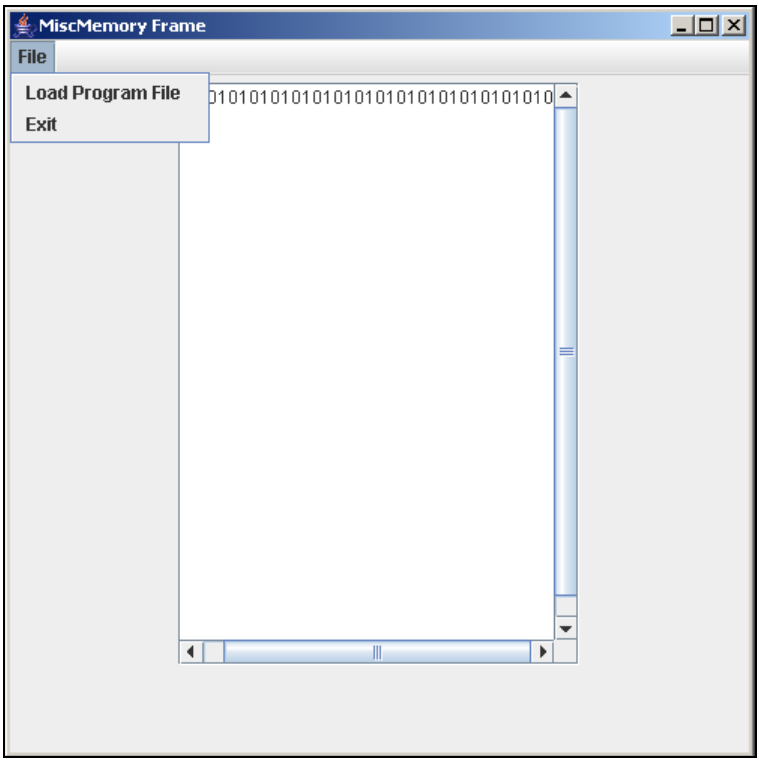


Рис. 21.13

MiscRegAndMem

Эта программа соединяет регистры и память. Щелчок мыши по кнопке не только вызывает обмен значений между регистрами, но и приводит к тому, что содержимое регистра `A` копируется в сдвиг 0, т. е. по адресу 0 памяти. Обратите внимание, что когда происходит обмен, область также заполняется значением 0 во всех других местах (рис. 21.14).

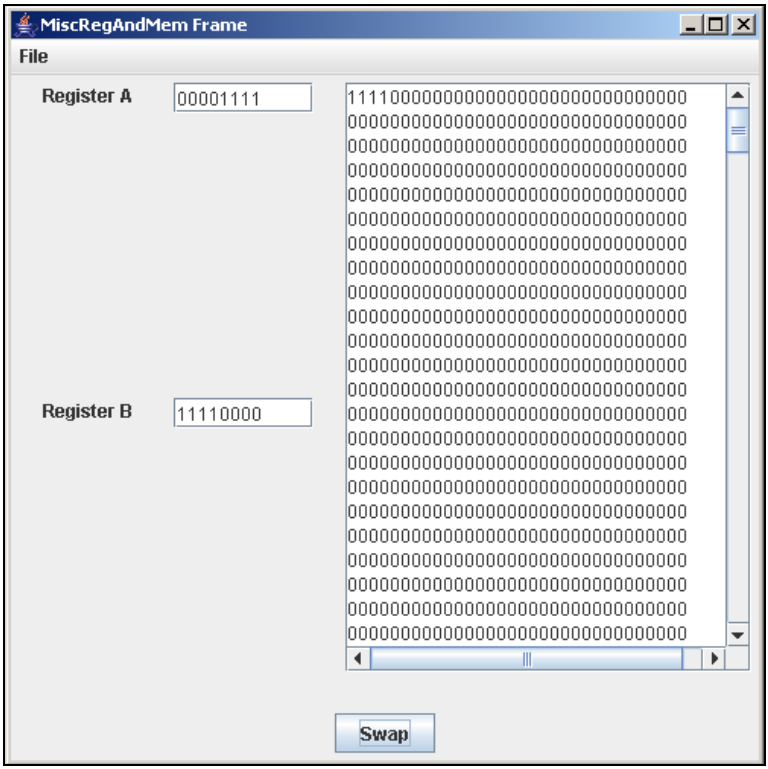


Рис. 21.14

21.5. Потоки

MiscThread

Этот пример показывает, как множественные фреймы могут быть добавлены к выполняемому приложению, где каждый фрейм имеет свой поток выполнения. Для того чтобы показать, что потоки работают одновременно, регистры постоянно обмениваются своим содержанием. Вызов метода `sleep()` помещается в метод `run()` с целью замедлить обмен так, чтобы его можно было легко заметить. Если вам нужно освежить в памяти метод `sleep()`, посмотрите *главу 5*. Для данного примера приводятся три рисунка.

В главном фрейме имеется меню **File** с опциями **Make a Sub Frame** (Создание подфрейма) или **Exit** (Выход). В подфреймах имеются регистры, а в меню — опции для файла: **Restart** (Запустить заново) или **Close** (Закрыть подфрейм) (рис. 21.15—21.17).

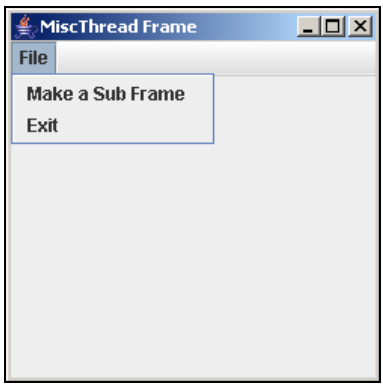


Рис. 21.15

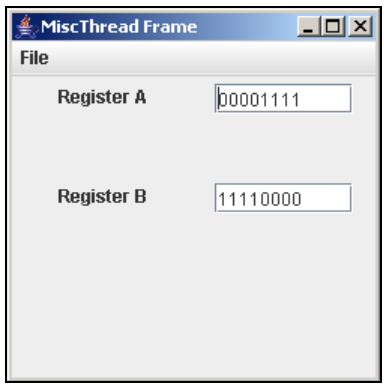


Рис. 21.16

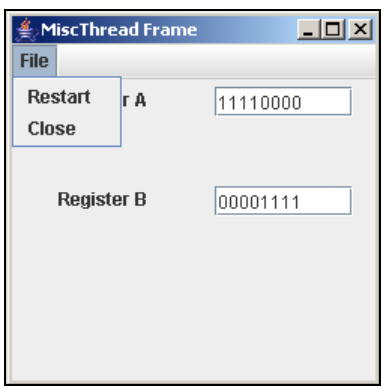


Рис. 21.17

Для того чтобы объяснить поточный код, необходимо более четко установить, что означают термины *"поток"* (thread) и *"параллельный"* (concurrent). Поток означает последовательное выполнение строк кода в программе. Параллельность значит то, что Java может управлять выполнением более чем одной программы

в одно и то же время. Программы не выполняются одновременно. Вместо этого система выполняет определенное количество команд из одной программы, переключается на какое-то время на другую программу, снова возвращается к первой, и т. д. Синтаксически, это обычно достигается не с помощью двух совершенно отдельных программ, а с помощью двух копий одной и той же программы. Поток — это термин, который используется для описания каждой из двух выполняемых копий программы. Более полное описание ситуации будет выглядеть так: имеются два потока выполнения одной и той же программы или фрагмента кода.

Потоки — это сложная тема, которая обычно включается в курсы по операционным системам. Здесь она не будет полностью описываться. Сложность возникает тогда, когда два различных потока имеют доступ к общим данным, с возможностью у обоих чтения и изменения этих данных. Управление такой ситуацией обычно называется *контролем за параллельностью* (concurrency control), и в языке это осуществляется с помощью понятия, известного как *синхронизация*. В данном разделе мы не будем рассматривать эту проблему или ее решения. Мы только намерены познакомить читателей с синтаксисом для создания множественных потоков. Необязательно, чтобы поточные коды имели общие ресурсы, и приложение MiscThread создано так, чтобы продемонстрировать потоки без общих ресурсов. Одно из заданий итогового проекта касается потоков, но требуется выполнить только то, что сделано в MiscThread. Если вы не будете писать коды потоков с общими ресурсами, то сможете без проблем создавать простые коды для потоков.

Синтаксически, существуют два пути для написания поточных кодов. Вот краткий обзор первого из них:

1. Напишите класс, являющийся расширением имеющегося в системе класса `Thread`.
2. Переопределите метод `run()` в этом классе.
3. В программе приложения создайте один или более экземпляров этого класса.
4. В программе приложения вызовите метод `start()` на объект (объекты).

Объекты этого класса и код в определении класса имеют поточное строение. Количество создаваемых объектов, к которым вызывается метод `run()`, определяет, сколько параллельных потоков будет существовать при выполнении программы.

Обратите внимание, что перед нами еще один пример ситуации, когда программа пользователя вызывает наследованный метод, который не виден, и никогда прямо не вызывает переопределенный метод, который виден в коде. Метод `start()` отводит место в памяти и инициализирует новый поток в системе Java. Затем он вызывает метод `run()` объекта. Если код пользователя прямо вызывает метод `run()`, то инициализация не происходит.

Такой подход к реализации потоков является прямым и будет работать во многих случаях. Не забывайте, что класс может расширить только один другой класс. Язык Java не поддерживает множественное наследование. Если класс, который должен быть разделен на потоки, уже является подклассом другого класса, то такой подход не сработает. Например, предположим, что вы хотите написать поточный апплет. Класс апплета должен будет расширить класс `JApplet`, и уже не будет возможности расширить класс `Thread`.

Вместо множественного наследования язык Java поддерживает интерфейсы, и можно реализовать поточный код путем реализации интерфейса `Runnable`. Вот краткое описание второго подхода к написанию кода для потоков:

1. Напишите класс, который реализует имеющийся в системе интерфейс `Runnable`.
2. Реализуйте метод `run()`, определенный интерфейсом в классе.
3. В программе приложения создайте один или более экземпляров этого класса. Поскольку в интерфейсе определен только метод `run()`, обратите внимание, что интерфейс сам по себе не предоставляет все необходимое для создания и запуска потоков.
4. В программе приложения для каждого объекта данного класса создайте поток за счет вызова конструктора `Thread()` и передачи ему ссылки на объект.

5. В программе приложения вызовите метод `start()` на экземпляры класса `Thread`, созданные таким образом.

Как обычно, в документации Java API имеется гораздо больше информации о классе `Thread` и интерфейсе `Runnable`. Кстати, сам класс `Thread` реализует интерфейс `Runnable`. Этим объясняется, почему класс содержит метод `run()`. Как уже говорилось, класс также имеет более одного конструктора, что делает возможным создание потоков для выполняемых объектов, которые передаются в качестве параметров. Вся эта схема разваливается, если программист не реализует метод `run()` в данном классе. Обратите внимание еще на два момента, которые, впрочем, могут быть очевидны и так:

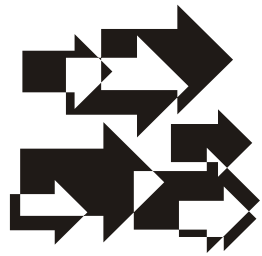
- конструкторы для самого выполняемого класса будут зависимы от приложения. То, что они содержат, не зависит от конструктора для класса `Thread`;
- на протяжении всего обсуждения мы не касались содержания метода `run()`. Этот метод содержит код приложения, который должен выполняться параллельно разными потоками.

Наконец, обратите внимание на соответствующую строку кода в примере MiscThread:

```
class MiscThreadSubFrame extends JFrame implements Runnable
```

В приложении именно подфреймы существуют параллельно. Они не имеют общих данных и выполняются независимо. Поскольку эти подфреймы, определенные в приложении, являются подклассами класса `JFrame`, необходимо, чтобы они реализовывали интерфейс `Runnable`. Для них невозможно будет написать код за счет расширения класса `Thread`.

ГЛАВА 22



Компьютер с минимальным набором команд

22.1. Использование MISC

MISC — это аббревиатура для Minimal Instruction Set Computer (компьютер с минимальным набором команд). MISC является моделью простого компьютерного чипа, написанного на языке Java. Используя эту модель, можно загрузить и выполнить программы на машинном языке. Данная глава дает необходимую информацию для использования модели и понимания ее структуры. MISC является основой для заключительного проекта, который состоит из изменения модели таким образом, чтобы она имела графический интерфейс пользователя. Полные требования к проекту даются в *разд. 22.7*.

По сравнению с примерами, которые мы рассматривали до сих пор, MISC является большой программой. Она состоит из следующих пяти классов: `MachineByteV1`, `MachineWordV1`, `MachineV1`, `OsystemV1` и `MachineOSProgramV1`. Начальная, неграфическая версия использует класс `MyTerminalIO`. Полезно сначала выяснить, что выполняет эта программа, прежде чем рассматривать содержание отдельных классов и их работу.

Метод `main()` находится в классе MachineOSProgramV1.java. Если классы скомпилированы и выполнены, то появится следующее:

```
Valid commands are:
load = load a program file
```

```
run = run program file
dump = dump memory contents
exit = quit program
This is the Osystem prompt.  Enter a command here:
```

Вот краткое описание команд.

- `load =` — загрузить файл программы. При введении этой команды пользователь получает приглашение ввести имя файла из программы машинного языка, который необходимо загрузить. В качестве примера приведен файл sumtenV1.txt (см. компакт-диск). Эта программа складывает первые десять положительных целых чисел.

  Обратите внимание, что файлы машинного языка должны быть простыми текстовыми файлами, и при приглашении ввести имя файла операционная система ожидает имя файла с расширением txt. При введении имен файлов с другими расширениями программа вроде бы примет такое имя, но не будет работать правильно.

- `run =` — выполнить программный файл. Эта команда вызывает выполнение той программы, которая загружена в данный момент в модель MISC.

- `dump =` — разгрузить содержание памяти. При введении этой команды пользователь получает приглашение создать имя файла вывода. Затем система поместит в память содержимое этого файла в виде 256 строк по 32 бита в каждой. MISC не имеет функции ввода/вывода. Результаты выполненной программы можно просмотреть только за счет разгрузки содержания памяти в файл, который затем открывается с помощью редактора.

  Обратите внимание, что файл вывода также должен быть текстовым файлом с расширением txt.

- `exit =` — завершить работу модели. Это не команда операционной системы. Эта команда останавливает выполнение класса `MachineOSProgramV1`, являющегося драйвером для модели.

Посмотрим на последовательность команд и содержания файлов, которые приводятся далее для иллюстрации использования

MISC. Для наших примеров не потребуются все 256 строк, так что показаны только фрагменты содержания файлов. Предположим, что модель MISC запущена и пользователь вводит команду `dump`. Поскольку сначала ничего не было загружено, то содержанием файла вывода будут одни нули, отражающие тот факт, что память пуста.

Итак, содержание памяти:

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
...
```

Теперь предположим, что пользователь ввел команду `load`. После введения имени файла sumtenV1.txt пользователь вводит команду `dump`. Далее показаны начальные строки памяти MISC, демонстрирующие загруженный файл sumtenV1.txt. Затем даются пояснения.

Теперь содержание памяти:

```
00001011000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

```
00000000000000000000000000000000
10000101000001000000000100000000    Примечание 1
10000111000000010000010000000000
10001010000001000000000100000000
10000011000000110000000000000000
10001011000000110000010000000000    Примечание 2
10010001000010010000000000000000
00000000000000000000000000000000
...
```

Первая строка в блоке вывода содержит значение, а не команду. Имеют значение только первые восемь битов в строке. В двоичной системе 00001011 равняется 11 в десятичной. Это верхний предел цикла в коде, который складывает первые десять целых чисел. Идея заключается в том, что цикл выполняется до тех пор, пока индекс строго меньше 11. Строки внизу, которые не содержат одни нули, являются командами на машинном языке.

В предыдущей главе было показано использование регистров с помощью команд на машинном языке для перемещения и вычитания. Команда по перемещению на языке ассемблера (`MOVE D, X01`) была переведена на машинный язык следующим образом:

```
10000101  00000100  00000001
```

Если вы посмотрите на примечание 1 в файле вывода, то увидите эту команду там, за которой следуют дополнительные восемь нулей. Аналогичным образом, команда на языке ассемблера для вычитания (такая как `SUB C, D`) была переведена на машинный язык следующим образом:

```
10001011  00000011  00000100
```

Посмотрев на примечание 2, вы увидите, что эта команда также имеется в файле вывода, и за ней следуют дополнительные восемь нулей.

Теперь предположим, что пользователь вводит команду `run`, за которой следует команда `dump`. Первые строки памяти MISC, показывающие результаты выполнения файла sumtenV1.txt, таковы:

```
00001011000000000000000000000000
00110111000000000000000000000000
```

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10000101000001000000000100000000
10000111000000010000010000000000
10001010000001000000000100000000
10000011000000110000000000000000
10001011000000110000010000000000
10010001000010010000000000000000
00000000000000000000000000000000
...
```

Вторая строка в памяти MISC уже не представлена одними нулями. Она содержит двоичное число 00110111. Это равно 55 в десятичной системе, т. е. сумма первых десяти целых чисел. Такая последовательность шагов показывает, как может быть использована модель MISC. Как работает программа, будет объяснено далее.

22.2. Структура MISC

Классы, входящие в MISC, были перечислены в предыдущем разделе. Теперь мы рассмотрим их содержание и функции.

- `MachineByteV1`. Это контейнер для массива из 8 символов. Каждый символ является или 1, или 0, так что это представляет один байт в модели.
- `MachineWordV1`. Это контейнер для массива из четырех объектов `MachineByteV1`. В машинной архитектуре один сегмент доступной памяти, т. е. одно слово, равен 4 байтам.
- `MachineV1`. Это сердце модели и его содержимое можно разделить на несколько категорий.
  - Аппаратное обеспечение машины, ее регистры и память моделируются с помощью элементов массивов необходимых типов. Они объявляются и создаются в классе `MachineV1`.

Класс `MachineV1` имеет метод общего назначения `showStuff()`, который вызывается каждый раз, когда выполняется программа. Он помещает полное содержимое машины, включая регистры, в файл showfile.txt. При работе над заданием может оказаться полезным включить вызовы этого метода в различных местах вашего кода, для того чтобы помочь обнаружить ошибки.

- Машина имеет несколько специальных методов, которые не поддерживают общие команды машинного языка. Вместо этого они могут быть использованы моделируемой операционной системой для выполнения ввода/вывода. Это:

  ```
  loadWordToOffset()
  getWordFromOffset()
  ```

- Машина имеет несколько методов, которые содержат логику для выполнения программы машинного языка. Это:

  ```
  totalReset()
  resetOffsets()
  takeControl()
  ```

  Самым фундаментальным из них является `takeControl()`. Он вызывается после того, как программа загружается. Этот метод содержит встроенную логику перехода к следующей команде программы, загруженной в память, и вызова метода в модели, который реализует ее.

- Машина имеет методы, которые реализуют команды "переместить", "добавить", "вычесть" и "перейти" машинного языка. Она также имеет некоторые вспомогательные методы, которые поддерживают арифметические действия. Один из них поддерживает арифметические действия с целыми числами, когда содержание машины находится в двоичной форме. Другой устанавливает регистр флагов для согласования с результатом арифметических операций.

- `OsystemV1`. Этот класс имеет конструктор, в котором создается копия `MachineV1`. Он также содержит три метода:

  ```
  loadProgramFile()
  runProgramFile()
  dumpMemoryContents()
  ```

Это методы, которые реализуют команды "загрузить", "выполнить" и "разгрузить содержание памяти" соответственно.

- `MachineOSProgramV1`. Это класс, содержащий метод `main()`. Это драйвер модели. В нем создается экземпляр класса `OsystemV1`. Остальная часть программы фактически представляет собой цикл, который приглашает пользователя ввести команды и проверяет, ввел ли пользователь команды класса `OsystemV1` и соответствующие имена файлов. Он также поддерживает команду выхода из программы, что вызывает остановку цикла `MachineOSProgramV1`.
- `MyterminalIO`. Это знакомый класс ввода/вывода.

22.3. Машинный класс

MISC имеет некоторое сходство с архитектурой и машинным языком чипа Intel 8088. Ни в коей мере модель не является точной реализацией чипа, и конструктивные решения в модели направлены на простоту понимания и реализации, а не на реальное моделирование какой-либо конкретной архитектуры.

Чип имеет 17 регистров, размером 8 бит каждый. Машина использует слова размером в 4 бита. Память состоит из 256 адресуемых слов, или 1024 байтов. Имеются отдельные регистры смещения для кода и данных. Эти регистры ведут учет того, какая команда и какая переменная машинного языка задействованы в любой момент времени. Память, выделенная для программы, может быть рассмотрена как состоящая из двух частей. Фиксированное количество памяти отводится для данных программы, а фиксированное максимальное количество памяти — для кода программы. Команды машинного языка и объявления данных состоят из блоков в 4 байта. Содержимое регистра, равно как и содержимое одного байта в памяти, моделируется объектом, содержащим символьный массив в 8 байтов.

Реализация регистров в виде массива с именем `reg` показана далее. Индекс массива указывает на конкретный регистр. Даны обычные, описательные имена регистров, значение индекса каждого элемента в массиве регистра и значение индекса в виде

8-битного двоичного кода, как он обозначался бы в программе машинного языка.

Регистры:

```
идентификация     десятичный индекс     двоичный код
имени регистра    в массиве  reg        индекса

unused            reg[0]                "00000000"

общие
A                 reg[1]                "00000001"
B                 reg[2]                "00000010"
C                 reg[3]                "00000011"
D                 reg[4]                "00000100"
```

Память отведенная:

```
codeoffset        reg[5]                "00000101"
dataoffset        reg[6]                "00000110"
unused1           reg[7]                "00000111"
unused2           reg[8]                "00001000"
unused3           reg[9]                "00001001"

flag              reg[10]               "00001010"
```

Контролирующие регистры:

```
instruction       reg[11]               "00001011"
operand1          reg[12]               "00001100"
operand2          reg[13]               "00001101"
extra             reg[14]               "00001110"
```

ALU-регистры:

```
aluinreg1         reg[15]               "00001111"
aluinreg2         reg[16]               "00010000"
aluoutreg         reg[17]               "00010001"
```

Регистры общего назначения `A`—`D` находятся в распоряжении программиста при написании команд машинного кода. Регистр `codeoffset` фактически является счетчиком программы или команд. Регистр `dataoffset` зарезервирован для хранения адреса памяти. Команды могут принимать адреса операндов напрямую,

поэтому регистр `dataoffset` мало употребляется в коде. Имеются три дополнительных регистра смещения, которые в настоящее время не используются. Регистр флагов, контролирующие регистры и регистры ALU не используются непосредственно в коде машинного языка. Они будут объяснены далее.

Память также реализуется в модели в виде массива. Каждый элемент массива — это 4-байтовое машинное слово. Индекс массива представляет смещение слова в памяти.

```
имя массива      memory[]
```

Последний раздел данной главы *(см. разд. 22.7)* подробно описывает итоговый проект. Однако уже сейчас можно указать на связь между примерами к игре Wari и MISC и на то, какое это имеет отношение к проекту. Примеры Wari начались с использования чашек, содержащих семена, и чашки были реализованы в виде массива, а не как отдельные объекты. Для того чтобы сделать игру Wari графической, необходимо было для чашек создать класс с атрибутами, которые могли быть использованы для изображения чашек, а также запрограммировать манипуляцию объектами данного класса. Обратите внимание, что неграфическая версия Wari имеет регистры в качестве контейнеров для байтов, и регистры реализуются в виде массива. Целью итогового проекта является создание класса для регистров с атрибутами, которые могут быть использованы для их (регистров) изображения. В предыдущей главе этот процесс был начат. В ней были даны примеры регистров, реализованных в виде текстовых файлов, и памяти, реализованной в виде области текста.

22.4. Машинные команды

Модель MISC реализует небольшой набор машинных инструкций, достаточных для написания простых программ. Например, для выполнения циклов, которые ведут подсчет, достаточно команд для перемещения данных, выполнения простых арифметических действий с этими данными и перехода от одной команды к другой во время выполнения программы. Набор команд поддерживает использование переменных памяти, регистров и констант. В данной версии MISC команды реализуются в виде

методов класса `MachineV1`. Важным решением при создании программы будет вопрос о том, где реализовать набор команд, когда имеется класс для регистров. В данном разделе предполагается, что вы помните вводную информацию о регистрах и командах машинного языка, которая была дана в предыдущей главе.

Основные правила как для перемещения данных, так и для арифметических действий следующие:

- местом назначения может быть регистр или переменная памяти;
- источником может быть константа, регистр или переменная памяти;
- не допускаются операции от памяти к памяти.

Поскольку все в машинном языке представлено простыми двоичными кодами, то нет возможности различать константы, регистры или смещения памяти. Существуют специальные машинные команды для всех допустимых комбинаций операндов, и программист, пользующийся машинным языком, должен выбирать соответствующую команду при написании кода. Посмотрим на описание конкретных машинных команд, поддерживаемых MISC.

Команда *MOVE*

Не допускаются перемещения с константами в качестве получателя и перемещения с двумя переменными памяти в качестве операндов. Каждое допустимое перемещение параметров получает свою машинную команду. Далее представлены возможные комбинации, показанные на языке ассемблера, имена методов, которые реализуют их в модели MISC, и их коды двоичных команд.

```
MOVE регистр, регистр
    void moveDestRegSrcReg()    "10000001"
MOVE память, регистр
    void moveToMemFromReg()     "10000010"
MOVE регистр, память
    void moveToRegFromMem()     "10000011"
```

```
MOVE память, константа
    void moveToMemFromConst()  "10000100"
MOVE регистр, константа
    void moveToRegFromConst()  "10000101"
```

Команда *ADD*

Приведенная далее информация излагается по той же схеме, что и для команды MOVE. Содержание операнда-источника не изменяется в данной операции. Содержание операнда-адресата заменяется результатом.

```
ADD регистр, регистр
    void addDestRegSrcReg()    "10000110"
ADD память, регистр
    void addToMemFromReg()     "10000111"
ADD регистр, память
    void addToRegFromMem()     "10001000"
ADD память, константа
    void addToMemFromConst()   "10001001"
ADD регистр, константа
    void addToRegFromConst()   "10001010"
```

Команда *SUB*

Информация для команды SUB излагается по той же схеме, что и для команды ADD. Содержание операнда-источника вычитается из операнда-адресата.

```
SUB регистр, регистр
    void subDestRegSrcReg()    "10001011"
SUB память, регистр
    void subFromMemSrcReg()    "10001100"
SUB регистр, память
    void subFromRegSrcMem()    "10001101"
SUB память, константа
    void subFromMemSrcConst()  "10001110"
SUB регистр, константа
    void subFromRegSrcConst()  "10001111"
```

Команда *JUMP*

Переход (jump) не является арифметической командой. Это управляющая команда. Единственным параметром ее является значение целого числа без знака. Адреса, или строки кода на машинном языке, к которым совершается переход, следующие:

```
JMP целое_число_без_знака
    void jumpUnconditional()    "10010000"
JPOS целое_число_без_знака
    void jumpOnPositive()       "10010001"
JNEG целое_число_без_знака
    void jumpOnNegative()       "10010010"
JZERO целое_число_без_знака
    void jumpOnZero()           "10010011"
JOVER целое_число_без_знака
    void jumpOnOverflow()       "10010100"
```

22.5. Программы машинного языка

Цель данного раздела — более полно объяснить пример программы sumtenV1.txt. Программы машинного языка MISC загружаются, начиная с адреса 0 в памяти. В программах есть сегмент с данными, который указывается сначала, и сегмент с кодом, который следует за ним. Каждая строка кода состоит из 4 байтов или одного слова. В объявлении данных первый байт является изначальным значением переменной и остальные байты не используются. Единица данных определяется по ее смещению. Смещение единицы данных — фактически это номер строки в программе, начиная с 0. Слова 0—7 зарезервированы для переменных с данными, что значит, что программа может иметь максимум восемь переменных. Неиспользованные слова будут заполнены нулями.

Командная строка кода состоит из одного байта для команды, двух байтов для не более чем двух операндов, и неиспользуемого байта. Фрагмент кода с программой начинается на смещении 8. Команды по переходу должны быть написаны с операндами с инкрементом 8. Внутри машины метод `takeControl()` остановит-

ся, если встретит команду со всем нулями. Поэтому в памяти за полной программой должно следовать хотя бы одно слово, где первый байт — это нули. При необходимости, это может быть 32-я строка, означающая, что возможна максимум 31 строка для программы, или до 8 переменных и до 23 строк кода.

Рассмотрим пример программы sumtenV1.txt снова. Код машинного языка показан с искусственными разрывами строк и пометками сегментов. Настоящий файл программы на машинном языке не может содержать символ возврата каретки или новых строк. Он должен быть полностью линейным. Звездочки (*) в вводимом файле обозначают конец программы.

Сегмент данных:

```
00001011000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Сегмент кода:

```
10000101000001000000000100000000
10000111000000010000010000000000
10001010000001000000000100000000
10000011000000110000000000000000
10001011000000110000010000000000
10010001000010010000000000000000
********************************
```

Теперь посмотрим на эту программу еще раз, с разрывами между байтами в строках, с соответствующими кодами на языке ассемблера и краткими пояснениями. В языке ассемблера слэш (/) используется в качестве разделителя.

```
/.DATA///
сегмент данных:
/LOOPLIM/X0B//
loop limit offset 0, value 11
```

```
00001011  00000000  00000000  00000000
/ACCUM/X00//
accumulator offset 1, value 0
00000000  00000000  00000000  00000000
6  неиспользуемых строк с нулями не показаны.
/.CODE///
сегмент кода:
/MOVE/D/X01/
move reg D, const 1
moveToRegFromConst 4, 1
10000101  00000100  00000001  00000000
/.LABEL/LOOPTOP//
/ADD/ACCUM/D/
(LABEL) add data offset 1, reg D
addToMemFromReg 1, 4
10000111  00000001  00000100  00000000
/ADD/D/X01/
add reg D, 1
addToRegFromConst 4, 1
10001010  00000100  00000001  00000000
/MOVE/C/LOOPLIM/
move reg C, data offset 0
moveToRegFromMem 3, 0
10000011  00000011  00000000  00000000
/SUB/C/D/
sub reg C, reg D
subtractDestRegSrcReg 3, 4
10001011  00000011  00000100  00000000
/JPOS/LOOPTOP//
```

Поскольку имеется место для 8 переменных, то первая команда кода начинается на слове 8. Следующая команда `jump` передает контроль на вторую команду — на слово 9:

```
jump on positive to "LABEL"
jumpOnPositive 9
10010001  00001001  00000000  00000000
/.END///
********  ********  ********  ********
```

22.6. Машинная арифметика

Понимание данной реализации является отправным моментом для ее изменения. Выполнение итогового проекта не должно включать в себя изменение основного выполнения команд MISC или арифметических действий. Выполнение команды происходит следующим образом. После того как программа загружена, машина берет на себя контроль выполнения с помощью вызова метода `takeControl()` за счет моделируемой операционной системы. Она начинает со значения 0 в регистре смещения кода, считывает содержание четырех последовательных байтов памяти кода и помещает их в регистры `instruction`, `operand1`, `operand2` и `extra registers`, т. е. помещает их в регистры `reg[11]`, `reg[12]`, `reg[13]` и `reg[14]` соответственно. После получения каждой команды и до ее выполнения смещение кода инкрементируется для следующего получения. Модель проходит через содержимое сегмента кода до тех пор, пока не встретит пустой байт команд. Иными словами, она продолжает работать, пока командный байт, полученный из кода, не будет заполнен нулями.

Прохождение через программный код является автоматической характеристикой системы. Поскольку инкремент производится в методе `takeControl()`, очень важно, чтобы это произошло до выполнения полученной команды. Если бы команда была, например, "перейти" (`jump`), то это поместило бы новую величину в регистр смещения кода — адрес команды "перейти". Если бы автоматический инкремент производился после выполнения команды, то он бы инкрементировал адрес, к которому нужно перейти, что было бы неверно.

Представление числовых величин является отправным пунктом для обсуждения регистров и арифметических действий в MISC. MISC обладает способностью обрабатывать как положительные, так и отрицательные числа. Используется представление целых чисел с дополнением до 2.

При дополнении до 2 самый старший, или самый левый, бит из ряда в 8 битов зарезервирован для знака, оставляя 7 битов для значения. Положительные целые числа представлены просто двоичными числами, при условии, что старший бит всегда 0. Это

эквивалент знака "плюс". Представление отрицательных величин образуется так. Берем полное 8-битное представление соответствующей положительной величины. Образуем его дополнение до 1. Это означает перемену всех битов с 1 на 0 и всех битов с 0 на 1. Эта процедура всегда приводит к 1 в старшем бите. Теперь добавляем 1, используя обычное сложение двоичных чисел.

Обратите внимание, что если вы попробуете перевести значение нуля — 00000000 — в отрицательное представление, то когда вы поменяете все биты, получите 11111111, и, добавив 1, получаете 100000000. Это, однако, 9-битное число. 1 является "переносом", который теряется от остающегося 8-битного представления. 8-битное число с дополнением до 2 представляет десятичное число –128. Когда положительное значение добавляется к его отрицательному, 1 бит вызывает *переполнение* (overflow), давая желаемый ответ, равный нулю. Может быть полезным создать некоторые примеры положительных и отрицательных значений с дополнением до 2 и попытаться сложить их.

Запись дополнения до 2 имеет следующие характеристики:

- существует только одно представление числа 0;
- вычитание реализуется добавлением отрицательного значения вычитаемого числа;
- если положительное значение инкрементировано на одну единицу выше наибольшего возможного положительного значения, то результатом будет отрицательная величина наибольшего масштаба, какая только может быть представлена в машине; и когда значение –1 инкрементируется, при игнорировании бита "переноса" остающиеся 8 битов просто содержат значение 0. Таким образом, увеличение и уменьшение просто вызывают *перекат* (rollover), когда в представлении достигаются максимально допустимые величины.

В модели значения смещения всегда являются положительными, и машина поддерживает 256 смещений в каждом сегменте памяти. Это означает, что все 8 битов должны быть использованы для того, чтобы содержать смещения, и когда производятся арифметические действия на значениях смещений, то они представляют собой простую двоичную арифметику на положительных числах.

И хотя существует концептуальная разница между целыми числами со знаками и без знаков, в модели на уровне сумматора различия отсутствуют. Какое бы число ни вводилось, сумматор просто произведет сложение. Сложение целых чисел без знаков с использованием простого двоичного представления может привести к переполнению. Машина "не знает", служит ли переполнение сигналом ситуации, вышедшей за пределы диапазона. Только программист знает, должны ли значения быть со знаками или без знаков, и, следовательно, о чем сигнализирует перекат, и являются ли результаты, полученные сумматором, арифметически правильными.

В модели имеются три регистра с арифметическими/логическими элементами (ALU): `aluinreg1`, `aluinreg2` и `aluoutreg`. Это `reg[15]`, `reg[16]` и `reg[17]` соответственно. Когда выполняется команда, операнды копируются в эти "входящие" регистры, и там выполняются арифметические действия. Результат, который помещается в "исходящие" регистр, затем копируется в регистр-получатель. Система использует содержимое регистра "исходящие" для установки значения регистра флагов, который указывает, был ли результат последней арифметической операции положительным, отрицательным или нулевым. Регистр флагов также отмечает, было ли переполнение во время последнего арифметического действия.

Три возможных результата арифметической операции — это 0, положительный результат и отрицательный результат. Вот как первые два бита сигнального регистра используются для записи этих результатов:

```
flag[0]  flag[1]
0        0
Результат операции был равен 0
1        0
Результат операции был положительным
0        1
Результат операции был отрицательным
1        1
Эта комбинация не используется
```

Третья позиция в регистре флагов используется для отметки переполнения. Если `flag[2]` содержит 0, то это указывает, что переполнение не произошло. Если он содержит 1, это сигнализирует о переполнении. Остальные 5 позиций в регистре флагов не используются.

В целом, арифметические действия в модели были реализованы с использованием двоичных алгоритмов. Однако на различных внутренних стадиях модели необходимо работать с содержимым регистров, и вовсе не обязательно делать все это с помощью двоичной арифметики. Для того чтобы создать такие элементы кода — инкрементирование регистра, содержащего выделение — сравнительно легкими и прозрачными, существует метод `incrementWord()`. Он использует методы `copyIntToWord()` и `getIntFromWord()`, чтобы преобразовать 8-битный двоичный массив в десятичный, добавить 1 и затем снова преобразовать результат обратно в представление в виде массива. Выполнение такого рода операций над содержимым регистра является удобным при написании модели, но в реальной машине арифметические операции таким образом не выполняются.

Наконец, еще немного о том, как выполняются команды MISC. Команда `MOVE` понятна. Она не зависит от сигнального регистра или регистров ALU. Остальные типы команд взаимодействуют с этими регистрами. Каждая команда устанавливает первые три бита сигнального регистра в зависимости от результата команды. Прибавление целых чисел со знаком или без знака может привести к перекату и переполнению, соответственно, но это задача программиста — следить за результатом.

Вычитание достигается путем преобразования второго операнда в его отрицательное представление в дополнении до 2 и прибавления его к первому операнду. Операнд-источник сам по себе не изменяется, потому что преобразование с дополнением до 2 появляется только в регистре `aluinreg2`. Содержание регистра-адресата заменяется результатом. Каждая команда на вычитание устанавливает первые три бита сигнального регистра в зависимости от результата операции. Вычитание целых чисел со знаками может привести к переполнению. Вычитание целых чисел без знаков проблематично. Если результатом будет отрицательное

число, тогда то, что представлено в машине, не будет ни результатом переполнения, ни результатом переката — это просто не будет соответствовать любому типу данных, определенному в модели. Программист не может полагаться на автоматическое преобразование между целыми числами со знаком и без знака и должен позаботиться о том, чтобы избегать неверных арифметических операций с неверными типами.

Когда машина декодирует команду `jump`, получатель помещается в регистр `operand1`. Когда команда выполняется, это значение помещается в регистр со смещением кода. Важно, чтобы программист проверил счет по 4 при создании смещения для `jump`. Безусловный переход понятен без объяснений. Условные переходы зависят от текущего содержания сигнального регистра. Как правило, программа написана так, чтобы команда `jump` следовала сразу же за арифметической операцией, которая зависит от нее.

22.7. Подробное описание проекта

Проект состоит из десяти частей, описываемых далее. В ряде случаев между частями существует взаимосвязь. Также в конце этого раздела приведена схема, иллюстрирующая существующие взаимосвязи. Две части проекта отмечены звездочками (***). Это означает, что их успешная реализация касается тех аспектов языка Java, которые не иллюстрировались примерами программ. Для того чтобы выполнить эти части, необходимо посмотреть соответствующие разделы документации Java API.

1. Создайте фрейм с меню. В меню должны быть опции для выполнения команд: загрузить (`load`), выполнить (`run`), разгрузить (`dump`) и выход (`exit`), вместе со своими блоками прослушивания. Используйте класс `JFileChooser` для обработки информации об именах файлов, которые должны передаваться для выполнения команд `load` и `dump`.

   Эту часть выполните в первую очередь.

2. В графическом фрейме приложения выведите на экран содержание регистров машины. Представления регистров должны

обладать способностью к вводу и выводу. Это означает использование экземпляров класса `JTextField`. Вы должны пометить регистры с помощью экземпляров класса `JLabel`. Поскольку приложение выполняет файлы от начала до конца без перерывов, достаточно показать содержание регистров в конце выполнения программы машинного языка. Нет необходимости обновлять представление во фрейме во время выполнения программы.

Выполните эту часть после завершения части 1.

3. Добавьте экземпляр класса `JButton` к графическому фрейму и внесите изменения в код приложения так, чтобы следующая команда выполнялась только тогда, когда нажимается кнопка (щелчком мыши). В данной версии приложения содержание регистров, как они показаны во фрейме, должно обновляться после каждого щелчка по кнопке, т. е. после выполнения каждой команды. Разумно предположить, что следующая команда все еще находится в памяти, и то, что отображается в регистрах после щелчка — это предыдущая выполненная команда вместе с результирующим состоянием после ее выполнения.

   Выполните эту часть после завершения части 2.

4. Эта часть состоит из двух отдельных компонентов:

   - добавьте фокус в приложение, чтобы, когда вы вводите что-либо в один из регистров с клавиатуры, курсор перемещался к следующему регистру;
   - добавьте кнопку, которая будет очищать содержание всех регистров без изменения загруженной программы. В общем, очищение означает установку на 0.

   Вам будет нужно подтвердить, что, какое бы значение ни вводилось в регистр смещения кода, оно является верным, так чтобы выполнение начиналось с начала программы машинного языка, когда нажимается кнопка `run`. Обратите внимание, что это не является идеальным решением, поскольку переменные значения памяти могли быть уже изменены до того, как регистры очищаются для нового запуска программы с самого начала.

Вы можете выполнить эту часть после завершения части 3. При желании, вы можете пропустить эту часть и перейти к следующим.

5. Выведите во фрейме содержание памяти машины с использованием экземпляра класса `JTextArea`. Поскольку содержание памяти слишком велико, чтобы оно могло целиком поместиться во фрейме, не забудьте проверить, чтобы в `JTextArea` имелись полосы прокрутки.

   Выполните эту часть после завершения части 3.

6. *** После того как память выведена на экран во фрейме, добавьте какую-либо визуальную индикацию того, которая из машинных команд в памяти является текущей. Сделайте это за счет выделения строки в области, где выведено на экран содержимое памяти, изменив цвет фона. Вы можете посмотреть документацию Java API для более подробной информации о том, что можно сделать с помощью класса `JTextArea`. Будет полезным также посмотреть класс `JTextComponent`, который имеет методы `selectAll()`, `cut()`, `setCaretPosition()` и `moveCaretPosition()`. Эти методы — прекрасная отправная точка для нахождения информации о том, как выделить строку.

   Вы можете выполнить эту часть после завершения части 5. При желании вы можете пропустить эту часть и перейти к следующим.

7. *** Сделайте память редактируемой во фрейме, так чтобы по мере выполнения программы, если код машинного языка в памяти изменяется, выполнялась бы именно измененная версия. Это не означает внесения каких-либо изменений в изначальный исходный файл — просто изменения в текущее состояние машины. Это открывает возможность использования области памяти в качестве редактора и введения программ либо за счет написания их там с нуля, либо за счет копирования и вставки их через буфер обмена из других источников. Вы можете посмотреть документацию Java API для более подробной информации о том, как использовать `JTextArea`.

   Вы можете выполнить эту часть после завершения части 5. При желании, вы можете пропустить эту часть и перейти к следующим.

8. Добавьте к меню опции сохранить (**Save**) и загрузить (**Load**), так чтобы во время выполнения программы машинного языка, между щелчками мышью по кнопке, существовала возможность сохранения состояния машины в файле и перезагрузки его позднее для продолжения выполнения программы. Если программа достаточно объектно-ориентирована, то свойство сериализации должно поддерживать эту возможность.

   Вы можете выполнить эту часть после завершения части 5. При желании, вы можете пропустить эту часть и перейти к следующим.

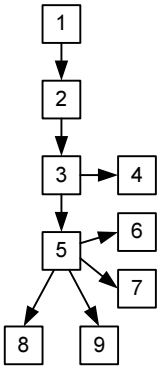
9. Добавьте новую, вторую кнопку, которая позволяет запустить программу от начала до конца без необходимости повторных щелчков по уже имеющейся кнопке. Преобразуйте это в поточное приложение. Должна быть возможность создать более одной копии машины, каждой в своем собственном фрейме. Должна существовать возможность загрузки и выполнения во множественных фреймах и просмотра, как копии MISC выполняют программу в своих отдельных фреймах одновременно. Если все совершается слишком быстро, так что у вас нет возможности запустить вторую копию до того, как первая останавливается, добавьте к коду механизм задержки.

   Вы можете выполнить эту часть после завершения части 5.

10. Нарисуйте схему UML для окончательной версии вашего кода, т. е. версии, с которой вы работаете. Обратите внимание, что эта схема будет зависеть от того, какое количество частей вы решили выполнить.

Схема зависимостей

Данная схема (рис. 22.1) показывает отношения зависимости между различными частями проекта. Любая часть программы должна быть выполнена только после того, как выполнены те части, которые находятся перед ней по прямой линии. 10-я часть зависит от того, какое количество частей вы решили использовать.



10 10-я часть зависит от количества предыдущих частей в реализации

Рис. 22.1

ПРИЛОЖЕНИЕ

Описание компакт-диска

Для каждой из 22 глав книги на диске имеется соответствующая папка. Например, папка, соответствующая *главе 7*, называется Unit7. Рассмотрим на примере *главы 7*, что может находиться в папке. Каждая папка содержит один или более разделов (подпапок):

- задания к главе 7 — unit7 assignment files:
  - упражнения к главе 7 — unit7assignment.doc;
  - ключи к упражнениям — unit7key.doc;
- программы к заданиям в главе 7 — unit7 assignment programs;
- программы-решения к заданиям — unit7 assignment solution programs;
- примеры программ к главе 7 — unit7 example programs.

Файл с упражнениями содержит вопросы или задачи по программированию для данной главы. Файл с ключами содержит ответы. В нем также могут быть ответы на короткие задачи по программированию, в виде текста, а не программного кода. Для некоторых заданий по программированию требуются определенные классы или программы, или часть программы может быть дана в качестве отправной точки задания; весь этот материал находится в разделе программ к заданиям. Кроме того, для тех заданий на программирование, которые требуют объемных ответов, полный исходный код для решений можно найти в подпапке, содержащей

программы-решения. Наконец, раздел с примерами программ содержит файлы кодов для примеров, рассматриваемых в главах.

Не для всех глав приведены все папки и файлы. Это сделано сознательно. Так, например, ключи даны не ко всем упражнениям, не все задания содержат сложные проблемы на программирование, для которых требуются коды, и т. д. Для того чтобы было совершенно ясно, что помещено на компакт-диске и чего на нем нет, в табл. П1 приводится список, указывающий содержание каждой папки, соответствующей той или иной главе. В *главе 15* акцент в изложении материала переносится с синтаксиса на программирование, и еще раз переносится на описание итогового проекта в *главах 21* и *22*. Пояснения к списку указывают, как эти изменения сказываются на содержании папок и файлов.

Таблица П1. *Содержимое компакт-диска*

Папки и файл	**Присутствует на диске**
❒ unit1 assignment files:	Нет
• unit1assignment.doc;	Нет
• unit1key.doc;	Нет
❒ unit1 assignment programs;	Нет
❒ unit1 assignment solution programs;	Нет
❒ unit1 example programs.	Да
❒ unit2 assignment files:	Да
• unit2assignment.doc;	Да
• unit2key.doc;	Да
❒ unit2 assignment programs;	Да
❒ unit2 assignment solution programs;	Да
❒ unit2 example programs.	Да
❒ unit3 assignment files:	Да
• unit3assignment.doc;	Да
• unit3key.doc;	Да
❒ unit3 assignment programs;	Нет
❒ unit3 assignment solution programs;	Да
❒ unit3 example programs.	Да

Таблица П1 *(продолжение)*

Папки и файл	**Присутствует на диске**
❒ unit4 assignment files:	Да
• unit4assignment.doc;	Да
• unit4key.doc;	Да
❒ unit4 assignment programs;	Да
❒ unit4 assignment solution programs;	Да
❒ unit4 example programs.	Да
❒ unit5 assignment files:	Да
• unit5assignment.doc;	Да
• unit5key.doc;	Да
❒ unit5 assignment programs;	Нет
❒ unit5 assignment solution programs;	Да
❒ unit5 example programs.	Да
❒ unit6 assignment files:	Да
• unit6assignment.doc;	Да
• unit6key.doc;	Да
❒ unit6 assignment programs;	Да
❒ unit6 assignment solution programs;	Да
❒ unit6 example programs.	Да
❒ unit7 assignment files:	Да
• unit7assignment.doc;	Да
• unit7key.doc;	Да
❒ unit7 assignment programs;	Да
❒ unit7 assignment solution programs;	Да
❒ unit7 example programs.	Да
❒ unit8 assignment files:	Да
• unit8assignment.doc;	Да
• unit8key.doc;	Да
❒ unit8 assignment programs;	Да
❒ unit8 assignment solution programs;	Да
❒ unit8 example programs.	Да

Таблица П1 *(продолжение)*

Папки и файл	Присутствует на диске
❒ unit9 assignment files:	Да
• unit9assignment.doc;	Да
• unit9key.doc;	Да
❒ unit9 assignment programs;	Да
❒ unit9 assignment solution programs;	Да
❒ unit9 example programs.	Да
❒ unit10 assignment files:	Да
• unit10assignment.doc;	Да
• unit10key.doc;	Да
❒ unit10 assignment programs;	Да
❒ unit10 assignment solution programs;	Да
❒ unit10 example programs.	Да
❒ unit11 assignment files:	Да
• unit11assignment.doc;	Да
• unit11key.doc;	Да
❒ unit11 assignment programs;	Да
❒ unit11 assignment solution programs;	Да
❒ unit11 example programs.	Да
❒ unit12 assignment files:	Да
• unit12assignment.doc;	Да
• unit12key.doc;	Да
❒ unit12 assignment programs;	Да
❒ unit12 assignment solution programs;	Да
❒ unit12 example programs.	Да
❒ unit13 assignment files:	Да
• unit13assignment.doc;	Да
• unit13key.doc;	Да
❒ unit13 assignment programs;	Да
❒ unit13 assignment solution programs;	Да
❒ unit13 example programs.	Да

Таблица П1 *(продолжение)*

Папки и файл	**Присутствует на диске**
❒ unit14 assignment files:	Да
• unit14assignment.doc;	Да
• unit14key.doc;	Да
❒ unit14 assignment programs;	Да
❒ unit14 assignment solution programs;	Да
❒ unit14 example programs.	Да
❒ unit15 assignment files:	Да
• unit15assignment.doc;	Да
• unit15key.doc;	Нет
❒ unit15 assignment programs;	Да
❒ unit15 assignment solution programs;	Нет
❒ unit15 example programs.	Нет
❒ unit16 assignment files:	Да
• unit16assignment.doc;	Да
• unit16key.doc;	Да
❒ unit16 assignment programs;	Нет
❒ unit16 assignment solution programs;	Нет
❒ unit16 example programs.	Нет
❒ unit17 assignment files:	Да
• unit17assignment.doc;	Да
• unit17key.doc;	Нет
❒ unit17 assignment programs;	Нет
❒ unit17 assignment solution programs;	Нет
❒ unit17 example programs.	Да
❒ unit18 assignment files:	Да
• unit18assignment.doc;	Да
• unit18key.doc;	Нет
❒ unit18 assignment programs;	Да
❒ unit18 assignment solution programs;	Нет
❒ unit18 example programs.	Да

Таблица П1 *(окончание)*

Папки и файл	Присутствует на диске
❒ unit19 assignment files:	Да
• unit19assignment.doc;	Да
• unit19key.doc;	Нет
❒ unit19 assignment programs;	Да
❒ unit19 assignment solution programs;	Нет
❒ unit19 example programs.	Да
❒ unit20 assignment files:	Да
• unit20assignment.doc;	Да
• unit20key.doc;	Нет
❒ unit20 assignment programs;	Нет
❒ unit20 assignment solution programs;	Нет
❒ unit20 example programs.	Да
❒ unit21 assignment files:	Нет
• unit21assignment.doc;	Нет
• unit21key.doc;	Нет
❒ unit21 assignment programs;	Нет
❒ unit21 assignment solution programs;	Нет
❒ unit21 example programs.	Да
❒ unit22 assignment files:	Нет
• unit22assignment.doc;	Нет
• unit22key.doc;	Нет
❒ unit22 assignment programs;	Нет
❒ unit22 assignment solution programs;	Нет
❒ unit22 example programs.	Да

Главу 15 можно рассматривать как последнюю главу первой логической части книги или как первую главу второй логической части. Задание к главе представляет собой одну большую задачу на программирование. Программы, приведенные в файле задания, должны помочь начинающему программисту, но здесь не

даны ни программа-решение, ни ключ к заданию. Эта же схема повторяется в папках к *главам 17—22*. К некоторым главам могут быть даны программы для заданий, в других могут быть примеры программ, но ни к одной из них нет программ-решений.

Глава 16 вводит иную тему, и задание к ней, как и ранее, представляет собой вопросы, к которым даются ключи с ответами.

В *главе 21* излагается материал, который подготавливает читателя к итоговому проекту, поэтому к этой главе нет задания.

В *главе 22* описывается итоговый проект по программированию. Заданием является реализовать этот проект. Само задание является неотъемлемой частью объяснений, и оно изложено в последнем разделе главы, а не в отдельном файле с заданием.

Предметный указатель

C

Callback 317
Cardinality 304
Concurrency control 409

M

Mancala 292
Minimal Instruction Set Computer (MISC) 412

N

Navigability 304

R

Relative record number (RRN) 219

T

TextPad 22
Thread 408
Togiz Kumalak 295
Token 202

W

Wari 290
Whitespace 202

А

Автоупаковка 193
Апплет 12, 82, 367
Ассоциация 304

Б

Байт-код 11
Блок прослушивания 329
◊ событий 316

В

Ввод/вывод в файл 199

Д

Динамическая диспетчеризация 263
Документирование кода 97
Доступность 304

И

Иерархия наследования 84, 307
Инкапсуляция 63
Интерфейс 188, 269
◊ Cloneable 279
◊ Collections 188

Интерфейс *(прод.)*
◊ Iterable 191
◊ Map 188
◊ Runnable 411
◊ Serializable 381
Исключение 207
◊ необрабатываемое 207
◊ обрабатываемое 207
Итерация 191

К

Класс 15, 29
◊ ArrayList 189, 193
◊ Color 90
◊ Double 44, 59
◊ FileWriter 210
◊ FocusListener 402
◊ Font 93
◊ Integer 59
◊ IOException 208
◊ JApplet 87
◊ JFrame 315
◊ JPanel 315
◊ KeyBoardFocusManager 402
◊ Math 52, 160
◊ MyTerminalIO 60, 199
◊ Object 237, 279
◊ Point 37, 83
◊ PrintWriter 217
◊ PropertyChangeListener 402
◊ public 29
◊ RandomAccessFile 218
◊ Scanner 201
◊ String 55, 115
◊ System 179, 199
◊ Thread 411
◊ адаптер 341
◊ вложенный 309
◊ внутренний 85, 309, 329
◊ дочерний 237
◊ определение 29
◊ определенный пользователем 62
◊ подкласс 237
◊ потомок 237
◊ родительский 237
◊ суперкласс 237
Клонирование 74, 278
Ключевое слово:
◊ catch 208
◊ char 206
◊ false 111
◊ finally 209
◊ instanceof 254
◊ new 30, 38
◊ public 29
◊ static 29
◊ super 249
◊ this 69
◊ true 111
◊ try 208
Коллекция 187, 189
Комментарий 30
Компиляция 11
Компоновка 400
Компьютер с минимальным набором команд 412
Конкатенация 32, 57
Константа 53
◊ символьная 121
Конструктор 36, 63
Контейнер 400
Контроль за параллельностью 409
Копия:
◊ поверхностная 280
◊ полная 280

Л

Лексема 202

М

Массив 173
◊ двумерный 176
◊ копирование 178
Метод 16
◊ clone() 279
◊ close() 210
◊ compareTo() 117
◊ equals() 115, 240, 276
◊ equalsIgnoreCase() 117
◊ main() 29
◊ paint() 87
◊ parseInt() 59
◊ print() 217
◊ printf() 200
◊ println() 31, 217
◊ read() 212
◊ toString() 59, 273
◊ write() 210
◊ абстрактный 259
◊ доступа 157
◊ модифицирующий 157
◊ перегрузка 245
◊ переопределение 245
◊ синтаксис вызова 39, 54, 56
◊ статический 53, 160
Множественность 304

Н

Наследование 238
◊ множественное 269

О

Обработка исключений 207
Обратный вызов 317
Объект 16
Объектная ссылка 252
Объектно-ориентированное программирование (ООП) 15
Оператор:
◊ арифметический 49
◊ логический 108
◊ модульный 51
◊ присваивания 48
◊ условного перехода if 102
▫ вложенный 107
Относительный номер записи 219
Отношение наследования 197
Ошибка 40, 45, 51, 54, 122
◊ бесконечная рекурсия 231
◊ бесконечный цикл 231
◊ повторного объявления переменной 128

П

Память 393
Параметр:
◊ скрытый 153
◊ фактический 148
◊ формальный 147, 168
◊ явный 153
Перегрузка 58, 245
Переменная 16, 45
◊ инициализация 47
◊ локальная 166, 167
◊ область действия 128
◊ ошибка повторного объявления 128
◊ правила именования 45
◊ статическая 53, 162, 169
◊ экземпляра 16, 167
Побочный эффект 158
Полиморфизм 263
Поток 408
◊ параллельный 408
Приведение типа 48

Р

Регистр 390

Рекурсия 222
Рисование 317
◊ геометрических форм 89

С

Сборка мусора 73
Сериализация 381
Символ:
◊ " 33
◊ % 51
◊ * 36
◊ + 32
◊ = 48
◊ пробельный 202
◊ точка 33, 39
Синхронизация 409
Ссылка:
◊ на null 75
◊ на объект 38
◊ объектная 72
Стандартный поток:
◊ ввода 201
◊ вывода 199
Строка 30, 55
◊ пустая 59
Схема UML 301

Т

Текст 93
Тип данных 44
◊ byte 220
◊ char 206
◊ double 44
◊ long 220
◊ булев 111

Ф

Фокус 397

Х

Хэш-код 274, 276

Ц

Цвет 90
Цикл 123
◊ do 126
◊ for 126
◊ for each 190
◊ while 123
◊ бесконечный 125
◊ вложенный 135