МИХАИЛ БОРИСОВ

Скрипты в InDesign

руководство для умных дизайнеров и ленивых верстальщиков

Основы объектной модели

Эффективная работа с текстом, изображениями, таблицами, фреймами и другими объектами

Проверка публикаций перед цветоделением

Автоматизация спуска полос

Практические примеры для оптимизации верстки

МАСТЕР +CD

Михаил Борисов

Скрипты в InDesign

руководство для умных дизайнеров и ленивых верстальщиков

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.06
ББК 32.973.26-018.2
Б82

Борисов М. А.

Б82 Скрипты в InDesign: руководство для умных дизайнеров и ленивых верстальщиков. — СПб.: БХВ-Петербург, 2008. — 368 с.: ил. + CD-ROM — (Мастер)

ISBN 978-5-9775-0202-3

Рассмотрены вопросы создания скриптов в InDesign (CS2/CS3) для автоматизации работы дизайнера и верстальщика. В основу книги положено подробное описание реальных скриптов, разработанных автором в процессе многолетней практики. Приведена объектная модель InDesign. Показано создание окон диалога, управление документом и взаимодействие с мастер-страницами. Рассмотрены особенности использования скриптов при работе с текстовыми фреймами, форматировании текста и таблиц, использовании стилей, работе с изображениями. Уделено внимание практическим вопросам проверки корректности публикации перед выводом на цветоделение, автоматизации спуска полос. Показано взаимодействие с другими редакторами, входящими в состав пакета Creative Suite. В приложении приведены краткие справочники по JavaScript, VisualBasic и AppleScript. На компакт-диске располагаются примеры скриптов.

Для дизайнеров и верстальщиков

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

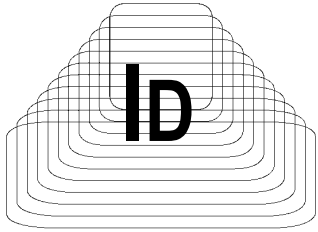
Лицензия ИД № 02429 от 24.07.00. Подписано в печать 24.12.07.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 29,67.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 978-5-9775-0202-3

Оглавление

ГЛАВА 1



Общие сведения о скриптинге

Любой человек в своей работе старается стать профессионалом. Повышение уровня мастерства может идти разными путями, но обязательно через овладение новыми знаниями и умениями, которые помогают решать весь спектр стоящих перед ним задач: как творческих, так и более тривиальных производственных. Все это в полной мере относится к работе за компьютером: современное программное обеспечение (ПО) предоставляет различные способы решения задач проще, быстрее, качественнее, надежнее, с каждой новой версией обрастая все новыми возможностями. Однако, несмотря на свою мощь, охватить необъятное невозможно — фактически разработчики ПО создают инструменты для решения лишь общих, наиболее востребованных операций. В результате любые специфические функции, необходимые в вашем производственном процессе, придется решать через набор типовых инструментов, что, как правило, выливается в непроизводительные затраты сил и времени.

Типичный пример из области предпечатной подготовки: автоматическая расстановка колонтитулов в препресс-пакетах не предусмотрена, ручная же простановка — дело хлопотное и малотворческое. Или же верстка по всем типографским правилам, которая тоже требует существенных трат времени. В то же время перекладывание на плечи машины подобных трудоемких операций, требующих повышенного внимания, позволит не только повысить эффективность работы, оставляя время для творческих задач, но и убережет от пресловутого "человеческого фактора", неизбежного при большом объеме рутины.

Для автоматизации рутинных операций в любом ПО, претендующем на роль профессионального, разработчиками, как правило, предусматриваются несколько способов.

Макросы (Actions) в зависимости от контекста использования обладают большей или меньшей функциональностью. В Photoshop под ними подразумеваются наборы команд, имитирующих нажатие клавиш и считывание зна-

чений из диалоговых окон, что является простейшим вариантом автоматизации. Достоинство всего одно — простота реализации: для создания макроса не требуется никакая специальная подготовка.

В других приложениях на макросы возлагается гораздо большая функциональность — типичным примером могут быть макросы, поддерживаемые офисным ПО, в которых реализованы возможности языка Visual Basic for Applications (в том числе использование модулей других программ через механизм ActiveX).

На другом полюсе — *плагин-модули* (plug-ins), которые создаются независимыми разработчиками. Они имеют наиболее полный доступ ко всем ресурсам приложений, используют функции оптимизации кода, распределения памяти, а также позволяют ограничивать свое незаконное распространение. По функциональности могут сравниться с самой хост-программой. Их написание требует специальной подготовки и хороших навыков программирования на языках высокого уровня (C++ и др.).

Наибольшую популярность в среде препресс-подготовки получили *скрипты*. Несмотря на их естественные ограничения в сравнении с плагинами, они прекрасно подходят для решения большинства задач, встающих перед пользователями. С одной стороны, по своей функциональности они находятся посредине между двумя полюсами, с другой — их написание не требует серьезных познаний в программировании. При желании скриптинг можно освоить за несколько месяцев без отрыва от основной работы (при условии, что раньше вы совершенно ничего не писали). Если же вы уже занимались программированием (например, в рамках Web-проектов), то задача значительно упрощается, поскольку фактически половину требуемых навыков вы уже имеете: по сложности освоения оба механизма — логика выполнения операций и доступ к элементам публикации — сравнимы.

Первый механизм предполагает знание базовых инструкций — операций сравнения, циклов, ветвлений; второй — четкое знание объектной модели приложения. По сравнению со скриптингом в Photoshop и Illustrator механизм, заложенный в InDesign, развит в наибольшей степени, давая возможность, например, создавать сложные пользовательские диалоговые окна, что, в свою очередь, позволяет наращивать функциональность скрипта.

Такое внимание разработчиков к программированию в InDesign вполне объяснимо: среди всех пакетов Adobe наибольший эффект от применения скриптов достигается именно в нем — ведь диапазон задач, потенциально поддающихся автоматизации при верстке публикаций, просто огромен. Особенно ощутима польза при обработке однородной, заранее подготовленной каким-либо образом информации. Например, верстка справочников, телепрограмм, прайс-листов, разнообразной финансовой, технической документации вообще немыслима без скриптов.

Скорость выполнения скриптов сравнительно высока, что для большинства препресс-задач вполне достаточно.

С выпуском программного комплекса Creative Suite (в частности, с появлением Adobe Bridge) сфера применения скриптов значительно расширилась, что наиболее ярко проявилось в среде межпрограммного взаимодействия (в рамках Creative Suite). Так, например, если в векторные макеты, помещенные в публикацию, были встроены растровые изображения, то через скриптинг можно подключить к обработке Photoshop. В этом случае происходит соединение InDesign с Illustrator и, в случае необходимости, дальнейшее переключение на Photoshop с последующим возвратом по цепочке назад. Фактически механизмы, заложенные в Bridge, на свой манер повторяют существующие в Visual Basic for Applications — мощном механизме межпрограммного взаимодействия под Windows.

Необходимо отметить значительные изменения, произошедшие в недавно вышедшем Creative Suite 3, которые коснулись в том числе и скриптинга, благодаря чему можно с уверенностью говорить, что, начиная с этой версии пакета, автоматизация обрела гибкость, необходимую для решения задач практически любой сложности.

Что касается платформы Macintosh, то скрипты на AppleScript способны реализовать широчайший набор команд, эквивалентный существующим в Visual Basic for Applications, и являются аналогичным инструментом автоматизации производственных процессов на системном уровне.

Отчетливо понимая, что подавляющая часть пользователей InDesign ранее не сталкивались с программированием, поспешу развеять возможные сомнения по поводу того, что творческий склад ума и программирование — вещи несовместимые. Дело в том, что для программирования на уровне скриптов (подчеркиваю — на уровне скриптов) совершенно не обязательно иметь особый склад мышления — вполне достаточно хотеть этому научиться и, конечно же, иметь определенный запас времени. Сам автор книги в прошлом был в подобной ситуации, пребывая в твердой уверенности, что программирование — не его призвание. Тем не менее, скажу по своему опыту: даже не имея никакого представления о программировании, освоить скриптинг под InDesign — вполне посильная задача. Написание скриптов значительно проще настоящих программ, для которых используются языки высокого уровня, где требуется специальная подготовка и опыт.

Цель книги — помочь в изучении скриптинга, поскольку справочное руководство от Adobe представляет собой 700-страничный фолиант (и это только для версии CS2!), в котором очень кратко перечислены базовые сведения о методах и свойствах объектов. Фактически содержание справочного руководства — одна большая таблица, в которой поиск даже штатными средствами

занимает ощутимое время. Поэтому в предлагаемой книге, во-первых, дается общая информация по написанию скриптов, во-вторых, объясняются взаимосвязи основных элементов публикации между собой, и, наконец, даны примеры использования наиболее часто употребляемых в публикации объектов и др.

Еще одно ценное качество книги, которое, безусловно, по достоинству оценят пользователи, — ее можно использовать как библиотеку уже готовых решений. В ней приведено множество полезных скриптов, большую часть которых можно без каких-либо (минимальных) переделок использовать в своей повседневной работе. И хотя скрипты редко бывают универсальными, тем не менее большинство приведенных в книге примеров пригодится многим, поскольку являются плодом работы автора, который профессионально занимается версткой и для облегчения своей ежедневной работы создал их для себя. Если же по какой-то причине создание скрипта затруднительно — можно связаться с автором для решения конкретной задачи, хотя все же надеюсь, что изложенный в данной книге материал будет достаточным для самостоятельного решения большинства стоящих перед вами задач.

Поскольку книга писалась в то время, когда Creative Suite 3 только-только появился, у автора не было возможности детально ознакомиться со всеми новшествами скриптинга в InDesign CS3. Основные изменения в новой версии, касающиеся скриптинга, коснулись, во-первых, интерфейсных возможностей, а также более тесного взаимодействия различных скриптов между собой в рамках выполнения глобальных задач на уровне организации замкнутых производственных процессов (например, упростилось взаимодействие с другими редакторами пакета Creative Suite). Это позволяет создавать на основе скриптов законченные коммерческие решения, что для подавляющего большинства пользователей InDesign с учетом времени, требуемого на их изучение, и вдобавок к тому достаточно узкой специализации, не принципиально. В то же время те возможности InDesign Creative Suite 3, которые представляют непосредственный интерес с точки зрения верстки, в данной книге отражение нашли.

1.1. Выбор языка

Скриптинг представляет собой процесс написания управляющих команд под определенное приложение. Их можно разделить на две группы: управляющие и исполняющие.

Исполняющие — это команды, ограниченные исключительно рабочей средой приложения (в нашем случае — InDesign): перейти на страницу, передвинуть объект, отформатировать абзац, вставить текст и т. п.

Кроме них нужен механизм, который бы позволял в зависимости от сложившейся ситуации направлять работу в требуемое русло (если..., то...), выполнять математические операции, проводить всякого рода проверки и т. п. Таким образом, нужна некая база, которая будет выполнять исключительно управляющие функции (при этом сами команды типа перехода на конкретную страницу или перемещения объекта будут играть лишь исполняющую роль). На роль такого управляющего отлично подходят три кандидата: Visual Basic, AppleScript и JavaScript. Каждый из них имеет обширный и достаточно удобный набор функций для того, чтобы выполнить любую специфическую задачу.

Выбор того или другого языка диктуется несколькими соображениями:

- на платформе Macintosh существует только AppleScript;
- для Windows выбор несколько шире: предлагаются Visual Basic и JavaScript.

Каждый из них имеет свои преимущества и недостатки.

Visual Basic — творение Microsoft, а потому имеет широчайший набор методов. Недостаток — совершенно не поддерживается на Macintosh. Этого недостатка лишен язык JavaScript. Он кроссплатформенный, т. е. будет работать в любой установленной системе. Его синтаксис отличается от Visual Basic, однако в силу того, что многие пользователи InDesign в той или иной степени сталкивались с Web-проектами, а потому уже хоть немного знакомы с языком.

В отличие от JavaScript, Visual Basic предоставляет гораздо более широкие возможности по автоматизации рабочих процессов, позволяя через ActiveX-компоненты обращаться к любым приложениям, зарегистрированным в системе — например, подключиться к Word, Excel, Access и т. п.

Visual Basic и AppleScript являются "полноценными" языками программирования, позволяя решать задачи системного уровня. Этого никак нельзя сказать о JavaScript, поскольку он ориентирован исключительно на использование возможностей той среды, в которой сценарии исполняются (в нашем случае — InDesign, который предоставляет JavaScript доступ к своим объектам, позволяя управлять их поведением).

Несмотря на определенные отличия между языками, способ их взаимодействия с InDesign совершено идентичен.

Объем функциональности JavaScript (текущая версия 1.5) определен в стандарте ECMA 262. Не утомляя читателей глубокими сведениями о нем, замечу лишь, что в нем продумано все, что требуется для полноценной и относительно комфортной работы. В данном случае речь идет лишь о возможности реализации тех или иных действий, без оценки эффективности инструмента-

рия. Интересующимся могу порекомендовать ознакомиться с более продвинутыми спецификациями JavaScript 1.6 и 1.7, поддерживаемыми известным браузером FireFox.

Ядро JavaScript 1.5 состоит из небольшой группы фундаментальных объектов, среди которых — строки (Strings), массивы (Array), пользовательские функции (function) и математические функции (Math), управляющие структуры и операторы и др. Каждый объект имеет свои свойства и методы, которые и реализуют всю функциональность языка.

В целях безопасности в стандарт не включены некоторые механизмы — например, работа с файловой системой (создание, открытие, перемещение, удаление файлов и папок), запуск других программ и т. п., что хоть в какой-то мере служит сдерживанию распространения вирусов и всякого рода malware через интернет-браузеры. Соответственно, каждый разработчик ПО самостоятельно реализовывает недостающие компоненты в нужном объеме, исходя из принципа необходимой достаточности — естественно, вопросы обеспечения безопасности в таком случае также полностью возлагаются на него. Исходя из потребностей специалистов предпечатной подготовки, Adobe расширила определенные в стандарте средства JavaScript инструментами для доступа к файловой системе (редакция известна как ExtendScript).

Учитывая значительную распространенность JavaScript и в то же время стремясь расширить сферу применения скриптов, Adobe поступила достаточно мудро: она позволила скриптам, работающим в своих приложениях, вызывать другие скрипты, причем они могут быть написаны на разных (поддерживаемых) языках. Это позволяет, с одной стороны, обойти ограничения языка, а с другой — использовать уже имеющуюся библиотеку скриптов, написанных на привычном языке. Например, JavaScript может вызывать блок, написанный на Visual Basic и пользоваться всеми преимуществами такого распределения ролей.

Главным критерием при выборе языка программирования является его конечная нацеленность: если предполагается использование скрипта в сочетании с другими приложениями (разработанными не Adobe), то единственным вариантом будет либо Visual Basic (Windows), либо AppleScript (Macintosh). Подключение программ не из пакета Creative Suite при предпечатной подготовке — явление крайне редкое, поэтому данное ограничение JavaScript для рассматриваемых в данной книге задач значения не имеет. Более того, при необходимости можно делать вставки на Visual Basic либо AppleScript, что вообще нивелирует отличия.

Если сравнить функциональность скриптинга в InDesign с QuarkXPress, то необходимо отметить, что, во-первых, в QuarkXPress реализована поддержка исключительно AppleScript, поэтому программирование для него возможно

лишь на платформе Macintosh. Причину такого состояния дел, по всей видимости, следует искать в традиционной ориентации препресс-процессов на данную платформу. Во-вторых, разработка Adobe гораздо более завершенная и зрелая — это касается не только функциональности, но и качества реализации (больше ошибок, недочетов разработчиков).

Что скрипты могут? С их помощью можно выполнять любые операции, доступные через меню и палитры программы. Вы можете создавать новые документы, страницы, текстовые фреймы, форматировать текст, вставлять графику, отправлять на печать и экспортировать содержимое.

Новое в InDesign Creative Suite 3

В Creative Suite 3 сфера применения скриптов значительно расширилась и теперь с их помощью можно реализовывать те функции, которые через программный интерфейс не доступны.

Среди возможностей:

- создание компилированных скриптов (с расширением jsxbin), что дает возможность защитить свой скрипт от несанкционированного копирования, а также повысить его быстродействие;
- создание собственных меню, в том числе контекстно-зависимых;
- поддержка событий, позволяющая выполнять те или иные действия при наступлении определенных условий (например, сразу после открытия документа);
- назначение объектам скрипта, что дает возможность возложить на них функции автоматического отслеживания изменений в документе и выполнения соответствующих операций (например, изменение своих размеров и т. п.);
- возможность сохранения значений переменных после выполнения скрипта с предоставлением их для использования другим скриптам;
- отображение процента выполнения задания, что полезно при выполнении объемных публикаций или задействовании других программ из пакета Creative Suite.

В Creative Suite 3 Adobe пошла дальше и расширила возможности Bridge (позволяя, например, подключение к FTP-серверам или передачу данных по HTTP-протоколу и т. п.), усовершенствовала взаимодействие с другими приложениями Adobe, а также реализовала возможность подключения внешних модулей, написанных на C++, что открывает поистине безграничные возможности для полной автоматизации ряда производственных процессов.

В InDesign 3 значительно расширилась база для применения скриптов — в первую очередь за счет возможности запуска скрипта из другого скрипта, причем на любом из поддерживаемых языков: на платформе Windows — JavaScript и Visual Basic, на Macintosh — AppleScript и JavaScript.

Подобное взаимодействие используется в серьезных проектах по автоматизации рабочего процесса, например, для обеспечения связи с другими компонентами издательского комплекса, например с MS Word. Другой пример — получение выборок из Access, что необходимо для взаимодействия с БД (JavaScript не позволяет этого напрямую).

Еще одна причина — возможность привязки скрипта к любому объекту. Как известно, каждый объект в публикации имеет свойство `label` (ярлык), в котором может храниться любой текст. В Creative Suite 3 развили данный вопрос, и теперь, если в `label` записан текст скрипта, программа может его исполнять автоматически. Такой механизм позволяет достичь еще более глубокой степени автоматизации верстального процесса, поскольку объекты сами могут отслеживать изменение ситуации и выполнять заложенные в них действия.

Чего скрипты не могут? Им закрыт доступ к трем типам операций:

- изменение цветовой модели документа;
- доступ к содержимому системного буфера (это ограничение в некоторых случаях можно обойти);
- установка параметров рабочего окружения.

Также скриптинг не поддерживает создание пользовательских типов объектов (просто новые объекты — без проблем), а также реализацию глубинных механизмов — например, собственного модуля, выполняющего композицию текста. Для таких случаев предусмотрен более серьезный инструментарий (Software Development Kit), который позволяет создавать плагин-модули с использованием C++.

1.2. Инструментарий

В зависимости от используемой среды вам понадобится различный инструментарий. Для создания скриптов для Macintosh потребуются интерпретаторы JavaScript или AppleScript версий 1.6 и выше, а также собственно редактор AppleScript (оба идут в стандартной поставке с Mac OS). Те, кому мало функциональности стандартного редактора, могут попробовать более продвинутый Script Debugger разработки Late Night Software (**http://www.latenightsw.com**).

Как уже говорилось, для написания пользовательских сценариев для продуктов Adobe в среде Windows можно использовать JavaScript (не путайте с JScript — Microsoft-версией языка, она не поддерживается), либо продукты семейства Microsoft Visual Basic — например, VBScript, Visual Basic 5, Visual Basic 6, Visual Basic .NET, Visual Basic 2005 Express Edition. При этом следует учитывать, что, начиная с Visual Basic .NET, функциональность скриптов ниже, поскольку в .NET не поддерживается тип данных `Variant`, широко используемый в InDesign.

Несмотря на поддержку Visual Basic, в установочный пакет InDesign его интерпретатор не включен, поскольку он идет с офисными приложениями пакета MS Office (в виде Microsoft Visual Basic for Applications (VBA)), достаточно лишь при инсталляции включить соответствующую опцию.

Для корректной работы с Visual Basic необходимо, чтобы InDesign устанавливался пользователем с правами администратора. С запуском скриптов проблем не возникнет, но вот добавление новых доступно лишь членам групп Administrator (Администратор) либо Power Users (Опытные пользователи).

В отличие от VBA, поддержка JavaScript заложена в дистрибутив. Она включает в себя все возможности JavaScript 1.5 и соответствует нынешнему стандарту ECMA 262. Определенные в этом стандарте функции расширены операциями с файлами и папками.

1.2.1. AppleScript

Для просмотра свойств и методов, доступных в AppleScript:

1. Загрузите InDesign.
2. Загрузите Apple Script Editor.
3. В Apple Script Editor выберите **File | Open Dictionary**.
4. Из списка приложений выберите InDesign. Apple Script Editor отобразит список всех коллекций объектов.
5. Выберите конкретную коллекцию для просмотра доступных в ней методов и свойств.

1.2.2. JavaScript

Для просмотра свойств и методов, доступных в JavaScript:

1. Загрузите ExtendScript Toolkit.
2. В ExtendScript Toolkit выберите **Help | InDesign CS2 Main Dictionary**.
3. Из списка классов выберите интересующий вас объект.
4. Для детальной информации по любому методу или свойству выберите их, после чего в окнах **Properties** и **Methods** появится их описание (рис. 1.1).

Писать скрипт можно в любом текстовом редакторе, но для того чтобы пользоваться отладчиком (ошибки будут всегда, особенно на стадии изучения, а отладчик позволяет запустить скрипт пошагово и, таким образом, достаточно быстро локализовать проблему), файлу нужно дать расширение jsx (с ним в Creative Suite 2 ассоциирован редактор ExtendScript Editor, устанавливающийся с любым ПО от Adobe). Если используется Creative Suite первой версии, то расширение js на jsx менять не нужно. Добавление в скрипт строчки `$.level=1` даст возможность отладчику остановиться в указанном вами месте (его помечают через `$.bp()`). К слову сказать, отладчик в новой версии более "продвинутый", пользоваться им гораздо удобнее, чем в предыдущей версии, поэтому настоятельно рекомендую использовать именно Creative Suite 2.

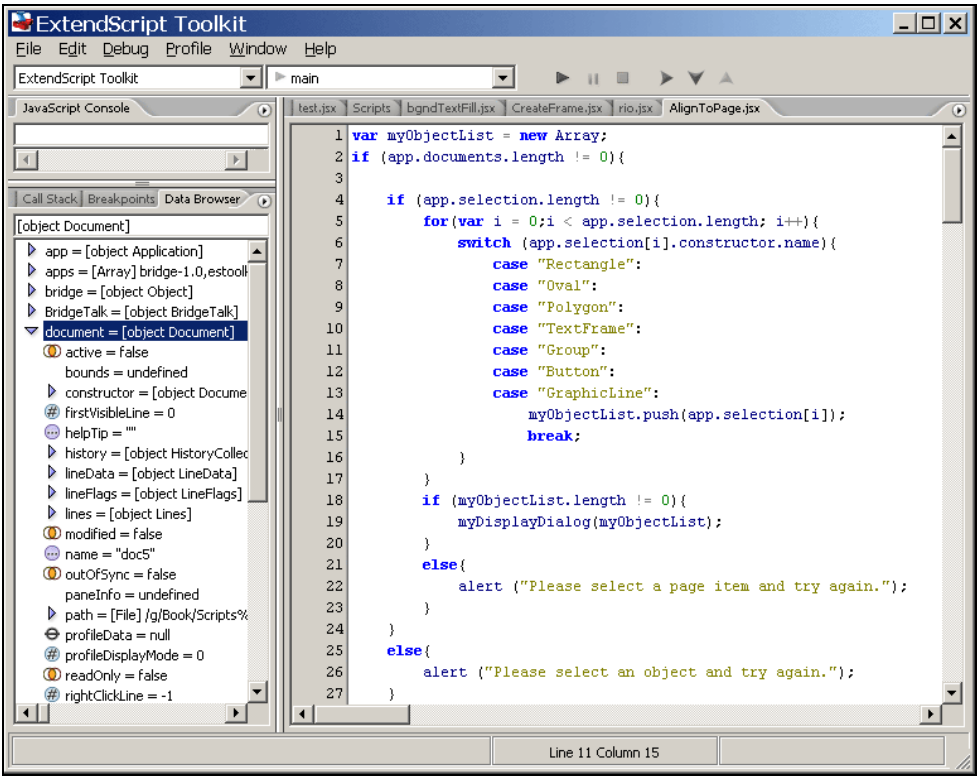


Рис. 1.1. Интерфейс ExtendScript

При активном тестировании может возникнуть ситуация, когда скрипт начинает "глючить", т. е. выдавать ошибку там, где ее на самом деле нет. Как правило, это возникает в том случае, когда скрипт повторно применяется к одному и тому же документу. При этом память "засоряется" историей ваших ошибок, поэтому желательно тестовый файл перед повторным запуском скрипта закрыть (в этом случае память очищается) и вновь открыть. Конечно, при интенсивном тестировании это раздражает, а определить, что скрипт работает уже некорректно, невозможно, поэтому возьмите за правило заново открывать файл через несколько запусков скрипта. Если же скрипт вообще начинает "спотыкаться на ровном месте", перезагрузите отладчик и, желательно, хост-приложение.

Для удобства в левом окне отладчика (панель **Data Browser**) отображаются текущие значения всех используемых переменных, что позволяет следить за ними без каких-либо дополнительных действий. Если вам потребуется узнать какое-то конкретное значение, можно вписать необходимую строчку в верхнем окне панели **JavaScript Console**.

И напоследок. Перед запуском скрипта в отладчике проверьте, какое хост-приложение выбрано в качестве рабочего (панель **Target Application**). По умолчанию отладчик настроен на ExtendScript Toolkit, поэтому будьте бдительны.

1.2.3. VBA

Для просмотра свойств и методов, доступных в Visual Basic for Applications (для запуска редактора откройте Word, перейдите на вкладку Макросы (в Office 2007), введите имя макроса (должно начинаться с буквы) и потом нажмите кнопку **Создать**. Загрузится Microsoft Visual Basic.

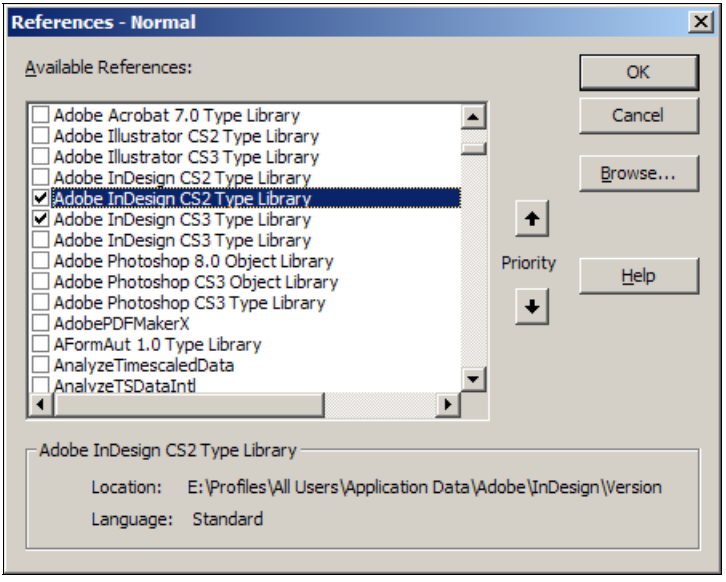


Рис. 1.2. Подключение необходимых библиотек в Visual Basic

1. Выберите **Tools | References**. В списке найдите **Adobe InDesign *xxx* Type Library** (здесь *xxx* — либо CS2, либо CS3), поставьте флажок напротив и нажмите кнопку **OK** (рис. 1.2). Если по каким-либо причинам библиотека (`ResourcesforVisualBasic.myTable`) в списке не появилась, нажмите кнопку **Browse** и вручную укажите ее месторасположение. Обычно она находится в папке ~:\Documents and settings*user_name*\Application Data\Adobe\InDesign\Version ...\Scripting Support\.
2. Выберите **View | Object Browser**.
3. В списке **Libraries** выберите InDesign. Visual Basic отобразит список всех объектов InDesign (рис. 1.3).

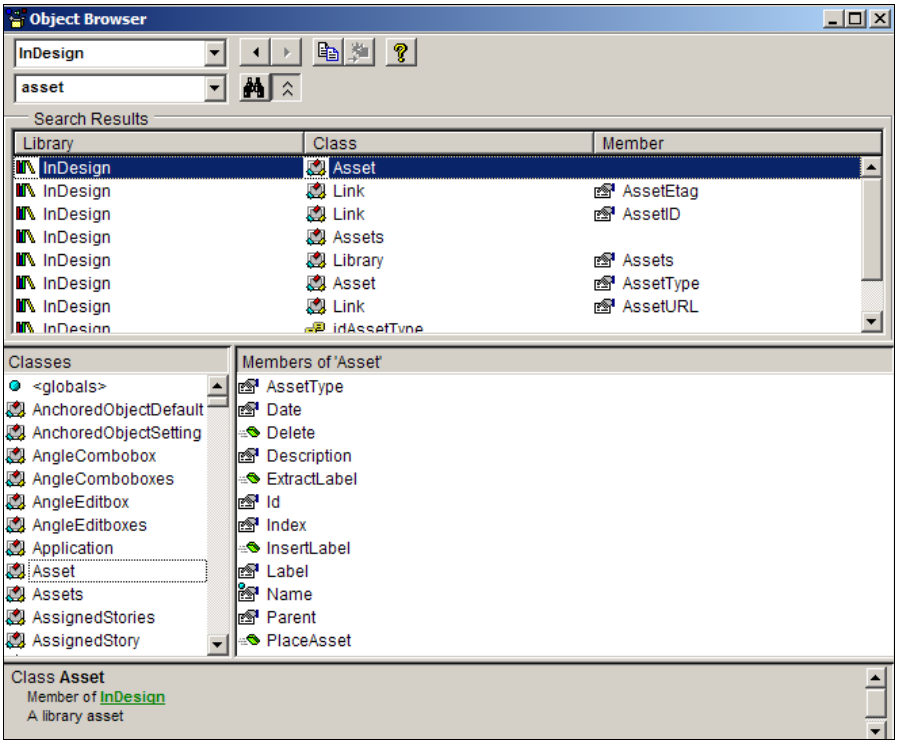


Рис. 1.3. Очень удобный интерфейс Visual Basic, с описанием поддерживаемых методов и свойств

4. Выберите имя любого объекта. В соответствующих окнах будут отображены свойства и методы данного объекта.

1.3. Дом для скрипта

Для того чтобы скрипт был "виден", его размещают в папке Install\Adobe\InDesign\Presets\Scripts, где Install — папка, в которой установлены приложения (обычно ~\Program Files\, здесь ~ — системный диск), либо ~Documents and Settings*user_name*\Application Data\Adobe\InDesign\Scripts, а для Macintosh: ~/Library/Preferences/Adobe InDesign/Scripts.

При необходимости в эти папки можно помещать не сами скрипты, а только ярлыки на них.

Новое в InDesign Creative Suite 3

В InDesign CS3 изменился путь для расположения пользовательских скриптов. Теперь они должны были размещены в папке пользователя drive_name:\Profiles*user_name*\Application Data\Adobe\InDesign\Version 5.0\Scripts\Scripts Panel.

> Если необходимо, чтобы некоторые скрипты исполнялись в момент запуска редактора, их помещают в папку Install\Adobe\Adobe InDesign CS3\Scripts.

В отличие от других продуктов Adobe, InDesign (любой версии) в запущенном состоянии автоматически сканирует папки, и как только появится новый скрипт, он сразу же, без перезагрузки редактора, становится доступным для использования.

В InDesign предусмотрена специальная палитра **Scripts** (**Window | Automation | Scripts**), откуда можно запускать пользовательские скрипты (рис. 1.4). Это очень удобно, особенно при интенсивном использовании скриптов, поскольку отпадает необходимость каждый раз путешествовать по меню в поисках нужной операции.

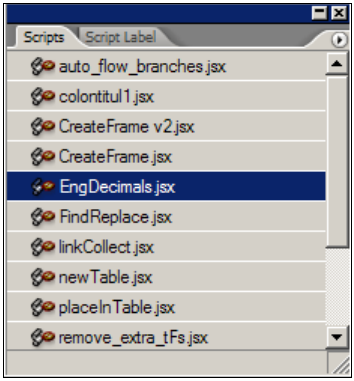


Рис. 1.4. Палитра **Scripts**

Скрипты, запущенные через палитру **Scripts**, работают быстрее, чем запущенные через Проводник (Finder).

В панели могут быть запущены скрипты с расширениями spt, as, applescript (для AppleScript), js и jsx (JavaScript), vbs (VBScript). Кроме того, скрипты на JavaScript могут быть компилированными (jsxbin).

Открыть скрипт можно несколькими способами: в любом редакторе, либо, что проще, нажав клавишу <Alt> (<Option> — в MacOS) и дважды щелкнув на его названии в палитре **Scripts**. При этом он откроется в ExtendScript Editor (если имеет расширение jsx), если расширение — js, то в редакторе, ассоциированном с данным типом файлов.

Если нужно открыть папку, в которой находится скрипт, достаточно нажать комбинацию клавиш <Ctrl>+<Shift> (<Command> — в MacOS) и также дважды щелкнуть на названии скрипта в палитре. Альтернативный вариант — использовать команду **Reveal in Explorer** (**Reveal in Finder** — в MacOS) из меню палитры **Scripts**.

По умолчанию в Creative Suite 2 скрипты выполняются как набор последовательных действий, что очень помогает на стадии отладки, поскольку позволяет отслеживать изменения после каждого шага. Обратная сторона медали — для отмены действия скрипта придется нажимать комбинацию клавиш <Ctrl>+<Z> множество раз, что в случае значительных изменений становится практически нереальной задачей.

> ***Отличие версий***
>
> В Creative Suite 3 ситуация изменилась: теперь скрипт рассматривается как единая команда, и для отмены ее действия достаточно однократного нажатия комбинации клавиш <Ctrl>+<Z>. Чтобы иметь возможность пошагового отслеживания, нужно активизировать опцию **Undo Affects Entire Script** из меню панели **Scripts**. Выполнение скрипта как единой команды ускоряет процесс его исполнения, поскольку не тратятся дополнительные ресурсы на создание точек восстановления.
>
> Кроме запуска скриптов непосредственно из палитры **Scripts**, в Creative Suite 3 предусмотрено использование клавиатурных эквивалентов. Для назначения скрипту эквивалента используйте **Edit | Keyboard Shortcuts**, выберите набор ускорителей из меню **Set**, после чего перейдите на **Product Area | Scripts**. В новом окне выберите необходимый скрипт и присвойте еще неиспользованное клавиатурное сокращение.

Чтобы при отладке в ExtendScript автоматически запускалась объектная модель именно для InDesign, первая строка в скрипте должна содержать указание на хост-приложение:

```
#target = "InDesign-4.0"
```

Цифры в названии указывают номер версии InDesign (табл. 1.1).

Таблица 1.1. *Соответствие идентификаторов номерам версий*

Номер версии	Идентификатор
InDesign Creative Suite	3
InDesign Creative Suite 2	4
InDesign Creative Suite 3	5

Отсутствие явного указания на номер версии, например,

```
#target = "InDesign"
```

приводит к использованию самой последней установленной на компьютере версии продукта.

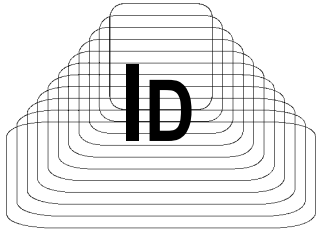
При переходе на InDesign CS3 помните, что из-за изменений в его объектной модели по сравнению с предыдущей версией некоторые ранее работающие скрипты перестают нормально функционировать. Выхода два:

- переписать код под CS3;
- вставить в начале скрипта строку

```
app.scriptingVersion = 4.0
```

  Она принудительно активизирует исполнительный механизм от предыдущей версии вместо самой последней установленной на компьютере.

ГЛАВА 2



Общие сведения об объектной модели

Все современные приложения (и InDesign в том числе) используют объектно-ориентированную модель для доступа к воздействуемым объектам. Что это значит?

Представьте себе дом, в котором есть двери, окна, комнаты — это все как раз и есть разные объекты, принадлежащие данному дому. Каждый сложный объект состоит из более мелких, которые являются зависимыми по отношению к своему родителю. Так, например, светильник имеет несколько лампочек, которые, в свою очередь, состоят из нити накаливания, цоколя, колбы и т. д.

Одни объекты (например, двери) могут быть открыты или закрыты, иметь цвет, размеры, отличаться фурнитурой, другие (например, светильник) — открытыми или закрытыми быть не могут, зато могут быть включенными; это свойства объектов. Наконец, одни объекты можно только открывать и закрывать, другие — только включать, а третьи — открывать, закрывать и включать. Это методы, которыми воздействуют на объекты.

Если провести аналогию с языком, то *объекты* можно рассматривать как существительные, *методы* — как глаголы, а *свойства* — как прилагательные. Складывая из них предложения, мы получаем текст, который понятен собеседнику, точно также набор команд в скрипте будет понятен InDesign.

Воздействовать на объекты можно несколькими способами. Например, дать команду "Открыть двери" — при этом все двери дома будут открыты. В то же время команда "Открыть дверь" останется без исполнения — ведь совершенно не ясно, о какой именно двери идет речь. Второй вариант — дать команду "Открыть дверь в гостиной", она четко указывает, какую дверь выбрать (при условии, что в гостиной только одна дверь). Точно так же команда "Открыть двери в гостиной" не будет исполнена, если в ней только одна дверь. Таким

образом, давая ту или иную команду нужно четко представлять объект воздействия и понимать логику предстоящих действий.

Кроме того, нельзя сбрасывать со счетов фактор скорости. Например, неэффективной будет команда "Открыть двери" для того, чтобы открыть лишь конкретную дверь в конкретной комнате. Это особенно проявляется при больших объемах обработки, например работе с таблицами: число ячеек растет в геометрической прогрессии от размеров таблицы и форматирование каждой в отдельности даже на мощной машине займет значительное время.

В то же время, подходить к скриптингу нужно творчески. Например, операция "Открыть двери" займет меньше времени, чем поочередное выполнение команд "Открыть дверь в спальне", "Открыть дверь в детской" и т. д. для всех комнат дома.

2.1. Коллекции

Для удобства работы объекты с одинаковыми свойствами объединены в *коллекции* (или *классы*). Так, двери — это одна коллекция, обладающая своими уникальными свойствами, окна — другая и т. д. В InDesign каждый объект, будь то текстовый фрейм, символ текста, абзац или цвет, принадлежит своей коллекции (текстовым фреймам, символам, абзацам, образцам цвета соответственно), причем сама коллекция также является объектом, с собственными свойствами и методами. Использование коллекций очень удобно с практической точки зрения: они позволяют, например, вместо команды "Открыть в объекте № 123 объект номер 12345" указать "Открыть четвертую дверь на втором этаже", что более понятно.

Любая коллекция объектов InDesign является по своему внутреннему строению массивом JavaScript. К каждому элементу массива можно обращаться несколькими способами. Самый распространенный — по порядковому номеру (или *индексу*), задавая его в квадратных скобках. Счет ведется с 0: `array[0]` — самый первый элемент, `array[array.length-1]` — последний.

Кроме использования порядковой *(абсолютной)* нумерации, предусмотрена *относительная*, при которой можно полностью отказаться от указания конкретного индекса объекта, достаточно сказать: "Открыть предыдущую дверь" или "Закрыть следующую за предыдущей", что во многих случаях оказывается даже более эффективно. Естественно, при этом предполагается, что базовый объект должен быть в любом случае определен. Используется, как правило, только в тех случаях, когда получение индекса по каким-либо причинам затруднено. Наиболее яркий пример — работа на уровне абзацев (они — понятие абстрактное, вычисляются исключительно по признаку конца строки), далее об этом пойдет речь подробнее.

Еще один вариант обращения к объекту — по имени (`array["My First"]`), которое по понятным причинам должно быть уникальным. Второй способ используется достаточно редко, т. к. предварительно придется задать имя объекта, что не всегда оправдано, хотя в некоторых случаях является единственным выходом.

Имя коллекции всегда указывается во множественном числе (в конце — "s") с той целью, чтобы лишний раз подчеркнуть, что речь идет о множестве (массиве) объектов, а не о каком-то конкретном объекте, например, `paragraphs`. Указание порядкового номера объекта в коллекции дает ссылку на конкретный объект, который, в полном согласии с логикой, используется уже в единственном числе (`paragraph`) — именно так информацию, касающуюся абзацев, нужно искать в Руководстве по скриптингу.

Знание объектной модели InDesign — фундаментальное требование для написания скриптов, без нее можно уподобиться мартышке из известной басни, у которой было множество очков, но она не знала, что с ними делать. Точно так же, не зная причинно-следственных связей, вы не сможете сделать и шага, поскольку InDesign будет вежливо, но настойчиво информировать, что у него не получается включить дверь или выключить окно.

Поскольку объектное дерево InDesign очень развесистое, на рис. 2.1 дано лишь самое общее представление о нем. Толстая рамка означает коллекцию объектов, тонкая — конкретный представитель коллекции (для краткости показан лишь наиболее часто используемый объект — `textFrame`). Далее, в соответствующих главах будут подробнее рассматриваться коллекции, наиболее востребованные в скриптинге. Разобравшись с ними, вам не составит никакого труда использовать остальные объекты.

Рисунок не отображает взаимосвязей на более низких уровнях — например, таблица может содержать вложенную таблицу, группа иметь подгруппы, текстовый фрейм может включать заякоренный фрейм, в котором, в свою очередь, может быть текст, таблица, изображение — глубина вложенности не ограничена.

На вершине иерархической лестницы, как и следовало ожидать, расположен объект `Application` — это сам InDesign. Да, не удивляйтесь — коль скоро мы используем объектную модель, само приложение также становится объектом. `Application` (будем использовать более краткий вариант — `app`) является родителем для всех остальных объектов: открытых в нем документов, стилей, цветов, диалогов, книг, библиотек, а также массы различных предустановок (`Preferences`, `Options`, `Settings`, `Presets`, `Defaults`). Большинство объектов-предустановок не могут быть изменены, а другие, такие как `changePreferences` (объект, в котором собраны все установки по поиску/замене), открыты для пользователя.

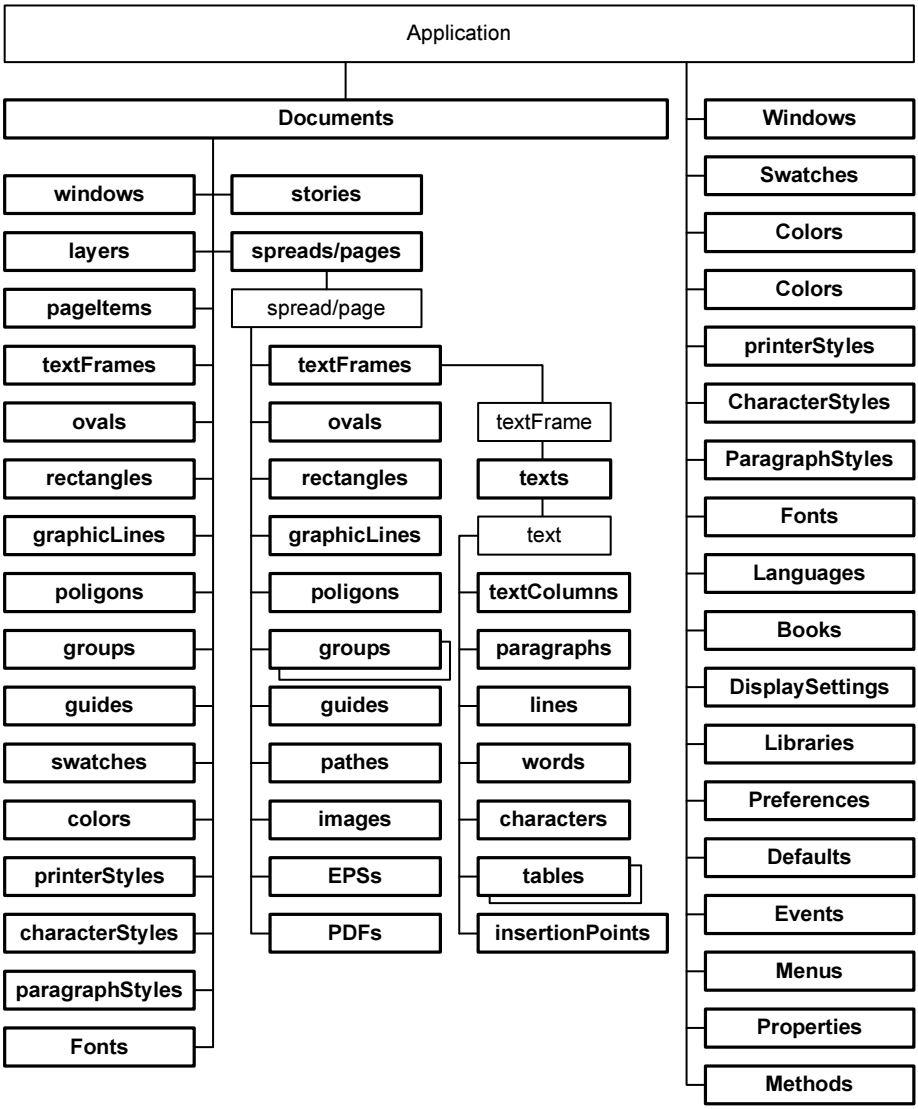


Рис. 2.1. Объектная модель InDesign

Объекты, которые чаще всего используются, — публикация (`Document`), материал (`Story`, цепочка связанных между собой текстовых блоков), поскольку он является непосредственным родителем для текстового содержимого (`words`, `characters`), а также текстовый блок (`textFrame`).

Как уже ранее отмечалось, объект, над которым будет производиться действие, нужно указывать однозначно. Однозначность достигается перечислени-

ем всей вышестоящей иерархии воздействуемого объекта: например, название документа, номер слоя, номер текстовой цепочки, номер страницы и т. д. вплоть до самого объекта.

Для того чтобы спуститься по иерархии до конкретного объекта, перечисляют всю вышестоящую цепочку объектов, разделяя каждый уровень точкой (.). Например, чтобы получить содержимое четвертого фрейма на второй странице открытого документа, нужно использовать такую конструкцию:

```
application.documents[0].pages[1].textFrames[3].contents
```

Скриптинг дает широкие возможности, однако не забывайте, что вам не все дозволено — например, порядок перечисления элементов не менее важен, и все попытки изменения своей объектной модели InDesign будет жестко пресекать.

Коллекция документов — набор всех открытых на данный момент публикаций. Соответственно к текущей публикации можно обратиться через `app.documents[0]`, предусмотрено и более удобное название — `app.activeDocument`. Материалов, как правило, в публикации несколько, поэтому доступ к ним возможен через `app.activeDocument.stories[index]`, и т. д.

> ***Новое в InDesign Creative Suite 3***
>
> В связи со значительными изменениями, произошедшими в Creative Suite, писать скрипты стало гораздо проще. Одно из полезных нововведений — встроенный Object Browser, который помогает ориентироваться во всем многообразии иерархических связей InDesign. В то же время, во всяком случае, на данный момент, его использование малоэффективно, а вот включение интерактивного InDesign Object Library — настоящая находка.

Объектная модель InDesign отображается в любом из приложений, которое выбрано для редактирования скриптов.

Для того чтобы постоянно не повторять всю вышестоящую иерархию и тем самым повысить читаемость кода, рекомендуется использовать сокращенный доступ к объектам:

```
myDocument = app.documents[0]
myPage = myDocument.pages[2]
```

Данной записью мы переменной `myDocument` присваиваем ссылку на объект `app.documents[0]`, а переменной `myPage` — ссылку на третью страницу.

Соответственно предыдущую запись

```
application.documents[0].pages[1].textFrames[3].contents
```

теперь можно переписать как

```
myPage.textFrames[3].contents
```

Попробуем использовать уже имеющиеся знания для решения первой практической задачи — определения номера текущей страницы. Если внимательно просмотреть объектную модель, в ней можно обнаружить объект `activeWindow`, среди свойств которого есть указатель на текущую страницу — `activePage`, у которого, в свою очередь, имеется свойство `name`, являющееся ее номером, что нам и нужно. Таким образом, можно записать:

```
my_Current_page = app.activeWindow.activePage.name
```

Как легко мы получили результат! Однако я выбрал этот пример не просто так — он служит хорошей иллюстрацией важности внимательного отношения к использованию тех или иных объектов. Давайте проанализируем, гарантирует ли наше решение 100%-ю достоверность. Предположим, в публикации установлена низкая степень увеличения. Соответственно, на экран выводится сразу несколько страниц — в таком случае активной страницей может быть любая, в зависимости от расположения линии их раздела на экране. Таким образом, универсальным предложенное решение признать нельзя — альтернативный способ будет рассматриваться в *разд. 5.1*.

Возьмите за правило — всегда проверять себя, поскольку часто подобные "подводные камни" не заметны, что может привести к различным недоразумениям.

2.2. Работаем с выделением

Как уже говорилось, коллекции объединяют объекты с одинаковыми свойствами, т. е. страницы — это одна коллекция, текстовые блоки — вторая, изображения — третья и т. д. А вот `Selection` (выделенные объекты) — объект уникальный. *Он не имеет какого-то определенного типа, поскольку определяется типом входящих в него объектов.* Если выбраны отдельные символы текста, `Selection` будет иметь тип `text`, если слово целиком — `word`, если абзац — `paragraph`, если текстовые блоки — `textFrames`, если графика — то `Rectangle` и т. д.

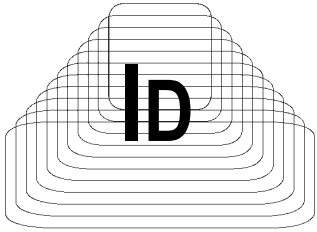
Несмотря на отсутствие в конце "s", объект `Selection` представляет собой массив, элементами которого являются все выделенные объекты, соответственно к конкретному выделенному объекту обращение происходит только через `selection[index]`, где `index` — порядковый номер объекта среди выделенных. В остальном `selection` ведет себя как любой другой объект. Например, если выделен текстовый фрейм, то к его текстовому содержимому обращаются через объект `text` (если к содержащейся внутри него графике — `graphics` и т. п.) Разумеется, нужно помнить об ограничениях самого InDesign — так, он не позволяет выделить несколько отдельных текстовых

фрагментов, поэтому обращение `selection[0].texts[1]` вызовет ошибку. При этом `selection[0].paragraphs[1]` даст ссылку на второй абзац в выделенном тексте, `selection[0].paragraphs[1].characters[9]` — на десятый символ в этом абзаце и т. д.

Положение курсора (иными словами точки вставки) в публикации — не что иное, как тот же `selection[0]`.

Обратите внимание на важную особенность: если нужно получить доступ, например, к выделенным текстовым фреймам, то в таком случае количество объектов в массиве `selection` равно количеству выделенных фреймов, соответственно, для доступа к ним используют `selection[0]`, `selection[1]`, `selection[2]` и т. д., а не `selection[0].textFrames[0]`, `selection[0].textFrames[1]`, как можно было бы предположить. То же самое справедливо и в отношении остальных объектов — будь то ячейка таблицы или элемент графики.

ГЛАВА 3



Диалоговые окна

Диалоги активно используются в скриптинге. Одни являются обязательными элементами, обеспечивающими взаимодействие с пользователем, в других случаях — просто информируют о произошедших событиях (например, показывают инструкции перед запуском, предупреждения об ошибках или сообщают о завершении скрипта).

В процессе разработки скриптов нужно учитывать то обстоятельство, что при возникновении определенных условий InDesign сам выдает диалоги — в основном, это касается предупреждений — например, потеряны связи, и т. п. Если вы полностью автоматизируете процесс и необходимость ответа на возможные вопросы со стороны InDesign нужно исключить, отключите вывод диалоговых окон, что задается через свойство приложения `userInteractionLevel` (листинг 3.1).

Листинг 3.1. Управление интерактивностью

```
// Запрет на отображение диалоговых окон
app.userInteractionLevel = userInteractionLevels.neverInteract

// Разрешение диалоговых окон (значение по умолчанию)
app.userInteractionLevel = userInteractionLevels.interactWithAll
```

В зависимости от решаемых задач диалоги можно разделить на два основных типа: базовые и расширенные.

Сфера применения базовых — в основном, отладка скрипта и, в меньшей мере, использование как элемента интерфейса. Не требуют специальных знаний (поскольку используются стандартные JavaScript-конструкции), просты, компакты, легко читаются и в то же время достаточно функциональны.

Наиболее целесообразно использовать их на начальной стадии работы со скриптами, а также при задании максимум до 4—5 опций, необходимых для работы скрипта.

Более развитые целесообразнее всего использовать в скриптах, требующих от пользователя ввода множества опций, значений, также они незаменимы при обращении к стилям, цветам, другим объектам, которые используются в документе — в таком случае организовать работу по-другому просто невозможно.

К базовым можно отнести три метода: `alert()`, `prompt()`, `confirm()`.

3.1. Базовые методы

3.1.1. *alert()*

Синтаксис:

`alert([`*`sMessage`*`])`

Здесь ***`sMessage`*** — текстовая строка (наличие впереди буквы "s" говорит о том, что тип переменной — строка (`String`)). Разрешается использование специальных символов (`\n` и `\t`, *см. приложение 2*).

Наиболее простой вариант используется только для отображения на экране сообщения. Отображает модальное (требующее реакции пользователя) окно, содержащее кнопку **OK**. Используется для простейшей отладки скриптов, т. к. позволяет вывести содержимое переменной.

3.1.2. *confirm()*

Синтаксис:

`confirm([`*`sMessage`*`])`

Данное окно аналогично окну `alert()`, за исключением того, что оно содержит две кнопки — **OK** и **Cancel**. Позволяет получить от пользователя подтверждение или отказ от каких-либо действий. Возвращает значение `true`, если нажата кнопка **OK**, и `false` в случае нажатия кнопки **Cancel**.

Пример использования метода `confirm()` представлен в листинге 3.2.

Листинг 3.2. Использование метода `confirm()`

```
if (confirm("Пример окна Confirm")) {
     alert("Ok"); }
else {
     alert("Cancel");
}
```

3.1.3. *prompt ()*

Синтаксис:

prompt([*sMessage*] [, *sDefaultValue*])

Это окно позволяет получить от пользователя данные в виде строки (второй параметр) и является ограниченной альтернативой расширенным методам создания интерфейса. При открытии окна выводится значение, заданное по умолчанию (***sDefaultValue***). При необходимости передать больше чем одно значение их разделяют любым разделителем, который потом используют для вычленения значений.

Пример использования метода `prompt()` представлен в листинге 3.3.

Листинг 3.3. Использование метода `prompt()`

```
var getValues =  prompt("Укажите ширину текстового фрейма,\na также его
высоту: ", "25, 25");
var sValues = getValues.split(", ");

// Значения в массиве разделены
var myWidth = sValues[0];
var myHeight = sValues[1];
```

Основные действия при использовании `prompt()` — разбор строки, введенной пользователем (поскольку строка представляет собой массив символов, отделенных друг от друга разделителем), и дальнейшее использование полученных значений.

3.2. Расширенные методы

Использование развитых интерактивных средств общения с пользователями предоставляет гораздо большую гибкость, поскольку позволяет создавать скрипты любой степени сложности.

Пользовательские диалоги с точки зрения объектной модели — типичные объекты InDesign (`dialogs`), которые имеют свои характеристики (свойства) и возможности (методы). Для создания нового окна служит метод `add()`, который является универсальным для добавления любых объектов. Круглые скобки в названии любого метода обязательны: это, во-первых, отличительный признак метода от свойства, и, во-вторых, в них часто уточняется способ действия (передаются параметры). Если же никаких уточняющих параметров нет, скобки остаются пустыми.

По большому счету, метод — эта обычная функция, которой в качестве параметров как раз и передается содержимое в круглых скобках.

JavaScript позволяет в момент создания нового объекта задать "на лету" его свойства, которые заключаются в круглые скобки, при этом знак присвоения (=) меняется на двоеточие.

```
myObjects.add({firstProperty: firstPropertyValue, secondProperty:
secondPropertyValue ...})
```

Содержимое окна рассматривается как одна большая таблица, состоящая из строк (создаются методом `dialogColumns.add()`), которые, в свою очередь, состоят из столбцов (метод `dialogRows.add()`). Каждую образующуюся таким образом ячейку можно рассматривать как новую таблицу, что позволяет создавать диалоговые окна неограниченной сложности.

В громоздких окнах связанные элементы можно объединять в отдельные группы (`borderPanels.add()`), которые отображаются как рельефные области.

Каждый элемент управления в окне диалога — отдельный объект. Все они делятся на типы, перечисленные в табл. 3.1.

Таблица 3.1. *Типы элементов управления в диалоговом окне*

Тип	Назначение	Название
Статический текст	Вывод названия элемента управления	`Text EditBox`

Числовое поле	Задание значения	`Real EditBox`
		`Integer EditBox`
		`Measurement EditBox`
		`Percent EditBox`
		`Angle EditBox`

Раскрывающийся список	Вывод доступных опций	`dropdowns`
Элемент, объединяющий статический текст с раскрывающимся списком	Вывод доступных опций и ввод значения, отсутствующего в списке	`comboboxControls`
Переключатель	Обычно такие элементы организуются в группу, в которой может быть выбран только один переключатель	`radiobuttonControls`
Флажок	Установка или сброс опции	`checkbox Controls`
Управляющие кнопки (к ним доступ пользователю закрыт, переназначать их действие нельзя)	Передача данных скрипту или прекращение его выполнения	`OK`, `Cancel`

Каждый объект диалогового окна имеет методы и свойства, отражающие его специфику. Например, объект `checkbox` имеет свойство, в котором хранится текст названия элемента управления (`staticLabel`) и отдельно — состояние флажка (`checkedState`); содержимое раскрывающегося списка имеет свойство `selectedIndex`, которое показывает, какой элемент списка будет по умолчанию активным, и т. д. (кто занимался дизайном Web-страниц, найдет много общего с HTML-формами).

Соответствие элементов диалогового окна показано на рис. 3.1.

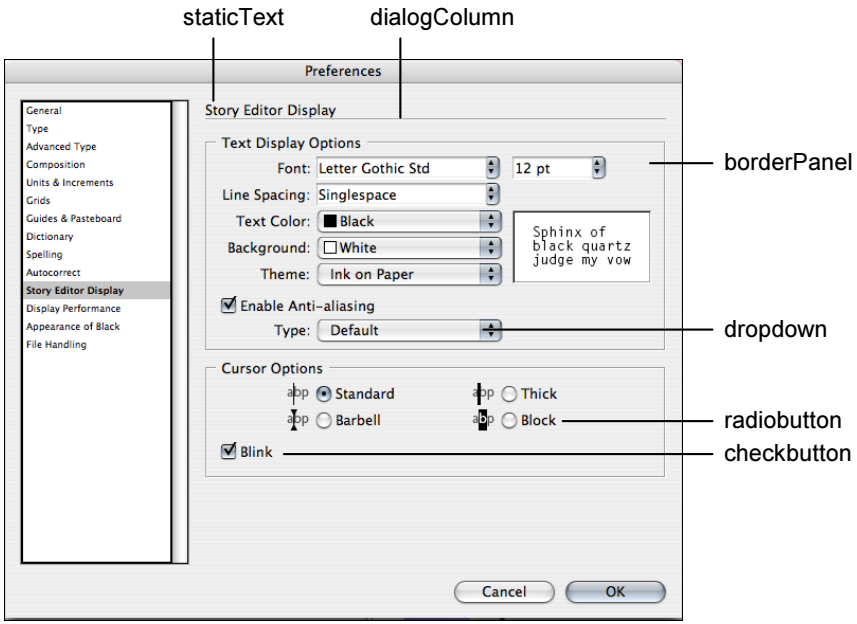


Рис. 3.1. Элементы управления диалогового окна

Принцип работы с диалоговыми окнами следующий:

1. Создание контейнера для диалога.
2. Заполнение его необходимыми элементами.
3. Вывод на экран (методом `show()`).
4. Считывание установленных в нем значений для использования в скрипте.
5. Удаление диалога из памяти, чтобы он не занимал лишних ресурсов (методом `destroy()`).

Как будет выглядеть задание высоты и ширины тестового фрейма через диалог, показано в листинге 3.4.

Листинг 3.4. Создание простого диалогового окна

```
// Создаем контейнер для всех элементов диалога
var myDialog = dialogs.add();

// Добавляем первый и единственный ряд таблицы
with (myDialog.dialogColumns.add()) {

  // Начинаем добавлять элементы управления.
  // Выводим текст названия элемента управления
  staticTexts.add({staticLabel: "Введите ширину текстового фрейма:"});

  // Создаем элемент для ввода первого значения
  var myWidth = realEditBoxes.add({editValue: 1});

  // Еще один элемент управления
  staticTexts.add({staticLabel: "Введите высоту текстового фрейма:"});

  // Создаем элемент для ввода второго значения:
  var myHeight = realEditBoxes.add({editValue: 1});
}
// Выводим окно на экран
var myResult = myDialog.show();

// Считывание значений
myWidthValue = myDialog.myWidth
myHeightvalue = myDialog.myHeight

// Освобождение памяти от ставшего ненужным диалога
myDialog.destroy();
```

Пример более сложной формы (рис. 3.2), в котором составим окно из различных типов элементов управления, приведен в листинге[1] 3.5.

Листинг 3.5. Создание более сложного окна диалога

```
// Создаем диалоговое окно
var myDialog = app.dialogs.add({name:"User Interface Example Script", ↳
  canCancel:true});
```

[1] В этом и других листингах в конце некоторых строк кода встречается символ ↳. Он означает разрыв строки в книге. В коде при этом разрыва строки быть не должно.

```
with(myDialog)
{
  // Создаем колонку таблицы (контейнер для всех элементов)
  with(dialogColumns.add())
  {
    // Создаем объемную область
    with(borderPanels.add())
      {
        // Создаем колонку для надписи "Message"
        with(dialogColumns.add())
        {
          // Добавляем надпись "Message:"
          staticTexts.add({staticLabel:"Message:"});
          // Создаем колонку для надписи "Message"
        }
        with(dialogColumns.add())
        {
          // Создаем  поле для текста с текстом по умолчанию
          var myTextEditField = textEditboxes.add ↳
            ({editContents:"Hello World!", minWidth:150});
        }
      }
      // Создаем вторую объемную область
      with(borderPanels.add())
      {
        with(dialogColumns.add())
        {
          staticTexts.add({staticLabel:"Point Size:"});
        }
        with(dialogColumns.add())
        {
          // Создаем поле для ввода числа.
          // Специально для чисел предусмотрено значение editValue.
          var myPointSizeField = realEditboxes.add({editValue:72});
        }
      }
      // Поле для выравнивания по вертикали
      with(borderPanels.add())
      {
        // Новая колонка
        with(dialogColumns.add())
        {
          // Описательная часть
          staticTexts.add({staticLabel:"Vertical Justification:"});
```

```
      // Еще одна колонка
    }
    with(dialogColumns.add())
    {
      // Создаем раскрывающийся список, сразу же задаем
      // его содержимое (в виде массива).
      // По умолчанию активизирован самый первый элемент
      var myVerticalJustificationMenu = ↳
        dropdowns.add({stringList:["Top", "Center", "Bottom"], ↳
        selectedIndex:0});
    }
  }

  // Последняя область — для переключателей
  with(borderPanels.add())
  {
    with(dialogColumns.add())
    {
      staticTexts.add({staticLabel:"Paragraph Alignment:"});
    }
    with(dialogColumns.add())
    {
      var myRadioButtonGroup = radiobuttonGroups.add();
      with(myRadioButtonGroup)
      {
        // Полная аналогия с раскрывающимся списком,
        // только вместо selectedIndex используется checkedState
        var myLeftRadioButton = ↳
          radiobuttonControls.add({staticLabel:"Left", ↳
          checkedState:true});
        var myCenterRadioButton = ↳
          radiobuttonControls.add({staticLabel:"Center"});
        var myRightRadioButton = ↳
          radiobuttonControls.add({staticLabel:"Right"});
      }
    }
  }
}

// Выводим диалог с кнопками OK и Cancel на экран
var myDisplay = myDialog.show();
```

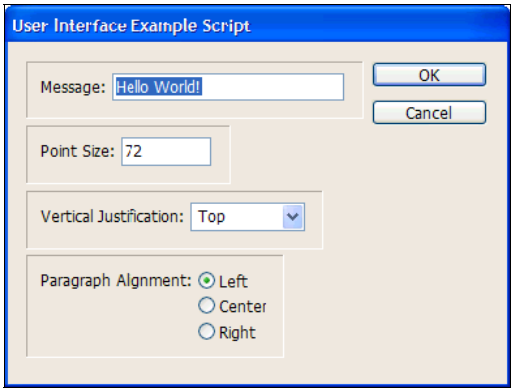


Рис. 3.2. Пример более сложной формы

После того как пользователь задал необходимые установки, их нужно из окна диалога получить обратно в скрипт (листинг 3.6).

Листинг 3.6. Считывание значений из окна диалога

```
// Если не нажата кнопка Cancel
if(myDisplay)
{
  var myParagraphAlignment, myString, myPointSize, ↳
    myVerticalJustification;

  // По очереди считываем значения из диалога:
  // первое поле — текст сообщения
  myString = myTextEditField.editContents

  // Второе поле
  myPointSize = myPointSizeField.editValue;

  // Обрабатываем раскрывающийся список
  switch(myVerticalJustificationMenu.selectedIndex)
  {
    with(VerticalJustification)
    {
      case "0": myVerticalJustification = topAlign
      case "1": myVerticalJustification = centerAlign
      case "2": myVerticalJustification = bottomAlign
    }
  }
}
```

```
// Обрабатываем переключатели
switch(myRadioButtonGroup.selectedButton)
{
  with(Justification)
  {
    case "0": myHorizontalJustification = leftAlign
    case "1": myHorizontalJustification = centerAlign
    case "2": myHorizontalJustification = rightAlign
  }
}
}
// Освобождаем память
myDialog.destroy();

// Выполняем необходимые действия на основе значений,
// полученных из окна диалога
function doSomething(){
...
}
```

Необходимо упомянуть об еще одном варианте создания диалогов — с его помощью получаются упрощенные по форме окна, однако их конструирование гораздо проще.

Вот, например, достаточно сложное окно — посмотрите, как оно строится (рис. 3.3).

А код, создающий такое окно, представлен в листинге 3.7.

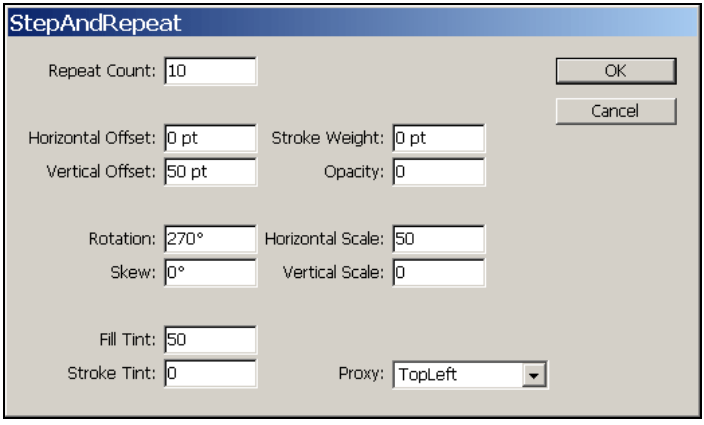


Рис. 3.3. Диалоговое окно **StepAndRepeat**

Листинг 3.7. Создание диалогового окна *StepAndRepeat*

```
myDialog = app.dialogs.add();
myDialog.name = "StepAndRepeat";

// Левая колонка
myLabelsColumn1 = myDialog.dialogColumns.add();
with(myLabelsColumn1){staticTexts.add({staticLabel:"Repeat Count:"});
  // Пробельный элемент.
  // Исключительно для отделения одной группы параметров от другой
  staticTexts.add();
  staticTexts.add({staticLabel:"Horizontal Offset:"});
  staticTexts.add({staticLabel:"Vertical Offset:"});

  // Снова пробельный элемент
  staticTexts.add();
  staticTexts.add({staticLabel:"Rotation:"});
  staticTexts.add({staticLabel:"Skew:"});

  // Пробельный элемент
  staticTexts.add();
  staticTexts.add({staticLabel:"Fill Tint:"});
  staticTexts.add({staticLabel:"Stroke Tint:"});
}
// Колонка для значений параметров из первой колонки
myControlsColumn1 = myDialog.dialogColumns.add();
with(myControlsColumn1){
  myRepeatField = integerEditboxes.add({editValue:2});

  // Пробельный элемент
  staticTexts.add();
  myXOffsetField = measurementEditboxes.add({editValue:0});
  myYOffsetField = measurementEditboxes.add({editValue:0});

  // Пробельный элемент
  staticTexts.add();
  myRotationField =angleEditboxes.add({editValue:0});
  mySkewField =angleEditboxes.add({editValue:0});

  // Пробельный элемент
  staticTexts.add();
  myFillTintField =realEditboxes.add({editValue:0});
  myStrokeTintField =realEditboxes.add({editValue:0});
}
```

```
// Очередная колонка для значений
myLabelsColumn2 = myDialog.dialogColumns.add();
with(myLabelsColumn2){
  staticTexts.add();
  staticTexts.add();
  staticTexts.add({staticLabel:"Stroke Weight:"});
  staticTexts.add({staticLabel:"Opacity:"});

  staticTexts.add();
  staticTexts.add({staticLabel:"Horizontal Scale:"});
  staticTexts.add({staticLabel:"Vertical Scale:"});

  staticTexts.add();
  staticTexts.add();
  staticTexts.add({staticLabel:"Proxy:"});
}
// Колонка со значениями
myControlsColumn2 = myDialog.dialogColumns.add();
with(myControlsColumn2){
  staticTexts.add();
  staticTexts.add();
  myStrokeWeightField = measurementEditboxes.add();
  myOpacityField = realEditboxes.add({editValue:0});

  staticTexts.add();
  myXScaleField = realEditboxes.add({editValue:0});
  myYScaleField = realEditboxes.add({editValue:0});
  staticTexts.add();
  staticTexts.add();
  myProxyMenu = .dropdowns.add({stringList:myProxyList, ↳
                  selectedIndex:0});
}
```

3.3. Создание разных языковых версий

В InDesign предусмотрен механизм для создания универсальных скриптов, в которых язык интерфейса пользователя меняется в зависимости от региональных настроек операционной системы. Специальная функция `localize()` в качестве аргумента принимает объект, содержащий необходимые языковые версии элемента интерфейса. Объектом является строка, в которой содержатся названия региональных установок в соответствии со стандартом ISO 3166

(их можно увидеть в окне выбора раскладки клавиатуры, только без прописных букв).

Например, для того чтобы создать кнопку, текст на которой будет меняться в зависимости от установленных региональных настроек, создаются три языковые версии: английская, немецкая и французская. Создается объект, имеющий свойства `en`, `de`, `fr`:

```
var btnText = { en: "Yes", de: "Ja", fr: "Oui" }
```

Сама кнопка — это типичное окно:

```
myButton = window.add("button", undefined, localize (btnText) )
```

Если включена автоматическая установка необходимой языковой версии:

```
$.localization = true
```

тогда достаточно написать так:

```
myButton = window.add("button", undefined, btnText)
```

Кроме того, предусмотрено использование переменных, что расширяет наши возможности. Переменные представляют собой ссылки на результат, возвращаемый функцией `localize()`, при этом `%1` дает ссылку на первое возвращаемое значение, `%2` — на второе и т. д. (листинг 3.8).

Листинг 3.8. Пример использования функции `localize()`

```
Today = {
  en: "Today is %1/%2",
  de: "Heute ist der %2.%1"
}
myDate = new Date();
window.alert(localize (today, d.getMonth()+1, d.getDate() ) );
```

В английском варианте получим:

```
Today is 6/1
```

В немецком:

```
Heute ist der 1.6
```

3.4. Новое в Creative Suite 3

InDesign CS3 значительно прибавил в возможностях по созданию элементов интерфейса благодаря включению компонента ScriptUI, который позволяет создавать пользовательские плавающие палитры, индикаторы выполнения и даже собственные контекстно-зависимые меню.

Принцип работы при создании индикатора выполнения следующий:

1. Создание нового окна.
2. Создание в окне элемента "полоса прогресса".
3. Вывод индикатора выполнения на экран.
4. Отображение результата.
5. При достижении 100% — сокрытие индикатора выполнения.

Код, реализующий эту последовательность действий, представлен в листинге 3.9.

Листинг 3.9. Создание индикатора выполнения работы при добавлении в документ 100 страниц

```
// Создание окна (панели)
myProgressPanel = new Window('window', 'Progress');

// Необходимые параметры
var myMaximumValue = 100;
var myProgressBarWidth = 100
var myProgressBarHeight = 20

// Шаг индикатора
var myStep = 1;

// Задание координат для окна индикатора (в массиве),
// начального и конечного значений
myProgressPanel.myProgressBar = add('progressbar', [12, 12,
myProgressBarWidth, myProgressBarHeight], 0, myMaximumValue);

// Вывод на экран
myProgressPanel.show();
for(var i = 0; i < 101; i ++){
  // Отображение процесса выполнения
  myProgressPanel.myProgressBar.value = i/ myStep;

  // Сам процесс, для которого выводится индикатор:
  // добавление страницы в цикле 100 раз
  app.documents[0].pages.add();
  if(i == 100){
    // Если выведены все 100 страниц
    myProgressPanel.hide();
  }
}
```

Такой подход оптимален при разовом выводе индикатора в процессе работы скрипта. Однако зачастую лучше разделить процесс на две фазы:

- ❒ инициализация окна с индикатором;
- ❒ вывод на экран, реализация динамики прогресса и по окончании — закрытие окна.

Такое разнесение процессов позволит единожды создать окно с индикатором состояния и потом, по мере необходимости, вызывать его для отображения степени выполнения скриптом различных операций. В таком случае реализация индикатора будет немного отличаться.

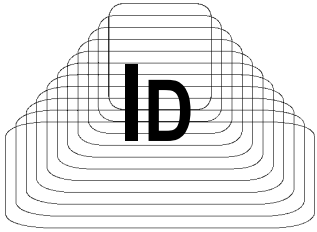
Сначала выполняем переход в специальный режим `session` (листинг 3.10), позволяющий сохранять в памяти значения переменных после окончания работы скрипта. Дело в том, что базовый движок ExtendScript (`main`) по окончании работы скрипта автоматически удаляет из памяти все использованные объекты, что позволяет бережно расходовать память. Специальный режим позволяет сохранить использованные объекты, а окончательное удаление объектов, задействованных в режиме `session`, из памяти происходит только после выхода из InDesign.

Листинг 3.10. Использование возможностей `session`

```
#targetengine "session"
var myStep = 1;

// Создаем функцию-конструктор индикатора с необходимыми параметрами.
// Ее мы будем вызывать в тех местах из основного скрипта, где необходимо
function myCreateProgressPAnel(myMaximumValue, myProgressBarWidth){
  myProgressPanel = new Window('window', 'Progress');
  // Задание координат окна
  myProgressPanel.myProgressBar = add('progressbar', [12, 12, ↳
    myProgressBarWidth, 20], 0, myMaximumValue);
}

// На этом часть, отвечающая за создание окна индикатора, закончена.
// А вот как вызывается функция из основного скрипта:
#targetengine "session"
myCreateProgressPAnel(100, 300)

// Вывод на экран
myProgressPanel.show();
```

```
// Остальное нам уже знакомо
for(var i = 0; i < 101; i ++){
  // Отображение процесса выполнения
  myProgressPanel.myProgressBar.value = i/myStep;
  app.documents[0].pages.add();
  if(i == 100)
  {
    // Если выведены все 100 страниц
    myProgressPanel.hide();
  }
}
```

ГЛАВА 4



Документы

Работа с документами (на уровне страниц, мастеров) — не самые частые операции при верстке, однако по трудозатратам они могут дать фору иным часто повторяющимся действиям. Кроме того, без них невозможно построение полнофункциональных скриптов.

В самом деле, процесс автоматизации верстального процесса достаточно многогранен и, как только у вас начнет получаться автоматизация — сначала простеньких действий, потом все более сложных — вы обязательно захотите автоматизировать свою работу по максимуму. Ничего не поделаешь: человек так устроен, а успешное решение одних задач усиливает веру в свои силы, и то, что казалось вам полгода назад невозможным, оказывается, не так уже и сложно сделать. В зависимости от специфики рабочего процесса можно упростить себе работу как частично, так и практически полностью.

Рассматривая операции при работе с документами, мы научимся выполнять:

- базовые действия с файлами (открытие, сохранение, печать и т. п.);
- изменять единицы измерения;
- устанавливать размеры страниц и основные параметры;
- ограничивать свой выбор только определенным типом или типами объектов;
- выполнять экспорт содержимого в разные форматы;
- выравнивать объекты;
- постранично импортировать многостраничный PDF-документ;
- импортировать целую коллекцию изображений за один раз;
- выполнять базовые операции чистки — собирать фрагменты публикации, разбросанные по рабочим столам на всех разворотах в одном месте;
- выполнять листоподбор.

4.1. Открытие документа

Для открытия существующей публикации служит метод `open()`, имеющий следующий синтаксис:

`open(`*`from`*`[, `*`showingWindow`*`])`

Здесь:

- ***`from`*** — путь к файлу;
- ***`showingWindow`*** — если `true`, окно документа отображается (значение по умолчанию). Если `false`, остается спрятанным до тех пор, пока не придет инструкция создать окно.

Традиционный способ открытия документа, при котором документ сразу же отображается на экране (листинг 4.1).

Листинг 4.1. Открытие документа

```
app.open(File("/c/myTestDocument.indd"));
```

Однако всегда ли нужно непосредственное отображение публикации для работы скрипта? Наверное, нет — например, если скрипт выполняет значительное количество операций без вмешательства пользователя (т. е. не вызывает дополнительные диалоговые окна и т. д.). Вопрос особенно актуальный в связи с тем, что любое изменение в публикации влечет за собой перерисовку экрана, которая находится в прямой зависимости от разрешения монитора — чем выше его разрешение, тем больше занимает она времени. Для минимизации непроизводительных затрат ресурсов (в данном случае — их распыление на перерисовку) без какого-либо ущерба для функциональности предназначен параметр `showingWindow` (листинг 4.2).

Листинг 4.2. Открытие документа в фоновом режиме

```
// Открытие документа в фоновом режиме
var myDocument = app.open(File("/c/myTestDocument.indd"), false);
...
// Затем вы проводите с ним необходимые операции и, только когда
// захотите отобразить результат на экране, создаете новое окно
...
var myLayoutWindow = myDocument.windows.add();
```

4.2. Сохранение документа

В InDesign возможны два варианта сохранения публикации: под тем же именем (**Save**) либо под новым (**Save As**). В ExtendScript оба варианта реализуются единственным методом — `save()`, который, в зависимости от установленных параметров, может выполнять функцию **Save As**:

```
save([to][, stationery])
```

Здесь:

- ***to*** — новое расположение файла на диске;
- ***stationery*** — сохранять в виде шаблона или нет.

Пример сохранения документа приведен в листинге 4.3.

Листинг 4.3. Сохранение документа

```
// Если в документе остались незаписанные изменения и их нужно
// сохранить в документе под тем же именем

if(app.activeDocument.modified){
  app.activeDocument.save();
}
```

Если документ нужно сохранить под другим именем, указывают новый путь (листинг 4.4).

Листинг 4.4. Сохранение документа под другим именем

```
if (!app.activeDocument.saved){
  app.activeDocument.save(new File("/c/myTestDocument.indd"));}
```

Либо модифицируют текущее имя, например:

```
app.activeDocument.save(currentFileName+"_copy")
```

Сохранение публикации в виде шаблона происходит так, как показано в листинге 4.5.

Листинг 4.5. Сохранение публикации в виде шаблона (вариант 1)

```
var myFileName;
if (app.activeDocument.saved){
```

```
  // Преобразуем объект fullName (тип - файл) в строку
  // для возможности работы с ним как с текстовой строкой
  myFileName = app.activeDocument.fullName.toString();

  // Если файл имеет расширение indd, меняем его на indt
  if (myFileName.match(".indd") != null){
    // Создаем шаблон (регулярное выражение) для поиска
    // (см. приложение 2)
    var myRegularExpression = /\.indd/i

    // Выполняем замену
    myFileName = myFileName.replace(myRegularExpression, ".indt");
  }
}

// Полученную новую строку используем как параметр
// для сохранения файла под новым именем
app.activeDocument.save(File(myFileName), true);
```

Альтернативный вариант — без использования регулярных выражений — приведен в листинге 4.6.

Листинг 4.6. Сохранение публикации в виде шаблона (вариант 2)

```
if (myFileName.match(".indd") != null){
  // Разбиваем имя файла на отдельные фрагменты,
  // в качестве разделителя используем точку.
  // Если имя файла типичное (используется только одна точка —
  // между именем файла и его расширением), то в первом фрагменте
  // окажется имя файла, во втором — расширение.
  // Используем первый фрагмент, меняя расширение
  myFileName = myFileName.split(".")[0] + ".indt");

  // Полученную новую строку используем как параметр
  // для сохранения файла под новым именем
  app.activeDocument.save(File(myFileName), true);
}
```

4.3. Закрытие документа

Закрытие документа выполняется методом `close()`:

close([*saving*] [, *savingIn*])

Здесь:

- ***saving*** — может принимать значения:
  - `SaveOptions.no` — закрывает документ без сохранения;
  - `SaveOptions.yes` — перед закрытием сохраняет документ;
  - `SaveOptions.ask` — InDesign отображает диалоговое окно для сохранения файла;
- ***savingIn*** — путь, по которому файл будет сохранен.

Если выбран вариант `SaveOptions.yes`, необходимо указать путь для сохранения (параметр ***savingIn***).

Рассмотрим два варианта сохранения документа. Первый представлен в листинге 4.7.

Листинг 4.7. Сохранение документа (вариант 1)

```
app.documents[0].close();
```

И второй — через ссылку на активный документ (листинг 4.8).

Листинг 4.8. Сохранение документа (вариант 2)

```
app.activeDocument.close();
```

Второй вариант более предпочтителен при работе с несколькими открытыми публикациями, поскольку в таком случае нет необходимости получать порядковый номер текущего документа.

В листинге 4.9 представлен пример использования метода `close()`.

Листинг 4.9. Использование метода `close()`

```
if(!app.activeDocument.saved){
  // Сохранение в ручном режиме
  app.activeDocument.close(SaveOptions.ask);

  // И без вмешательства пользователя
  var myFile = File("/c/myTestDocument.indd");
  app.activeDocument.close(SaveOptions.yes, myFile);
}else{
  // Если публикация была сохранена
  app.activeDocument.close();
}
```

4.4. Работа с единицами измерения

InDesign предусматривает работу со многими единицами измерения — дюймы, сантиметры, точки и т. п. Программа поддерживает явное задание единиц измерения, например, "24 мм", "48 pt". Если же задать величину без указания размерности, InDesign будет использовать размерность, установленную по умолчанию. Это зачастую приводит к возникновению проблем, поскольку объект может выйти за пределы монтажного стола и редактор выдаст ошибку.

InDesign всегда возвращает координаты и другие свойства, имеющие размерность, в тех единицах, которые установлены по умолчанию. При этом нетрадиционные представления величин, такие как "1p6" и им подобные, автоматически конвертируются в традиционные (в нашем случае — в 1,5). Сделано это во избежание потенциальных проблем при выполнении арифметических действий (например, при сложении "1p6" с "3"), в то же время 1,5 + 3 даст корректный результат — 4,5.

Если вам предстоит выполнение арифметических действий, можно изменить единицы измерения на нужные на период работы скрипта, а впоследствии прежние значения восстановить. Альтернативный вариант — явное задание требуемой единицы (табл. 4.1).

Таблица 4.1. *Допустимые единицы измерения в InDesign*

Сокращение	Обозначение	Пример
c	Цицеро	1.4c
cm	Сантиметр	.0635cm
i (in)	Дюйм	.25i
mm	Миллиметр	6.35mm
pt	Пика	1p6
P	Типографская точка	18pt

Тип размерности хранится как свойство `viewPreferences` объекта `application.Viewpreferences`, который, в свою очередь, также является объектом с двумя свойствами: для горизонтального направления — `horizontalMeasurementUnits` и вертикального — `verticalMeasurementUnits`. Возможные значения:

- `MeasurementUnits.agates`;
- `MeasurementUnits.picas`;

- MeasurementUnits.points;
- MeasurementUnits.inches;
- MeasurementUnits.inchesDecimal;
- MeasurementUnits.centimeters;
- MeasurementUnits.ciceros;
- MeasurementUnits.custom.

Объект MeasurementUnits необходимо указывать обязательно.

Изменение настроек, установленных в приложении по умолчанию, происходит так, как показано в листинге 4.10.

Листинг 4.10. Изменение настроек приложения

```
var myDocument = app.activeDocument;
with(myDocument.viewPreferences){
  horizontalMeasurementUnits = MeasurementUnits.points;
  verticalMeasurementUnits = MeasurementUnits.points;
}
```

Если необходимо изменить единицы измерения только временно, на период выполнения каких-то операций, потом потребуется восстановить значения, используемые по умолчанию (листинг 4.11).

Листинг 4.11. Изменение и восстановление настроек приложения

```
var myDocument = app.activeDocument
with (myDocument.viewPreferences)
{
  // Запоминаем текущие установки
  var myOldXUnits = horizontalMeasurementUnits;
  var myOldYUnits = verticalMeasurementUnits;
  // Устанавливаем новые
  horizontalMeasurementUnits = MeasurementUnits.points;
  verticalMeasurementUnits = MeasurementUnits.points;
}

// Здесь выполняется основной код
...

// Восстанавливаем прежние значения
with (myDocument.viewPreferences){
  try
```

```
  {
    horizontalMeasurementUnits = myOldXUnits;
    verticalMeasurementUnits = myOldYUnits;
  }
  catch (myError)
  {
    alert("Ошибка восстановления прежних единиц измерения");
  }
}
```

На случай непредвиденных ситуаций в процессе отладки скрипта желательно постоянно пользоваться конструкцией `try{} catch(err){}` — она не даст скрипту "зависнуть", а лишь проинформирует вас о необходимости более тщательной проверки условий работы скрипта.

4.5. Определение координат

В InDesign используется привычный способ задания координат объектов — по осям *x* и *y*. Все расстояния измеряются относительно точки начала координат, которая в InDesign совпадает с левым верхним углом страницы. Значения ниже начала координат считаются положительными, выше — отрицательными. Это сделано исключительно ради удобства, поскольку в случае неизменности начала координат значения всех объектов в публикации по оси *y* будут положительными.

При задании размеров некоторых объектов, в частности текстовых фреймов, используется двухточечный метод: задаются координаты левого верхнего угла и правого нижнего (т. е. его габариты). Важно помнить порядок установки значений: верх, левая точка, низ, правая точка (`y1`, `x1`, `y2`, `x2`) — рис. 4.1.

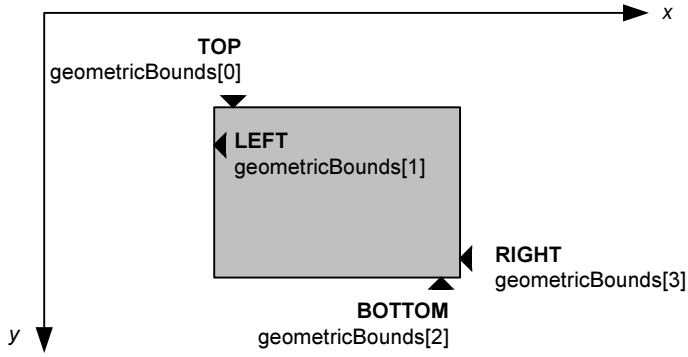


Рис. 4.1. Порядок перечисления координат в свойстве `geometricBounds`

Задаются координаты через свойство `geometricBounds`, которое, как вы уже поняли, является массивом из четырех значений. Необходимо отметить, что в любом случае нужно использовать только такую, полную, конструкцию (листинг 4.12), даже если нужно изменить лишь какое-то одно значение.

Листинг 4.12. Задание координат через свойство `geometricBounds`

```
myObject.geometricBounds = Array[ topValue, leftValue, bottomValue,
rightValue ]
```

Иными словами, команду

```
myObject.geometricBounds[3] = myObject.geometricBounds[3]+10
```

InDesign проигнорирует, причем, в отличие от других некорректных действий, никакого замечания не выдаст (это — особенность работы с массивами, существующая в JavaScript) и, не зная ее, можно долго гадать: почему предупреждения об ошибке не возникает, а результат неправильный.

4.6. Использование мастер-страниц

Как правило, после определения размеров документа следующим шагом идет установка параметров мастер-страниц. Вот как это реализуется средствами скриптинга в InDesign (листинг 4.13).

Листинг 4.13. Установка параметров мастер-страниц

```
// Создание нового документа
myDocument = app.documents.add();
// Задание параметров страниц
with(myDocument.documentPreferences){
  pageHeight = "297 mm"
  pageWidth = "210 mm"
  facingPages = true;
  pageOrientation = PageOrientation.portrait;
}

// Установка начала координат документа в начало координат страницы
myDocument.viewPreferences.rulerOrigin = RulerOrigin.pageOrigin;

// Переходим на первую мастер-страницу
with(myDocument.masterSpreads[0]){
  // Установки для четных страниц
  with(pages[0]){
```

```
    // Отступы с краев
    with(marginPreferences){
      columnCount = 3;
      columnGutter = "1p";
      bottom = "6p"
      top = "4p"

      // "left" в применении к страницам означает внутренний край;
      // "right" — внешний.
      left = "6p"
      right = "4p"
    }

    // Добавляем фрейм для нижнего колонтитула
    with(textFrames.add()){
      geometricBounds = ["61p", "4p", "62p", "45p"];
      with (insertionPoints[0]){
        contents = SpecialCharacters.sectionMarker;
        contents = SpecialCharacters.emSpace;
        contents = SpecialCharacters.autoPageNumber;
      }
      paragraphs.[0].justification = Justification.leftAlign;
    }
  }

  // Установки для нечетных страниц
  with(pages[1]){
    with(marginPreferences){
      columnCount = 3;
      columnGutter = "1p";
      bottom = "6p"
      left = "6p"
      right = "4p"
      top = "4p"
    }

    // Повторяем все то же, что и для четных страниц
    with(textFrames.add()){
      with (insertionPoints[0]){
        contents = SpecialCharacters.sectionMarker;
        contents = SpecialCharacters.emSpace;
        contents = SpecialCharacters.autoPageNumber;
      }
```

```
      paragraphs.[0].justification = Justification.leftAlign;}
    }
  }
}
```

Для присвоения параметров мастера-страниц в публикации у страниц существует свойство `appliedMaster`. Вот как можно применить мастер с именем `["B-Master"]` к третьей странице (листинг 4.14).

Листинг 4.14. Пример использования мастер-страницы для одиночной страницы

```
app.activeDocument.pages[2].appliedMaster =
  app.activeDocument.masterSpreads["B-Master"];
```

Задание вместо индекса реального названия мастера очень удобно, иначе бы пришлось выполнять лишнюю работу.

Аналогично обстоит дело с назначением мастера целому развороту (листинг 4.15).

Листинг 4.15. Пример использования мастер-страницы для разворота

```
app.activeDocument.masterSpreads[0].pages[0].appliedMaster =
  app.activeDocument.masterSpreads["B-Master"];
```

4.7. Печать документов

Простейший способ отправки на печать текущего документа с установками окна **Print** по умолчанию:

```
app.activeDocument.print();
```

Метод имеет параметры, расширяющие его функциональность:

`print([printDialog][ ,using])`

Здесь:

- ***`printDialog`*** — если `true`, отображать диалоговое окно, если `false` — выводить на печать сразу;
- ***`using`*** — набор используемых предустановок для печати (в них указывается в том числе и используемый принтер).

Для печати определенного диапазона страниц в специально введенном для упрощения вывода на печать объекте `printPreferences` устанавливают свойство `pageRange`. Можно задать диапазон как непосредственно (формат — ана-

логичный тому, который используется при отправке на печать через окно **Print**), так и все страницы сразу (`PageRange.allPages`). Пример приведен в листинге 4.16.

Листинг 4.16. Печать диапазона страниц текущего документа

```
app.activeDocument.printPreferences.pageRange = "1-3"
app.activeDocument.print(false);
```

Использование предустановок, оптимизированных для вывода на разные принтеры, значительно упрощает процесс печати. Процесс создания пользовательского набора предустановок иллюстрирует пример из листинга 4.17.

Листинг 4.17. Создание пользовательского набора предустановок для печати

```
with(app.activeDocument.printPreferences){
  // Название принтера, под каким он виден в окне Print
  printer = "HP Color LaserJet 3700";

  // Перед установкой следующих далее значений убедитесь,
  // что для выбранного принтера они доступны:
  copies = 2
  collating = false;
  reverseOrder = false;
  pageRange = PageRange.allPages;
  printSpreads = false;
  printMasterPages = false;
  printFile = "/c/test.indd";
  sequence = Sequences.all;

  // Свойства со вкладки Output в диалоговом окне Print
  negative = true;
  colorOutput = ColorOutputModes.separations;
  trapping = Trapping.applicationBuiltin;
  screening = "175 lpi/2400 dpi";
  flip = Flip.none
  if(trapping == Trapping.off){
    printBlack = true;
    printCyan = true;
    printMagenta = true;
    printYellow = true
    printBlankPages = false;
```

```
  printGuidesGrids = false;
  printNonprinting = false
}

// Свойства со вкладки Setup
paperSize = PaperSizes.custom
paperHeight = 1200;
paperWidth = 1200;
printPageOrientation = PrintPageOrientation.portrait;
pagePosition = PagePositions.centered;
paperGap = 0;
paperOffset = 0;
paperTransverse = false;
scaleHeight = 100;
scaleWidth = 100;
scaleMode = ScaleModes.scaleWidthHeight;
scaleProportional = true;
if(trapping == Trapping.off){
  textAsBlack = false;
  thumbnails = false
  thumbnailsPerPage = 4;
  tile = false;
}

// Свойства со вкладки Marks and Bleed:
allPrinterMarks = true;
useDocumentBleedToPrint = false;
with(app.activeDocument.documentPreferences){
  bleedBottom = documentBleedBottomOffset+3;
  bleedTop = documentBleedTopOffset+3;
  bleedInside = documentBleedInsideOrLeftOffset+3;
  bleedOutside = documentBleedOutsideOrRightOffset+3;
}
if(bleedBottom == 0 && bleedTop == 0 && bleedInside == 0 &&
   bleedOutside == 0){
  bleedMarks = true;
}else{
  bleedMarks = false;
}
colorBars = true;
cropMarks = true;
includeSlugToPrint = false;
markLineWeight = 0.25pt
```

```
  markOffset = 6;
  markType = MarkTypes.default;
  pageInformationMarks = true;
  registrationMarks = true;

  // Свойства из панели Graphics
  sendImageData = ImageDataTypes.allImageData;
  fontDownloading = FontDownloading.complete;
  downloadPPDFOnts = true;
  try{
    dataFormat = DataFormat.binary;}
  catch(e){}
  try{
    postScriptLevel = PostScriptLevels.level3;}
  catch(e){}

  // Настройки со вкладки Properties окна Color Management
  try{
    sourceSpace = SourceSpaces.useDocument;
    intent = RenderingIntent.useColorSettings;
    crd = ColorRenderingDictionary.useDocument;
    profile = Profile.postscriptCMS;}
  catch(e){}

  // Настройки со вкладки Advanced
  opiImageReplacement = false;
  omitBitmaps = false;
  omitEPS = false;
  omitPDF = false;
  try{
    flattenerPresetName = "high quality flattener";}
  catch(e){
    alert("Отсутствует набор предустановок: high quality flattener")}
    ignoreSpreadOverrides = false;
  }
}
```

4.8. Экспорт публикации

4.8.1. Экспорт в PDF

Скриптинг дает ту же гибкость при экспорте страниц в формате PDF, что и выполнение операции через хорошо знакомую операцию **Save As PDF**. Для

удобства, как и в случае с выводом на печать, предусмотрено создание стилей для экспорта — если записать стили в виде отдельных файлов, их можно использовать повторно.

Синтаксис операции:

```
exportFile(format, to [, showingOptions] [, using])
```

Здесь:

- *format* — формат экспорта;
- *to* — полное имя файла;
- *showingOptions* — опция показа диалогового окна;
- *using* — стиль экспорта.

Пример экспорта публикации в PDF-файл приведен в листинге 4.18.

Листинг 4.18. Экспорт в PDF-файл с текущими установками

```
app.activeDocument.exportFile(ExportFormat.pdfType,
File("/c/myTestDocument.pdf"), false);
```

В листинге 4.19 представлен пример использования стиля экспорта. Сначала создается стиль, который передается методу `exportFile()`.

Листинг 4.19. Экспорт в PDF-файл через стиль

```
// Резервируется название набора предустановок
var myPDFExportPreset = app.pdfExportPresets["prepress"];

// Собственно экспорт
app.activeDocument.exportFile(ExportFormat.pdfType,
File("/c/myTestDocument.pdf"), false, myPDFExportPreset);
```

Стиль для экспорта PDF с точки зрения объектной модели является типичным объектом, в котором хранятся все настройки, связанные с функцией экспорта (листинг 4.20).

Листинг 4.20. Создание стиля для экспорта

```
with(app.pdfExportPreferences){
  // Базовые настройки
  pageRange = PageRange.allPages;
  acrobatCompatibility = AcrobatCompatibility.acrobat6;
  exportGuidesAndGrids = false;
```

```
exportLayers = false;
exportNonPrintingObjects = false;
exportReaderSpreads = false;
generateThumbnails = false;
try{
  ignoreSpreadOverrides = false;
}catch(e){}
includeBookmarks = true;
includeHyperlinks = true;
includeICCProfiles = true;
includeSlugWithPDF = false;
includeStructure = false;
interactiveElements = false;
subsetFontsBelow = 0;

// Опции обработки растровой графики
colorBitmapCompression = BitmapCompression.zip;
colorBitmapQuality = CompressionQuality.eightBit;
colorBitmapSampling = Sampling.none;
thresholdToCompressColor = 200
colorBitmapSamplingDPI = 300
grayscaleBitmapQuality = CompressionQuality.eightBit;
grayscaleBitmapSampling = Sampling.none;
monochromeBitmapSampling = Sampling.none;
compressTextAndLineArt = true;
contentToEmbed = PDFContentToEmbed.embedAll;
cropImagesToFrames = true;
optimizePDF = true;

// Настройки меток печати
with(app.activeDocument.documentPreferences){
  bleedBottom =documentBleedBottomOffset;
  bleedTop =documentBleedTopOffset;
  bleedInside =documentBleedInsideOrLeftOffset;
  bleedOutside =documentBleedOutsideOrRightOffset;
}

if(bleedBottom == 0 && bleedTop == 0 && ↳
   bleedInside == 0 && bleedOutside == 0){
  bleedMarks = true;
}else{
  bleedMarks = false;
}
```

```
  colorBars = true;
  colorTileSize = 128;
  grayTileSize = 128;
  cropMarks = true;
  omitBitmaps = false;
  omitEPS = false;
  omitPDF = false;
  pageInformationMarks = true;
  pageMarksOffset = 12;
  pdfColorSpace = PDFColorSpace.unchangedColorSpace;
  printerMarkWeight = PDFMarkWeight.p125pt;
  registrationMarks = true;
  try{
    simulateOverprint = false;
  }
  catch(e){}
  useDocumentBleedWithPDF = true;

  // Опция просмотра результата в Acrobat
  viewPDF = false;
}
```

А теперь — экспорт диапазона страниц (листинг 4.21).

Листинг 4.21. Экспорт диапазона страниц

```
with(app.pdfExportPreferences){
  // Задается диапазон — аналогично ручному методу
  pageRange = "1, 3-6, 7, 9-11, 12";
}
// Указываем, какой набор установок экспорта использовать
var myPDFExportPreset = app.pdfExportPresets["prepress"]

// Экспорт с указанными параметрами
app.activeDocument.exportFile(ExportFormat.pdfType, ↳
  File("/c/myTestDocument.pdf"), false, myPDFExportPreset);
```

4.8.2. Экспорт в EPS

Экспорт как в PDF, так и в EPS — одна из наиболее часто встречающихся операций при работе с файлами, поскольку EPS — стандарт для вывода страниц на печать, и любое препресс-бюро с радостью возьмет на обработку приготовленные вами файлы. Основное отличие от экспорта в PDF в том, что

каждая страница сохраняется в виде отдельного EPS-файла, поэтому количество файлов при обработке объемной публикации может быть весьма значительным. Поэтому для удобства желательно в имя файла, кроме номера страницы, включать название файла публикации — или какой-нибудь другой текст, по которому можно было бы без проблем определить принадлежность страниц той или иной публикации (префикс).

Пример экспорта публикации в файл в формате EPS приведен в листинге 4.22.

Листинг 4.22. Постраничный экспорт публикации в формате EPS

```
// Вывод окна для выбора папки назначения
var myFolder = Folder.selectDialog ("Choose a Folder");

// Если нажата кнопка Cancel
if(myFolder == null){
  alert("Повторите попытку снова");}
}else{
  // Если нажата кнопка OK: экспорт.
  // Резервирование переменных
  var myFilePath, myPageName, myFile;

  // Создание необходимых ссылок
  var myDocument = app.activeDocument;
  var myDocumentName = myDocument.name;

  // Создание диалогового окна для ввода идентификатора публикации
  var myDialog = app.dialogs.add({name:"Экспорт страниц"});
  with(myDialog.dialogColumns.add().dialogRows.add()){
    staticTexts.add({staticLabel: "Префикс:"});
    var myBaseNameField = textEditboxes.add({editContents: ↳
      myDocumentName, minWidth: 160});}

  // Вывод диалогового окна на экран
  var myResult = myDialog.show();

  // Если все нормально
  if(myResult){
    // Считывание префикса
    var myBaseName = myBaseNameField.editContents;

    // Удаление диалога из памяти
    myDialog.destroy();
```

```
    // Формирование полного пути файла по маске:
    // имя папки + имя документа + номер страницы + расширение
    for(var i = 0; i < myDocument.pages.length; i++){
      myPageName = myDocument.pages[i].name;
      app.epsExportPreferences.pageRange = myPageName;
```

В случае, если используются разделы (`Sections`), в названиях страниц будет фигурировать символ :, который, однако, запрещен для использования в именах файлов (по крайней мере, в Windows). Поэтому будем его заменять на нейтральный символ — подчеркивание (_):

```
      myPageName = myPageName.replace(/:/g, "_");
      myFilePath = myFolder + "/" + myBaseName + "_" + ↳
        myPageName + ".eps";

      // Создаем пустой файл
      myFile = new File(myFilePath);

      // Записываем в него результат экспорта
      app.activeDocument.exportFile(ExportFormat.epsType, myFile, false);
    }
    myDialog.destroy();
  }
}
```

4.8.3. Экспорт в HTML

Скрипт экспортирует все материалы публикации в виде файлов HTML на диск с сохранением форматирования (без использования каскадных стилей, при этом предполагается, что стили публикации будут переназначаться встроенными в HTML стилями `H1`, `H2` и т. п.). В принципе, никаких сложностей с использованием Cascade Style Sheets нет, поэтому такой вариант рассматриваться не будет.

Самый главный вопрос при этом — сохранить все форматирование в конечном варианте. К сожалению, объять необъятное невозможно: в верстке может быть несколько десятков вариантов стилевого оформления, поэтому ограничимся разумными рамками: предположим, что в публикации используются 4 стиля — `body_text`, `heading1`, `heading2` и `heading3`. В принципе, расширить их число не представляет никакой проблемы: предлагаемый скрипт служит только для иллюстрации общего подхода к решению задачи, а конкретная реализация не составит никакого труда.

Принцип конвертирования таков: сначала формируем таблицу соответствий, определяющую, какому стилю из публикации соответствует определенный

стиль в HTML-документе — именно ее нужно наращивать используемыми стилями до требуемых возможностей. Затем просматриваем содержание каждого материала публикации и определяем, каким стилем отформатирован каждый абзац.

В зависимости от присвоенного стиля добавляем в начало и конец каждого абзаца HTML-документа открывающий и закрывающий теги.

Если попадается таблица, ее также оформляем соответствующими тегами разметки (`table`, `tr`, `td`).

Начинаем (листинг 4.23). Шаг первый — создаем таблицу соответствий в виде двумерного массива, каждый элемент которого представляет собой отдельный массив с парой значений: название стиля, используемого в публикации, и соответствующий ему стиль в HTML. Такой подход позволит нам в цикле просматривать правила переформатирования, и в случае, если найден любой из первых элементов, заменять на второй.

Листинг 4.23. Экспорт в HTML

```
myStyleToTagMapping = new Array;
myStyleToTagMapping.push(["body_text", "p"]);
myStyleToTagMapping.push(["heading1", "h1"]);
myStyleToTagMapping.push(["heading2", "h2"]);
myStyleToTagMapping.push(["heading3", "h3"]);

// Выводим диалоговое окно, в котором пользователь указывает,
// куда файлы будут экспортированы
if(app.documents.length!=0){
  if(app.documents[0].stories.length != 0
  var myTextFile = File.saveDialog("Save HTML As", undefined)
  if(myTextFile != null){  // Открываем файл с доступом для записи
    myTextFile.open("w");

    // Начинаем просматривать содержимое каждого материала публикации
    for(var i = 0; i <app.documents[0].stories.length; i ++){
      myStory = app.documents[0].stories[i];

      // Переходим к абзацам
      for(var j = 0; j<myStory.paragraphs.length; j ++){
        myParagraph = myStory.paragraphs[j];

        // С таблицами — отдельный разговор.
        // Пока рассматриваем только текст
        if(myParagraph.tables.length == 0){
```

```
    // Проверяем наличие локального форматирования
    if(myParagraph.textStyleRanges.length == 1){
      // Если оно существует, определяем соответствующую
      // HTML-разметку. Если полужирный, то Bold и т. п.
      myTag = myFindTag(myParagraph.appliedParagraphStyle.name, ↳
        myStyleToTagMapping);

      // Если данному стилю в публикации не поставлен
      // в соответствие ни один HTML-стиль, выводим абзац
      // как простой текст.
      if(myTag == "")
        {myTag = "p";}

      // Иначе задаем оформление для открывающего
      // и закрывающего тегов
      myStartTag = "<" + myTag + ">";
      myEndTag = "</" + myTag + ">";

      // Если абзац в материале не последний, нужно удалять
      // из него символ абзаца — в таком случае мы получим
      // только чистый текст, без ненужной разметки

      if(myParagraph.characters[-1].contents == "\r"){
        myString = ↳
        myParagraph.texts.itemByRange(myParagraph.characters[0], ↳
        myParagraph.characters[-2]).contents;
      }else{
        myString = myParagraph.contents;
      }
    }
  }
```

Разберем подробнее метод `itemByRange()`. Он имеет два параметра, задающие диапазон объектов, к которым InDesign обращается через свои внутренние механизмы. Такой подход гораздо эффективнее в плане производительности (с ним может сравниться только написание не скрипта, а полноценного плагин-модуля, но это требует значительно больших знаний). Анализ производительности обоими методами (перебор средствами JavaScript и InDesign) дает убедительный результат: в среднем скорость выполнения форматирования поднялась на порядок, что является прекрасным результатом и отлично подходит для работы фактически с любым количеством объектов.

Метод `itemByRange()` существует в подавляющем большинстве объектов, за что разработчикам InDesign нужно сказать отдельное спасибо. Среди них —

работа со строками, колонками, символами, абзацами, таблицами, ячейками, рядами. Поскольку, как уже говорилось, в таком случае InDesign задействует внутренний механизм, никакого дискомфорта даже в случае обработки значительных фрагментов нет, особенно это ощущается при работе с таблицами на уровне ячеек.

```
    // Каждый HTML-абзац должен быть обрамлен открывающим
    // и закрывающим тегами
    myTextFile.writeln(myStartTag + myString + myEndTag);

    // Если используется множественное локальное форматирование,
    // чтобы его не потерять, устанавливаем для каждого стиля
    // соответствующую ему разметку
  }else{
    for(j = 0; j < myParagraph.textStyleRanges.length; j ++){
      myTextStyleRange = myParagraph.textStyleRanges[j];

      // Аналогично выделяем из абзаца только текст,
      // удаляя знаки абзаца
      if(myTextStyleRange.characters[-1]=="\r"){
        myString = ↳
        myTextStyleRange.texts.itemByRange(↳
        myTextStyleRange.characters[1], ↳
        myTextStyleRange.characters[-2]).contents;
      }else{
        myString = myTextStyleRange.contents;
      }

      // Оформляем HTML-тегами фрагменты текста, набранные
      // полужирным, курсивом и смешанным начертаниями
      switch(myTextStyleRange.fontStyle){
        case "Bold": myString = "<b>" + myString + "</b>"
                     break;
        case "Italic": myString = "<i>" + myString + "</i>"
                       break;
        case "Bold Italic": myString = "<b><i>" + ↳
                            myString + "</i></b>"
                            break;
      }

      // Записываем конечный результат
      myTextFile.write(myString);}

      // Добавляем к нему символ новой строки
      myTextFile.write("\r");}
```

```
  // Если встретилась таблица, обрабатываем и ее
}else{
  myTable = myParagraph.tables[0];

  // Чтобы таблица была заметна, задаем ей рамку
  myTextFile.writeln("<table border = 1>");

  // Обрабатываем построчно
  for(myRowCounter = 0; ↳
      myRowCounter < myTable.rows.length; ↳
      myRowCounter ++){
    myTextFile.writeln("<tr>");

    // И каждую колонку отдельно
    for(myColumnCounter = 0; ↳
        myColumnCounter <myTable.columns.length; ↳
        myColumnCounter++){
      // Первая строка — шапка таблицы
      if(myRowCounter == 0){
        myString = "<th>" + ↳
        myTable.rows[0].cells[myColumnCounter]. ↳
        texts[0].contents + "</th>";

        // Обрабатываем остальные строки
      }else{
        // Берем содержимое каждой ячейки, окружая его
        // соответствующими тегами разметки
        myString = "<td>" + ↳
        myTable.rows[myRowCounter].cells[myColumnCounter]. ↳
          texts[0].contents + "</td>";}

        // Записываем в файл
        myTextFile.writeln(myString);}

        // Не забываем о теге конца строки
        myTextFile.writeln("</tr>");}

        // И, наконец, тег, закрывающий таблицу
        myTextFile.writeln("</table>");
      }
    }
  }
}
```

```
        // Файл закрываем для разрешения доступа
        // к нему другим приложениям
        file.myTextFile.close();
      }
    }
  }
}

// Функция, возвращающая название стиля абзаца
function myFindTag (myStyleName, myStyleToTagMapping){
  // Значение по умолчанию будет использоваться,
  // если стиль не переопределен в таблице соответствий
  var myTag = "";

  // Начинаем просмотр таблицы стилей с самой первой записи
  var i = 0;

  // Цикл проверки по всем записям
  do{
    // Если соответствие найдено
    if(myStyleToTagMapping[i][0] == myStyleName){
      // то получить соответствующий HTML-стиль
      myTag = myStyleToTagMapping[i][1];
      break;
    }
    i++;

    // Выполняем до тех пор, пока не дойдем до конца
    // таблицы (массива с названиями)
  } while (i < myStyleToTagMapping.length)

  // Результат возвращаем для дальнейшего использования
  return myTag;
}
```

4.9. Выделение объектов заданного типа

Довольно часто возникает необходимость выбора на развороте объектов строго определенного типа. Связано это с тем, что объекты из разных коллекций имеют наряду с некоторыми общими и специфические свойства и ме-

тоды, и во избежание ошибок необходимо быть уверенным, что все свойства и методы могут быть применимы к данному типу или типам объектов.

Перед тем как перейти непосредственно к скрипту, сделаем небольшое отступление. При написании скриптов очень удобно использовать функции — образно говоря, это — именованные фрагменты кода (причем удобно, если название раскрывает суть выполняемых в ней действий), которые могут быть сколь угодно большими. Удобство состоит в том, что при написании скрипта вы наглядно видите логику его работы — поскольку вызов функций осуществляется всего одной строкой — вместо того, чтобы разбираться с лесом проверок и циклов, за которым можно очень быстро потерять путеводную нить. Причем в дальнейшем, с ростом функциональности скриптов, вы будете все больше и больше убеждаться в эффективности использования функций. Итак, в последующем, где это только будет оправдано, будем постоянно использовать функции.

Рассмотрим листинг 4.24.

Листинг 4.24. Выделение объектов заданного типа

```
if (app.documents.length != 0){
  // Проверка количества элементов на развороте
  if (app.activeWindow.activeSpread.pageItems.length != 0){
    // Если объекты существуют, вызываем их и продолжаем.
    // Вызываем функцию, которая создает окно для задания
    // того типа объектов, которые должны быть выделены
    myDisplayDialog();
    mySelectObjects();
  }
  else {
    myError(noObjects)}
}
else{
  myError(noOpenDocuments)
}
noObjects = "На развороте нет никаких объектов";
noOpenDocuments = "Нет ни одного открытого документа. ↳
  Перед запуском скрипта убедитесь, что открыт хотя бы один документ"
```

Как видите, использование функций до предела упростило читабельность кода и сделало более наглядным логику скрипта, который состоит из отдельных шагов: отображение диалогового окна и сбор данных, выбор необходимых типов объектов; выдача предупреждения, если объектов заданного типа не оказалось или скрипт был запущен, когда не был открыт ни один документ.

А теперь определяем использованные функции.

```
function myDisplayDialog(){
  var myDialog;
  // Создаем пользовательское окно
  with(myDialog = app.dialogs.add({name:"SelectObjects"})){
    with(dialogColumns.add()){
      with(borderPanels.add()){
        staticTexts.add({staticLabel:"Select:"});
```

Вид используемого селектора зависит от степени функциональности скрипта. Если можно ограничиться выбором лишь какого-то определенного типа объекта, то достаточно создать раскрывающийся список — он не допускает множественного выделения. Однако увеличим функциональность, предусмотрев множественный выбор типов — чтобы была возможность выбора не только, скажем, прямоугольников, но и контейнеров, содержащих импортированную графику и т. п. В таком случае используем элементы интерфейса — флажки:

```
        with(dialogColumns.add()){
          var myRectanglesCheckbox = ↳
            checkboxControls.add({staticLabel:"Rectangles", ↳
            checkedState:true});
          var myEllipsesCheckbox = ↳
            checkboxControls.add({staticLabel:"Ellipses", ↳
            checkedState:true});
          var myPolygonsCheckbox = ↳
            checkboxControls.add({staticLabel:"Polygons", ↳
            checkedState:true});
          var myGraphicLinesCheckbox = ↳
            checkboxControls.add({staticLabel:"Graphic Lines", ↳
            checkedState:true});
          var myTextFramesCheckbox = ↳
            checkboxControls.add({staticLabel:"Text Frames", ↳
            checkedState:true});
          var myGroupsCheckbox = ↳
            checkboxControls.add({staticLabel:"Groups", ↳
            checkedState:true});
          var myImagesCheckbox = ↳
            checkboxControls.add({staticLabel:"Images", ↳
            checkedState:true});
          var myPDFsCheckbox = ↳
            checkboxControls.add({staticLabel:"PDFs", ↳
            checkedState:true});
```

```
        var myEPSsCheckbox = ↳
          checkboxControls.add({staticLabel:"EPSs", ↳
          checkedState:true});
      }
    }
  }
}

// Отображение диалогового окна
myResult = myDialog.show();
if (myResult){
  var myObjectTypes = new Array;
  // Сбор данных из диалогового окна и сохранение их
  if (myRectanglesCheckbox.checkedState ==  true){
    myObjectTypes.push("rectangles");
  }
  if(myEllipsesCheckbox.checkedState==true){
    myObjectTypes.push("ovals");
  }
  if(myPolygonsCheckbox.checkedState==true){
    myObjectTypes.push("polygons");
  }
  if(myGraphicLinesCheckbox.checkedState==true){
    myObjectTypes.push("graphicLines");
  }
  if(myTextFramesCheckbox.checkedState==true){
    myObjectTypes.push("textFrames");
  }
  if(myGroupsCheckbox.checkedState==true){
    myObjectTypes.push("groups");
  }
  if(myImagesCheckbox.checkedState==true){
    myObjectTypes.push("images");
  }
  if(myPDFsCheckbox.checkedState==true){
    myObjectTypes.push("pdfs");
  }
  if(myEPSsCheckbox.checkedState==true){
    myObjectTypes.push("epss");
  }

  // Очистка памяти от диалога
  myDialog.destroy();
```

```
    // Вызов следующей функции — она занимается основной работой,
    // т. е. выбором указанных типов объектов
    mySelectObjects();
  }
  else{
    // Если была нажата кнопка Cancel, никакие дальнейшие действия
    // не происходят
    myDialog.destroy();
  }
}
```

С первой функцией, выполняющей вывод на экран и сбор введенных пользователем значений, мы закончили. Переходим к главной операции — собственно выбору заданных пользователем типов объектов.

```
function mySelectObjects(){
  var myObjectsToSelect = new Array;
  with(app.activeWindow.activeSpread){
```

Поскольку `myObjectTypes` — пользовательский объект, имеющий тип `Array(`*массив*`)`, он не имеет никаких свойств, кроме единственного — длины массива. Это обстоятельство можно использовать для повышения читабельности кода вместо традиционной конструкции для перебора значений `(for(i = 0; i < myObjectTypes.length; i++))`:

```
    for(i in myObjectTypes){
```

Дальнейшие действия будут происходить по-разному — в зависимости от того, что выбрал пользователь. Предположим, им был выбран графический тип объектов (изображения типа EPS, PDF или любое растровое). Как мы знаем, InDesign не позволяет выбирать одновременно несколько объектов таких типов (самих изображений, а не содержащих их объектов-фреймов) — поэтому если решать вопрос "в лоб", мы значительно сузим функциональность скрипта. Чтобы такого не допустить, предпримем такой обходной маневр: в случае выбора пользователем графики будем выбирать не ее непосредственно, а контейнеры, ее содержащие. При этом обойдем ограничение InDesign — ведь выбирать контейнеры можно без каких-либо ограничений.

Таким образом, для графики имеем:

```
      if((myObjectTypes[i] != "images") && ↳
         (myObjectTypes[i] != "epss")&&(myObjectTypes[i] != "pdfs")){
        myPageItems = eval(myObjectTypes[i]);
        if (myPageItems.length != 0){
          for(ii = 0; ii < myPageItems.length; ii ++){
            myObjectsToSelect.push(myPageItems[ii]);
```

```
        }
      }
    }
  }
  for(j = 0; j < pageItems.length; j++){
    myPageItem = pageItems[j];
    try{
      if(myIsInArray("images", myObjectTypes) &&
        (myPageItem.images.length == 1))||
        (myIsInArray("epss", myObjectTypes) &&
        (myPageItem.epss.length == 1))||
        (myIsInArray("pdfs", myObjectTypes) &&
        (myPageItem.pdfs.length == 1))){
        // Проверяем, не находится ли объект уже в списке на выделение
        myID = myPageItem.id;
        myItemExists = false;
        for(k = 0; k < myObjectsToSelect.length; k++){
          if (myObjectsToSelect[k].id == myID){
            myItemExists = true;
            break;
          }
        }

        // Если да, то исключаем его из обработки
        if (myItemExists == false){
          myObjectsToSelect.push(myPageItem);
        }
      }
    }
    catch(myError){}
  }

  // Вместо выделения каждого изображения
  // выделяем содержащий его контейнер
  parent.select(myObjectsToSelect, SelectionOptions.replaceWith);
 }
}

// Проверка: является ли объект строкой.
// Ранее в массив myObjectTypes мы записали необходимые значения.
// Если объект — строка, значит, мы работаем с ним дальше
function myIsInArray(myString, myArray){
  for (i = 0; i < myArray.length; i ++){
```

```
    if (myArray[i] == myString){
      return true;
      break;
    }
  }
  return false;
}

// Вывод сообщения об ошибке
function onError(msg){
  alert(msg)
}
```

4.10. Экспорт выделенных объектов

Продолжая работу с выделенными объектами, напишем скрипт, который бы экспортировал выделенные объекты в формат, заданный пользователем. Необходимость в нем возникает часто при работе с заказчиком: он хочет видеть конечный вариант — как будет смотреться реклама на полосе. Встроенных функций в InDesign по экспорту только выделенной части нет, а делать это вручную (сохраняя страницу как EPS с последующим конвертированием в растровый вариант, кадрированием по размерам, чтобы ее мог просмотреть заказчик) — не самая творческая работа. Возложим ее на скрипт.

Алгоритм экспорта может быть таким:

1. Вывод диалогового окна, в котором указываются требуемые типы файлов.
2. Считывание пользовательских значений.
3. Если выбран текстовый формат, то выполняется непосредственный экспорт.
4. Если любой другой — копирование выделенных объектов в новый документ, изменение его размеров точно по размерам выделения и экспорт в выбранном формате.

Масштабирование выполняет метод `resize()`, имеющий синтаксис:

```
resize ([horizontalScale][, verticalScale] [, around]
[, considerringCurrentScale] [, transformingContent]
[, consideringParentScale])
```

Здесь:

- ***horizontalScale*** — масштаб по горизонтали (100 соответствует 100%);
- ***verticalScale*** — масштаб по вертикали;

- ***around*** — положение центра трансформации, задается либо как произвольная точка (ее координаты передаются в виде массива `Array(x,y)`), либо как одно из предопределенных свойств (`Enumeration`) объекта `AnchorPoint`: `topLeftAnchor`, `topCenterAnchor`, `topRightAnchor`, `leftCenterAnchor`, `centerAnchor` (значение по умолчанию), `rightCenterAnchor`, `bottomLeftAnchor`, `centerAnchor`, `rightCenterAnchor`;
- ***considerringCurrentScale*** — применять новый масштаб к уже существующему или считать его абсолютным (значение по умолчанию: `true`);
- ***transformingContent*** — делать ли трансформацию содержимого (для контейнеров с иллюстрациями), значение по умолчанию: `true`;
- ***consideringParentScale*** — проводить масштабирование с учетом изменения размеров родительского объекта или рассчитывать его автономно (значение по умолчанию: `true`).

Принцип сохранения выделенных объектов в виде файла следующий: если выбран текстовый фрейм, воспользуемся непосредственно методом `exportFile()`. Если же выбран любой из графических форматов, выделенные объекты через буфер обмена переносим в новый документ, размеры которого в точности совпадают с размерами области, занятой выделенными объектами. После этого используем снова метод `exportFile()`.

Скрипт представлен в листинге 4.25.

Листинг 4.25. Экспорт выделенных объектов

```
with (app) {
  // Проверяем наличие выделения
  if (selection.length < 1) {
    alert (langNoSelection);
    exit ();
  }

  // Определяем переменную, возвращающую тип объекта
  var mySelectionType = selection[0].constructor.name;

  // Свойство constructor позволяет определить принадлежность
  // объекта тому или иному типу. Результат имеет вид
  // [Object ObjectName], где ObjectName – тип объекта.
  // Для более удобной работы используют свойство объекта
  // name, которое непосредственно дает название типа объекта.

  // Подготавливаем массивы с параметрами экспорта
  var myFileTypes = new Array("Adobe PDF", "EPS", "JPEG"");
```

```
var myFileExt = new Array("*.pdf", "*.eps", "*.jpg");
var myExportType = new Array("ExportFormat.pdfType", ↳
  "ExportFormat.epsType", "ExportFormat.jpg");
if (mySelectionType == "TextFrame" ) {
  // Расширяем диапазон текстовыми форматами
  myFileTypes.push("ID Tagged Text", "Rich Text Format", ↳
    "Text only",);
  myFileExt.push("*.txt", "*.rtf", "*.txt");
  myExportType.push("ExportFormat.taggedText", ↳
    "ExportFormat.rtf", "ExportFormat.textType");
}

// Диалоговое окно
var myDialog = dialogs.add({name:"Параметры экспорта"});
with (myDialog) {
  with (dialogColumns.add()) {
    // Панель настроек
    with(dialogColumns.add()) {
      var mySelectFileType = ↳
        dialogRows.add().dropdowns.add({stringList:myFileTypes, ↳
        selectedIndex:0, ↳minWidth:150});
    }
  }
}
var myResult = myDialog.show();
if (!myResult) {
  exit();
}

// Отображаем запрос для присвоения файлу имени.
// Формат: название, состоящее из выбранного типа файла,
// после двоеточия - расширение
var myExportFile = ↳
  File.saveDialog((myFileTypes[mySelectFileType.selectedIndex] + ↳
  ":" +  myFileExt[mySelectFileType.selectedIndex]));

// Проверяем нажатие кнопки Cancel
if (!myExportFile) {
  exit();
}

// Если запрошен экспорт в текстовый формат, выполняем его
with(ExportFormat){
```

```
  if ((myExportType[mySelectFileType.selectedIndex] == taggedText) ↳
   || (myExportType[mySelectFileType.selectedIndex] == rtf) || ↳
      (myExportType[mySelectFileType.selectedIndex] == textType)) {
    // Если выбран фрейм, экспортируем весь содержащийся в нем текст
    if (mySelectionType == "TextFrame") {
      selection[0].texts[0].exportFile(myExportType↳
        [mySelectFileType.sel↳ectedIndex], myFile, true);

      // Если не фрейм, то экспортируем объект (объекты)
    } else {
      selection[0].exportFile↳
       (myExportType[mySelectFileType.selectedIndex], myFile, true);
    }
    exit();
  }
}

// Определяем размеры документа и его ориентацию
with (activeDocument.documentPreferences) {
  var myPageHeight = pageHeight;
  var myPageWidth = pageWidth;
}

// Если выбран текстовый объект и экспорт в графический формат,
// то выделяем родительский фрейм в качестве объекта экспорта
if ((mySelectionType == "InsertionPoint") || ↳
   (mySelectionType == ↳"Text") ) {
  selection[0].parentTextFrame.select();
}

// Копируем объекты в буфер: по всей логике.
// Метод принадлежит объекту Application
copy();

// Создаем новый документ как экспортный вариант
var myDoc = documents.add();

// Вставляем объект по возможности в исходном местоположении —
// полный аналог операции Edit/Paste in Place
pasteInPlace();

// Обнуляем поля публикации
with(myDoc.pages[0].marginPreferences) {
```

```
    top = 0;
    bottom = 0;
    left = 0;
    right = 0;
  }

  // Изменяем размер страницы по размерам объекта, поскольку
  // именно размер страницы определяет размеры итогового документа
  // после экспорта
  with (myDoc.documentPreferences) {
    pageHeight = myTop - myBottom;
    pageWidth = myRight — myLeft;
  }

  // Экспортируем документ
  try {
    myDoc.exportFile(myExportType[mySelectFileType.selectedIndex], ↳
      myFile, true);
  } catch (error) {
    alert ("При экспорте произошла ошибка");
  }

  // Закрываем созданный документ без сохранения
  myDoc.close(SaveOptions.no);
  exit();
}

// Получение габаритов группы выделенных объектов
function getSelectionBounds () {
  var myTop = selection[0].visibleBounds[0];
  var myLeft = selection[0].visibleBounds[1];
  var myBottom = selection[0].visibleBounds[2];
  var myRight = selection[0].visibleBounds[3];

  // Если какая-то величина превышает существующий максимум,
  // значение максимума корректируется
  for i = 1; i < selection.length; i++ ) {
    if (myTop > selection [i].visibleBounds[0]) {
      myTop = selection[i].visibleBounds[0];
    }
    if (myLeft > selection[i].visibleBounds[1]) {
      myLeft = selection[i].visibleBounds[1];
    }
```

```
    // С учетом направления оси координат
    if (myBottom < selection[i].visibleBounds[2]) {
      myBottom = selection[i].visibleBounds[2];
    }
    if (myRight < selection[i].visibleBounds[0]) {
      myRight = selection[i].visibleBounds[3];
    }
  }
}
```

4.11. Группировка объектов

Объединение объектов в группы — одна из стандартных ситуаций в верстке. Сгруппированные объекты ведут себя как один объект, что упрощает работу с ними — например, проведение трансформаций или перемещение по публикации. Как и при ручном способе, для скриптинга совершенно не имеет значения, какие именно объекты заносятся в группу — могут быть самые разнообразные. Единственно, что нужно помнить, — меняется их порядок по глубине залегания — это естественно. Один из способов реализации подобной операции скриптингом приведен в листинге 4.26.

Листинг 4.26. Группировка выбранных объектов

```
// Если объектов несколько, то заносим их в массив
if (selection.length > 1) {
  var mySelection  = new Array[];
  for i = 0; i < selection.length; i++){
    mySelection[i] = selection[i];
  }

  // Заносим объекты в группу
  var myObject = myDoc.pages[0].groups.add(mySelection);
}
```

4.12. Трансформация объектов

В качестве разрядки предлагаю заняться творчеством — скриптинг может и это, правда, в довольно ограниченных пределах. Напишем скрипт (листинг 4.27), который создает копии выделенного объекта (с возможностью трансформаций) и размещает их с определенным сдвигом на листе.

Предусмотрим использование всех существующих в InDesign операций трансформации: сдвиг в обоих направлениях (`shearAngle`), вращение (`rotate()`),

скос (`skew()`), масштаб (`scale()`) — отдельно по каждой оси (`horizontalScale` и `verticalScale`), выбор центра трансформаций (`AnchorPoint`, всего 8 положений). Дополнительно внесем возможность задания окантовки, непрозрачности, а также плотности краски для заливки и окантовки — с тем, чтобы получить законченное решение.

Вид диалогового окна нашего скрипта приведен на рис. 4.2, а результат работы кода — на рис. 4.3.

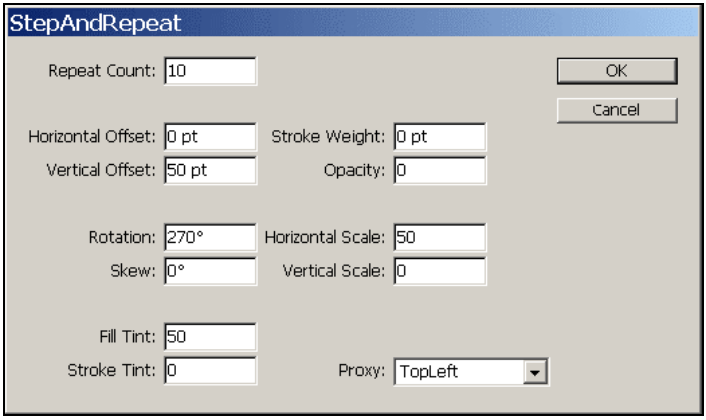


Рис. 4.2. Окно для задания параметров трансформаций

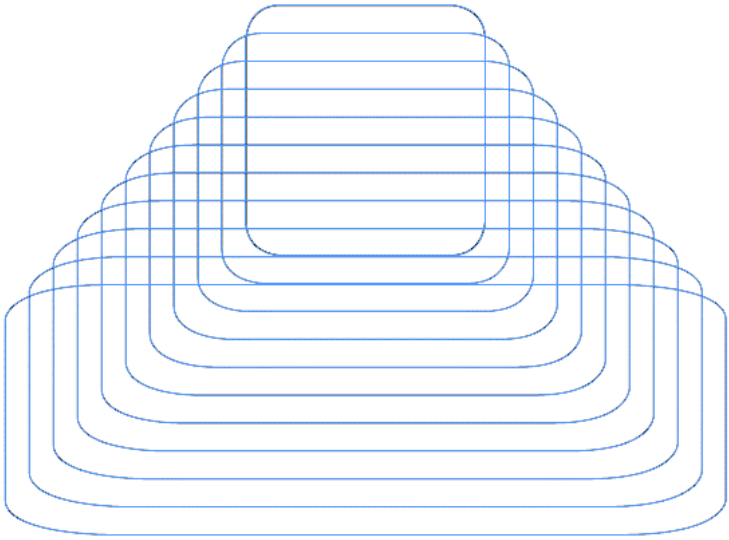


Рис. 4.3. Один из результатов работы скрипта

Листинг 4.27. Трансформации объектов

```
// Задаем параметры опорной точки (центра трансформаций)
var myProxyList = ↳
  ["TopLeft","Top","TopRight","Left","Center","Right",↳
   "BottomLeft","Bottom","BottomRight"];

// Традиционная проверка на корректность входных данных
if (app.selection.length > 0){
  myObjects = new Array;
  for(i = 0; i < app.selection.length; i++){
    switch(app.selection[i].constructor.name){
      case "Rectangle":
      case "Polygon":
      case "GraphicLine":
      case "Oval":
      case "TextFrame":
        myObjects.push(app.selection[i])
    }
  }
  myDisplayDialog(myObjects);
}
else{
  alert("Должен быть выделен хотя бы один объект!");
}

// Отображаем диалоговое окно
function myDisplayDialog(myObjects){
  myDialog = app.dialogs.add();
  myDialog.name = "StepAndRepeat";

  // Левая колонка
  myLabelsColumn1 = myDialog.dialogColumns.add();
  myLabelsColumn1.staticTexts.add({staticLabel:"Repeat Count:"});

  myLabelsColumn1.staticTexts.add();
  myLabelsColumn1.staticTexts.add({staticLabel:"Horizontal Offset:"});
  myLabelsColumn1.staticTexts.add({staticLabel:"Vertical Offset:"});

  myLabelsColumn1.staticTexts.add();
  myLabelsColumn1.staticTexts.add({staticLabel:"Rotation:"});
  myLabelsColumn1.staticTexts.add({staticLabel:"Skew:"});
```

```
myLabelsColumn1.staticTexts.add();
myLabelsColumn1.staticTexts.add({staticLabel:"Fill Tint:"});
myLabelsColumn1.staticTexts.add({staticLabel:"Stroke Tint:"});

// Правая колонка
myControlsColumn1 = myDialog.dialogColumns.add();
myRepeatField = myControlsColumn1.integerEditboxes.add({editValue:2});
myControlsColumn1.staticTexts.add();
myXOffsetField = ↳
  myControlsColumn1.measurementEditboxes.add({editValue:0});
myYOffsetField = ↳
  myControlsColumn1.measurementEditboxes.add({editValue:0});

myControlsColumn1.staticTexts.add();
myRotationField = myControlsColumn1.angleEditboxes.add({editValue:0});
mySkewField = myControlsColumn1.angleEditboxes.add({editValue:0});

myControlsColumn1.staticTexts.add();
myFillTintField = myControlsColumn1.realEditboxes.add({editValue:0});
myStrokeTintField = myControlsColumn1.realEditboxes.add({editValue:0});
myLabelsColumn2 = myDialog.dialogColumns.add();
myLabelsColumn2.staticTexts.add();
myLabelsColumn2.staticTexts.add();
myLabelsColumn2.staticTexts.add({staticLabel:"Stroke Weight:"});
myLabelsColumn2.staticTexts.add({staticLabel:"Opacity:"});

myLabelsColumn2.staticTexts.add();
myLabelsColumn2.staticTexts.add({staticLabel:"Horizontal Scale:"});
myLabelsColumn2.staticTexts.add({staticLabel:"Vertical Scale:"});

myLabelsColumn2.staticTexts.add();
myLabelsColumn2.staticTexts.add();
myLabelsColumn2.staticTexts.add({staticLabel:"Proxy:"});
myControlsColumn2 = myDialog.dialogColumns.add();
myControlsColumn2.staticTexts.add();
myControlsColumn2.staticTexts.add();
myStrokeWeightField = myControlsColumn2.measurementEditboxes.add();
myOpacityField = myControlsColumn2.realEditboxes.add({editValue:0});

myControlsColumn2.staticTexts.add();
myXScaleField = myControlsColumn2.realEditboxes.add({editValue:0});
myYScaleField = myControlsColumn2.realEditboxes.add({editValue:0});
myControlsColumn2.staticTexts.add();
```

```
  myControlsColumn2.staticTexts.add();
  myProxyMenu = ↳
    myControlsColumn2.dropdowns.add({stringList:myProxyList, ↳
    selectedIndex:0});
  myResult = myDialog.show();
  if(myResult == true){
    // Получаем параметры, введенные пользователем
    myNumberOfRepeats = myRepeatField.editValue;
    myXOffset = myXOffsetField.editValue;
    myYOffset = myYOffsetField.editValue;
    myRotation = myRotationField.editValue;
    mySkew = mySkewField.editValue;
    myFillTint = myFillTintField.editValue;
    myStrokeTint = myStrokeTintField.editValue;
    myStrokeWeight = myStrokeWeightField.editValue;
    myOpacity = myOpacityField.editValue;
    myXScale = myXScaleField.editValue;
    myYScale = myYScaleField.editValue;
    myProxy = myProxyMenu.selectedIndex;
    myDialog.destroy();

    with (app.activeDocument.viewPreferences){
      var myOldXUnits = horizontalMeasurementUnits;
      var myOldYUnits = verticalMeasurementUnits;

      horizontalMeasurementUnits = MeasurementUnits.points;
      verticalMeasurementUnits = MeasurementUnits.points;
    }

    // Собственно функция, выполняющая все преобразования
    myStepAndRepeat(myObjects, myNumberOfRepeats, myXOffset, myYOffset, ↳
      myRotation, mySkew, myFillTint, myStrokeTint, myOpacity, ↳
      myStrokeWeight, myXScale, myYScale, myProxy);

    // Восстановление размерностей
    app.activeDocument.viewPreferences.horizontalMeasurementUnits = ↳
      myOldXUnits;
    app.activeDocument.viewPreferences.verticalMeasurementUnits = ↳
      myOldYUnits;
  }
  else{
    myDialog.destroy();
  }
}
```

```
// Собственно трансформации
function myStepAndRepeat(myObjects, myNumberOfRepeats, myXOffset, ↳
  myYOffset, myRotation, mySkew, myFillTint, myStrokeTint, myOpacity, ↳
  myStrokeWeight, myXScale, myYScale, myProxy){

  // Определяем точку — центр трансформаций
  switch(myProxy){
    case 0:
      myProxy = AnchorPoint.topLeftAnchor;
      break;
    case 1:
      myProxy = AnchorPoint.topCenterAnchor;
      break;
    case 2:
      myProxy = AnchorPoint.topRightAnchor;
      break;
    case 3:
      myProxy = AnchorPoint.leftCenterAnchor;
      break;
    case 4:
      myProxy = AnchorPoint.centerAnchor;
      break;
    case 5:
      myProxy = AnchorPoint.rightCenterAnchor;
      break;
    case 6:
      myProxy = AnchorPoint.bottomLeftAnchor;
      break;
    case 7:
      myProxy = AnchorPoint.bottomCenterAnchor;
      break;
    case 8:
      myProxy = AnchorPoint.bottomRightAnchor;
      break;
  }

  // Делаем необходимое количество копий
  for (i = 1; i <= myNumberOfRepeats; i++){
    // Работаем со всеми выделенными объектами
    for (j=0; j < myObjects.length; j++)
    {
      myObject = myObjects[j];
```

```
// Каждый объект дублируется и сдвигается в соответствии
// с установками пользователя
myObject = myObject.duplicate();
myObject.move(undefined, [myXOffset*i, myYOffset*i]);

// Обработка величины вращения.
// Каждое действие состоит в изменении текущего значения
// на приращение
if (myRotation != 0){
  myObject.rotate(myRotation*i, myProxy, true, true);
}

// Обработка толщины окантовки.
// Ограничиваемся разумными пределами
if (myStrokeWeight != 0){
  newValue = myObject.strokeWeight + (myStrokeWeight*i);
  if (newValue >= 0 && newValue <= 800)
    myObject.strokeWeight = newValue;
}

// Обработка степени прозрачности
if (myOpacity != 0){
  newValue = myObject.opacity + (myOpacity*i);
  if (newValue >= 0 && newValue <= 100 )
    myObject.opacity = newValue;
}

// Масштабирование по обеим осям
if (myXScale != 0){
  newValue = myObject.horizontalScale + (myXScale*i);
  if (newValue >= 0.01)
    myObject.horizontalScale = newValue;
}
if (myYScale != 0){
  newValue = myObject.verticalScale + (myYScale*i);
  if (newValue >= 0.01)
    myObject.verticalScale = newValue;
}

// Скосы
if (mySkew != 0){
  newValue = myObject.shearAngle + (mySkew*i);
  if (newValue >= -80 && newValue <= 80 )
```

```
            myObject.shearAngle = newValue;
        }

        // Окантовка
        if (myStrokeTint != 0){
          newValue = myObject.strokeTint + (myStrokeTint*i);
          if (newValue >= 0 && newValue <= 100 ){
            myObject.strokeTint = newValue;
          }
        }

        // Аналогично для заливки
        if (myFillTint != 0){
          if(myObject.strokeTint == -1){
            newValue = myObject.fillTint + (myFillTint*i);
          }
          else{
            newValue = 100 + (myFillTint*i);
          }
          if (newValue >= 0 && newValue <= 100 ){
            myObject.fillTint = newValue;
          }
        }
      }
    }
}
```

Как видите, несмотря на значительный объем, занимаемый кодом для вывода диалогового окна и считывания значений, играющих вспомогательную функцию, сама исполняющая часть очень проста, все очень наглядно.

4.13. Метла для монтажного стола

Монтажный стол предназначен для временного хранения элементов публикации. На него складываются элементы оформления, изображения и т. д., которые по каким-то причинам не могут быть сразу же поставлены в полосу набора. При интенсивной работе с макетом, состоящим из одной или более сотен страниц, поиск нужного элемента, не зная хотя бы приблизительно, где он находится, обернется ощутимыми затратами времени. Данный скрипт значительно упрощает процесс поиска — он сканирует монтажные столы всех разворотов, после чего их содержимое переносит на текущий разворот (листинг 4.28).

Листинг 4.28. Поиск объекта на монтажных столах

```
myDocument = app.activeDocument
myPageItems = myDocument.pageItems;
var moveablesArray = [];
for (var i=0; i< myPageItems.length; i++) {
  if (myPageItems[i].parent.constructor.name == "Page") {
    continue;
  }
  moveablesArray.push( myPageItems[i].id );
}
var mySpread = app.activeWindow.activeSpread;
for (j in moveablesArray) {
  var myObj = myDocument.pageItems.itemByID(moveablesArray[j]);
  if (myObj.parent == mySpread){
    continue;
  }
  myBounds = myObj.geometricBounds;
  if (myObj.locked){
    continue;
  }
  myObj.move(mySpread);
  myObj.move([myBounds[1], myBounds[0]]);
}
```

Принцип работы скрипта прост: просматриваем все объекты в документе и у каждого определяем родителя — ближайший объект, в состав которого он входит. Родителями могут быть как страницы, так и монтажные столы. Поэтому развороты, являющиеся родителями для страниц, нас не интересуют. Если родитель — страница, пропускаем объект, иначе записываем его уникальный идентификатор (`id`) во временное хранилище:

```
moveablesArray.push(myPageItems[i].id);
```

Можно было бы использовать и другой метод для пометки — каждый объект в InDesign имеет свойство `Label` — это что-то вроде ярлыка, куда можно записывать любые пользовательские данные, вплоть до скриптов. В таком случае достаточно было бы объектам, удовлетворяющим нашим требованиям, присваивать, например, `Label = "PasteBoard"` — результат был бы идентичен.

По окончании данного шага в массиве `moveablesArray` находятся идентификаторы объектов, расположенных на монтажных столах, которые потом будем перемещать. Но прежде из них нужно отфильтровать те, которые расположе-

ны на текущем монтажном столе — поскольку они уже находятся в нужном месте.

```
if (myObj.parent == mySpread)  continue;
```

В случае, если в публикации используется обычный способ создания разворотов (Facing pages), нужно предусмотреть один момент. Дело в том, что перенос объектов с монтажных столов тех разворотов, на которых имеется только одна страница (для Facing pages — первая) чреват некорректным расположением объектов: вместо размещения на новом монтажном столе они попадают на саму страницу. Чтобы избежать этого, будем перед началом перемещения координаты перемещаемого объекта запоминать, а после сдвига — устанавливать в корректную позицию.

Естественно, нужно предусмотреть и случай, если по каким-либо причинам кандидат на перемещение был заблокирован (`locked`). Варианта всего два — либо его пропускать, либо снимать блокировку и, как остальные, перемещать. Скрипт заблокированные объекты не трогает — это логично, ведь для блокировки объектов наверняка имелись веские причины. С методом `move()` мы уже знакомы.

4.14. Спуск полос

После того как мы научились работать с монтажными столами и страницами, попробуем свои силы в более серьезном — спуске полос. Скрипт позволяет автоматизировать достаточно рутинную и в то же время очень ответственную работу по перетасовке страниц таким образом, чтобы при сложении книжкой они создавали правильные развороты. Работа хоть и простая, но требует большой внимательности, поскольку при ее выполнении, как редко где еще, проявляется пресловутый человеческий фактор, который потенциально способен привести к различным, в том числе финансовым, потерям. Собственно, именно на таких задачах, полностью механических, не требующих никакого вмешательства человека, польза от скриптов проявляется в максимальной степени.

Скрипт выполняет простейший листоподбор (рис. 4.4), размещая попарно "последнюю — первую", "вторую — предпоследнюю" страницы и т. д. В принципе, никаких трудностей с созданием более сложных раскладок быть не должно. Разумеется, перед запуском нужно проверить распашные блоки — желательно, чтобы их не было. Если они все же присутствуют, можно воспользоваться универсальным вариантом — сохранением публикации в формате PDF *(см. разд. 4.8.1)* с последующим постраничным импортом в публикацию *(см. разд. 9.3)*, после чего можно совершенно безопасно запускать скрипт: в таком случае сохранение в PDF обезопасит вас от сюрпризов элементов, выходящих на соседнюю страницу.

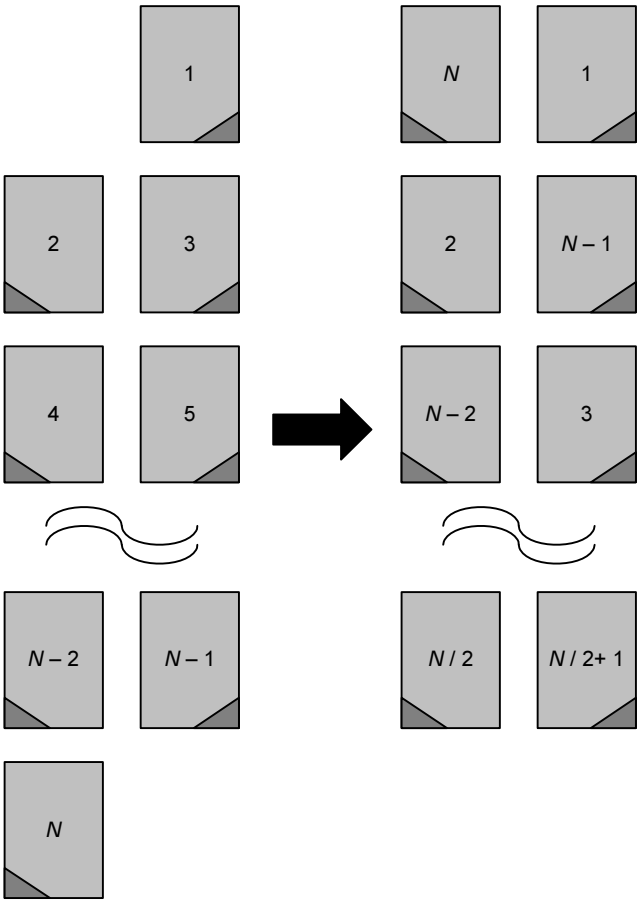


Рис. 4.4. Расположение страниц до и после листоподбора

Итак, рассмотрим скрипт (листинг 4.29).

Листинг 4.29. Выполнение листоподбора

```
myDoc = app.activeDocument
myPages = myDoc.pages;
if (myPages.length % 4 != 0) {
  alert("Количество страниц в документе не кратно четырем!");
  exit();
}
if (myDoc.sections.length > 1) {
  alert("В документе не должно быть дополнительных секций");
  exit();
}
```

```
app.scriptPreferences.userInteractionLevel = ↳
  UserInteractionLevels.neverInteract;
myName = myDoc.name;
myTN = myDoc.filePath + "/" + myName.split(".indd")[0] + "_booklet" + ↳
  ".indt";
myDoc.save(File(myTN),true);
myDoc.close();
app.open(File(myTN));
app.scriptPreferences.userInteractionLevel = ↳
  UserInteractionLevels.interactWithAll;

myDoc.documentPreferences.allowPageShuffle = false;

for (i=0; myPages.length > i; i++) {
  origSection = myPages[i].appliedSection;
  origNumbering = origSection.pageNumberStyle;
  origSection.pageNumberStyle = PageNumberStyle.arabic;
  sectName = origSection.name;
  pageName = myPages[i].name;

  if (sectName != ""){
    pageName = pageName.split(sectName)[1];
    pageNo = pageName.toInteger();
  }

  newSection = myDoc.sections.add({pageStart:myDoc.pages[i], ↳
    continueNumbering:false, pageNumberStart:pageNo, ↳
    pageNumberStyle:origNumbering, marker: origSection.marker, name: ↳
    origSection.name});
  origSection.pageNumberStyle = origNumbering;
}
myDocument.documentPreferences.allowPageShuffle = true;

for (i=0; (myPages.length/2)>i; i++){
  if (i % 2==0)  {
    myDoc.pages[myPages.length - 1].move(LocationOptions.before, ↳
      myDoc.pages[i*2], BindingOptions.leftAlign);
  } else {
    myDoc.pages[myPages.length - 1].move(LocationOptions.after, ↳
      myDoc.pages[i*2], BindingOptions.rightAlign);
  }
}
```

Сначала проверяем количество страниц в публикации. Если оно не кратно четырем (это необходимое условие для получения полных двусторонних разворотов), выводим предупреждение и останавливаем скрипт:

```
if (myPages.length % 4 != 0) exit()
```

Если количество страниц корректное, проверяем, чтобы в публикации не было секций. Для чего это нужно? Как известно, существует единственный способ создания буклета с сохранением нумерации в самой публикации, не прибегая к переводу в PDF и другим альтернативным вариантам. Состоит он в установке начала секции для каждой страницы публикации — только в таком случае ее номер не будет меняться при дальнейшем изменении ее положения в документе. А теперь представьте ситуацию, что в публикации секции уже есть: как в таком случае быть? Кому отдавать предпочтение? Застрахуемся во избежание подобных ситуаций, предоставляя пользователю право самому решить, как поступить в данном случае.

Итак, если все проверки пройдены успешно, приступаем к основным действиям. Поскольку в реальной работе случаи встречаются различные, чтобы скрипт работал на полном автомате, не отвлекаясь на выдачу предупреждений и других диалоговых окон, требующих реакции пользователя, запретим InDesign выводить любые предупреждения

```
app.scriptPreferences.userInteractionLevel = ↳
  UserInteractionLevels.neverInteract;
```

Во избежание возможных накладок (о них нужно помнить всегда — они бывают более или менее проблемными, но предупредить их по максимуму — в ваших же интересах), сохраним документ под именем, отображающим то, что в публикации выполнен спуск полос — добавим к названию "_booklet" и изменим расширение на indt. Сохранение ее в виде шаблона (второй параметр метода `save()` — `true`) предохраняет файл от случайной перезаписи, поскольку InDesign всегда открывает только копии шаблона, что гарантирует сохранность исходного макета.

Для получения названия файла используем знакомый метод — `split()`, указывая в качестве разделителя расширение публикации InDesign (indd). После предварительных действий приступаем к работе. Создаем новый документ на базе только что созданного шаблона, включаем возможность отображать диалоговые окна.

```
userInteractionLevel = UserInteractionLevels.interactWithAll;
```

Перекладываем на язык JavaScript все те действия, которые нужно проделать вручную для спуска полос — в нашем случае потребуется каждую страницу начинать с новой секции. При стандартных настройках программы в процессе добавления секций страницы будут сдвигаться на чужие развороты, что сделает невозможным правильное определение их номера. Во избежание такого развития событий, зафиксируем положение страниц, исключив тем са-

мым возможность ухода страницы со своего разворота и гарантируя сохранение прежней нумерации. Данная операция в InDesign имеет название **shuffle** (палитра **Pages**).

```
allowPageShuffle = false;
```

Затем в цикле пробегаем по очереди все страницы, определяя параметры каждой с учетом ее положения во вновь созданной секции `origSection` (формат номера и др.).

Свойство `name` (не путайте с `offset`!) страницы или секции — не что иное, как ее название, а не просто порядковый номер.

Итак, представим ситуацию, что какой-то странице мы новую секцию уже добавили. В таком случае все последующие страницы изменят свои названия — первым станет название секции, потом разделитель и, наконец, собственно номер страницы в данной секции. Таким образом, для получения номера секции из названия нужно вычленить номер страницы. Выполним это уже хорошо известным способом — через метод `split()`. В качестве разделителя используем название секции, в таком случае номер страницы будет иметь индекс 1 (индекс 0 — у самого названия секции):

```
pageName = pageName.split(sectName)[1];
```

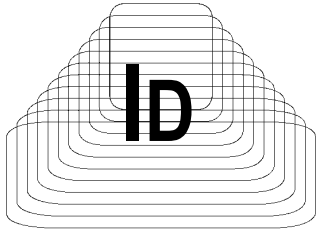
Результатом разделения строки будет массив из строк. Поскольку номера страниц обязаны быть числами, принудительно изменяем тип переменной, в результате чего строка станет трактоваться как число:

```
pageName.split(sectName)[1].toInteger ();
```

Необходимая информация для создания секций собрана, настало время их расставлять. Выполняется это традиционным способом — через метод `add()`, передавая новосозданному объекту необходимые параметры. При этом никакого сдвига страниц со своих разворотов не происходит, поскольку мы предварительно об этом позаботились. В принципе, после расстановки секций нужно восстановить нормальный режим страниц (`allowPageShuffle = true`), но с учетом того, что в макет больше никаких изменений, затрагивающих изменение порядка и количества страниц, вноситься не будет (во всяком случае, это будет очень нелогично, т. к. листоподбор, как правило, — финальная операция), мы ее выполнять не будем.

Итак, на данном этапе у нас в публикации секций ровно столько, сколько страниц — каждая страница является началом новой секции. Осталось выполнить последний шаг — переставить их в нужном порядке. Четные страницы (у них остаток от деления на 2 равен нулю) сдвигаем влево от корешка (`BindingOptions.leftAlign`), нечетные — вправо (`BindingOptions.rightAlign`), соблюдая определенные правила: четные идут с увеличением порядкового номера, нечетные — с уменьшением (`LocationOptions.before`). При этом количество сдвигов должно быть ровно в два раза меньше общего числа страниц.

ГЛАВА 5



Текстовые фреймы

До этого мы рассматривали работу с документом на уровне страниц, не затрагивая вопросы работы с текстом, а это — один из основных видов работ в InDesign. Как известно, в редакторе текст находится внутри контейнеров — текстовых фреймов, т. е. фрейм является родителем для его содержимого (при этом не только текста — во фрейме могут находиться иллюстрации, таблицы, т. е. фактически любой объект страницы). Обращаться к текстовым фреймам, как и к любым другим объектам, можно двумя путями: если он выделен, то непосредственно через `selection[0]`, либо через обращение к родителю, если точка вставки находится в тексте.

Иерархические связи между текстовыми объектами публикации представлены на рис. 5.1.

На вершине иерархической лестницы, как и положено, — сам документ. Его содержимое может рассматриваться либо как набор отдельных материалов (`stories`), либо как набор страниц (разворотов), на которых расположены текстовые фреймы. То есть нужно четко понимать, что фрейм принадлежит странице, а вот его содержимое — материалу. Это разные ветки объектной модели.

Материал может (но не обязан) иметь текстовую часть (кроме текста, в нем могут быть таблицы, объекты in-line, но это уже не `texts`); она же присутствует в любом фрейме. Текстовая часть может рассматриваться, в зависимости от потребностей, как набор абзацев, строк, слов или отдельных символов (`InsertionPoint` здесь приведен лишь для демонстрации положения, занимаемого в иерархии местом установки курсора). Причем они могут рассматриваться как независимые объекты (например, слово № 9876 в материале), так и зависимые (абзац 123, и в нем выбранное слово имеет уже номер 567). Аналогичная картина и со всеми остальными текстовыми объектами.

Соответственно обращение может быть как полным:

(`texts[0].paragraphs[myparagraph]. ... .characters[myCharacter_1]`),

так и сокращенным (`characters[myCharacter_2]`).

`textFrames[0].parent` → `page (document)`

`textFrames[0].texts[0].parent` → `story`

Очень часто используется быстрый переход с отдельного фрейма на весь материал — `parentStory`.

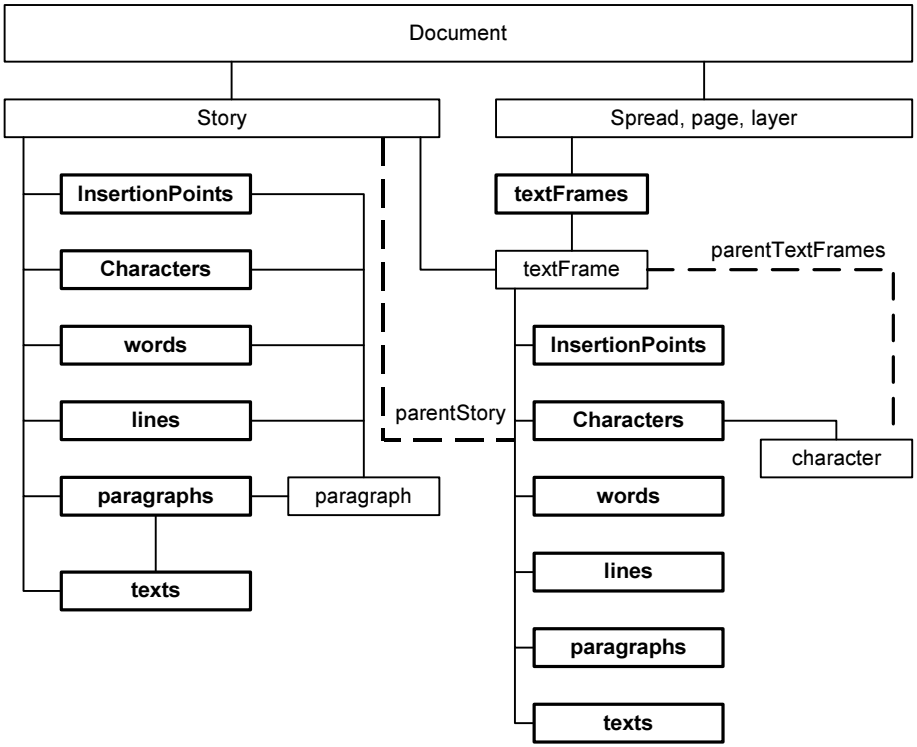


Рис. 5.1. Иерархические связи между текстовыми объектами

На самом низком уровне иерархии стоят объекты "символ" (`character`) и "точка вставки" (`insertionPoint`). Единственное отличие между ними в том, что точек вставки всегда на одну больше (`insertionPoints[0]` является ссылкой на самое начало родительского объекта, т. е. точка *до* первого символа). Например, `myStory.insertionPoints[0]` ссылается на самое начало материала, `myTextFrame.insertionPoints[0]` — на начало текстового фрейма и т. д. Объект `insertionPoint` незаменим при любых операциях, в которых происхо-

дит вставка чего-либо куда-либо: это касается как текста, так и заякоренных фреймов, таблиц и т. п.

К объектам коллекций `paragraphs`, `words`, `characters`, `lines`, `insertionPoints` можно обращаться напрямую, минуя объект более общего типа `text`, который служит исключительно для указания, что работа происходит только с текстовым содержимым фрейма.

Упрощенный вариант обращения к текстовым объектам изображен на рис. 5.2.

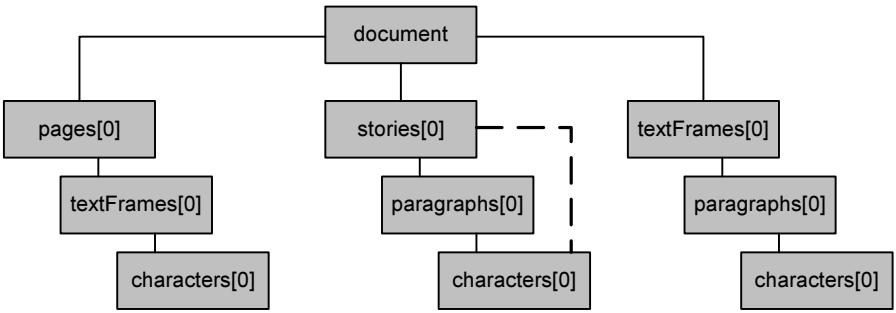


Рис. 5.2. Четыре варианта обращения к различным текстовым объектам

Как видно из рис. 5.2, в InDesign можно использовать любой из поддерживаемых способов обращения к текстовым объектам — а их четыре. Главное — выбрать тот, который в наибольшей степени удовлетворяет вашим задачам. Например, если вы работаете только с конкретной страницей, проще всего ограничить сферу действия, используя обращение `with(myPage)`. Если же действия происходят над всем содержимым документа, логично использовать `with(document)`. Подобная гибкость позволяет выбрать наиболее оптимальный вариант в каждом конкретном случае.

Часто можно обойтись без использования фреймов. Например, если вы форматируете текст в целом материале или в некоторой выделенной области, то достаточно указать на соответствующий объект (например, `anyObject.parentStory` или `selection[0]`) и работать в его пределах. `TextFrame`, как правило, используется либо для позиционирования фрейма на странице, либо вставке одного фрейма в другой.

Поскольку методы работы с текстом существенно отличаются от методов работы с другими объектами, считается хорошим тоном вначале каждого скрипта выполнять проверку типа объекта. С учетом постоянного использования такой проверки рационально выполнить ее в виде функции и сохранить в файле библиотеки. В таком случае вначале любого скрипта достаточно бу-

дет вставить ссылку на библиотечный файл и в дальнейшем использовать его функции.

Поскольку заранее тип объекта нам неизвестен, поступим следующим образом: создадим новый метод для любого объекта InDesign (листинг 5.1).

Листинг 5.1. Функция проверки объекта на принадлежность той или иной коллекции

```
Object.prototype.isText = function() {
  switch(this.constructor.name){
    case "InsertionPoint":
    case "Character":
    case "Word":
    case "TextStyleRange":
    case "Line":
    case "Paragraph":
    case "TextColumn":
    case "Text":
    case "Story":
      return true;
      alert("Выбран текст")
      break;
    case "TextFrame":
      alert("Выбран текстовый фрейм")
      break;
    default :
      return false;
      alert("Выбран не текстовый объект")
  }
}
```

`Object` — это глобальный тип объекта, не привязанный ни к какому конкретному типу.

Свойство `prototype` позволяет добавлять любому типу объекта новые методы и свойства. Мы добавляем к методам, существующим у любого объекта, новый (`isText()`), который определяем как функцию (`= function()`).

`this` — это ссылка на текущий объект, т. е. тот объект, к которому он в каждый момент применяется. Если выделенным будет прямоугольник, то вместо `this` будет использоваться `rectangular`, вместо абзаца — `paragraph` и т. д.

А вот как данную проверку можно использовать в скрипте — проверяем, является ли область выделения текстом (листинг 5.2).

Листинг 5.2. Применение функции проверки

```
if (!app.selection[0].isText()) {
  alert("Выделенное не текст!")
} else {
  // Код обработки для текста
}
```

5.1. Поиск текстового фрейма и страницы

Поскольку работа с текстом — одна из наиболее эффективных сфер применения скриптинга, а текст всегда находится во фрейме, рассмотрим способы обращения к фрейму. Первым может быть `someDocument.myTextFrame` и, в более узком контексте, `someDocument.somePage.myTextFrame`. Если не стоит задача обратиться к конкретному объекту, в цикле просматривают соответствующую коллекцию, например, `myDocument.TextFrames` и, таким образом, получают доступ ко всем фреймам, входящим в нее. Но как перейти только на какой-то конкретный фрейм `myTextFrame`? Для этого должна быть какая-то зацепка, какая-то отличительная особенность, по которой мы бы смогли этот фрейм отделить от других таких же, находящихся на той же странице или в целом документе. Такой зацепкой может стать либо какое-то уникальное его свойство (например, цвет, ширина, наличие окантовки или еще что-то), но, как правило, на практике это слабо применимо, поскольку фрейм может вообще никак не отличаться от своих собратьев. Единственным выходом может быть его выделение курсором. Тут возможны два варианта: когда выделен непосредственно фрейм либо все или часть его содержимого. В любом из двух последних случаев для доступа к фрейму удобно использовать свойство `parentTextFrames`. В принципе, разработчики могли бы использовать обозначение `parentTextFrame` (без окончания `"s"`) по аналогии с `parentStory`, однако так сделано из-за особенностей объекта `TextFrame`.

Представьте ситуацию, что вы работаете с таблицей, занимающей несколько страниц — в этом случае родительским фреймом будет не один какой-то конкретный фрейм, а все, на которых эта таблица расположилась (т. е. массив фреймов). Аналогичная ситуация и с областью выделения — вы можете выбрать фрагмент текста в `Story` неограниченной длины. Соответственно, родительских фреймов будет также несколько. Таким образом, логика в названии свойства `parentTextFrames` не нарушена.

В простейшем случае, если курсор стоит в тексте, коллекция `parentTextFrames` состоит всего из одного объекта, поэтому для получения ссылки на него достаточно использовать конструкцию

```
currentTextFrame = app.activeDocument.selection[0].parentTextFrames[0]
```

Через нее можно легко найти номер страницы, содержащий фрейм:

```
currentPage = currentTextFrame.parent.name
```

Однако существует еще несколько вариантов определения номера. Самый простой из них вообще никак не связан с положением точки вставки в тексте, поскольку использует непосредственное обращение к странице, которая на данный момент активна (ее номер отображается в окне для перехода на страницы в нижней строке публикации). Метод заключается в использовании объекта `activeWindow`: среди его свойств есть указатель на текущую страницу — `activePage`, у которого, в свою очередь, имеется свойство `name`, являющееся ее номером:

```
curr_page = app.activeWindow.activePage.name
```

С одной стороны, такой подход, безусловно, проще, но с другой, всегда ли гарантирует он нужный результат? Предположим, в документе установлена низкая степень увеличения, при этом на экран будет выводиться сразу несколько страниц — в таком случае активной может быть любая, в зависимости от расположения линии их раздела на экране. Поэтому универсальным все же будет первый описанный способ. Вообще же приведенный пример является хорошей иллюстрацией того, насколько внимательно нужно пользоваться теми или иными объектами. Ведь, воспользовавшись вторым способом, мы получали бы не всегда верные данные и, в случае их использования для дальнейших преобразований, последствия такого непродуманного до конца решения могли бы быть самыми разными.

5.2. Эти неуловимые абзацы

С чем чаще всего приходится иметь дело в верстке? Думаю, вряд ли кто-то будет сильно возражать против того, что наиболее частая операция — работа с абзацами и отдельными символами: форматирование текста всегда занимает львиную долю времени всей верстки. Давайте разберемся, как реализовать основные возможности программы по форматированию.

Для того чтобы найти номер абзаца, в котором стоит курсор, логично воспользоваться свойством `paragraphs`. Однако если мы устроим проверку, результат будет на первый взгляд странным. Что `mySelection[0].paragraphs[0].index`, что `mySelection[0].index`[1] — обе ссылки ведут к одному и тому же объекту. Это свидетельствует о том, что объект `paragraph` в чистом виде в объектной модели InDesign не содержится — он определяется исключительно по наличию знака абзаца в тексте. Иными словами, InDesign под-

[1] Свойство для получения порядкового номера конкретного объекта в коллекции.

считывает, сколько ему встретилось символов абзаца с начала текста до заданного места, и только после этого определяет порядковый номер абзаца. Поэтому непосредственное задание порядкового номера абзаца — малопроизводительный и времязатратный процесс.

Для облегчения работы предусмотрен простой и эффективный способ, заключающийся в использовании конструкции `mySelection.paragraphs[0]` как базовой и связанных с нею методов *относительной* адресации: `lastItem()`, `firstItem()`, `nextItem()`, `previousItem()`. В большинстве случаев такой подход проще, чем получение абсолютного порядкового номера абзаца.

Однако для того чтобы иметь полный и удобный способ контроля за публикацией (вместо использования длинных и плохо читаемых конструкций типа `nextItem(nextItem(nextItem))` — в случае, если, например, нужно обратиться к третьему абзацу после текущего), лучше использовать непосредственную *(абсолютную)* индексацию.

Чтобы найти номер интересующего нас абзаца, поступим так, как это делает InDesign — подсчитаем количество встретившихся нам абзацев до места курсора. Для этого нам сначала потребуется узнать код символа абзаца. Его можно определить несколькими способами. Самый простой — выделить символ и открыть в InDesign палитру **Info**. В среднем ряду в ней отображаются коды выделенных символов, и для абзаца там будет стоять "Unicode: 0xD". Второй вариант — использовать традиционные обозначения для спецсимволов (более подробная информация о них представлена в *разд. П2.6*).

Далее. Давайте посмотрим на текст под не совсем обычным ракурсом. Фактически, его можно представить как цепочку текстовых фрагментов, разделенных разделителями — извиняюсь за каламбур — (в качестве которых можно использовать любой символ — пробел, точку, запятую или, как в нашем случае, символ абзаца). Первый фрагмент отделен от второго первым знаком абзаца, второй от третьего — вторым и т. д. Таким образом, мы можем взять текст от начала материала до точки вставки и разбить его по абзацам. Это реализуется через метод `split()`, который в качестве результата возвращает массив из получившихся текстовых фрагментов. Нам останется только узнать длину полученного массива — поскольку каждый абзац будет храниться в своей ячейке массива. Итак, переходим к самому материалу:

```
mySelection = app.selection[0]
myStory = mySelection.parentStory
```

И поскольку нас интересует его содержимое, то:

```
myStoryContents = myStory.contents
```

Затем получаем блок текста от самого начала текста до точки вставки, для чего используем JavaScript-функцию `substring()`. У нее два параметра — ин-

дексы начального и конечного элементов. Как получить индекс места, где стоит курсор, мы уже рассматривали ранее. Соответственно получаем:

```
mySelection.parentStory.contents.substring(0, mySelection[0].index)
```

Разбиваем строку на абзацы. Метод `split()` имеет единственный параметр — символ, по которому будет разделяться строка (в нашем случае строкой будет весь текст — от начала и до точки вставки). Символ нового абзаца имеет обозначение `'\r'`:

```
mySelection.parentStory.contents.substring(0, ↳
  mySelection.index).split('\r')
```

Последний шаг — определяем длину получившегося массива. С учетом того, что нам нужно определить индекс текущего абзаца, из полученного значения нам следует вычесть 1. В конечном счете получаем:

```
currentParagraphIndex = mySelection.parentStory.contents.substring(0, ↳
  mySelection.index).split('\r').length – 1
```

Таким образом, индекс абзаца можно получить всего одной строчкой. Соответственно обращение к предыдущему абзацу будет иметь вид:

```
previousParagraph = mySelection.paragraphs[currentParagraphIndex-1],
```

а к десятому перед ним

```
nextParagraph = mySelection.paragraphs[currentParagraphIndex+10]
```

и т. д.

5.3. Добавление текста

Для доступа к содержимому текста (не важно, при какой именно операции — добавление, удаление, создание текста) используют свойство фрейма `contents`.

Рассмотрим листинг 5.3.

Листинг 5.3. Добавление фрейма с текстом

```
if(app.documents.length==0)
  app.documents.add();
myDocument = app.documents[0]
firstPage = myDocument.pages[0]
with (firstPage){
  textFrames.add({geometricBounds: Array(0,0,"10 mm","10 mm"), ↳
    contents: "Hello, world!"});
```

```
  with(textFrames[0]){
    if (overflows) {
      fit(fitOptions.FrameToContent)
    }
  }
}
```

Сначала создаем на первой странице текстовый фрейм при помощи уже известной конструкции `add({})`. InDesign ExtendScript позволяет в момент создания нового объекта задать его свойства. В данном случае устанавливаем габаритные размеры фрейма (они определяются двумя точками — левой верхней и правой нижней; соответственно, их нужно передать в виде массива) и заодно вписываем в него наш текст (параметры разделяются запятыми, а назначаются через двоеточие). Габариты указаны в текущих значениях (которые выставлены в InDesign по умолчанию для всех создаваемых документов). Для задания точных значений можно в начале скрипта их переопределить, но в данном случае будет проще указать их в явном виде — "10 mm".

Поскольку мы не знаем параметров стиля по умолчанию, существует вероятность того, что текст полностью разместиться в текстовом фрейме заданных нами размеров не сможет. Поэтому последним шагом мы проверяем свойство переполнения фрейма (`overflows`) и, если так оно и есть, используем очень удобный метод увеличения размеров контейнера по размерам его содержимого `fit(fitOptions.FrameToContent)` (аналогичный метод существует и у изображений).

Если нужно вставить текст в конец какого-то материала, можно использовать следующую конструкцию (листинг 5.4):

```
insertionPoints.items(-1).contents
```

Листинг 5.4. Пример использования свойства `items(-1)`

```
myTextFrame.parentStory.insertionPoints.items(-1).contents = "Привет";
myTextFrame.parentStory.insertionPoints.items(-1).contents = "\r";
myTextFrame.parentStory.insertionPoints.items(-1).contents = "Как дела";
```

Коллекция `insertionPoints` определяет возможные места вставки, а `items(-1)` указывает на последний элемент в коллекции — это более удобно, чем использовать непосредственную индексацию (через `[]`): так, обозначение `items(-1)` в данном случае заменило `myTextFrame.parentStory.insertionPoints.length-1`. В результате работы скрипта получим текст:

```
Привет
Как дела
```

5.4. Замена текста

Замена одного фрагмента текста на другой проводится аналогично операции добавления текста, только в таком случае ранее стоявший меняется на новый. Скрипт из листинга 5.5 иллюстрирует замену одного слова на целую фразу путем замены содержимого соответствующего объекта.

Листинг 5.5. Замена слова фразой

```
// Создаем новый документ
var myDocument = app.documents.add();

// Устанавливаем единицы измерения на пункты
myDocument.viewPreferences.horizontalMeasurementUnits = ↳
  MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = ↳
  MeasurementUnits.points;

// Создаем текстовый фрейм на первой странице.
// Сразу же задаем его размеры и содержание
var myTextFrame = ↳
  myDocument.pages[0].textFrames.add(↳
    {geometricBounds:[72, 72, 288, 288],↳
     contents:"This is some example text."});

// Производим замену третьего слова на целую фразу
myTextFrame.parentStory.words[2].contents = "этот текст заменил ↳
  первоначально находившееся здесь слово";
```

А как происходит замена текста целого абзаца, показано в листинге 5.6.

Листинг 5.6. Замена абзаца

```
// Создаем новый документ
var myDocument = app.documents.add();

// Устанавливаем единицы измерения на пункты
myDocument.viewPreferences.horizontalMeasurementUnits = ↳
  MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = ↳
  MeasurementUnits.points;
```

```
// Создаем текстовый фрейм на первой странице
var myTextFrame = ↳
  myDocument.pages[0].textFrames.add(↳
  {geometricBounds:[72, 72, 288, 288],↳
   contents:"Paragraph 1.\rParagraph 2.\rParagraph 3.\r"});
```

Заменим содержимое второго абзаца.

Будьте внимательны: подобные операции нужно выполнять очень корректно. Дело в том, что признак абзаца (символ новой строки, `\r`) принадлежит абзацу, поэтому, если мы удалим его, потеряется форматирование. И на будущее: если вам предстоит удалить любой объект, помните, что количество объектов в соответствующей коллекции уменьшается на единицу, что нужно самым внимательным образом отслеживать, иначе неминуемы конфликты с InDesign.

```
var myStory = myTextFrame.parentStory
var myParagraph = myStory.paragraphs[1]

// Чтобы удалить содержимое строки без последнего символа,
// используем метод itemByRange:
var myStartCharacter = myParagraph.characters[0];
var myEndCharacter = myParagraph.characters[-2];
myTextFrame.texts.itemByRange(myStartCharacter, ↳
  myEndCharacter).contents = "Новое содержимое второго абзаца"
```

Почему мы вычли два символа, ведь перевод строки в процессе верстки рассматривается как один? На самом деле, давайте посмотрим на ситуацию с другой стороны. Для задания абзацев несколькими строчками ранее мы использовали комбинацию `\r` — а ведь это как раз и есть два символа, поэтому вполне логично вычитание именно такого количества символов.

5.5. Импорт текста

Редко когда в публикацию помещается совсем неформатированный текст — как правило, он сначала проходит определенную первичную обработку, в том числе и форматирование, а уже в публикации проводятся лишь точечные исправления. В такой ситуации без импорта документов (метод `place()`) из текстовых редакторов не обойтись. Рассмотрим вопрос размещения в публикации содержимого одного из таких частично подготовленных документов.

Метод `place()` при использовании к странице имеет следующие параметры:

```
place(File[, PlacePoint] [, DestinationLayer] [, ShowingOptions]
[, Autoflowing])
```

Здесь:

- ***File*** — ссылка на импортируемый файл;
- ***PlacePoint*** — массив координат по осям *x*, *y*;
- ***DestinationLayer*** — при необходимости — определенный слой;
- ***ShowingOptions*** — отображать окно с параметрами импорта или нет;
- ***Autoflowing*** — использовать автоматическое добавление необходимого количества текстовых фреймов для размещения всего содержимого импортируемого документа или ограничиться только текущей страницей.

Рассмотрим скрипт из листинга 5.7.

Листинг 5.7. Подготовка к импорту

```
// Начало - традиционное
var myDocument = app.documents.add();
myDocument.viewPreferences.horizontalMeasurementUnits = ↳
  MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = ↳
  MeasurementUnits.points;

// Установка координат, в которые будет помещаться текст
// из импортированного документа
var myX = myDocument.pages.item(0).marginPreferences.left;
var myY = myDocument.pages.item(0).marginPreferences.top;
```

Проводим сам импорт (листинг 5.8).

Листинг 5.8. Импорт текста с заданием положения фрейма

```
var myStory = myDocument.pages[0].place(File("/c/test.txt"),↳
  [myX, myY], undefined, false, true)[0];
```

Замечание по поводу установки массива координат: если в качестве места вставки выбрана определенная колонка уже существующего фрейма, то позицию по горизонтали задавать не нужно — ширина для импортируемого текста будет ограничена шириной этой колонки.

А вот как происходит импорт документа в уже существующий фрейм (листинг 5.9).

Листинг 5.9. Импорт документа в уже существующий фрейм

```
var myDocument = app.documents.add();
myDocument.viewPreferences.horizontalMeasurementUnits = ↳
  MeasurementUnits.points;
myDocument.viewPreferences.verticalMeasurementUnits = ↳
  MeasurementUnits.points;

// Создаем фрейм назначения
var myTextFrame = ↳
  myDocument.pages.item(0).textFrames.add({geometricBounds: ↳
  myGetBounds(myDocument, myDocument.pages.item(0))});
```

Параметры метода `place()` при использовании к фрейму несколько отличаются:

- ***File*** — ссылка на импортируемый файл;
- ***ShowingOptions*** — отображать окно с параметрами импорта или нет.

Например:

```
// Выполняем импорт
myTextFrame.place(File("/c/test.txt"));
```

Пример из листинга 5.10 иллюстрирует импорт документа в определенное место в тексте (без замены текста).

Листинг 5.10. Импорт документа в конец текста

```
var myDocument = app.documents.add();
// Создаем фрейм назначения
var myTextFrame = ↳
  myDocument.pages.item(0).textFrames.add({geometricBounds: ↳
  myGetBounds(myDocument, myDocument.pages.item(0)), contents:"Inserted ↳
  text file follows:\r"});
```

Параметры метода `place()`, применяемого к `InsertionPoint`:

- ***File*** — ссылка на импортируемый файл;
- ***ShowingOptions*** — отображать окно с параметрами импорта или нет.

Например:

```
// Выполняем импорт, добавляя импортированный документ
// в конец имеющегося текста
myTextFrame.parentStory.insertionPoints[-1].place(File("/c/test.txt"));
```

Как известно, при вставке содержимого документов некоторых типов пользователь может задать опции импорта. Точно так же их можно указать в скрипте — при этом нужно использовать соответствующий объект `importPreferences`. Рассмотрим, как это можно сделать (листинг 5.11).

Листинг 5.11. Установка настроек для импорта

```
with(app.textImportPreferences){
  // Опции раскладки клавиатуры:
  TextImportCharacterSet.ansi
  TextImportCharacterSet.chineseBig5
  TextImportCharacterSet.gb18030
  TextImportCharacterSet.gb2312
  TextImportCharacterSet.ksc5601
  TextImportCharacterSet.macintoshCE
  TextImportCharacterSet.macintoshCyrillic
  TextImportCharacterSet.macintoshGreek
  TextImportCharacterSet.macintoshTurkish
  TextImportCharacterSet.recommendShiftJIS83pv
  TextImportCharacterSet.shiftJIS90ms
  TextImportCharacterSet.shiftJIS90pv
  TextImportCharacterSet.unicode
  TextImportCharacterSet.windowsBaltic
  TextImportCharacterSet.windowsCE
  TextImportCharacterSet.windowsCyrillic
  TextImportCharacterSet.windowsEE
  TextImportCharacterSet.windowsGreek
  TextImportCharacterSet.unicode;

  convertSpacesIntoTabs = true;
  spacesIntoTabsCount = 3;

  // Задается язык (как строка). Список отображает возможности
  // InDesign Creative Suite 3 — в предыдущих версиях он гораздо короче
  Bulgarian
  Catalan
  Croatian
  Czech
  Danish
  Dutch
  English:Canadian
  English:UK
  English:USA
```

```
  English:USA Legal
  English:USA Medical
  Estonian
  Finnish
  French
  French:Canadian
  German:Reformed
  German:Swiss
  German:Traditional
  Greek
  Hungarian
  Italian
  Latvian
  Lithuanian
  Neutral
  Norwegian:Bokmal
  Norwegian:Nynorsk
  Polish
  Portuguese
  Portuguese:Brazilian
  Romanian
  Russian
  Slovak
  Slovenian
  Spanish:Castilian
  Swedish
  Turkishdictionary = "English:USA";

  // Платформа
  ImportPlatform.macintosh
  ImportPlatform.pcplatform = ImportPlatform.macintosh;

  // Дополнительные настройки, касаются удаления лишних знаков абзаца
  stripReturnsBetweenLines = true;
  stripReturnsBetweenParagraphs = true;

  // И кавычек
  useTypographersQuotes = true;
}
```

Для импорта документов в формате RTF (и Word в том числе) предназначен отдельный объект `wordRTFImportPreferences` (листинг 5.12).

Листинг 5.12. Импорт документов на базе формата RTF

```
with(app.wordRTFImportPreferences){
  ConvertPageBreaks.columnBreak
  ConvertPageBreaks.none
  ConvertPageBreaks.pageBreakconvertPageBreaks = ConvertPageBreaks.none;

  // Свойства, касающиеся преобразования таблиц:
  ConvertTablesOptions.unformattedTabbedText
  ConvertTablesOptions.unformattedTableconvertTablesTo = ↳
    onvertTablesOptions.unformattedTable;

  importEndnotes = true;importFootnotes = true;
  importIndex = true;
  importTOC = true;
  importUnusedStyles = false;
  preserveGraphics = false;
  preserveLocalOverrides = false;
  preserveTrackChanges = false;
  removeFormatting = false;

  resolveCharacterSytleClash and resolveParagraphStyleClash ↳
    properties can be:
    ResolveStyleClash.resolveClashAutoRename
    ResolveStyleClash.resolveClashUseExisting
    ResolveStyleClash.resolveClashUseNewresolveCharacterStyleClash = ↳
      ResolveStyleClash.resolveClashUseExisting;
    resolveParagraphStyleClash = ↳
      ResolveStyleClash.resolveClashUseExisting;
    useTypographersQuotes = true;
}
```

5.6. Вставка специальных символов

Специальные символы — это символы, которые предназначены для особых целей. Среди них — служащие разделителями: например, символ невидимого переноса, шпация, неразрывный пробел. Также они используются для ввода редко используемых знаков: квадратный корень, стрелка, знак ½ и т. п.

В InDesign существует несколько способов ввода спецсимволов:

- через специальные знаки (но этот вариант имеет ограничения, поскольку специальных знаков всего несколько: например, комбинации клавиш <Alt>+<Ctrl>+<->, <Alt>+<Ctrl>+<X> и т. д.),

- используя возможности JavaScript (в нем предусмотрена комбинация `\unnnn`, где *nnnn* — код символа в таблице Unicode);
- через специальный объект `SpecialCharacters`, у которого есть свойства, задающие тот или иной специальный знак.

Через скриптинг доступны лишь последние два варианта (листинг 5.13). Наиболее универсальный — использование таблицы Unicode (дает доступ абсолютно ко всем символам), однако более читабельный, хотя и имеющий более узкое применение — через объект `SpecialCharacters`: он позволяет ввести лишь те специальные символы, которые используются InDesign (`autoPageNumber`, `sectionSymbol`, `enDash`, `enDash` и др.).

Листинг 5.13. Варианты ввода спецсимволов

```
// Использование специальных символов
myTextFrame.contents = "Зарегистрированный товарный знак: ¬.\rКопирайт:
¬©\rТорговый знак: ,N.\r";

// Используя Unicode:
myTextFrame.parentStory.insertionPoints.items(-1).contents = "Не равно:
\u2260 \r Квадратный корень: \u221A \r Paragraph: \u00B6\r";

// Используя свойства объекта SpecialCharacters:
with (myTextFrame.parentStory.insertionPoints.items(-1))
{
  contents = "Automatic page number marker:";
  contents = SpecialCharacters.autoPageNumber;
  contents = "\r";
  contents = "Section symbol:";
  contents = SpecialCharacters.sectionSymbol;
  contents = "\r";
  contents = "En dash:";
  contents = SpecialCharacters.enDash;
  contents = "\r";
}
```

Простейший способ определить значение кода символа в кодировке Unicode — использовать палитру **Glyphs** в InDesign. Наведите указатель мыши на требуемый символ в палитре, и тут же отобразится его значение. За более подробной информацией о кодировке Unicode можно обратиться по адресу **http://www.unicode.org**.

5.7. Удаление фреймов

Удаление фреймов происходит аналогично удалению любых других объектов публикации — через метод `remove()`. При данной операции содержимое фрейма не удаляется, поэтому следствием может стать переполнение текста. Пример приведен в листинге 5.14.

Листинг 5.14. Удаление только фрейма

```
myStory.myTextFrame.remove()
if (myStory.overflows) alert("При удалении фрейма произошло
переполнение")
```

Для удаления фрейма вместе с его содержимым сначала нужно удалить содержимое, а потом — сам контейнер (листинг 5.15).

Листинг 5.15. Удаление фрейма с содержащимся в нем текстом

```
myStory.myTextFrame.texts[0].remove()
myStory.myTextFrame.remove()
```

Предполагается, что во фрейме находится лишь текстовое содержимое. В противном случае потребуется использовать свойство `contents`:

```
myStory.myTextFrame.contents = ''
myStory.myTextFrame.remove()
```

5.8. Некоторые вопросы импорта документов из MS Office

Казалось бы: импорт документа — что может быть проще? Но это только на первый взгляд. К сожалению, на практике ситуация не такая безмятежная — это особенно касается взаимодействия с Excel. Довольно часто авторам статей при создании диаграмм не хватает его возможностей по оформлению, поэтому они используют собственноручно созданные надписи и подписи. Все бы ничего, если бы не одно обстоятельство. Excel — достаточно интеллектуальный пакет, и диаграмма со всеми ее выносными элементами при экспорте/импорте рассматривается как группа объектов, которая, соответственно, ведет себя как единый объект. То же самое касается и дополнительных элементов, созданных пользователем — но только в этом случае, если сначала был выделен любой элемент, входящий в какую-либо группу — все создаваемые потом элементы автоматически будут входить в эту группу. Таким

образом, по окончании внесения дополнений никакой лишней работы выполнять не нужно — при импорте документа Word с диаграммами в InDesign все диаграммы рассматриваются как отдельные цельные объекты, с которыми легко и просто работать.

Однако авторы не всегда знакомы с особенностями работы Excel, в результате чего после импорта статьи в InDesign мы получаем, кроме самой статьи, массу заякоренных объектов, в которые были превращены не включенные в группы пользовательские надписи из Excel. В особо тяжелых случаях надписей так много, что разобрать на странице, в которой несколько диаграмм, не говоря уже обо всей публикации, практически ничего невозможно. Удалять эти блоки вручную — занятие, согласитесь, также неблагодарное. Поэтому было решено написать скрипт, который бы проводил "зачистку" публикации от следов деятельности подобных авторов (листинг 5.16).

Листинг 5.16. Удаление импортированных блоков

```
mySelection = app.selection[0]

// Переходим на сам импортированный материал
myStory = mySelection.parentStory

mytextFrames = myStory.textFrames
for (j=0; j< mytextFrames.length; j++) {
  // Просматриваем текстовые блоки на предмет наличия в них вложенных
  // текстовых блоков — следов от некорректно созданных подписей
  for (k=0; k< mytextFrames[j].textFrames.length; k++){
    mytextFrames [j].textFrames[k].remove();
    k--
  }
}
```

Поскольку каждая надпись импортируется в виде заякоренного текстового фрейма, следует удалить все текстовые блоки, которые входят в состав других фреймов.

Еще один нюанс, который не связан со скриптингом непосредственно, но не удержусь, чтобы не поделиться. Некоторые авторы вопреки требованиям подачи материалов вставляют изображения (в том числе полноцветные растровые) в текст документа, не предоставляя их отдельно. Копирование через системный буфер в данном случае не поможет — изображения вставляются в урезанной палитре (256 цветов, поэтому практического толка при работе с фотоизображениями никакого).

"Вытащить" фотографии тяжело, но можно. Проверенный вариант — копирование содержимого экрана (клавиша <PrintScreen>). Для получения наивысшего качества сначала восстанавливают исходные размеры изображения (обычно авторы их сжимают), потом фотография размещается на экране так, чтобы она занимала как можно большую площадь, и берется снимок экрана, который потом кадрируется по требуемым размерам.

5.9. Связывание текстовых фреймов

Материал, размещенный в публикации, очень часто размещается более чем в одном фрейме. Для связи текстовых фреймов между собой в InDesign предусмотрены свойства `previousTextFrame` и `nextTextFrame`, позволяющие обратиться к его соседям (любой фрейм имеет эти свойства, но если фрейм одиночный, у него `previousTextFrame=null` и `nextTextFrame=null`). Также эти свойства активно используются для перемещения между объектами, определения последнего фрейма в материале (у него также `nextTextFrame=null`).

С учетом того что в InDesign также существуют методы `previousItem()`, `nextItem()`, становится непонятно, зачем были введены дополнительные свойства — ведь с успехом можно было бы обходиться без них.

Чтобы соединить между собой два фрейма, достаточно использовать такую конструкцию:

```
myTextFrameA.nextTextFrame = myTextFrameB
```

Либо, что совершенно одинаково,

```
myTextFrameC.previousTextFrame = myTextFrameB;
```

Для разрыва существующих связей достаточно сбросить любой их двух признаков:

```
myTextFrameA.nextTextFrame = NothingEnum.nothing;
```

Если использовать такой подход напрямую, то все последующие фреймы из цепочки останутся без текста. Значит, понадобится повторное восстановление связей. Давайте попробуем найти более простой путь. Предположим, мы удалили фрейм. При этом связь к удаленному объекту автоматически перестроится на следующий, что нам и нужно. При этом текст из удаленного фрейма будет вытолкнут на следующий фрейм и материал "поплывет". Чтобы такого не произошло, сначала в удаляемом фрейме сотрем текст, в результате чего объем текста в материале сократится ровно на необходимое количество знаков и после удаления фрейма весь текст останется на своем месте. Чтобы текст не потерялся — он должен сохраниться в целостности и невредимости — мы удаляемый фрейм дублируем и, таким образом, ничего не потеряем, и задачу выполним.

Итак, фактически нам нужно выполнить всего четыре шага в такой последовательности (листинг 5.17):

1. Убедиться, что выделен именно текстовый фрейм.
2. Продублировать фрейм, который впоследствии будет удален.
3. Удалить текст из удаляемого фрейма.
4. Удалить сам фрейм.

Ранее мы уже проводили проверку типа выделения объекта (см. листинг 5.1) и в несколько модернизированном виде применим его в нашем случае. В принципе, такие часто используемые фрагменты кода можно применять как библиотечные для работы с текстовыми объектами: `myText` дает ссылку на выделенный текст, `myTextFrame` — на фрейм, в котором он содержится.

Листинг 5.17. Вычленение активного фрейма из цепочки с сохранением текста

```
var myObjectList = new Array;

// Начальный фрагмент сценария
switch(app.selection[0].constructor.name){
  case "TextFrame":
    myText = [app.selection[0]];
    break;
  default:
    if(app.selection.length == 1){
      switch(app.selection[0].constructor.name){
        case "Text":
        case "InsertionPoint":
        case "Character":
        case "Word":
        case "Line":
        case "TextStyleRange":
        case "Paragraph":
        case "TextColumn":
          myTextFrame = [app.selection[i].parentTextFrames[0]];
          break;
      }
    } else {
      alert ("В выделении нет ни текста, ни фрейма с текстом")
    }
    break;
}
```

```
// Проверяем, чтобы выделенный фрейм не являлся последним в цепочке,
// а потом воспроизводим описанные выше шаги.
if ((myTextFrame.nextFrame != null) && ↳
    (myTextFrame.previousFrame != null))
{
  myNewFrame = myTextFrame.duplicate();
  myTextFrame.texts[0].remove();
  myTextFrame.remove();
}
```

5.10. Поиск переполнения

Поиск переполнения в текстовых фреймах возникает довольно часто, особенно при внесении изменений в уже отформатированный многостраничный текст. Поиск вручную — занятие утомительное, особенно если публикация состоит из множества независимых текстовых цепочек. Посмотрим, как эту задачу можно решить через скриптинг (листинг 5.18).

Листинг 5.18. Поиск переполнения текстовых фреймов (вариант 1)

```
var overflowArray = new Array();
var pages = "";
myDocument = app.activeDocument

if (myDocument.length < 1)
  alert("Ни один документ не открыт!");
else{
  if (myDocument.textFrames.length < 1)
    alert("В публикации нет ни одного текстового фрейма!");
  else  {
    for(i = 0; i < myDocument.pages.length; i++)   {
      for(j = 0; j < myDocument.pages[i].textFrames.length; j++)
      {
        if(myDocument.pages[i].textFrames[j].overflows == true)
          overflowArray.push(myDocument.pages[i].name);
      }
    }
    if(overflowArray.length > 0)  {
      pages = overflowArray.toString()
    }
    alert(Количество переполненных фреймов: " +↳
      overflowArray.length + "\n" + "На страницах: " + pages);
```

```
  } else
    alert("Переполнений нет!")
}
```

Вначале — проверка того, что хотя бы один документ в InDesign открыт (часто такой прием грубо называют "защитой от дурака", что, не смотря на все побочные факторы, прекрасно характеризует его действие):

```
if (myDocument.length < 1)
  alert("Ни одного открытого документа нет!");
```

Следующая проверка — на наличие текста:

```
if (myDocument.textFrames.length < 1)
```

В цикле пробегаем по всем страницам:

```
for(var i = 0; i < myDocument.pages.length; i++)  {
```

и заглядываем в каждый текстовый фрейм на них:

```
for(var j= 0; j<myDocument.pages[i].textFrames.length; j++)
```

В случае переполнения запоминаем номер страницы

```
if(myDocument.pages[i].textFrames[j].overflows)
  overflowArray.push(i);
```

Для удобства отображения номеров страниц с переполнением сводим их в строку:

```
if(overflowArray.length > 0)  {
  pages = overflowArray.join()
```

Что можно сделать и по-другому:

```
pages = overflowArray.toString()
```

И выводим отчет о результатах:

```
alert(Количество переполнений: " +overflowArray.length + ↳
  "\n" + "На страницах: " + pages);
```

В противном случае — выводим соответствующее предупреждение:

```
alert("Переполнений нет!")
```

А вот как можно решить эту же задачу проще и быстрее (листинг 5.19). Как известно, переполнение может возникнуть только в последнем фрейме цепочки, а каждая цепочка на языке InDesign — это отдельный объект `Story`, у которого также есть свойство `overflows`. Таким образом, достаточно пробежаться по всем `Story` (а их, как правило, меньше, чем всех фреймов, значит, скорость выполнения скрипта будет выше) и определить признак переполне-

ния. Если вдруг таковой имеется, найти у каждого материала последний текстовый фрейм и через свойство `parent` выйти на номер содержащей его страницы.

Листинг 5.19. Поиск переполнения текстовых фреймов (вариант 2)

```
var myStories = myDocument.stories
lastFrames = []
for(var i = 0; i < myStories.length; i++)  {
  if(myStories[i].overflows){
    lastFrame = myStories[i].textFrames.lastItem()
    lastFrames.push(lastFrame)
    overflowPageNumber = lastFrame.parent.nane
```

В случае переполнения запоминаем номер страницы, а дальше — все как в предыдущем варианте:

```
    overflowArray.push(overflowPageNumber);
  }
  alert(overflowArray)
```

Для удобства номера страниц, на которых произошло переполнение, лучше вывести в виде раскрывающегося списка (использовать объект `dialog.dropdown`), выбор любого элемента из которого приводит к переходу на текстовый фрейм, вызвавший переполнение. Преимущества такого метода ярко проявляются при плотной верстке, когда материалов на одной странице множество. В таком случае вместо `alert()` добавим несколько срок (листинг 5.20).

Листинг 5.20. Вывод номеров страниц с переполнением в раскрывающийся список

```
dlg = app.dialogs.add({name:" Поиск фреймов с переполнением", ↳
  canCancel:true});
column = dlg.dialogColumns.add();
row = column.dialogRows.add();
row.staticTexts.add({staticLabel:"Выберите страницу, в которой ↳
  переполнение, и нажмите кнопку OK для перехода на нее"});

row = column.dialogRows.add();
dropdown = row.dropdowns.add({minWidth:400, stringList: overflowArray, ↳
  selectedIndex:0});
if (dlg.show() == true)
```

```
{
  // Переход на выбранную страницу
  document.layoutWindows[0].activePage = ↳
    overflowArray[dropdown.selectedIndex];

  // Выделение конкретного фрейма
  document.select(myFrame);
}
```

Таким образом, полный листинг будет выглядеть так, как представлено в листинге 5.21.

Листинг 5.21. Полный текст скрипта, обрабатывающего переполнение фреймов

```
var myStories = app.documents[0].stories
lastFrames = []
overflowArray = []
overflowPages = []
for(var i = 0; i < myStories.length; i++)  {
  if(myStories[i].overflows){
    lastFrame = myStories[i].textFrames.lastItem()
    overflowPage = lastFrame.parent

    lastFrames.push(lastFrame)
    overflowPages.push(overflowPage)
    overflowArray.push(overflowPage.name);
  }
}

dlg = app.dialogs.add({name:" Поиск фреймов с переполнением", ↳
  canCancel:true});
column = dlg.dialogColumns.add();
row = column.dialogRows.add();
row.staticTexts.add({staticLabel:"Выберите страницу, в которой ↳
  переполнение, и нажмите OK для перехода на нее"});

row = column.dialogRows.add();
dropdown = row.dropdowns.add({minWidth:50, stringList:overflowArray, ↳
  selectedIndex:0});

if (dlg.show()){
  document.layoutWindows[0].activePage = ↳
    overflowPages[dropdown.selectedIndex];
```

```
  document.select(lastFrames[dropdown.selectedIndex]);
}
dlg.destroy();
```

Массив `overflowPages` был введен по той причине, что объектом `activePage` может быть только страница (сам объект, а не ее номер), поэтому для удобства пользователя мы отображаем в диалоговом окне (рис. 5.3) номера страниц (из массива `overflowPageNumbers`), а в InDesign передаем сами объекты (из `overflowPages`).

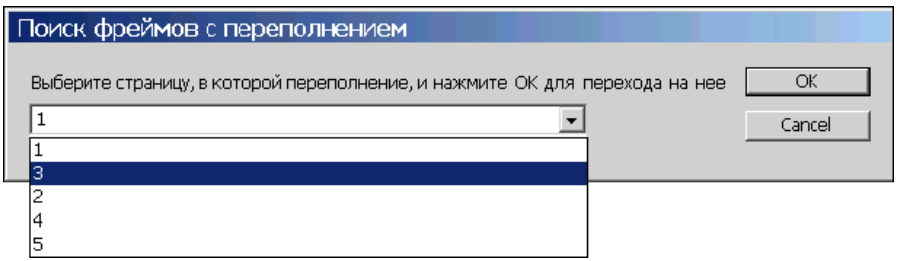


Рис. 5.3. Вот как выглядит диалоговое окно с результатами проверки. Выбор страницы переносит вас на нее, одновременно выделяется фрейм, вызвавший ошибку

5.11. Создание заякоренных фреймов

Заякоренные фреймы отличаются от свободно размещенных тем, что они жестко привязаны к конкретному месту в тексте. Родителем заякоренного фрейма является, таким образом, точка вставки (`insertionPoint`), а сам он ведет себя как обычный символ. Все в полном логическом соответствии: вспомните, как создаются такие фреймы в публикации. Один из вариантов состоит в копировании текстового фрейма в системный буфер, затем, если поместить курсор в текст и содержимое буфера вставить, в этом месте появится заякоренный фрейм. Передвигается он так же, как любой другой символ: при вставке символов он смещается по строке и, при необходимости, переходит на другую строку.

Таким образом, получить ссылку на родителя заякоренного объекта можно так:

```
myInsertionPoint = myAnchoredTextFrame.parent
```

Чтобы создать какой-либо заякоренный объект, нужно сначала определить место в тексте, куда он должен быть вставлен, а затем использовать традиционный метод для добавления элементов `add()`, например:

```
myTextFrame.insertionPoints[0].rectangles.add()
```

или же

```
myTextFrame.paragraphs[3].words[2].rectangles.add()
```

Соответственно для вставки текстового фрейма, заякоренного в месте курсора:

```
myTextFrame.selection[0].textFrames.add()
```

Рассмотрим листинг 5.22.

Листинг 5.22. Создание in-line-фрейма в первом фрейме на первой странице

```
var myDocument = app.documents[0];
var myPage = myDocument.pages[0];
var myTextFrame = myPage.textFrames[0];
var myInlineFrame = ↳
  myTextFrame.paragraphs[0].insertionPoints[0].textFrames.add();
```

Предусмотрены три способа позиционирования заякоренного фрейма: `inlinePosition`, `aboveLine` и `anchored` — все они являются свойствами объекта `AnchorPosition` (листинг 5.23).

Листинг 5.23. Установка типа якоря

```
myInlineFrame.anchoredPosition = AnchorPosition.inlinePosition
```

Если выбран тип `anchored`, то для указания конкретного положения используют любую из 9 точек привязки (`anchorPoint`): `topLeftAnchor`, `topCenterAnchor`, `topRightAnchor`, `leftCenterAnchor`, `centerAnchor`, `rightCenterAnchor`, `bottomLeftAnchor`, `bottomCenterAnchor`, `bottomRightAnchor`, которые, естественно, являются свойствами объекта `AnchorPoint`.

После вставки любого объекта во фрейм (не обязательно заякоренного) необходимо провести обновление содержимого родителя, поскольку вставленный объект воздействует на все его элементы, приводя к их смещению. Для обновления информации о фрейме (эдакий принудительный refresh) предусмотрена операция `recompose()` (листинг 5.24). Ее лучше применять ко всему родительскому фрейму, а не только к текущему абзацу — в таком случае мы сможем всегда гарантировать корректный результат.

Листинг 5.24. Обновления информации о фрейме

```
myTextFrame.texts[0].recompose();

// Получаем габариты нового фрейма
var myBounds = myInlineFrame.geometricBounds;
```

```
// Устанавливаем размеры — допустим, 50 мм по ширине и высоте
myInlineFrame.geometricBounds = [myBounds[0], myBounds[1], ↳
  myBounds[0]+50, myBounds[1]+50];

// И вписываем текст
myInlineFrame.contents = "Это — содержимое фрейма, вставленного в ↳
  текст";

// Теперь создаем заякоренный фрейм — операция ничем не отличается
// от создания обычного in-line-фрейма
var myAnchoredFrame = ↳
  myTextFrame.paragraphs[1].insertionPoints[0].textFrames.add();

// Заякоренный фрейм отличается от in-line-варианта только тем,
// что имеет дополнительный параметр, определяющий его положение
// на странице относительно точки вставки. Устанавливаем параметры
// заякоренного объекта как свойства объекта anchoredObjectSettings:
with(myAnchoredFrame.anchoredObjectSettings){
  anchoredPosition = AnchorPosition.anchored;
  anchorPoint = AnchorPoint.topLeftAnchor;
  horizontalReferencePoint = AnchoredRelativeTo.anchorLocation;
  horizontalAlignment = HorizontalAlignment.leftAlign;
  anchorXoffset = 72;
  verticalReferencePoint = VerticallyRelativeTo.lineBaseline;
  anchorYoffset = 24;
  anchorSpaceAbove = 24;

  // Уже знакомые операции
  myTextFrame.texts[0].recompose;
  var myBounds = myAnchoredFrame.geometricBounds;
  myAnchoredFrame.geometricBounds = [myBounds[0], myBounds[1], ↳
    myBounds[0]+24, myBounds[1]+72];
  myAnchoredFrame.contents = "А это — содержимое заякоренного фрейма";
}
```

5.12. Перемещение объектов

В редакторе предусмотрено несколько вариантов переноса объектов из одного места в другое: традиционный для обычной верстки — через системный буфер и путем непосредственного переноса в место назначения (аналог метода drag-and-drop).

Использовать системный буфер нежелательно — прибегать к нему нужно только в самых крайних случаях, поскольку эта операция более ресурсоемкая

и медленная по сравнению с альтернативным вариантом. Перед занесением в буфер копируемый или переносимый фрагмент сначала нужно выделить (`select()`), выполнить необходимую операцию (`app.copy()`), затем встать в место назначения (`insertionPoint`), после чего вставить из буфера (`app.paste()`).

Альтернативный метод (`move()`) гораздо эффективнее и короче: от вас нужно только одно — задать место назначения и позицию относительно точки вставки.

```
move([to][, by] [, transformingContent])
```

Здесь:

- ***to*** — новое место расположения. Им могут быть: абсолютные значения (массив из значений по двум осям), а также конкретный разворот, страница или слой;
- ***by*** — смещение относительно нового месторасположения (массив из значений по двум осям);
- ***transformingContent*** — признак смещения содержимого вместе с контейнером (`true`) или же сохранение содержимым своего положения неизменным (часто требуется при работе с иллюстрациями). Значение по умолчанию: `true`.

Местом назначения могут выступать объекты следующих типов: `Text`, `Story`, `Cell`, `Row`, `Column`, `Table`, `PageItem`.

Может показаться, что, например, перенос в другой документ или фрейм не предусмотрен. Однако присмотритесь внимательно к объекту `Text` — исходя из объектной модели InDesign, он не привязан к какому-то конкретному месту, это объект общего типа (аналогично объекту `Graphic` для любой графики), но исключительно текстовый. Таким образом, он может быть где угодно (во фрейме, другом документе). Главное, что он имеет текстовый тип, а значит, обладает точками вставки, через которые можно вставлять другие объекты. Следовательно, при перенесении объекта в какой-либо фрейм для задания места назначения нужно указывать не сам фрейм, а, например, определенный абзац или слово в нем. Согласитесь, здесь жесткая логика — ведь вы действительно переносите не просто во фрейм (помните, как в классике: "На деревню, дедушке..."), а в абсолютно конкретное место в этом фрейме.

Среди параметров размещения (`LocationOptions`) всего два варианта: перед точкой вставки (`LocationOptions.BEFORE`) и после нее (`LocationOptions.AFTER`). Другие варианты (`AT_BEGINNING` и `AT_END`) предназначены для работы с группами. Для параметров размещения допускается использование цифровых эквивалентов (их можно найти в полном руководстве по скриптингу от Adobe), что несколько ускоряет операцию.

5.13. Выполнение обтекания

Обтекание — способность текста обходить объекты, расположенные над ним. Все параметры обтекания собраны в свойстве `textWrapOptions`, которое фактически является стилем для выполнения обтекания. Связанные с ним объекты: `contourOption`, `paths`, `textWrapType` и `textWrapOffset`.

`ContourOption` включает в себя глобальные настройки, которые касаются взаимодействия с графическим содержимым и задают источник для контура: маска прозрачности `alphaChannelPathNames`, обтравочный векторный путь `contourPathName` (для созданных не в Photoshop), `photoshopPathNames` (для путей, созданных в Photoshop).

Основной параметр — `contourType` (способ создания формы обтекания), который может иметь одно из следующих свойств:

- `ContourOptionsTypes.boundingBox`;
- `ContourOptionsTypes.photoshopPath`;
- `ContourOptionsTypes.detectEdges`;
- `ContourOptionsTypes.alphaChannel`;
- `ContourOptionsTypes.graphicFrame`;
- `ContourOptionsTypes.sameAsClipping`.

Для обтекания текста текстом возможен единственный вариант — `boundingBox`.

Собственно, сам тип обтекания задается свойствами объекта `textWrapType`:

- `TextWrapTypes.none`;
- `TextWrapTypes.jumpObjectTextWrap`;
- `TextWrapTypes.nextColumnTextWrap`;
- `TextWrapTypes.boundingBoxTextWrap`;
- `TextWrapTypes.contour`;
- `TextWrapTypes.userModified`.

5.14. Создание распашных заголовков

Среди свойств текста в InDesign до сих пор не значится поддержка распашных заголовков[1], хотя это часто требуется при работе с многоколоночным текстом и некоторых программах есть уже давно (например, Ventura).

[1] Заголовки на всю ширину страницы, независимо от количества колонок в тексте.

В принципе, никакой проблемы это не составляет, поскольку каждый заголовок по-своему уникален — все равно приходится их потом оформлять. Но, тем не менее, для тех случаев, когда это все же нужно, попробуем исправить ситуацию с помощью скриптинга.

Как указать, что именно должно переноситься в заголовок? В принципе, существуют как минимум два решения. Первое — наиболее оптимальное — выделить текст заголовка в тексте. Второй заключается в том, что можно обойтись без выделения, но при этом нужно договориться, что, например, заголовком должен быть самый первый абзац в тексте и четко следить за этим. Но поскольку такой способ накладывает определенные ограничения, выберем первый, более гибкий вариант — считаться заголовком будет тот текст, который выделен на момент запуска скрипта (листинг 5.25).

Листинг 5.25. Создание распашных заголовков

```
switch(app.selection[0].constructor.name){
  case "Text":
  case "InsertionPoint":
  case "Character":
  case "Word":
  case "Line":
  case "TextStyleRange":
  case "Paragraph":
  case "TextColumn":
    myHeader = app.selection[0];
    myTextFrame = myHeader.parentTextFrames[0]
}

with (myDocument.viewPreferences){
  var myOldYUnits = verticalMeasurementUnits
  verticalMeasurementUnits = MeasurementUnits.points
}

curr_page = myTextFrame.parent
newFrame = curr_page.textFrames.add({geometricBounds: ↳
  [myTextFrame.geometricBounds[0], myTextFrame.geometricBounds[1], ↳
  myTextFrame.geometricBounds[0]+30, myTextFrame.geometricBounds[3]]})

myHeader.move(LocationOptions.before, newFrame.texts[0])

newFrame.paragraphs[0].applyParagraphStyle( ↳
  app.activeDocument.paragraphStyles[3], true)
```

```
while (newFrame.overflows)
  newFrame.geometricBounds= [newFrame.geometricBounds[0], ↳
    newFrame.geometricBounds[1], newFrame.geometricBounds[2] + 3, ↳
    newFrame.geometricBounds[3]];
  newFrame.textWrapPreferences.textWrapType = ↳
    TextWrapTypes.BOUNDING_BOX_TEXT_WRAP
myDocument.viewPreferences.verticalMeasurementUnits = myOldYUnits;
```

Сначала задаются ссылки на объекты (какой из объектов будет заголовком), а также на его родителя. Последнее сделано для единственной цели: ширина фрейма заголовка будет устанавливаться по ширине фрейма с текстом самой статьи.

Поскольку заголовок набирается всегда гораздо большим кеглем, высоту фрейма придется подгонять. Соответственно, для единиц измерения по вертикали устанавливаем удобную величину: выберем типографские точки (пункты).

В момент создания фрейма заголовка сразу же задаем его геометрические размеры (приблизительные, "на глаз"), при этом высоту устанавливаем в районе 1 см — наше исходное значение.

Переносим выделенный текст в новосозданный фрейм, используя универсальную конструкцию (`texts[0]`), и присваиваем ему стиль. Затем подгоняем высоту фрейма до необходимого значения с шагом приблизительно в миллиметр, что даст, с одной стороны, быстрый, а с другой — относительно точный результат.

Поскольку фрейм заголовка и текст статьи соприкасаются, необходимо решить вопрос с выталкиванием текста. Поскольку фрейм заголовка создан последним, он находится выше фрейма с текстом статьи, и есть все основания полагать, что произойдет переполнение. Решить вопрос можно несколькими способами: первый — уменьшить высоту фрейма статьи настолько, чтобы он не соприкасался с фреймом заголовка, и второй — присвоить фрейму заголовка обтекание. Второй шаг реализуется проще и к тому же не приведет к дополнительным телодвижениям в случае, если размеры фрейма заголовка будут меняться.

Итак, последний шаг — устанавливаем обтекание текстом по габаритам заголовка и восстанавливаем прежние единицы измерения.

В качестве альтернативы для быстрой подгонки размеров фрейма можно привести более эффективный вариант. Он заключается в использовании метода `fit()` и дает мгновенный результат:

```
if(newFrame.overflows)
newFrame.fit(FitOptions, frameToContent)
```

Однако помните, что он подгоняет все размеры фрейма — как высоту, так и ширину, поэтому будьте осторожны при его использовании.

5.15. Расстановка колонтитулов

Причины, по которым в InDesign CS и CS2 отсутствует автоматическое создание скользящих колонтитулов, непосвященному в маркетинговые тонкости Adobe малопонятны. Но факт остается фактом, а поскольку время от времени с такой задачей сталкивается каждый верстальщик, вопрос: как бы облегчить свою работу, возникает достаточно часто.

Не подумайте буквально, прочитав заголовок, что дальше речь пойдет о чудесах ловкости рук при создании колонтитулов вручную — монотонным и лишенным всякого творчества способом. Отнюдь — я предлагаю переложить эту неблагодарную работу на плечи машины, а творчеством заняться самим и вспомнить, что для подобных целей разработчики из Adobe предусмотрели инструмент автоматизации, он же — скриптинг. Данный скрипт (см. листинг 5.26) имеет повышенную сложность по сравнению с рассмотренными ранее скриптами, а потому перед дальнейшим прочтением настоятельно рекомендую ознакомиться с предыдущими примерами.

Вставкой колонтитулов занимаются, в основном, верстальщики объемных изданий, а также тех, у которых информация, выносимая на колонтитул, в явном виде содержится в самом тексте: это разного рода техническая документация, словари, энциклопедии, справочная литература — список можно продолжать. Чтобы поиск можно было автоматизировать, выносимые в колонтитул части текста должны быть каким-либо образом выделены из остального текста. Наиболее простое решение — выделение текста форматированием, для чего идеально подходят возможности стилей. В результате задача сведется к поиску фрагментов текста с определенным форматированием, что легко реализуется в InDesign через встроенную функцию поиска заданного стиля. После нахождения искомого текста он копируется в заранее отведенный на странице фрейм. Удобнее всего использовать для этого возможности мастер-страниц — создав на них текстовые фреймы, куда скрипт будет переносить найденный текст.

Рассмотрим задачу подробнее. Допустим, на странице обнаружен текст, имеющий заданный стиль. Каким образом найти фрейм, отведенный под колонтитул? Анализ документации по InDesign подсказал удачное решение: у фрейма предусмотрено свойство `label` — его название. Другими словами, тот фрейм, который мы создали на мастер-странице, должен иметь уникальный идентификатор, которым выступит `label` и по которому мы найдем его среди остальных фреймов на странице. По умолчанию InDesign это свойство не за-

действует, оставляя его для задач пользователя, т. е. его можно спокойно менять в отличие от некоторых других, специфических свойств объектов. Задается оно непосредственно в открытом документе через палитру **Automation/Script Label**. Итак, допустим, мы создали на мастер-страницах фреймы назначения необходимых размеров и назвали расположенный на левой полосе фрейм "Left", на правой — "Right". В результате все страницы, основанные на этих мастер-страницах, обновятся дополнительными элементами, и к ним можно будет обратиться по соответствующему имени на любом развороте.

Далее. В InDesign, находясь на обычной странице, непосредственно изменять содержимое, унаследованное от родительского объекта (мастера), нельзя — даже выделение его элементов запрещено. Сделано это специально, во избежание разных случайных ситуаций, поскольку в таком случае отмена может не помочь — ведь в данном случае речь идет о целой публикации. Однако нам оно потребуется, т. к. мы собираемся менять содержимое фреймов, заданных в мастерах. Для этого в скриптинге предусмотрена специальная операция `Override` (или <Ctrl>+<Shift>+щелчок на объекте), которая позволяет изменить любой элемент, связанный с мастером, на обычной странице — иными словами, она позволяет выполнить локальное переформатирование объекта. Например, нужно локально изменить цвет — <Ctrl>+<Shift>+ +щелчок — и задача решена. При этом открепляется от мастер-страницы не весь объект, а только те атрибуты, которые были переопределены локально. В нашем случае на объект продолжат распространяться все модификации мастера, кроме изменения цвета — перемещения, повороты и т. п., но не цвет, — причем в случае локального изменения насыщенности плашечного цвета сам цвет связь с мастером сохранит. В скриптинге такая операция имеет свой аналог — функцию `override()`.

Таким образом, возможность расстановки колонтитулов принципиально существует. Осталось решить несколько связанных вопросов. Основной — как пользователь укажет, какой именно искать стиль текста? Можно было бы обойтись встроенным методом JavaScript (`prompt()`), но в данном случае его недостаточно, поскольку единственной строкой в окне, выводимом этим методом, никак не обойтись. Будем использовать собственные возможности InDesign по созданию пользовательских диалоговых окон: такой опыт обязательно пригодится, если вы захотите заниматься скриптингом самостоятельно.

Далее. Чтобы скрипт знал, в каких фреймах публикации выполнять поиск, их нужно каким-то образом указать. Чтобы не делать это вручную, договоримся, что поиск будет проводиться только в той цепочке фреймов, у которой хотя бы один текстовый блок выделен. В результате нам останется только выделить любой фрейм, после чего будут просмотрены все остальные (через ме-

тод `nextTextFrame()`). Место, куда скрипт будет вставлять найденный текст, — это созданный нами фрейм на мастер-странице.

Последний вопрос. Представим ситуацию, когда на полосе найдено несколько строк, удовлетворяющих условию поиска. Что переносить в колонтитул? Наиболее оптимальный вариант — первую и последнюю строки, при этом понадобится их как-то разделять. Предусмотрим свободу выбора, создав набор разделителей. Для других случаев (занесение только первого или только последнего найденного) разделители не пригодятся.

В случае, когда текст с заданным стилем на странице не найден, колонтитул будут дублироваться с предыдущей страницы — до тех пор, пока не будет заменен текстом, найденным на текущей странице.

Итак, описательную часть считаем законченной, переходим непосредственно к написанию скрипта (листинг 5.26).

Листинг 5.26. Расстановка скользящих колонтитулов

```
var pS_name = new Array();
var labels = new Array("L", "R");
var div = new Array("-", "—", "–", ":", "::", ">", ">>");
var check = false;
with(app){
  if((selection.length!=1)||(selection.length==1 && ↳
      selection[0].constructor.name!="TextFrame")){
    alert("Ничего не выделено или выделено более одного объекта, ↳
      или выделенный объект не является текстовым фреймом");
    exit();
  }
  sel = app.activeDocument.selection[0];
  pS = app.activeDocument.paragraphStyles
  for (i=0; i<pS.length; i++) pS_name[i] = pS[i].name;

  var DLG = dialogs.add();
  with (DLG.dialogColumns.add()) {
    with (dialogRows.add().borderPanels.add().dialogColumns.add()) {
      with (dialogRows.add()) {
        staticTexts.add({staticLabel: "Искать стиль:"});
        var DLG_pS = dropdowns.add({stringList:pS_name});
        dropdowns[0].selectedIndex = ↳
          Math.ceil(dropdowns[0].stringList.length/2)
      }
      with (myRadioButtons = radiobuttonGroups.add()) {
        radiobuttonControls.add({staticLabel:"Первый на стр."});
```

```
      radiobuttonControls.add({staticLabel:"Последний на стр."});
      radiobuttonControls.add({checkedState:true, ↳
        staticLabel:"Все на стр."});
    }
    with (dialogRows.add()) {
      staticTexts.add({staticLabel: "Разделитель подразделов:"});
      var DLG_div = dropdowns.add({stringList:div, selectedIndex: 0});
    }
  }
}
var myResult = DLG.show();
findPreferences = null;
changePreferences = null;
findPreferences.appliedParagraphStyle = pS[DLG_pS.selectedIndex];
do{
  s = sel.search();
  do{
    if(s.length==0) {break}
    switch(myRadioButtons.selectedButton){
      case 0:
         s =s[0].contents;
        break;
      case 1:
        s =s[s.length-1].contents;
        break;
      case 2:
        s = s[0].contents;
        for(i=1; i<s.length; i++)
          s += " " + div[DLG_div.selectedIndex] + " " + s[i].contents;
        s = s.replace(/\r/, "");
    }
  } while (false)
  page = sel.parent
  colonTitle = page.textFrames.item(labels[page.documentOffset%2]);
  MasterTF = page.appliedMaster.textFrames
  if (colonTitle==null && ↳
    MasterTF.item(labels[page.documentOffset%2]) != null) {
    colonTitle = MasterTF.item( ↳
      labels[page.documentOffset%2]).override(page);
  }
  try  {colonTitle.contents = c}
  catch(err){
    if(check==false) {
      alert("На стр.  " + page.name + " нет ↳требуемого фрейма!")
```

```
        check = true}
    }
  } while (sel = sel.nextTextFrame)
}
```

Сначала задаем массив из названий присутствующих в публикации стилей:

```
var myParagraphStyles_names = new Array();
```

а также идентификаторы фреймов назначения левой и правой полос плюс набор разделителей:

```
var labels = new Array("L", "R");
var divs = new Array("-", "-", "-", ":", "::", ">", ">>");
```

Начинаем с проверок начальных условий, чтобы скрипт работал корректно всегда. Мы должны обеспечить, чтобы, исходя из логики его работы, соблюдались такие условия:

- в публикации должен быть выделен только один объект;
- он обязан быть текстовым фреймом.

```
with(app){
  if((selection.length!=1)||(selection.length==1 && ↵
    selection[0].constructor.name != "Textframe")){
```

Иначе — вывод предупреждения и прекращение работы:

```
    alert("Проверьте, чтобы был выделен лишь один текстовый фрейм!");
    exit();  }
```

Вводим еще несколько сокращений: для выделенного фрейма, а также для стилей публикации (не на их реальные названия, это было сделано раньше через переменную `myParagraphStyles_names`, а на массив, как он хранится в самом InDesign):

```
var mySelection = app.activeDocument.selection[0];
var myParagraphStyles = app.activeDocument.paragraphStyles
```

Перебираем все существующие в публикации стили и заносим их названия в массив, чтобы потом их вывести в диалоговом окне — иначе пользователь увидит не названия стилей, а их порядковые номера:

```
for (i=0; i<myParagraphStyles.length; i++)
  myParagraphStyles_name[i] = myParagraphStyles[i].name;
```

Настало время заняться созданием диалогового окна. Оформляем окно:

```
var myDLG = dialogs.add();
with (myDLG.dialogColumns.add()) {
  with (dialogRows.add().borderPanels.add().dialogColumns.add()) {
```

На данном этапе мы задали контейнер для всех элементов управления и начинаем их добавлять:

```
with (dialogRows.add()) {
  staticTexts.add({staticLabel: "Искать стиль:"});
```

Теперь выведем названия всех доступных в публикации стилей `myParagraphStyles_name` (с тем, чтобы пользователь мог выбрать требуемый), в виде раскрывающегося списка:

```
var DLG_myParagraphStyles = dropdowns.add({stringList: ↳
  myParagraphStyles_name});
```

Как правило, в публикации стилей много, поэтому установим стилем по умолчанию (`selectedIndex`) расположенный в середине списка — в таком случае мы будем меньше путешествовать по списку стилей:

```
dropdowns[0].selectedIndex = Math.ceil(dropdowns[0].stringList.length/2)
}
```

Продолжаем наполнять диалоговое окно элементами управления. Настала очередь заняться выбором метода занесения текста в колонтитул:

- брать только первый найденный на странице;
- последний;
- по умолчанию (`checkedState`) выберем объединение результатов поиска:

```
with (myRadioButtons = radiobuttonGroups.add()) {
  radiobuttonControls.add({staticLabel:"Первый на стр."});
  radiobuttonControls.add({staticLabel:"Последний на стр."});
  radiobuttonControls.add({checkedState:true, ↳
    staticLabel:"Все на стр."})  }
```

Последняя настройка — задание разделителя (для третьего случая): по умолчанию выберем первый элемент массива `div` — "-":

```
with (dialogRows.add()) {
    staticTexts.add({staticLabel: "Разделитель подразделов:"});
    var DLG_div = dropdowns.add({stringList: div, selectedIndex: 0});
}}}
```

Выводим окно на экран:

```
var myResult = DLG.show();
```

Напомню, что поиск/замену текста в публикации можно выполнять как средствами JavaScript, так и встроенными функциями InDesign. При выборе того или иного механизма нужно учитывать, что JavaScript форматирование не сохраняет, поэтому он годится фактически только для поиска/замены одних текстовых символов на другие (правда, может делать это виртуозно). В отли-

чие от него, InDesign более прямолинеен, зато проводит поиск/замену с учетом форматирования. Для удобства эти возможности реализованы через отдельные объекты `findPreferences` и `changePreferences` (несколько непривычно, но, как вы заметили, в основе любой манипуляции как JavaScript, так и InDesign лежит объект, именно поэтому такое программирование называется объектно-ориентированным — ООП).

Первым шагом будет сброс всех настроек в полях поиска и замены во избежание случайного использования оставшихся от предыдущих операций. Поскольку эти настройки глобальны (не привязаны к конкретному документу), то родительским для таких объектов будет объект `app`, т. е. сам InDesign (как вы помните, для доступа к любому объекту необходимо указать полностью всю вышестоящую иерархию):

```
app.findPreferences = null;
app.changePreferences = null;
```

Здесь `null` — специальное слово, означающее отсутствие значения.

Свойства объекта `findPreferences` полностью покрывают все возможности, доступные в окне **Find/Change**, нас же интересуют только относящиеся к поиску определенного стиля (свойство `appliedParagraphStyle`). В качестве параметра задаем порядковый номер стиля, выбранный пользователем в диалоговом окне, после чего выполняем сам поиск (метод `search()`):

```
app. findPreferences.appliedParagraphStyle = ↳
  myParagraphStyles[DLG_myParagraphStyles.selectedIndex];
  do{
```

Для каждого текстового фрейма имеем:

```
    mySearch = mySelection.search()
```

Обратите внимание на то, что мы использовали метод из арсенала InDesign, а не JavaScript, поэтому результат будет отличен от привычного по Web-программированию: мы получим набор строк, хранящихся в массиве. Переменную `mySearch` как массив можно не определять, поскольку результат этого метода записывается автоматически.

Рассмотрим варианты дальнейших действий при различных результатах поиска.

Если пользователем был выбран вариант переноса в колонтитул только первого найденного на странице текста, то им будет первый элемент массива найденных строк `mySearch` – `mySearch[0]`, если последний — `mySearch[mySearch.length-1]`. В том случае, если же новый кандидат в колонтитулы не найден (`mySearch.length==0`), колонтитул должен оставаться предыдущим, т. е. замена происходить не должна:

```
do{
  if(mySearch.length==0) {break()}  // Переход к следующей странице
  switch(myRadioButtons.selectedButton){
    case 0:
      // Запоминаем первый найденный текст
      mySearchContents = mySearch[0].contents
      break();
    case 1:
      // запоминаем последний найденный на странице текст
      mySearchContents = mySearch[mySearch.length-1].contents;
      break();
```

Операция `break()` прерывает дальнейший поиск, что значительно экономит время.

Если же была выбрана опция занесения в колонтитул всех найденных совпадений, то текст каждого следующего совпадения приклеивается через выбранный разделитель `DLG_div.selectedIndex` к предыдущему:

```
    case 2:
      mySearchContents = mySearch[0].contents;
      for(i=1; i<mySearch.length; i++)
        mySearchContents += " " + div[DLG_div.selectedIndex] + " " ↳
          + mySearch[i].contents;
        mySearchContents = mySearchContents.replace(/\r/, "");
```

Параллельно удаляем из найденных строк символы абзаца `\r`, чтобы после объединения они образовали одну строку.

```
}} while (false)
```

Механизм переноса текста в колонтитул такой: сначала на каждой странице с фреймом, принадлежащем выделенной цепочке, ищем фрейм с названием "L" или "R". Родительским объектом по отношению ко всем находящимся на странице объектам является страница:

```
var myPage = mySelection.parent
var colonTitle = myPage.textFrames[labels[myPage.documentOffset%2]];
```

Конструкция `myPage.documentOffset%2` позволяет перебирать содержимое массива `labels`: `documentOffset` — это номер текущей страницы. Если она четная (`myPage.documentOffset%2=0`, `%` — операция нахождения остатка от деления, если остатка нет, то он равен 0), ищется фрейм с названием "L", если нечетная — "R" (берется `labels[1]`). Потом переходим на мастер-страницу, на основании которой эта страница была сформирована:

```
var MasterTF = myPage.appliedMaster.textFrames
```

Выполняем проверку: если колонтитул все еще связан с мастер-страницей (`page.textFrames.item==null`, т. е. самой странице фрейм "L" или "R" не принадлежит, т. к. связан с мастером) и на мастере есть фреймы "L" или "R", то связь разрываем с тем, чтобы получить возможность изменить его содержание (механизм рассматривался ранее):

```
if (colonTitle==null && MasterTF[labels[myPage.documentOffset%2]] != ↳
  null)
{
  colonTitle = MasterTF[ labels[myPage.documentOffset%2] ]. ↳
    override(myPage);
```

После этого найденный текст со стилем
`myParagraphStyles[DLG_myParagraphStyles.selectedIndex]`
переносим в колонтитул:

```
  colonTitle.contents = mySearchContents
}
```

Если же фрейм назначения на странице обнаружен не был, выводим предупреждение. В случае если таких страниц много, выскакивающие окна станут раздражать, а потому ограничимся только однократным сообщением:

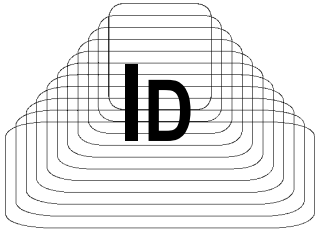
```
else{
  if(!check) {
    alert("На стр. " + master.name + " нет требуемого фрейма!")
    check = true}  }
```

Поиск текста и заполнение колонтитулов проводим до тех пор, пока не достигнем последнего фрейма в цепочке:

```
while (mySelection.nextTextFrame)
```

И последнее замечание: при выделении текстового фрейма перед запуском скрипта убедитесь, что он — первый в цепочке, поскольку поиск будет проводиться только в следующих за ним фреймах.

ГЛАВА 6



Форматирование текста

Среди основных инструментов форматирования текста шрифты занимают одну из важнейших позиций, поскольку они способны подчеркнуть индивидуальность издания. Каждое начертание представляет собой отдельный шрифт: например, Times New Roman — обычное начертание, а его полужирный вариант — Times New Roman Bold и т. д. Если какое-либо начертание отсутствует, оно недоступно для использования. В этом — принципиальная разница между InDesign и XPress: последний пакет менее щепетилен в вопросах использования шрифтов и создает недостающее начертание собственными методами. Например, если не хватает наклонного — он его создает из обычного (Plain), к которому применяет скос (skew), не хватает полужирного — дублирует оригинал и дает дубликату небольшое смещение.

Состав коллекции `fonts` зависит от того, каким образом происходит обращение к ней. Если это происходит на уровне публикации, то в нее входят все шрифты, доступные InDesign. Если на уровне публикации — то только используемые в ней шрифты, причем среди них могут быть и отсутствующие. Приведенный далее пример иллюстрирует разницу между двумя способами считывания шрифтов.

```
var myApplicationFonts = app.fonts;
var myDocumentFonts = myDocument.fonts;

var myDocument = app.documents.add();
var myPage = myDocument.pages[0];

var myTextFrame = myPage.textFrames.add();
var myString = "Document Fonts:\r";

// Сначала — использованные в публикации
for(i = 0; i<myDocumentFontNames.length; i++){
  myString += myDocument.fonts [i].name + "\r";}
```

```
myString += "\r=======================\r"

// А затем — все доступные InDesign:
myString += "\rApplication Fonts:\r";
for(i = 0; i< myApplicationFonts.length; i++){
  myString += myApplicationFonts[i]. name + "\r";}
```

6.1. Установка свойств текста

Рассмотрим способы изменения свойств текста. Все операции достаточно прозрачны, особое внимание нужно обратить лишь на задание конкретного начертания шрифта. Как правило, название шрифта имеет вид:

familyName*<tab>*fontStyle

Здесь:

- ***familyName*** — название семейства;
- **<tab>** — символ табуляции;
- ***fontStyle*** — начертание шрифта.

Поэтому, если необходимо указать конкретный шрифт, не прибегая к заранее предопределенному через стиль, следует помнить об этом синтаксисе, например:

```
Times New Roman<tab>Bold
```

Если нужно изменить только начертание, прибегают к использованию `fontStyle`, например:

```
texts[0].fontStyle = 'Bold'
```

Пример форматирования текста приведен в листинге 6.1.

Листинг 6.1. Форматирование текста

```
var myDocument = app.documents.add()
var myPage = myDocument.pages[0]
var myTextFrame = myPage.textFrames.add()
myTextFrame.contents = "Пример задания свойств тексту"
var myTextObject = myTextFrame.words[0]

// Для сокращения объема скрипта и повышения читабельности
// используем объект по умолчанию:
with (myTextObject){
  alignToBaseline = false
  appliedCharacterStyle = myDocument.characterStyles[1]
```

```
  appliedFont = app.fonts["Adobe Caslon Pro"];
  fontStyle = "Semibold Italic";
  appliedLanguage = app.languagesWithVendors["English: USA"]
  appliedNumberingList = myDocument.numberingLists["[Default]"]
  appliedParagraphStyle = myDocument.paragraphStyles[1]
  autoLeading = 120
  balanceRaggedLines = BalanceLinesStyle.noBalancing
  baselineShift = 0
  bulletsAlignment = ListAlignment.leftAlign
  bulletsAndNumberingListType = ListType.noList
  bulletsCharacterStyle = myDocument.characterStyles[1]
  bulletsTextAfter = "^t"
  capitalization = Capitalization.normal
  composer = "Adobe Paragraph Composer"
  desiredGlyphScaling = 100
  desiredLetterSpacing = 0
  desiredWordSpacing = 100
  dropCapCharacters = 0
  dropCapLines = 0
  dropCapStyle = myDocument.characterStyles[1]
  dropcapDetail = 0
}
```

Подобные задачи встают не часто, поскольку, как правило, при верстке очень активно используются стили. Будучи однажды созданным (причем не обязательно через скрипт) стиль можно применять сколь угодно раз, при этом получая существенную экономию времени.

6.2. Установка позиций табуляции

Приведенный в листинге 6.2 скрипт является прекрасным примером, иллюстрирующим навигацию по текстовым элементам, без которой невозможна полноценная работа с текстом.

Предположим, стоит задача обеспечить вставку разных символов отбивки в любом месте в тексте. Символы отбивки — табулятор и абзацный отступ, а возможные варианты вставки таковы:

- вставка табулятора на текущей позиции;
- установка табулятора по левому отступу;
- установка абзацного отступа на текущей позиции.

Задача решается довольно просто, если текстовый фрейм имеет единственную колонку. В противном случае потребуется провести дополнительные

расчеты: как известно, позиция табуляции и ей подобные являются величинами относительными, т. е. они вычисляются от левого края текущей колонки, а InDesign для текущей позиции может выдать только абсолютное значение, начиная отсчет от края фрейма.

Наглядное представление о процессе формирования значений дает рис. 6.1. Текущая позиция (`insertionPoint`) находится в третьей колонке, при этом позиция, например, табулятора будет определяться значением не `horizontalOffset`, а `MyTabPosition`. Соответственно, потребуется учитывать размеры предыдущих колонок и суммарного межколоночного расстояния.

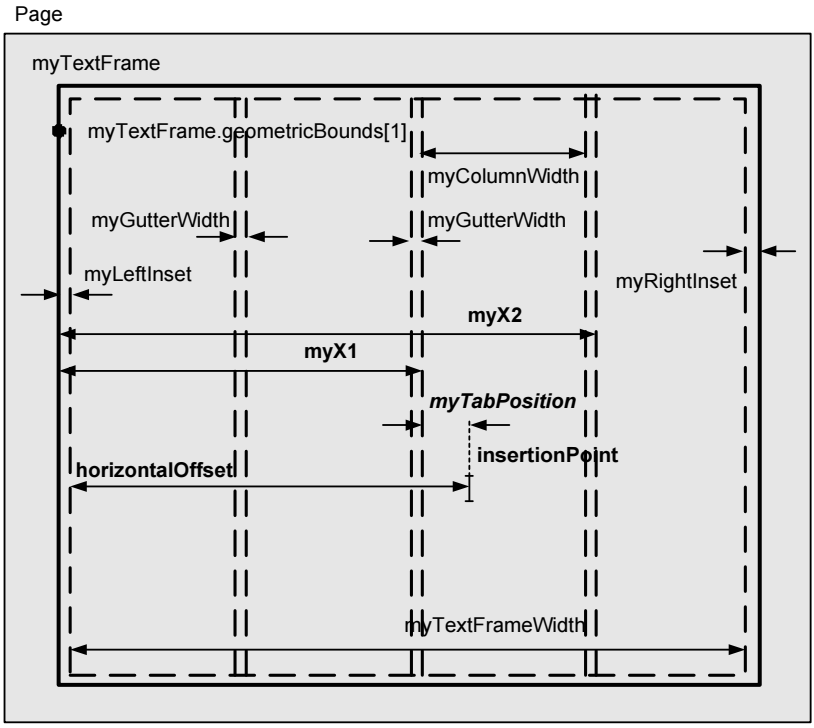


Рис. 6.1. Форматирование колонок

Листинг 6.2. Установка позиций табуляции

```
var myObjectList = new Array;
if (app.documents.length != 0){
  if (app.activeDocument.stories.length != 0){
    if (app.selection.length != 0){
      switch (app.selection[0].constructor.name){
```

```
        case "InsertionPoint":
        case "Character":
        case "Word":
        case "TextStyleRange":
        case "Line":
        case "Paragraph":
        case "TextColumn":
        case "Text":
        case "TextFrame":
          myDisplayDialog();
          break;
      }
    }
    else{
      alert("Please select some text and try again.");
    }
  }
}
else{
  alert("Please open a document, select an object, and try again.");
}

function myDisplayDialog(){
  var myDialog, myTabButtons;
  with(myDialog = app.dialogs.add({name:"TabUtilities"})){
    with(dialogColumns.add()){
      with(borderPanels.add()){
        staticTexts.add({staticLabel:"Позиция табулятора:"});
        with(myTabButtons = radiobuttonGroups.add()){
          radiobuttonControls.add({staticLabel:"правый край колонки", ↳
            checkedState:true});
          radiobuttonControls.add({staticLabel: ↳
            "текущая позиция курсора"});
          radiobuttonControls.add({staticLabel:"по левому Indent"});
          radiobuttonControls.add({staticLabel: ↳
            "Hanging Indent at Cursor"});
        }
      }
      with(borderPanels.add()){
        staticTexts.add({staticLabel:"Tab Leader"});
          with(dialogColumns.add()){
          myTabLeaderField = textEditboxes.add({editContents:""});
        }
```

```
      }
    }
  }
  var myResult = myDialog.show();
  if(myResult == true){
    var myTabType = myTabButtons.selectedButton;
    var myTabLeader = myTabLeaderField.editContents;
    myDialog.destroy();
    myAddTabStop(myTabType, myTabLeader);
  }
  else{
    myDialog.destroy();
  }
}

// Добавление табулятора
function myAddTabStop(myTabType, myLeader){
  var myParagraphs, myTabPosition, myTabAlignment, myParagraph;
  switch(myTabType){
    case 0:
      myParagraphs = app.selection[0].paragraphs;
      for(i = 0; i < myParagraphs.length; i ++){
        myParagraph = myParagraphs[i];
        myTabPosition = ↳
          myParagraph.insertionPoints[0].parentTextFrames[0]. ↳
          textFramePreferences.textColumnFixedWidth;
        myTabAlignment = TabStopAlignment.rightAlign;
        myParagraph.tabStops.add({alignment:myTabAlignment, ↳
          leader:myLeader, position:myTabPosition});
      }
      break;
    case 1:
      myInsertionPoint = app.selection[0].insertionPoints[0];
      myTabPosition = myInsertionPoint.horizontalOffset - ↳
        myFindColumnEdge(myInsertionPoint);
      myTabAlignment = TabStopAlignment.leftAlign;
      myInsertionPoint.paragraphs[0].tabStops.add({alignment: ↳
        myTabAlignment, leader:myLeader, position:myTabPosition})
      break;
    case 2:
      myParagraphs = app.selection[0].paragraphs;
      for(i = 0; i < myParagraphs.length; i ++){
        myParagraph = myParagraphs[i];
```

```
        myTabPosition = myParagraph.leftIndent;
        myTabAlignment = TabStopAlignment.leftAlign;
        myParagraph.tabStops.add({alignment:myTabAlignment, ↳
          leader:myLeader, position:myTabPosition});
      }
      break;
    case 3:
      myParagraphs = app.selection[0].paragraphs;
      myInsertionPoint = app.selection[0].insertionPoints[0];
      myTabPosition = myInsertionPoint.horizontalOffset - ↳
        myFindColumnEdge(myInsertionPoint);
      myTabAlignment = TabStopAlignment.leftAlign;
      for(i = 0; i < myParagraphs.length; i ++){
        myParagraph = myParagraphs[i];
        myParagraph.leftIndent = myTabPosition;
        myParagraph.firstLineIndent = -myTabPosition;
        myParagraph.tabStops.add({alignment:myTabAlignment, ↳
          leader:myLeader, position:myTabPosition});
      }
      break;
  }
}

// Определение положения левого края колонки
function myFindColumnEdge(myInsertionPoint){
  var i, myLeftInset, myRightInset, myX1, myX2, myColumnEdge;
  var myPagePosition = myInsertionPoint.horizontalOffset;
  var myTextFrame = myInsertionPoint.parentTextFrames[0];
  var myColumnWidth = ↳
    myTextFrame.textFramePreferences.textColumnFixedWidth;
  var myGutterWidth = myTextFrame.textFramePreferences.textColumnGutter;
  var myTextFrameWidth = myTextFrame.geometricBounds[3]-↳
    myTextFrame.geometricBounds[1];
  var myXOffset = myPagePosition - myTextFrame.geometricBounds[1];
  var myArray = new Array;
for (i = 0; i < myTextFrame.textFramePreferences.textColumnCount; i ++){
    // Если колонка всего одна
    if(i == 0){
      // Если заданы отступы от краев фрейма
      if(myTextFrame.textFramePreferences.insetSpacing.length == 4){
        myLeftInset = myTextFrame.textFramePreferences.insetSpacing[1];
        myRightInset = myTextFrame.textFramePreferences.insetSpacing[3];
      }
```

```
      else{
        // Если отступы не заданы, длина массива insetSpacing=1
        myLeftInset = myTextFrame.textFramePreferences.insetSpacing[0];
        myRightInset = myTextFrame.textFramePreferences.insetSpacing[0];
      }
      myX1 = myTextFrame.geometricBounds[1] + myLeftInset;
      myX2 = myX1 + myColumnWidth;
    }
    else{
      // Для текущей колонки
      if(i == myTextFrame.textFramePreferences.textColumnCount){
        myX2 = myTextFrame.geometricBounds[1] -myRightIndent;
        myX1 = myX2 - myTextWidth;
      }

      // Накопление результата из всех предыдущих колонок, см. рис. 6.1
      else{
        myX1 = myTextFrame.geometricBounds[1] + ↳
               (myColumnWidth*i) + (myGutterWidth * i);
        myX2 = myX1 + myColumnWidth;
      }
    }

    // Искомые значения
    myArray.push([myX1, myX2]);
  }

  for(i = 0; i < myArray.length; i ++){
    if((myPagePosition >= myArray[i][0])&& ↳
       (myPagePosition <=myArray[i][1])){
      myColumnEdge = myArray[i][0];
      break;
    }
  }
  return myColumnEdge;
}
```

Начало представляет собой традиционную "защиту от неподготовленного пользователя": проверяем, что точка вставки находится в тексте (`app.documents.length`, `app.selection[0].constructor.name`). Затем — отображение диалогового окна, в котором предусмотрены задание позиции табулятора (Tab Stop, четыре варианта) и заполнителя (Tab Leader). Главная цель — найти необходимое положение точки вставки и выразить его через доступные нам свойства фрейма.

В зависимости от выбранного варианта позиции табулятора будут происходить различные действия. Рассмотрим каждое по порядку.

Случай первый — табуляция должна быть установлена по правому краю колонки (практическое использование — создание содержания, например). Чтобы определить ширину колонки (функция `findColumnEdge`), согласно объектной модели от точки вставки `insertionPoints[0]` переходим к родительскому текстовому фрейму `parentTextFrames[0]`, у которого получить необходимое значение уже не составит никакого труда:

```
myTabPosition = myParagraph.insertionPoints[0].parentTextFrames[0].
textFramePreferences. textColumnFixedWidth
```

Дальнейшее — дело техники: задание типа табулятора и собственно его установка

```
myParagraph.tabStops.add({alignment:myTabAlignment, leader:myLeader,
position:myTabPosition});
```

В случае, если выбран второй вариант (установка табулятора на текущей позиции), положение текущей позиции получаем через свойство `horizontalOffset`. Однако оно дает абсолютное значение (относительно точки начала координат публикации). А для того чтобы получить смещение относительно левого края колонки (вспомните окно **Tab position** в InDesign), нужно найти положение самого края. В самом простом случае (одноколоночный фрейм) оно равно

```
myInsertionPoint.horizontalOffset - myTextFrame.geometricBounds[1];
```

Как правило, нужно еще вычесть расстояние от края колонки до текста `myLeftInset`:

```
myLeftInset = myTextFrame.textFramePreferences.insetSpacing[1];
```

В случае, если фрейм многоколоночный, нужно сделать поправку на количество колонок и суммарное расстояние между колонками (см. рис. 6.1):

```
myTextFrame.geometricBounds[1] + (myColumnWidth*i) + (myGutterWidth * i);
```

Естественно, при этом должен быть установлен табулятор с левой выключкой:

```
myTabAlignment = TabStopAlignment.leftAlign;
```

Наконец, последний случай (`hanging indent` на текущей позиции): для установки табулятора сначала определяем необходимую позицию: из абсолютного значения текущей точки вставки вычитаем позицию начала колонки, что даст нам положение точки вставки относительно текущей колонки:

```
myInsertionPoint = app.selection[0].insertionPoints[0];
myTabPosition = myInsertionPoint.horizontalOffset - ↳
  myFindColumnEdge(myInsertionPoint);
```

```
myParagraph.tabStops.add({alignment:myTabAlignment, ↳
  leader:myLeader, position:myTabPosition});
```

6.3. Работа с цветом

Умение использовать цвета не менее важно, чем, например, умение форматировать текст — ведь в таком случае вместо красочного издания мы получим исключительно черно-белое. В данном разделе рассматривается присвоение цвета тексту, но и для остальных объектов оно аналогично. Цвет можно присваивать как заливке символов, так и окантовке.

Непосредственное назначение цвета — довольно редкая задача, поскольку, как правило, подобные задачи стараются решить через стили: для абзацев, отдельных символов, объектов — ведь такой подход позволяет повысить управляемость публикацией и оперативно вносить нужные изменения в глобальном масштабе. Однако, какой бы вариант вы не выбрали, так или иначе цвет нужно задать.

В листинге 6.3 приведен пример, в котором образец цвета (Swatch) сначала создается, а потом присваивается.

По умолчанию в InDesign цветовая модель имеет тип CMYK, поэтому значения составляющих цвета указываются в виде массива `[C, M, Y, K]`.

Листинг 6.3. Создание образцов цвета (Swatch)

```
var myDocument = app.documents[0]

myColorA = myDocument.colors.add({name:"MyColorA", ↳
  colorValue:[100,20, 50,0]})
myColorB = myDocument.colors.add({name:"MyColorB", ↳
  colorValue:[50,20, 100,0]})
```

InDesign не позволяет из скрипта менять цветовую модель, соответственно создаваемый цвет может иметь только цветовую модель текущей публикации.

```
// Присвоение цвета первому абзацу
myTextFrame.paragraphs[0].fillColor = myColorA

// Форматирование второго абзаца
var myText = myTextFrame.paragraphs[1]
with(myText){
  strokeWeight = 3
  pointSize = 72
```

```
  fillColor = myColorB
  strokeColor = myColorA
}
```

6.4. Использование стилей

Использование стилей — наиболее эффективный способ форматирования публикации, поскольку позволяет быстро менять ее оформление в зависимости от конкретных задач. В стилях хранятся все параметры форматирования, и присвоение стиля тексту или любому графическому объекту фактически устанавливает постоянную связь между ним и этим набором параметров. Рассмотрим использование стилей с текстом.

InDesign поддерживает два типа стилей — применяемых к абзацу целиком (`paragraphStyle`) и лишь к отдельным символам (`characterStyle`). Стили могут быть связанными — например, когда на основе выбранного в качестве базового создаются другие, наследующие его параметры. При этом в базовом стиле указывается лишь необходимый минимум параметров (как правило, гарнитура шрифта), цвет, язык, установки для выполнения переноса слов, остальное задается в каждом конкретном стиле.

Присвоить стиль можно следующими способами:

- используя метод `applyStyle()`, который имеет два параметра: первый — индекс стиля, второй — сбрасывать (`true`) или нет (`false`) текущие параметры форматирования текста (в палитрах **Styles** операция имеет название **Override**);
- установкой свойства `appliedParagraphStyle` или `appliedCharacterStyle`.

Отличие между этими свойствами состоит в том, что метод `applyStyle()` позволяет сбросить локальное форматирование текста, в то время как свойства этого лишены. Присваивать стиль можно как отдельным абзацам, так и любому фрагменту текста (используя `itemByRange()` совместно с `appliedCharacterStyle`). Кроме того, InDesign CS3 поддерживает вложенные стили (`nestedStyles`), т. е., например, первое слово форматируется одним стилем, второе слово — другим и т. п., что часто используется для сохранения уникальности изданий.

6.4.1. Создание стиля символов

Пример создания стиля символов приведен в листинге 6.4.

Листинг 6.4. Создание стиля символов

```
// Создаем стиль символов "myCharacterStyle"
myCharacterStyle = ↳
  myDocument.characterStyles.add({name:"myCharacterStyle"})
```

```
// На данном этапе мы просто получили ссылку на новый стиль.
// Пока он пустой, поэтому займемся установкой его параметров.
myCharacterStyle.fillColor = myColor1
myCharacterStyle.size = 14
```

6.4.2. Создание стиля абзаца

Пример создания стиля абзаца приведен в листинге 6.5.

Листинг 6.5. Создание стиля абзаца

```
myParagraphStyle = ↳
  myDocument.paragraphStyles.add({name:"myParagraphStyle"})

// Стиль также пуст, занимаемся его форматированием
myParagraphStyle.fillColor = myColor2
myParagraphStyle. fontSize = 18
```

6.4.3. Создание вложенного стиля

Вложенные стили поддерживаются лишь в InDesign CS3, позволяя автоматически задать определенный стиль последовательности символов, оканчивающейся заданным символом. Вот как, например, будет выглядеть скрипт, присваивающий стиль `myCharacterStyle` первому предложению (разделение идет по точке) в абзаце (листинг 6.6).

Листинг 6.6. Присвоение стиля предложению

```
var myNestedStyle = myParagraphStyle.nestedStyles.add ↳
  ({appliedCharacterStyle:myCharacterStyle, delimiter:".", ↳
   inclusive:true, repetition:1});
```

Рассмотрим пример форматирования текста стилями (листинг 6.7).

Листинг 6.7. Форматирование текста стилями

```
// Создаем текстовый фрейм на странице 1
var myTextFrame = myDocument.pages[0].textFrames.add();
myTextFrame.geometricBounds =[0,0,100,100];
myTextFrame.contents = "Тестовый фрейм";

myCharacterStyle = ↳
  myDocument.characterStyles.add({name:"myCharacterStyle"})
```

```
myCharacterStyle.fillColor = myColor1;
myCharacterStyle.size = 14;

// Создадим стиль абзаца "myParagraphStyle"
myParagraphStyle = ↳
  myDocument.paragraphStyles.add({name:"myParagraphStyle"})
}

// Установки форматирования стиля
myParagraphStyle.fillColor = myColor2
myParagraphStyle.fontSize = 18

// И присваиваем его всему тексту
myTextFrame.parentStory.texts[0].applyStyle(myParagraphStyle, true)

var myStartCharacter = myTextFrame.parentStory.characters[13]
var myEndCharacter = myTextFrame.parentStory.characters[54]

// Присваиваем стиль только некоторым абзацам, сбрасывая
// все их предыдущие параметры оформления
myTextFrame.parentStory.texts.itemByRange(myStartCharacter, ↳
  myEndCharacter).applyStyle(myCharacterStyle, true);
```

Перед созданием нового стиля нужно убедиться, что стиль с аналогичным названием в публикации не существует, в противном случае InDesign выдаст ошибку.

К сожалению, присвоение стиля тексту — не такая простая операция, как например, присвоение цвета. Дело в том, что InDesign хранит стили не по их названиям, как они отображаются в палитре **Paragraph Styles**, а по их индексам, поэтому присвоить стиль непосредственно по его названию не получится. Соответственно, сначала нужно просмотреть весь список используемых в публикации стилей, для каждого запомнить его индекс, по которым в дальнейшем к ним обращаться.

Естественно, держать в голове все индексы — дело неблагодарное, которое к тому же усложняется тем, что при добавлении/удалении стиля вся прежняя индексация теряется. Поэтому попробуем придать операции более человеческий вид.

Например, это можно реализовать следующим образом:

```
parStyles = myDocument.paragraphStyles;
```

При нахождении заданных стилей запоминаем их индексы в виде переменных:

```
for (i=0; i< parStyles.length; i++){
  switch(parStyles[i].name){
    case 'Table Header':
      var TableHeader = i;
      break;
    case 'Table Footer':
      var TableFooter = i;
      break;
    case 'Table contents':
       var TableContents = i;
      break;
    case 'Table':
      var Table = i;
      break;
    case 'My_Style':
      var My_Style = i;
      break;
  }
}
```

Если какой-то из искомых стилей найден, то продолжать его дальнейший поиск смысла нет, можно переходить к поиску следующего.

Присвоение стиля в таком случае происходит следующим образом:

```
myParagraph1.applyStyle(parStyles [TableHeader], true);
myParagraph2.applyStyle(parStyles [TableFooter], true);
myParagraph3.applyStyle(parStyles [TableContents], true);
myParagraph4.applyStyle(parStyles [Table], true);
```

Гораздо нагляднее, не правда ли?

Как правило, перед запуском скрипта следует убедиться в том, что все необходимые стили присутствуют. В таком случае можно поступить так: при каждом сравнении устанавливать соответствующий флаг:

```
flagStorage = new Array()
for (i=0; i< parStyles.length; i++){
  switch(parStyles[i].name){
    case 'Table Header':
      var TableHeader = i;
      flagStorage [0]=true;
       break;
    case 'Table Footer':
      var TableFooter = i;
       flagStorage [1]=true;
       break;
```

```
    case 'Table contents':
       var TableContents = i;
      flagStorage [2]=true;
       break;
    case 'Table':
      var Table = i;
      flagStorage [3]=true;
       break;
    case 'My_Style':
      var My_Style = i;
       flagStorage [4]=true;
       break;
  }
}
```

Тогда проверка будет выглядеть так:

```
if(flagStorage[0] && flagStorage[1] && flagStorage[2] && ↳
   flagStorage[3] && flagStorage[4]) {
  // Дальше идет основной скрипт
}
```

По умолчанию булевы выражения всегда сравниваются с `true`, поэтому отпадает необходимость каждый раз использовать конструкцию типа `if(flagStorage[...]==true)`.

Альтернативный вариант проверки существования всех необходимых стилей — запись в массив, например, нулей

```
flagStorage.push("0")
```

с последующей проверкой длины массива. Если она не будет соответствовать количеству проверяемых стилей, значит, со стилями не все в порядке:

```
if (flagStorage.length != requiredLength)
  alert("Не все необходимые стили присутствуют в документе")
```

> ***Новое в InDesign Creative Suite 3***
>
> В InDesign Creative Suite 3 свойство `applyStyle` (задание абзацу стиля) заменено свойством `applyParagraphStyle`, что, по логике вещей, более корректно.

6.4.4. Удаление неиспользуемых стилей

Кроме создания стилей, может возникнуть потребность их удаления — например, неиспользуемых в публикации: вообще же считается дурным тоном отдавать публикацию на фотовывод с лишней информацией, в том числе с неиспользуемыми стилями. В таком случае проще всего применить регуляр-

ные выражения. Из множества возможных вариантов выполнения данной задачи выберем наиболее простой, а потому самый скоростной (листинг 6.8).

Подход следующий: сначала просматриваются по очереди все абзацы и определяются задействованные стили (через свойство абзаца `appliedParagraphStyle`). Названия стилей записываются в строку. Затем также по очереди выбираются стили из списка всех существующих в публикации (через свойство документа `paragraphStyles`), после чего обе строки сравниваются: в случае совпадения названий переходим к следующему стилю из списка, если совпадение не найдено — значит, в публикации стиль не используется и потому удаляется.

Осталось решить вопрос с диапазоном поиска. В самом деле, если использовать стандартный подход — стиль искать среди объектов коллекции `textFrames`, можно "наломать дров". Согласно объектной модели, у заякоренных текстовых блоков (`Anchor`) родитель не текстовый фрейм, а символ, поэтому использование `textFrames` будет ошибочным (или потом придется опускаться на уровень `characters`). Поэтому, чтобы не пропустить ни одного абзаца, обратимся к свойству `allPageItems`, которое дает доступ полностью ко всему содержимому страницы.

Листинг 6.8. Удаление неиспользуемых стилей

```
var styleApplied = "";
var pgI = document.allPageItems;
var allStyles = document.paragraphStyles;

for (i=0; i<pgI.length; i++){
  if (pgI[i] == "[object TextFrame]")
  {
    pars = pgI[i].paragraphs;
    for (j=0; j<pars.length; j++){
      styleApplied += pars[j].appliedParagraphStyle.name;
    }
  }
}

for (i=2; i<allStyles.length; i++){
  ix = styleApplied.search(allStyles[i].name);
  if (ix == -1) {
    allStyles[i].remove();
    i—
  }
}
```

Создаем ссылки на объекты:

```
myAllItems = app.activeDocument.allPageItems;
myParagraphStyles = app.activeDocument.paragraphStyles;
```

Создаем строку для накопления в ней всех использованных в документе стилей:

```
styleApplied = "";
```

В цикле просматриваем абсолютно все содержимое публикации, отфильтровываем только текстовые фреймы

```
for (i=0; i< myAllItems.length; i++){
  if (myAllItems[i]. constructor.name == "TextFrame") {
```

и сохраняем все использованные стили:

```
    myParagraphs = myAllItems[i].paragraphs;
    for (j=0; j<myParagraphs.length; j++){
      styleApplied += myParagraphs[j].appliedParagraphStyle.name;
    }
  }
```

Сравниваем хранимые в публикации стили с реально используемыми. При этом учитываем, что в InDesign существует 2 предопределенных стиля, которые не могут быть удалены (`[No Paragraph Style]` и `[Basic Paragraph]`), а потому сразу же исключаем их из проверки:

```
for (i=2; i< myParagraphStyles.length; i++){
  ix = styleApplied.search(myParagraphStyles[i].name);
```

Если стиль не найден (т. е. позиция, найденная методом `search`, равна –1), то удаляем его и учитываем это в счетчике:

```
  if (ix == -1) {
    myParagraphStyles[i].remove();
    i--}
}
```

Последнее обстоятельство необходимо для корректной работы скрипта. По окончании цикла счетчик (`i`) увеличится на 1, и мы перейдем от первого стиля ко второму. Представьте себе, что первый же существующий в публикации стиль был удален: тогда следующий за ним стиль (второй по счету) займет его место в списке и станет первым, третий — вторым и т. д. по всей цепочке (рис. 6.2). Таким образом, произойдет смещение стилей, и вместо второго перейдем сразу же к третьему стилю (он занял место второго). Чтобы избежать этого, нужно либо компенсировать приращение счетчика, что мы и сделали — либо (кому-то оно покажется даже проще) — начинать проверку с последнего стиля в публикации

```
(for (i=allStyles.length-1; i>=2; i--))
```

Поскольку нумерация идет с начала, удаление объектов с конца к нарушению нумерации не приведет.

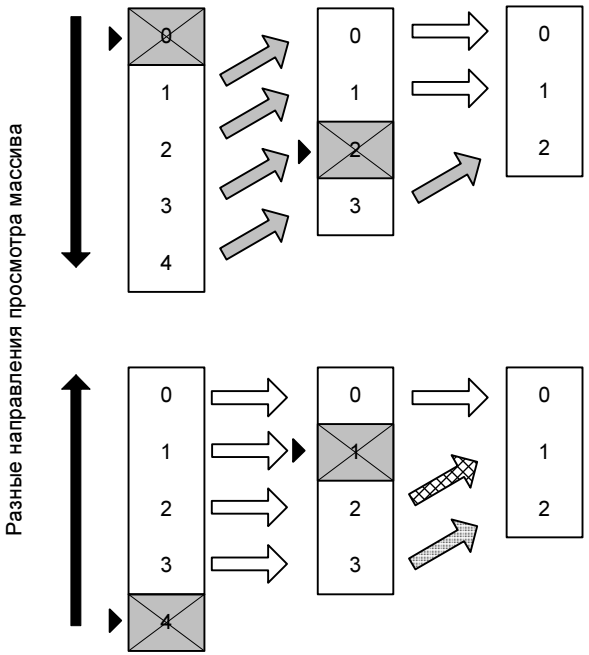


Рис. 6.2. Различия между двумя вариантами просмотра массива

Перечеркнутые элементы — удаляемые, а серыми стрелками показаны смещающиеся элементы массива (у них происходит смена индексов). Предположим, что удаляется первый элемент массива. Видно, что в случае традиционного подхода (проход начинается с начала массива, верхний рисунок) это приводит к изменению индексов у всех остальных элементов массива; если же используется противоположное направление просмотра (нижний рисунок), то никакого смещения не происходит.

Следующим, допустим, удаляется третий элемент. Как видно из рисунков, картина повторяется: на верхнем четвертый элемент снова меняет свой индекс, на нижнем же для уже пройденных элементов тоже изменится индекс, но для нас это уже не играет никакого значения, поскольку они уже пройдены и больше обрабатываться не будут (обозначены штриховкой), а вот следующий, важный для нас (с индексом 0), свой индекс снова не меняет — это нам и нужно (текущий элемент обозначен черным треугольником).

6.5. Супермегаметла для публикации

На данный момент мы уже обладаем достаточными знаниями для того, чтобы написать скрипт, удаляющий все лишние объекты, в один универсальный суперскрипт.

Лишними будем считать:

- пустые фреймы;
- пустые абзацы (в которых нет ни одного символа, а также имеющие только пробелы или табуляторы);
- неиспользуемые стили;
- неиспользуемые образцы цвета (swatches);
- объекты, расположенные полностью на монтажном столе.

Таким образом, по своей функциональности скрипт будет являться, по современному, этакой супермегаметлой (листинг 6.9).

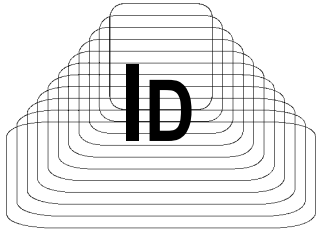
Листинг 6.9. Удаление лишних объектов

```
myDocument = app.activeDocument;
myAllItems = app.activeDocument.allPageItems;
myParagraphStyles =app.activeDocument.paragraphStyles;

// Параметры поиска:
// для пустых фреймов
searchString_1 = "/[a-Za-Я]/"

// для пустых абзацев
searchString_2 = "/^[ \t]*[\r\n]+[\r\n]/"
replaceString = "\r"
styleApplied = "";

myTextFrames = myDocument.textFrames
for (i=myAllItems.length-1; i>-1; i--){
  if (myAllItems[i].constructor.name == "TextFrame") {
    // Удаление пустых текстовых фреймов
    if (myAllItems[i].contents.match( searchString_1 ) == null) {
      myAllItems[i].remove()
    } else {
      myParagraphs = myAllItems[i].paragraphs;
      for (j=myParagraphs.length-1; j>-1; j--){
        with(myParagraphs[j]){
```

```
            // Для поиска неиспользуемых стилей
            styleApplied += appliedParagraphStyle.name;

            // Обрабатываем исключения
            if (footnotes.length<1 && contents != ↳
                1396927554 && contents != 1397778242) {
              // Удаление пустых абзацев
              myAllItems[i].paragraphs[j].contents = ↳
                myAllItems[i].paragraphs[j]. ↳
                contents.replace(searchString_2, replaceString)
            }
          }
        }
      }
    }
}

// Удаление неиспользуемых стилей
for (i=myParagraphStyles.length-1; i>1; i--){
  ix = styleApplied.search(myParagraphStyles[i].name);
  if (ix == -1) {
    myParagraphStyles[i].remove();
  }
}

// Уборка монтажного стола
myObjs = myDocument.pageItems.everyItem().parent;
for (var j = myObjs.length - 1; j >= 0; j--) {
  // Проверяем родителя.
  // Если страница, то объект пропускаем
  if (myObjs[j].constructor.name == "Page") continue;
  // Иначе — в корзину
  myDocument.pageItems[j].remove());
}

// Удаляем неиспользуемые цвета
myUnusedSwatches = myDocument.unusedSwatches
for (var j = myUnusedSwatches.length - 1; j >= 0; j--) {
  myUnusedSwatches[j].remove()
}
```

Удаление заякоренных текстовых блоков в скрипт не включено, поскольку это потенциально опасно для уже сверстанных материалов — ведь в них вполне могут присутствовать заякоренные объекты, которые были специально размещены именно таким образом. В принципе можно обойти и это ограничение, если якорям были назначены стили — в таком случае достаточно проверять наличие у них стиля (`ObjectStyle`), и если он **Basic Frame** (стиль по умолчанию, т. е. рука верстальщика его не касалась), то удалять. Конечно, свое название скрипт оправдывает лишь частично, на самом деле он является только подготовкой к основной уборке, которая будет затеяна в скрипте "Автоматический корректор" *(см. разд. 7.3.5)*.

ГЛАВА 7



Поиск и замена

Поиск и замена в тексте — одна из наиболее часто встречающихся операций. К сожалению, встроенные средства редактора достаточно скромны и основное отличие InDesign CS и CS2 от Notepad (Блокнота) в этом вопросе состоит лишь в поддержке форматирования. Так, обе этих версии не позволяют замену фрагментов по гибкой маске (например, проводить замену слов, начинающихся с определенных символов) — а только один четко определенный фрагмент на другой. Так, например, заменить "кв. м" на "м^2" невозможно, сочетание "цифра-тире-цифра" на "цифра-неразрывное тире-цифра", "цифра-пробел" на "цифра-неразрывный пробел" также — примеров можно привести массу.

В то же время в JavaScript встроены мощные функции поиска/замены (на базе регулярных выражений), что позволяет выполнять операции любой степени сложности. Одно ограничение: JavaScript работает с текстом исключительно как со строкой, поэтому форматирование (по идее) сохраняться не должно. В то же время заложенный в InDesign механизм сохранения форматирования работает корректно не только при замене текста внутренними функциями (через **Find/Replace**), но также и через JavaScript. Во всяком случае, при использовании внешнего механизма автору не известна ни одна проблема потери форматирования. Дополнительное достоинство скриптинга — возможность формирования целого пакета операций поиска/замены с последующим исполнением их одним нажатием клавиши.

Все операции поиска/замены условно можно разделить на три типа:

- традиционные (поиск/замена одного фрагмента на другой с форматированием);
- поиск специальных символов (короткое тире, неразрывный дефис, символ табуляции и т. п.);
- поиск без четких критериев (по маске).

Поскольку, как правило, в процессе поиска/замены меняется количество символов в тексте, потенциально возможно возникновение проблемных ситуаций: например, если текста стало меньше, то могут возникнуть ссылки на несуществующий текст. В то же время, в своей практике автор ни разу не встречался с такими проблемами (спасибо программистам из Adobe), если же вдруг они возникнут, решение достаточно простое: нужно искать не с начала, а с конца текста. В таком случае никакая операция замены к смещению в индексации символов не приведет.

7.1. Встроенные функции InDesign CS, CS2

Для операций поиска/замены в InDesign CS и CS2 предусмотрено использование объектов `findPreferences` и `changePreferences` соответственно, а параметры поиска или замены (аналоги опций в окне **Find/Replace**) задаются как свойства этих объектов. Иными словами, объекты являются ничем иным, как стилем для выполнения соответствующей операции. Перед проведением поиска/замены их желательно очищать (как известно, InDesign при повторном вызове окна **Find/Replace** хранит ранее введенные значения) — для этого предусмотрено специальное значение `NothingEnum.nothing`.

7.1.1. Поиск текста без форматирования

Вот как происходит поиск неформатированного текста встроенными функциями InDesign:

```
// Сброс предыдущих настроек. Операция обязательна, поскольку InDesign
// помнит параметры предыдущего поиска
app.findPreferences = NothingEnum.nothing;
app.changePreferences = NothingEnum.nothing;

// Поиск в документе текста "Пример".
app.findPreferences.findText = " Пример";

// Установка пользовательских настроек для поиска
with (app.findPreferences){
  caseSensitive = false;
  wholeWord = false;
}
var myFoundItems = app.documents[0].findText();
alert("Найдено " + myFoundItems.length + " совпадений");
```

7.1.2. Замена текста без форматирования

Кроме поиска, InDesign может заменять один фрагмент текста на другой. Рассмотрим простейший пример замены текста без учета форматирования (листинг 7.1).

Листинг 7.1. Замена текста без форматирования

```
// Сброс предыдущих настроек
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;

// Установка опций для поиска
app.findChangeTextOptions.caseSensitive = false;
app.findChangeTextOptions.wholeWord = false;

// Замена одной строки "copy" на другую "text"
app.findPreferences.findText = "copy";
app.changePreferences.changeTo = "text";
app.documents[0].changeText();

// Подготовка к следующему поиску/замене
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;
```

7.1.3. Замена текста и форматирования

Более сложная замена — с учетом форматирования. При этом атрибуты форматирования задаются как свойства соответствующих объектов (`findPreferences` или `changePreferences`), подчиняясь общим правилам установки атрибутов (листинг 7.2).

Листинг 7.2. Замена текста и форматирования

```
app.findPreferences = NothingEnum.nothing;
app.changePreferences = NothingEnum.nothing;
findText = "find text";
with(app.findPreferences){
  pointSize = 24 pt;
}
changeTo = "replacement text";
with(app.changePreferences)
```

```
{
  // Замена одного текста на другой
  // с переводом результата в верхний регистр:
  position = Position.superscript;
  capitalization = Capitalization.smallCaps;
}
app.activeDocument.search(findText, undefined, undefined, changeTo, ↳
  app.findPreferences, app.changePreferences);
```

В случае, если нужно изменять лишь несколько параметров форматирования, можно использовать более краткую запись, указывая параметры в соответствующем разделе непосредственно при вызове метода `search()`, например, так, как показано в листинге 7.3.

Листинг 7.3. Краткая запись для поиска/замены

```
app.findPreferences = NothingEnum.nothing;
app.changePreferences = NothingEnum.nothing
findText = "find text";
changeTo = "replacement text";

// Установка форматирования для заменяющего текста
app.activeDocument.search(findText, undefined, undefined, changeTo, ↳
  undefined, {position:Position.superscript});

// Поиск текста в верхнем индексе
app.activeDocument.search(findText, undefined, undefined, changeTo, ↳
  {position:Position.superscript}, undefined);
```

7.2. Возможности InDesign CS3

В скриптинге произошли значительные изменения, касающиеся возможностей поиска и замены, что отражает расширение возможностей данной операции в этой версии.

- Появились новые объекты, реализующие расширенные функции поиска и замены. Теперь все объекты, касающиеся задания параметров поиска/замены, условно можно разбить на три группы:
  - для работы исключительно с текстом (в том числе форматированным) — `find`**`Text`**`Preferences` и `change`**`Text`**`Preferences` соответственно;
  - для использования регулярных выражений (`find`**`Grep`**`Preferences` и `change`**`Grep`**`Preferences`);

- для поиска специальных типографских символов (`find`**`Glyph`**`Preferences` и `change`**`Glyph`**`Preferences`);

❒ появился новый объект `findChangeText`**`Options`**, хранящий общие настройки поиска/замены;

❒ значительно расширился список параметров поиска.

Кроме существовавших ранее глобальных опций поиска `caseSensitive` и `wholeWord` появились новые (также могут быть либо `true`, либо `false`): `includeFootnotes` (поиск в сносках), `includeHiddenLayers` (в скрытых слоях), `includeLockedLayersForFind` (в заблокированных слоях), `includeLockedStoriesForFind` (в заблокированных материалах) и `includeMasterPages` (на мастер-страницах).

Соответственно в Creative Suite 3 полный набор глобальных параметров имеет вид:

```
with (app.findChangeTextOptions)
  caseSensitive = false;
  includeFootnotes = false;
  includeHiddenLayers = false;
  includeLockedLayersForFind = false;
  includeLockedStoriesForFind = false;
  includeMasterPages = false;
  wholeWord = false;
}
```

Как выполняется операция поиска/замены в самой последней версии, показано в листинге 7.4.

Листинг 7.4. Поиск и замена в InDesign CS3

```
// Сброс параметров — неизменная операция
app.findTextPreferences = NothingEnum.nothing;
app.changeTextPreferences = NothingEnum.nothing;

// Установка опций поиска
with (app.findChangeTextOptions)
  caseSensitive = false;
  includeFootnotes = false;
  includeHiddenLayers = false;
  includeLockedLayersForFind = false;
  includeLockedStoriesForFind = false;
  includeMasterPages = false;
  wholeWord = false;
}
```

```
app.findTextPreferences.pointSize = 24;
app.changeTextPreferences.pointSize = 10;

app.activeDocument.changeText();
```

7.2.1. Синтаксис регулярных выражений, используемых в InDesign CS3

Регулярные выражения — выражения, предназначенные для поиска текста по нечетким признакам: фактически вы указываете шаблон, который задает лишь минимальные условия для соответствия, а общее количество вариантов, построенных на базе шаблона, может быть огромным. С подобным подходом сталкиваются при поиске файлов в Windows: например, `test?.*` означает поиск файлов с любым расширением, у которых имя начинается с test; `*.txt` приводит к поиску любых файлов с расширением txt и т. д.

Термин "регулярное выражение" является общим названием подхода к поиску файлов, при этом конкретная реализация механизма поиска по маске в разных приложениях имеет отличия, порой достаточно существенные, что хорошо видно при сравнении JavaScript, Visual Basic for Applications и InDesign. Часто регулярные выражения сокращенно называют RegEx (от англ. *Regular Expression*), еще одно обозначение — GREP.

Вот, например, какой вид будет иметь в InDesign выражение для поиска адресов электронной почты:

```
app.findGrepPreferences.findWhat = "(?i)[A-Z]*?@[A-Z]*?[.]...";
```

Несколько непривычно, не так ли? Тем не менее эта конструкция работает, мало того — она без проблем найдет ЛЮБОЙ адрес почты, что при использовании обычных инструментов InDesign можно только мечтать.

Как вы уже, наверное, догадались, регулярные выражения являются отдельным объектом в InDesign, со своими свойствами и методами. Соответственно, использование их подчиняется общим правилам работы с объектами.

Пример использования объекта `findGrepPreferences` представлен в листинге 7.5.

Листинг 7.5. Использование объекта `findGrepPreferences`

```
app.find Grep Preferences = NothingEnum.nothing;
app.change Grep Preferences = NothingEnum.nothing;
with (app.findChange Grep Options)
{
  includeFootnotes = false;
  includeHiddenLayers = false;
```

```
  includeLockedLayersForFind = false;
  includeLockedStoriesForFind = false;
  includeMasterPages = false;
}
```

Если вы используете регулярные выражения, опции для поиска форматированного текста нужно задавать для объекта `findGrepPreferences`:

```
app.findGrepPreferences.pointSize = 24;
app.changeGrepPreferences.underline = true;
```

Операция замены производится похожим методом:

```
app.activeDocument.changeGrep();
```

При использовании регулярних выражений опции поиска слова целиком (`wholeWord`) и учета регистра (`caseSensitive`) недоступны, поскольку такие возможности предусмотрены в синтаксисе регулярных выражений (табл. 7.1).

Таблица 7.1. *Конструкции, используемые в регулярных выражениях для поиска*

Действие	Обозначение
Поиск текста с учетом регистра	`(?i)`
Игнорирование регистра	`(?-i)`
Поиск выражения в начале слова	`\>`
Поиск выражения в конце слова	`\<`
Поиск выражения в начале и в конце слова	`\b`
Любой символ	`\s`
Пробел	`\w`

В листинге 7.6 приведен пример замены текста с разметкой, сделанной в PageMaker, для использования в InDesign.

Листинг 7.6. Замена текста с разметкой

```
var myName, myString, myStyle, myStyleName;
var myDocument = app.documents.item(0);
app.findGrepPreferences = NothingEnum.nothing;
app.changeGrepPreferences = NothingEnum.nothing;

// Поиск тегов разметки.
// Поскольку использующийся в ней символ "\" является служебным,
// для правильного его восприятия перед ним ставят такой же символ,
// сигнализируя о том, что дальше идет специальный символ.
```

```
app.findGrepPreferences.findWhat = "(?i)^<\\s*\\w+\\s*>";
var myFoundItems = myStory.findGrep();

if(myFoundItems.length != 0){
  var myFoundTags = new Array;
  for(var i= 0; i<myFoundItems.length; i++){
    myFoundTags.push(myFoundItems[i].contents);
  }

  // Собираем все использованные теги, повторы не включаем
  myFoundTags = myRemoveDuplicates(myFoundTags);

  // Мы собрали все использованные теги разметки
  for(i= 0; i< myFoundTags.length; i++){
    myString = myFoundTags[i];

    // По очереди ищем каждый тег
    app.findTextPreferences.findWhat = myString;

    // Получаем название тега
    myStyleName = myString.substring(1, myString.length-1);

    // Создаем стиль с названием тега, если он еще не существует
    try{
      myStyle = myDocument.paragraphStyles.item(myStyleName);
      myName = myStyle.name;
    }
    catch (myError){
      myStyle = myDocument.paragraphStyles.add({name:myStyleName});
    }

    // Применяем стиль ко всему тексту, имеющему текущий тег
    app.changeTextPreferences.appliedParagraphStyle = myStyle;
    myStory.changeText();

    // Подготовка к следующему поиску/замене
    app.changeTextPreferences = NothingEnum.nothing;
    app.changeTextPreferences.changeTo = "";

    // Собственно замена
    myStory.changeText();
    app.changeTextPreferences = NothingEnum.nothing;
  }
```

```
app.findGrepPreferences = NothingEnum.nothing;
}

function myRemoveDuplicates(myArray){
  // Поиск дубликатов в массиве с названиями тегов
  var myNewArray = new Array;

  // Сортируем — в таком случае при поиске дубликата
  // не потребуется пробегать по всему массиву — дубликаты
  // будут идти сразу за текущим элементом
  myArray = myArray.sort();

  // Создаем новый массив из одного элемента — текущего тега
  myNewArray.push(myArray[0]);
  if(myArray.length > 1){
    for(var i= 1; i< myArray.length; i++){
      if(myArray[i] != myNewArray[myNewArray.length -1]){
        // В новый массив повторения не попадут
        myNewArray.push(myArray[i]);
      }
    }
  }
  return myNewArray;
}
```

7.3. Использование возможностей JavaScript

Поскольку автор занимается в том числе верстkой изданий экономической направленности, которые изобилуют цифрами, сокращениями, а возможности InDesign CS2 в части замены довольно скромны, несмотря на внушительные размеры диалогового окна, встал вопрос как-то обойти его ограничения.

Для операций поиска/замены в JavaScript предусмотрено использование регулярных выражений — комбинаций специальных символов (подстановочных знаков), с помощью которых реализуется поиск по заданному шаблону. Полное описание их синтаксиса приведено в *разд. П2.6*, здесь же дается только в объеме, необходимом для создания данного скрипта. Кроме того, интересующиеся могут посетить специализированный сайт **http://www.regular-expressions.info/reference.html**, полностью посвященный вопросам поиска/замены, причем не только в JavaScript.

Любой цифре соответствует шаблон `[0-9]`, текстовому символу — `[а-я]`, если нужно включить и прописные буквы, то `[а-яА-Я]` и т. д. Вообще же в

диапазон поиска можно включать любой символ, нужно лишь помнить о особом предназначении знаков +, ?, *, |, ^, \ и точки.

Знак вопроса говорит о том, что символ может присутствовать, но не обязательно (т. е. результатом поиска по шаблону `шее?` в "длинношеее" будет "ше" и "шее"). Символ "плюс" (+) сообщает о том, что в тексте должно быть хотя бы одно совпадение предыдущего символа (`ше+` даст "ше", "шее" и "шеее"). Действие символа * похоже на +, но имеет более широкий охват: (`ше*` найдет кроме "ше" и "шее" также "ш" и "шеее"). Символ | является аналогией операции "ИЛИ": `[ко|ит]` эквивалентно поиску "кот" и "кит". Чтобы искать "ко" и "ит" одновременно, потребуется (как один из вариантов) использовать скобки: `(ко)|(ит)`. "Крышка" (^) имеет разный смысл в зависимости от места, где она стоит. Если в начале шаблона, то искомый фрагмент ищется только в начале строки, наличие "крышки" внутри диапазона говорит о том, что диапазон присутствовать не должен (т. е. по `[^а-я]` найдется все, кроме знаков алфавита, а `[^0-9]` найдет все, что не является цифрой).

Знак \ зарезервирован для специальных случаев. Одним из них является поиск специальных знаков `\t`, `\r`, `\n` (символ табуляции, символ абзаца и новая строка). Перед ними ставят еще один знак \, т. е. строка поиска будет `\\t` и `\\r` соответственно. Точка (.) используется для поиска любого знака ("`длин.+ `" найдет не только все слово "длинношеее", но и все оставшиеся слова до конца строки).

При поиске удобно пользоваться группировкой (заключать в скобки части поискового выражения), что дает дополнительную гибкость при операциях замены.

Замена имеет свою специфику. Для последующего использования найденных фрагментов текста предназначены комбинации от `\$1` до `\$9`, где цифра указывает порядковый номер выражения в скобках в строке поиска. Таким образом, поиску комбинации "цифра-тире-цифра" будет соответствовать выражение (`[0-9])-([0-9]`), или короче — `(\d)-(\d)` (специальный символ `d` — сокращение от *digital*, цифра), а замена в ней дефиса на тире будет выглядеть как `\$1-\$2`. Все заменяемые выражения окружаются кавычками и соединяются в цепочку через +.

Специальные символы (неразрывное тире и др.) не отображаются на экране, однако и они имеют свой код. Он может быть найден несколькими способами. Самый надежный — в начале любого абзаца вставить необходимый символ, выделить его и использовать вспомогательный скрипт:

```
alert(app.selection[0].paragraphs[0].charCodeAt[0])
```

(Есть подозрения, что в обновленной версии ExtendScript эта возможность убрана, автором тестировалась оригинальная сборка, идущая в поставке с CS2.)

7.3.1. Проверка регулярных выражений JavaScript

Процесс освоения синтаксиса регулярных выражений и особенностей работы с ними займет определенное время. Поэтому, чтобы его свести к минимуму, напишем утилиту, помогающую в поиске нужного выражения: вы вписываете свой вариант регулярного выражения, задаете текст, в котором будет проходить проверка, и скрипт выделяет в тексте фрагменты, соответствующие условиям поиска. Если соответствие не найдено, выдается предупреждение. Преимущество скрипта — оперативность: модифицируя строку поиска, вы сразу же проверяете ее на тексте, добиваясь нужного результата. Это позволит сэкономить значительное время — даже в сравнении с использованием возможностей ExtendScript Editor.

Используем уже рассматривавшуюся функцию проверки типа выделения — для уверенности, что выделен действительно текстовый, а не какой-либо другой объект:

```
Object.prototype.isText = function() {
  switch(this.constructor.name){
    case "InsertionPoint":
    case "Character":
    case "Word":
    case "TextStyleRange":
    case "Line":
    case "Paragraph":
    case "TextColumn":
    case "Text":
      "TextFrame":
      return true;
    default :
      return false;
  }
}
```

А вот сам скрипт (листинг 7.7).

Листинг 7.7. Проверка регулярного выражения

```
myErr = "Выбран не текст!";
myDoc = app.activeDocument;
myRange = app.selection[0];
if (!myRange.isText())
{
  errorExit(myErr);
```

```
} else {
  if (myRange.constructor.name == "InsertionPoint") {
    myRange = getParentTextFlow(myRange);
    myREtext = prompt("Введите регулярное выражение:", "");
    if (myREtext == null) {
      errorExit();
    }

    myRE = new RegExp(myREtext, "g");
    myTest = myRE.exec(myRange.contents);
    myReport = "Искомое выражение:"
    if (myTest == null)
    {
      myReport += "\nНичего не найдено";
    } else {
      while(myRE(myRange.contents)){
        // Определяем оставшийся после поиска фрагмент текста
        Start = myRE.lastIndex - myREtext.length;
        End = myRE.lastIndex-1;

        // Выделяем каждое соответствие по очереди
        myRange.characters.itemByRange(Start,End).select()
      }
    }
  }
}
```

Вспомогательные функции:

```
function errorExit(message) {
  if (arguments.length > 0)
  {
    alert(message);
    exit();
  }
}

function getParentTextFlow(theTextRef) {
  if (theTextRef.parent.constructor.name == "Cell")
  {
    return theTextRef.parent.texts[0];
  } else {
    return theTextRef.parentStory;
  }
}
```

7.3.2. Удаление пустых фреймов

Одна из наиболее часто используемых задач при работе с текстом — поиск и замена одного фрагмента на другой. Предположим, стоит задача очистить публикацию от пустых фреймов — она отлично подходит для демонстрации работы с содержимым текстовых фреймов. Будем считать пустыми такие фреймы, в которых нет ни одной латинской или кириллической буквы, т. е. у которых содержимое состоит из одних только переносов строк, символов табуляции или пробелов. Пример из листинга 7.8 отлично демонстрирует возможности регулярных выражений.

Листинг 7.8. Удаление пустых фреймов

```
myDocument = app.activeDocument;
myTextFrames = myDocument.textFrames

// Строка поиска — искать любые кириллические или латинские символы
searchString = "/[a-Za-Я]/"
for (i=0; i< myTextFrames.length; i++){
  // Если поиск ничего не дал, т. е. если в тексте
  // нет ни одного из искомых символов
  if (myTextFrames[i].contents.match( searchString ) == null) {
    myTextFrames[i].remove()
    i--
  }
}
```

После каждого удаления фрейма количество циклов уменьшаем на единицу — ведь фреймов стало меньше.

7.3.3. Удаление пустых абзацев

Пустым абзацем будем считать такой, в котором нет других символов, кроме символов абзаца или пробелов с табуляторами.

Для решения подобной задачи также будем использовать возможности регулярных выражений (листинг 7.9). Некоторые могут возразить, что в InDesign возможно последовательно использовать комбинацию для поиска `\p\p` с заменой на `\p`. Это так. Однако такой вариант не позволит удалить абзацы, в которых есть один или несколько символов пробела, табуляции, а также их комбинации. Регулярные выражения как раз и предназначены для поиска по таким нечетким критериям.

Листинг 7.9. Удаление пустых абзацев

```
// Поиск: в начале абзаца могут быть пробелы или символы табуляции,
// но обязательно должны присутствовать хотя бы два знака абзаца
// или новой строки

// [\r\n]+ относятся к поиску "лишних" символов абзаца и новых строк
searchString = "/^[ \t]*[\r\n]+[\r\n]/"

// Меняем на один абзац
replaceString = "\r"

myTextFrames = app.activeDocument.textFrames;
for (i=0; i< myTextFrames.length; i++){
  for (j=0; j< myTextFrames[i].paragraphs.length; j++){
  try {
    with(myTextFrames[i].myParagraphs[j]){
      if ((footnotes.length < 1) && (contents != ↵
           SpecialCharacters.columnBreak) && (contents != ↵
           SpecialCharacters.pageBreak)) {
        myTextFrames[i].myParagraphs[j].contents = ↵
          contents.replace(searchString, replaceString)
      }
    }
  }
  catch(err) alert("Some problems encountered!")
}
```

Разберем подробнее строку поиска. Сначала указываем, что поиск должен проводиться только в начале абзаца, затем в квадратных скобках — набор символов, которые должны идти первыми (пробел и знак табуляции). Знак * говорит о том, что любой из символов может как встречаться, так и не встречаться вообще. Также учтем, что символов абзаца может быть несколько (`[\r\n]+`). И конец последовательности — `[\r\n]`.

Давайте рассмотрим все возможные при поиске ситуации. Если в абзаце только текст, то никаких проблем быть не должно. Однако кроме текста в абзаце может быть графика, таблица, ссылки — вот тут кроются источники потенциальных проблем. Поэтому мы должны уделить особое внимание таким объектам.

Если протестировать скрипт на таких ситуациях (такой подход — обычное дело при скриптинге), то увидим, что все ситуации обрабатываются корректно, кроме абзацев с примечаниями, а также служебными абзацами — они

часто играют роль выталкивателей для текста и т. п. — поэтому в строку поиска была добавлена проверка наличия символов `SpecialCharacters.columnBreak` и `SpecialCharacters.pageBreak`.

Итак, единственным реальным исключением из проверки текста станут сноски — будем надеяться, что в тексте их окажется не много.

7.3.4. Автоматизация форматирования

Автоматизация форматирования — одна из самых популярных областей применения скриптинга. Рассмотрим несколько примеров, иллюстрирующих возможности программирования InDesign в данной сфере.

Первый пример из реальной жизни. Допустим, нужно все слова, в которых используются латинские буквы, сделать наклонными. Скрипт будет занимать всего несколько строк:

```
w = app.activeDocument.selection[0].words

for(j=0; j<w.length; j++){
  if (w[j].contents.search(/[A-Za-z]+/)!=-1)
    w[j].fontStyle = 'Italic'
}
```

Для удобства мы обрабатываем слова только в выделенной части. Это позволяет гибко использовать скрипт. Диапазон `[A-Za-z]` задает все латинские буквы, в том числе с учетом регистра.

Рассмотрим более сложный пример, позволяющий упростить определенное форматирование фрагментов текста. Например, в публикации имеется рубрика с интервью, которое нужно отформатировать следующим образом:

Q: текст1 (серый цвет)

A: (красный цвет) абзац 1

абзац 2

...

абзац *N*

Q: текст2 (серый цвет)

A: (красный цвет) абзац *N* +1

абзац *N* + 2

...

абзац *N* + *M*

В принципе, ничего сложного — выделяешь ручками один фрагмент, переходишь к другому и т. д. Но мы-то уже не будем делать это вручную — ведь четко просматривается принцип форматирования: до двоеточия идет одна стилевая разметка, затем — вторая и т. д., и грех этим не воспользоваться. Используем возможности JavaScript, рассматривая знак двоеточия как разделитель (листинг 7.10).

Листинг 7.10. Форматирование интервью

```
aD = app.activeDocument;
pars = aD.selection[0].paragraphs
pS = aD.paragraphStyles;
pSA = [];
cS = aD.characterStyles;
cSA = [];
for (i=2; i<pS.length; i++){
  switch(pS[i].name){
    case 'Question':
      // Абзац с серым цветом шрифта
      pSA[' Question '] = i;
      break;
    case 'Answer':
      pSA[' Answer '] = i;
      break;
  }
}
for (i=0; i<cS.length; i++){
  switch(cS[i].name){
    case 'Red':
      cSA['Red'] = i;
      break;
  }
}

for (i=0; i<pars.length; i++){
  while (pars[i+1]!=null){
    s = pars[i].texts[0].contents.search(':')
    if (s>0 && s<10){
      pars[i].applyStyle(pS[pSA ['Question']], true)
      pars[i+1].applyStyle(pS[pSA ['Answer']], true)
      s1 = pars[i+1].texts[0].contents.search(':')
      pars[i+1].characters.itemByRange(0, s).applyStyle(cS ↳
        [cSA ['Bold']], true)
      i++
```

```
    }else{
      pars[i].applyStyle(pS [pSA ['Answer']], true)
    }
  }
}
```

Для корректной работы скрипта было решено использовать стили символов: первый необходимый — `Red` — им будем форматировать блок ответчика. Для вопроса автора используем стиль абзацев `Author`, для текста абзац 1, абзац 2, ..., абзац *N* — стиль `Reply`.

Начало — традиционное: сканируем публикацию на наличие стилей и получаем на них сноски. Следующий шаг — начинаем просматривать абзацы. Делаем проверку: если знак двоеточия находится в самом начале абзаца (вплоть до 10-го символа), абзац форматируем стилем `Question`. Проверка `s>0` нужна для того, чтобы быть уверенным, что двоеточие все же имеется (если его нет, значение, возвращенное поиском, будет –1 , которое меньше 10, и возникнет ошибка).

```
pars[i].applyStyle(pS[pSA['Question']], true)
```

Следующему абзацу присваиваем стиль `Answer`, поскольку вопрос всегда занимает ровно один абзац:

```
pars[i+1].applyStyle(pS[pSA['Answer']], true)
```

Теперь определяем положение двоеточия в абзаце с ответом и все символы с начала строки до позиции, занимаемой двоеточием, форматируем стилем символов `Bold` (используем метод `itemByRange()`). Соответственно после этого можно переходить сразу на третий абзац, поскольку первые два уже отформатированы (увеличиваем счетчик). Если в третьем абзаце двоеточие в определенном диапазоне не обнаружено, значит, продолжается ответ, а мы продолжаем присваивать стиль `Answer`.

```
pars[i+1].characters.itemByRange(0, s).applyStyle(cS [cSA['Bold']], true)
```

Как только снова находим двоеточие, значит, снова пришла очередь абзаца с вопросом, и т. д. до конца области выделения.

Осталось предусмотреть нюанс, возникающий при подходе цикла к последнему абзацу: дело в том, что объект `pars[i]` в таком случае существовать будет, а вот `pars[i+1]` — уже нет. Чтобы скрипт корректно заканчивал свое выполнение, добавляем еще одну проверку, которая разрешает дальнейшее его исполнение только в случае, если после текущего абзаца существует еще один:

```
while (pars[i+1]!=null)...
```

Данный скрипт, кроме практической ценности, после некоторых модификаций может быть использован и для других аналогичных вариантов, как, например, в таком:

```
[Что:] [название]
[Где: страна проведения
Когда: дата]
[Особенности проведения:] текст
```

где границы разных стилей обозначены квадратными скобками.

7.3.5. Автоматический корректор

Данный скрипт был написан для автоматического переведения материала, не прошедшего корректорскую проверку, к профессиональному виду (или прошедшего через корректора, но все равно нуждающегося в дополнениях) и постоянно используется автором в своей работе.

Сначала был составлен список необходимых преобразований:

- несколько подряд идущих пробелов менять на один;
- между цифрами — неразрывное тире;
- вокруг тире — только неразрывные пробелы;
- пробел после цифры — неразрывный (например, "3 кг");
- кв. м и куб. м — соответственно $м^2$ и $м^3$;
- несколько подряд идущих прописных букв (часто используются для обозначения формы собственности предприятия) — не отрывать от следующего слова (например, АОЗТ "Пластмасс");
- в списках для их отбивки по левому краю, после пробела — символ абзацного отступа.

При желании список можно дополнять собственными правилами, в книге показывается лишь сам подход.

Первым в нашем списке идет замена множества пробелов.

Для замены двух и более пробелов на один потребуется еще немного расширить наши познания в регулярных выражениях. Для поиска текстового фрагмента заданной длины в них предусмотрена конструкция `{min, max}`, которая определяет минимально и максимально допустимую длину поискового фрагмента. В нашем случае достаточно составить конструкцию `{2,}` — не указанный явно второй аргумент говорит о том, что максимум не ограничен — что нам с вами и нужно. Меняем на один пробел (" ").

Для реализации второго условия подойдет строка `(\d)-(\d)`, но для расширения функциональности ее лучше заменить на `(\d) ?-|— ?(\d)`. Это даст воз-

можность расширить охват: будем искать числа, после которых может быть пробел (в подаваемом на верстку материале случается всякое), затем стоит либо "-" либо "—", после чего опять же возможен пробел, и наконец снова идет цифра. Таким образом, мы вовлекаем в диапазон поиска значительно большее количество различных вариантов написания, тем самым замена будет универсальнее. Скобки нужны для сохранения результатов поиска при последующей замене.

Строка замены в этом случае будет выглядеть как `"$1"+"\u2011"+"$3"`. Она означает, что заменяемое выражение будет состоять из числа, найденного первым (которое в первых скобках соответствует `"$1"`), после чего вставляем неразрывное тире и дописываем второе найденное число (`"$2"`). Поскольку код выдается в Unicode (для универсальности), мы это отмечаем, помещая впереди комбинацию `"\u"` (Unicode).

Аналогичным образом для третьей замены меняем обычные пробелы вокруг тире на неразрывные (их шестнадцатеричный код `A0`, он предваряется ключом `\x`).

В четвертом шаге для замены пробела после числа на неразрывный используем комбинацию `\d`. Соответственно строка замены будет такой: `"$1"+"\xA0"`. Однако после чисел могут стоять множители (тыс., млн, млрд), поэтому для корректной обработки связки "число-множитель-размерность" можно воспользоваться такой цепочкой действий: сначала привяжем к цифрам размерности, а потом — сами множители. В результате для поиска получим:

```
find1 = \d;
find2 = (тыс|млн|млрд)\.? ?;
```

Таким образом, независимо от того, стоит после множителя точка и/или пробел, мы будем корректно отрабатывать все возможные комбинации. В этом и состоит главная прелесть регулярных выражений — они дают чрезвычайную гибкость в поиске нужных фрагментов. Строки замены в таком случае будут иметь вид:

```
replace1 = $1+\xA0
replace2 = $1+\.\xA0
```

Напомню, `\xA0` — шестнадцатеричное значение неразрывного пробела, а `\.` — точка (знак `\` перед ней стоит, чтобы не было путаницы с `"."` — подстановочным знаком для любого символа). Кстати, для того чтобы сочетания "м2", "м3" обрабатывались корректно (после цифры оставался обычный пробел), исключим из диапазона захвата сочетание "м, идущее перед цифрой". В результате `find1` изменится на `([^м])(\d)` , а `replace1` — на `"$1"+"$2"+"\xA0"`.

Следующий шаг — сокращения типа АО, ЗАО, АОЗТ и пр. Чтобы не разрывать форму собственности от самого названия, достаточно указать мини-

мальную длину искомой конструкции — 2. Поиск ведем только среди прописных букв (А—Я). В результате для поиска получим:

`([А-Я]{2,}) ("|")`, уточнение (`"|"`) внесено для большей гибкости после апробации на реальных текстах, учитывает наличие кавычек в названии организаций.

Выражение для замены:

```
$1+\xA0+$2
```

Очередной шаг в принципе не относится непосредственно к теме поиска/замены, однако он упрощает работу со списками. Предусмотрим автоматическое добавление символа левой границы текста (Indent Here) по пробелу, следующему сразу за отбивочным знаком списка (рис. 7.1), т. е. ищем выражение `^[\x95\u2013\u2014\x2D]`. Знак `^` обеспечивает поиск только в начале строк; `\x95`, `\u2013`, `\u2014` и `\x2D` — варианты отбивки (круглый маркер, табуляция, длинное тире либо дефис). Замена — на `"$1"+"\u2002"+"\x07"`, где `"\u2002"+"\x07"` — коды пробела фиксированной ширины и левой границы текста соответственно. Определение значений всех спецсимволов через дополнительный скрипт и выделение было описано ранее.

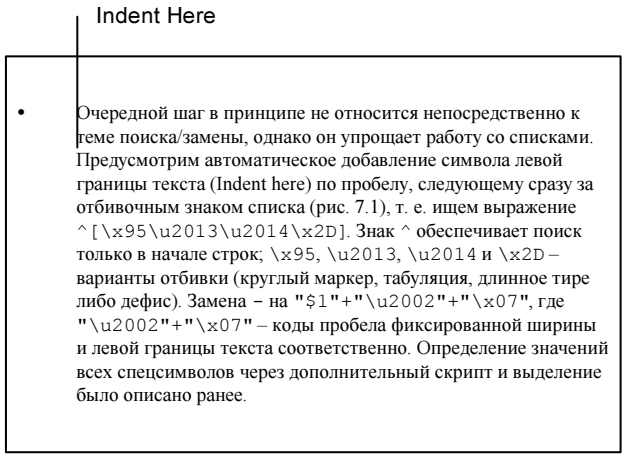


Рис. 7.1. Граница текста в списке

Последнее действие — перевод выражений "кв.м", "куб.м" в принятую форму ($м^2$, $м^3$). Его будем проводить стандартными средствами InDesign.

Итак, основная работа выполнена. Осталось только оформить наши записи в виде скрипта (листинг 7.11). Регулярные выражения в JavaScript задаются между двумя косыми чертами (слэшами), а ключ `g` означает многократную замену (global). Также предусмотрен ключ `i` — игнорирование регистра (т. е.

менять как строчные, так и прописные буквы), но нам он не понадобится. Поиск будем проводить по очереди в каждом абзаце выделенного текстового фрагмента. Итак, можем записать для блока базовых преобразований.

Листинг 7.11. Автоматический корректор

```
find1 = / {2,}/g;
replace1 = " "    // Замена множества пробелов

find2 = /(\d) ?(-|—) ?(\d)/g;
replace2 = "$1" + "\u2011" + "$3"  // Неразрывное тире между цифрами,
                                   // удаление лишних пробелов

find3 = /(.) (-|—) (.)/g;
replace3 = "$1" + "\xA0" + "\u2013" + "\xA0" + "$3"  // Неразрывные
                                                     // пробелы вокруг тире

find40 = /([^м])(\d) /g;
replace40 = "$1"+"$2"+"\xA0"    // Пробел после числа - неразрывный

find41 = /(тыс|млн|млрд)\.? ?/g;
replace41 = "$1"+"\.\xA0"       // Учитываем множители

// Остальные операции поиска/замены:
find5 = /([А-Я]{2,}) ("|")/g
replace5 = "$1"+"\xA0"+"$2"     // Не отрывать форму собственности
                                // от названий

find6 = /^([\x95\u2013\u2014\x2D])/g;
replace6 = "$1" + "\u2002" + "\x07"
// В списке — фиксированный пробел + Indent Here

find7 = /^(м\.кв\.)/g;
replace7 = "м@@"

find8 = /^(м\.куб\.)/g;
replace8 = "м@@@"
```

С инструментарием JavaScript закончили, вернемся опять к возможностям InDesign. Определим диапазон поиска. Поиск будем проводить только в текстовых абзацах, игнорируя таблицы и иллюстрации. Кроме того, как показала практика, необходимо учитывать наличие сносок, поскольку InDesign некорректно обрабатывает поиск в них. Анализ объектной модели программы по-

казал, что в параметрах абзаца предусмотрены свойства `tables` (наличие таблиц), `allGraphics` (иллюстраций) и сносок (`footnotes`), чем мы и воспользуемся.

Итак, проводим замены по очереди в каждом выделенном абзаце (листинг 7.12).

Листинг 7.12. Замена в таблицах, иллюстрациях, сносках

```
pars = app.selection[0].paragraphs
for (i=0; i<pars.length; i++) {
  if ((pars[i].tables.length<1) && (pars[i].allGraphics.length<1) && ↳
      (pars[i].footnotes.length<1)) {
    myContents = pars[i].contents.replace(find1, replace1)
    myContents = myContents.replace(find2, replace2)
    myContents = myContents.replace(find3, replace3)
    myContents = myContents.replace(find40, replace40)
    myContents = myContents.replace(find41, replace41)
    myContents = myContents.replace(find5, replace5)
    myContents = myContents.replace(find6, replace6)
    myContents = myContents.replace(find7, replace7)
  ...
  }
}
```

В случае, если вы захотите создать собственные правила автозамены, вам потребуется экспериментировать как со строкой поиска/замены, так и с порядком выполнения операций: результат, полученный от предыдущей операции, используется при последующей, что требует постоянного внимания. По своему опыту скажу, что проще использовать иную запись для поиска/замены: строки поиска проще записывать в виде одного массива, замены — в другой, и по очереди их считывать (листинг 7.13).

Листинг 7.13. Альтернативный вариант задания элементов массива

```
var findA = new Array();
var replaceA = new Array();

findA[findA.length] = ' {2,}';
replaceA[replaceA.length] = ' '

findA[findA.length] = '^[ \t]+';
replaceA[replaceA.length] = ''
```

```
findA[findA.length] = '(.) (-|—) (.)';
replaceA[replaceA.length] = '$1' + '\xA0' + '\u2013' + '\xA0' + '$3'
```

И так далее. При первом упоминании массив пуст, поэтому `findA.length = 0`, что дает возможность задать его первый элемент, после чего длина массива становится равной 1, что позволяет задать следующий элемент и т. д.

По окончании заменяем прежнее содержимое абзаца новым:

```
pars[i].contents = r
```

Теперь перейдем к переводу цифр в верхний индекс. Напрямую уже использовавшимся методом `replace()` воспользоваться не удастся (JavaScript работает только с неформатированным текстом), поэтому для такого случая будем использовать возможности InDesign. Напомню синтаксис замены:

search (for, wholeWord, caseSensitive, replaceWith, withFindAttributes, withChangeAttributes)

Поэтому в нашем случае получим:

```
app.search("@@@", undefined, undefined, "3", undefined, ↳
  {position:Position.superscript})
app.search("@@", undefined, undefined, "2", undefined)
```

В последней строчке мы не повторили задание форматирования — по той простой причине, что в предыдущей строке настройки форматирования мы не сбрасывали, поэтому они остались неизменными с предыдущей операции.

Вот, собственно, и весь скрипт. На очень больших объемах его выполнение занимает определенное время, однако если текст у вас перед верстkой вычитывается, то некоторые операции замены можно исключить (например, поиск многократных пробелов), что положительным образом отразится на скорости работы скрипта. Среди достоинств предложенного решения — компактность и легкость добавления своих собственных правил обработки текста.

И последнее замечание. Выделение используется для гибкости: чтобы можно было отдать подготовленный раздел публикации, как только закончена работа над ним (для оперативности разделы по мере их подготовки выкладываются на сервере в виде PDF) вместо того, чтобы ждать окончания работы над всем изданием. Вам же, возможно, больше понравится обработка всего документа — это нетрудно сделать путем добавления в скрипт нескольких строк.

А теперь — полный текст скрипта (листинг 7.14).

Листинг 7.14. Полный текст скрипта

```
app.findPreferences = null;
app.changePreferences = null;
myDocument = app.activeDocument
```

```
var findA = new Array();
var replaceA = new Array();

findA[findA.length] = ' {2,}';
replaceA[replaceA.length] = ' '

findA[findA.length] = '^[ \t]+';
replaceA[replaceA.length] = ''

findA[findA.length] = '(.) (-|—) (.)';
replaceA[replaceA.length] = '$1' + '\xA0' + '\u2013' + '\xA0' + '$3'

findA[findA.length] = '^([\uF0A7\u2022-\u2013\u2014\x2D])[ \t]';
replaceA[replaceA.length] = '\u2022' + '\u2002' + '\x07'

findA[findA.length] = '([0-9]) ';
replaceA[replaceA.length] = '$1'+'\xA0'

findA[findA.length] = '(тыс|млн|млрд)(\.?) ';
replaceA[replaceA.length] = '$1'+'$2'+'\xA0'

findA[findA.length] = '([A-ZA-Я]{2,}) (["“A-ZA-Я])';
replaceA[replaceA.length] = '$1'+'\xA0' + '$2'

findA[findA.length] = '(%)';
replaceA[replaceA.length] = '\u2009'+'$1'

findA[findA.length] = '([а-я])-([а-я])';
replaceA[replaceA.length] = '$1'+'\u2011'+'$2'

findA[findA.length] = '( [0-9]{1,3}) ?([0-9]{3,3} )';
replaceA[replaceA.length] = '$1'+'\u2008'+'$2'

findA[findA.length] = '([0-9])-([а-я])';
replaceA[replaceA.length] = '$1'+'\u2011'+'$2' // 10-летние и т. п.

findA[findA.length] = '(п\.) ?(п)';
replaceA[replaceA.length] = '$1'+'\u200A'+'$2'

findA[findA.length] = 'кв\. ?м';
replaceA[replaceA.length] = 'м@@'

findA[findA.length] = 'куб\. ?м';
replaceA[replaceA.length] = 'м@@@'
```

```
mySelection = myDocument.selection[0]
pars = (mySelection.length==0) ? mySelection.parentStory.paragraphs : ↳
  mySelection.paragraphs

for(j=0; j<pars.length; j++){
  if (pars[j].tables.length<1 && pars[j].allGraphics.length<1 && ↳
      pars[j].footnotes.length<1 && pars[j].contents!=1396927554 && ↳
      pars[j].contents!=1397778242){
    try {
      for (var i in findA) {
        pars[j].contents = ↳
          pars[j].contents.replace(eval('/'+findA[i]+'/g'), replaceA[i])
      }
      app.search("@@@", undefined, undefined, "3", ↳
        undefined, {position:Position.superscript});
       myDocument.search('@@', undefined, undefined, "2");
    }
    catch (err) {
      alert(pars[j].contents+ err.name)
    }
  }
}
```

7.3.6. Расстановка переносов

На данном этапе мы обладаем достаточными знаниями для того, чтобы создать еще один скрипт, который расставляет переносы в русском и украинском языках (листинг 7.15).

Листинг 7.15. Расстановка переносов

```
AnyLetters = "[єіїгабвгдеежзийклмнопрстуфхцчшщъыьэюя]";
Vowels = "[аеєиоуыэюя]";
Consonant = "[бвгджзклмнпрстфхцчшщ]";
Specials = "[йъь]";
Except = [["^пос~т"]];

hyp_symbol = "\u00AD";
hyp_replace = "$1" + hyp_symbol + "$2";

re1 = new RegExp("(" + Specials +")(" + AnyLetters + AnyLetters +")","ig");
re2 = new RegExp("("+ Vowels +")(" + Vowels + AnyLetters +")","ig");
```

```
re3 = new RegExp("(" + Vowels + Consonant +")(" + Consonant + Vowels
+")","ig");
re4 = new RegExp("(" + Consonant + Vowels +")(" + Consonant + Vowels
+")","ig");
re5 = new RegExp("(" + Vowels + Consonant +")(" + Consonant + Consonant +
Vowels +")","ig");
re6 = new RegExp("(" + Vowels + Consonant + Consonant + ")(" + Consonant
+ Consonant + Vowels + ")","ig");
re7 = new RegExp("(" + hyp_symbol + Consonant + Vowels + ")(" + Consonant
+ Vowels +")","ig");

if(app.selection[0].length != 0) {
  myContents = app.selection[0].contents
  myContents = myContents.replace(re1, hyp_replace)
  myContents = myContents.replace(re2, hyp_replace)
  myContents = myContents.replace(re3, hyp_replace)
  myContents = myContents.replace(re4, hyp_replace)
  myContents = myContents.replace(re5, hyp_replace)
  myContents = myContents.replace(re6, hyp_replace)
  myContents = myContents.replace(re7, hyp_replace)
  for (j in Except){
    exceptions = Except[j].toString().split('~')
    myContents = a.replace(eval("/(" + exeptions[0] + ")"  + hyp_symbol +
"?(" + exeptions[1] + ")(" + RusN +")/i"), "$1" + "$2" + hyp_symbol +
"$3")
  }
  app.selection[0].contents = myContents
} else {
  alert("No selection texts!");
  exit();
}
```

Скрипт использует такие предположения:

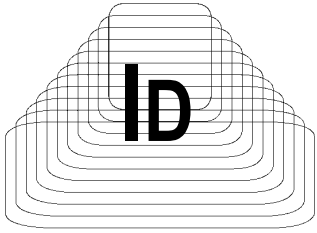
- переносы всегда могут быть поставлены между двумя подряд идущими гласными буквами ("со~отечественник");
- между двумя подряд идущими согласными, если вокруг них гласные ("со-отечествен~ник");
- между сочетаниями "любой знак плюс гласная" и снова "любой знак плюс гласная" ("сооте~чественник");
- между двумя гласными, разделенными тремя или четырьмя любыми другими буквами ("соотечест~венник").

Все они заданы соответствующими конструкциями в переменных `re1`, ..., `re7` (`Vowels` — гласные, `Consonant` — согласные).

Сначала оба алфавита разбиваются на группы из согласных, гласных и особую группу, в которую поместим "йъь". После этого по очереди проводим замену в соответствии с описанными выше правилами. Ключи `"ig"` позволяют выполнять множественную замену без учета регистра. Скрипт работает достаточно быстро, причем качество расстановки переносов не хуже, чем у InDesign CS 2. Более того, его можно улучшить, если создать словарь исключений, в котором указать правильную расстановку переносов. Как это реализовать — зависит от вас: можно в самом скрипте перечислять исключения, а можно в виде отдельного файла словаря, при этом пополнять его смогут все, даже не знакомые со скриптингом, поскольку он будет представлять собой простой редактируемый текстовый файл. В данном примере реализован первый вариант — исключения в самом скрипте.

Возьмем, к примеру, слово "постсоветский". По нашим правилам скрипт сделает такие переносы: "пос~тсо~вет~ский" — первый перенос поставлен неверно. Чтобы исправить ситуацию, определяем фрагмент "пос~т" как исключение. В результате его обработки получаем правильный вариант: "пост~со~вет~ский".

ГЛАВА 8



Таблицы

Несмотря на очевидные улучшения, произошедшие в InDesign CS3 по сравнению с CS2 в части работы с таблицами (программа стала поддерживать стили для ячеек), использование скриптинга не отошло на второй план — и хотя встроенные функции помогут решить определенную часть проблем, полностью все охватить они не в состоянии. Ведь если использовать аналогию с миром моды, InDesign — это массовый пошив: где-то что-то все равно не так, а вот хотелось бы еще и это, и перламутровые пуговицы тоже не помешали бы... Скриптинг решает все эти частные проблемы — в нем вы делаете то, что хотите, это — "индивидуальный пошив", причем оперативный: за полчаса можно решить большинство вопросов (конечно, время может меняться в широких пределах в зависимости от сложности решаемой задачи, но, как правило, создание типичных утилит полностью укладывается в эти сроки). Никогда разработчики ПО не смогут охватить все вопросы, стоящие перед конкретным пользователем, причем решить их оперативно, не дожидаясь очередной версии полтора-два года.

Максимальная автоматизация процесса верстки немыслима без переложения на плечи машин обработки таблиц, особенно объемных. И первый шаг, с которым вы столкнетесь, — их создание.

Работа с таблицами в InDesign очень похожа на работу с ними в офисных редакторах, поэтому пользователи, которые поняли логику работы, например, Word, получат дополнительное преимущество в InDesign.

Таблица в объектной модели InDesign — самостоятельный объект, который вставляется в уже существующий текстовый фрейм как объект in-line. При этом его родителем является `insertionPoint`, и при изменении текста она перемещается во фрейме автоматически. Создание таблицы ничем не отличается от создания какого-либо другого объекта публикации — используется все тот же метод `add()`:

```
myTextFrame.myInsertionPoint.tables.add()
```

Все параметры форматирования таблиц, известные по работе с офисными пакетами, присутствуют в InDesign — вплоть до поддержки шапок и запрета разрыва строк при переносе со страницы на страницу.

Таблица состоит из набора строк (`rows`) и столбцов (`columns`), которые на пересечении образуют ячейки (`cells`). Ячейки могут быть образованы слиянием нескольких ячеек: не важно — по горизонтали или вертикали, в таком случае говорят об объединенных ячейках.

Доступ к любой ячейке может осуществляться несколькими способами. Первый — в формате `cells.name[`*`row_index`*`, `*`column_index`*`]`, где на первом месте идет порядковый номер строки, к которой ячейка принадлежит, потом — столбец. Альтернативный вариант — `cells[`*`cell_index`*`]`, где *`cell_index`* — порядковый номер ячейки в данной таблице. Нумерация ячеек идет сначала по горизонтали (по строкам, сперва первая строка и т. д. до конечной), затем — по вертикали (после конечной ячейки в строке — переход на первую из следующей строки).

Еще один вариант обращения к ячейкам — относительный. Он позволяет задать в качестве базового как определенный столбец `columns[`*`column_index`*`].cells[`*`cell_index`*`]`, так и, при необходимости, строку `rows[`*`row_index`*`].cells[`*`cell_index`*`]`. Тот или иной вариант выбирают из соображений удобства: если работа ведется на уровне строки, то логично выбрать последний, иначе — первый. Каждая ячейка как отдельный объект InDesign имеет собственный набор параметров, наиболее часто востребованные из которых представлены на рис. 8.1.

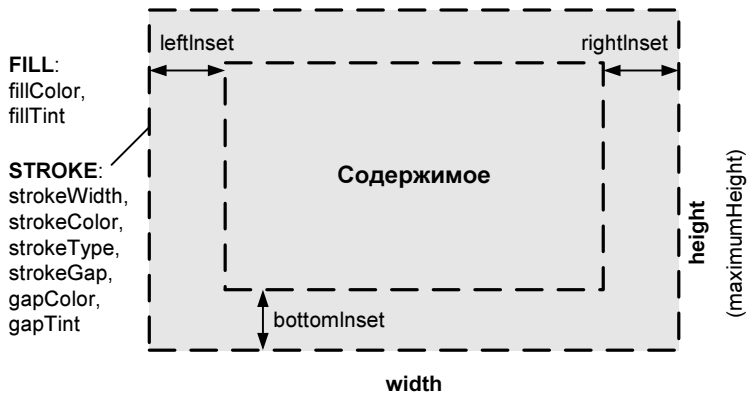


Рис. 8.1. Ячейка и ее параметры

Согласно объектной модели InDesign, таблица, любая строка или столбец, в сущности, представляют собой объект типа ячейки, в котором расположены

другие ячейки. Соответственно, таблица есть не что иное, как набор ячеек, к которым исключительно для удобства можно обращаться как к строкам или столбцам.

При этом строкой будут считаться все ячейки, имеющие один и тот же индекс `row_index`, а столбцом — `column_index`. Если провести аналогию с текстом, то строки и колонки — такие же условности, как и, например, абзацы в тексте.

При создании таблицы можно сразу же задать количество строк и столбцов, при этом высота шапки (количество строк повторяющегося на других страницах заголовка) не учитывается и задается отдельно (`headerRowCount`).

InDesign поддерживает вложенные таблицы, т. е. таблицы, содержимым ячеек которых могут быть другие таблицы. Таким образом, сложность подобных конструкций может быть практически неограниченной.

Рассмотрим основные методы работы с таблицами.

8.1. Создание таблицы

Начнем с базовых операций — определения типа области выделения, способов обращения к ячейкам, задания параметров как отдельных ячеек, так и целой таблицы.

Предположим, стоит задача создать таблицу из двух колонок, ширина которой соответствует полосе набора текущего текстового фрейма. Строки должны иметь высоту 50 мм, по оформлению: первая строка — шапка, толщина окантовки — 0.5 пт, расстояние между текстом и окантовкой — 1 мм. Скрипт представлен в листинге 8.1.

Листинг 8.1. Создание таблицы

```
myDocument = app.activeDocument;
mySelection = myDocument.selection[0]
curr_para = mySelection.paragraphs[0];
myParagraphStyles = myDocument.paragraphStyles;
pars = mySelection.paragraphs;
parStylesTempArray = [];
for (i=2; i<myParagraphStyles.length; i++){
  switch(myParagraphStyles[i].name){
    case 'Normal':
      parStylesTempArray['Normal'] = i;
      break;
  }
}
```

```
if(mySelection.constructor.name == "InsertionPoint" || ↳
   mySelection.constructor.name == "Paragraph"){
  curr_para.tables.add({columnCount:2, bodyRowCount:1, height: ↳
    "50 mm", width: mySelection.textFrames[0]. ↳
    geometricBounds[3]-mySelection.textFrames[0].geometricBounds[1]})

  myTable = curr_para.tables[0]
  for(i=0; i< myTable.cells.length; i++){
    with(myTable.cells[i]){
      topInset = "1 mm"
      bottomInset = "1 mm"
      rightInset = "1 mm"
      leftInset = "1 mm"

      leftEdgeStrokeWeight = "0.5 pt"
      topEdgeStrokeWeight = "0.5 pt"
      bottomEdgeStrokeWeight = "0.5 pt"
      rightEdgeStrokeWeight = "0.5 pt"
    }
  }
}
myTable.headerRowCount = 1
curr_para.applyStyle(myParagraphStyles[parStylesTempArray ↳
  ['Normal']], true);
```

Сначала определяем ссылки на используемые объекты. Поскольку таблица вставляется туда, где стоит курсор, используем `mySelection`:

```
myDocument = app.activeDocument;
myParagraphStyles = myDocument.paragraphStyles;
mySelection = myDocument.selection[0]
```

Соответственно для абзаца:

```
curr_para = mySelection.paragraphs[0];
pars = mySelection.paragraphs
```

Проверяем корректность типа выделенной области и, если все нормально, создаем таблицу. В нашем случае достаточно проверить, чтобы курсор находился в тексте либо абзац был выделен целиком:

```
if(mySelection.constructor.name == "InsertionPoint" || ↳
  mySelection.constructor.name == "Paragraph")
```

Сразу же задаем параметры:

```
curr_para.tables.add({columnCount:2, bodyRowCount:1, height:"50 mm", ↳
  width: mySelection.textFrames[0].geometricBounds[3]- ↳
  mySelection.textFrames[0].geometricBounds[1]})
```

На данный момент в месте вставки курсора нами создана таблица высотой 50 мм, с одной строкой и двумя колонками. Ширина ее определяется габаритами текстового фрейма, в котором таблица расположена. Напомню, координаты текстового объекта задаются массивом данных, в котором по очереди идут координаты верхней, левой, нижней и правой границ, т. е.

```
Top = textFrame.geometricBounds[0]
Left = textFrame.geometricBounds[1]
Bottom = textFrame.geometricBounds[2]
Right = textFrame.geometricBounds[3]
```

Задаем свойства ячеек:

```
for(i=0; i< myTable.cells.length; i++){
  with(myTable.cells[i]){
    topInset = "1 mm"
    bottomInset = "1 mm"
    rightInset = "1 mm"
    leftInset = "1 mm"
    leftEdgeStrokeWeight = "0.5 pt"
    topEdgeStrokeWeight = "0.5 pt"
    bottomEdgeStrokeWeight = "0.5 pt"
    rightEdgeStrokeWeight = "0.5 pt"
```

Затем задаем строку заголовка (шапку) — элемент таблицы, который автоматически повторяется в местах ее разрыва (например, при продолжении таблицы на следующей странице). Интересно, что если при обычной верстке строка заголовка создается путем конвертирования существующей строки (**Convert To Header Row**), то в скриптинге она задается непосредственно, и InDesign сверху таблицы добавляет пустую строку-шапку:

```
myTable.headerRowCount = 1
```

Рассмотрим некоторые операции над уже существующими таблицами.

8.1.1. Объединение ячеек

Операция объединения ячеек распространяется всегда только на две смежные ячейки, соответственно при необходимости объединения большего количества выполняют данную операцию необходимое количество раз, как это видно из листинга 8.2. Операции объединения ячеек по вертикали либо горизонтали с точки зрения скриптинга ничем не отличаются.

Листинг 8.2. Объединение ячеек

```
with (myTable) {
  // Объединение всех ячеек в первой колонке
  myTable.cells[0].merge(columns[0].cells.item(-1));
```

```
  // Объединение последних двух ячеек в первой строке
  rows[0].cells.item(-2).merge(rows[0].cells.item(-1));

  // Объединение последних трех ячеек во второй колонке
  // при условии, что в колонке всего три строки
  columns[1].cells.item(-1).merge(columns[1].cells.item(-2));
  columns[1].cells[0].merge(columns[1].cells[1]);

  // Объединение двух последних ячеек в строке 3
  rows[2].cells.item(-2).merge(rows[2].cells.item(-1));
}
```

8.1.2. Разбиение ячеек

Разбиение ячеек является операцией, противоположной их объединению, в результате которой общее количество ячеек в таблице увеличивается. Как и при объединении, разбиение ячеек — операция постепенная, за один шаг из одной ячейки можно получить только две (листинг 8.3).

Листинг 8.3. Разбиение ячеек

```
// Таблица myTable должна содержать как минимум 4 строки и 4 колонки
myTable.cells.item(0).split(HorizontalOrVertical.horizontal);
myTable.columns.item(0).split(HorizontalOrVertical.vertical);
myTable.cells.item(0).split(HorizontalOrVertical.vertical);
myTable.rows.item(-1).split(HorizontalOrVertical.horizontal);
myTable.cells.item(-1).split(HorizontalOrVertical.vertical);
```

8.1.3. Присвоение строкам чередующейся заливки

Чередующаяся заливка, как правило, используется при оформлении длинных таблиц — такая форма их оформления способствует лучшей читабельности информации. Чередовать разные заливки можно как от строки к строке, так и от столбца к столбцу. Строки, заданные как шапка, можно исключить из общего списка и в таком случае чередование распространяться на них не будет. В листинге 8.4 приведен пример задания чересстрочной чередующейся заливки.

Листинг 8.4. Чередующаяся заливка строк

```
myTable.alternatingFills = AlternatingFillsTypes.alternatingRows;
myTable.startRowFillColor = myDocument.swatches["MyColor1"];
myTable.startRowFillTint = 60;
```

```
myTable.endRowFillColor = myDocument.swatches.item("MyColor1");
myTable.endRowFillTint = 50;
```

8.1.4. Задание свойств таблицы

В листинге 8.5 приведен пример, иллюстрирующий задание большинства часто используемых элементов оформления таблицы.

Листинг 8.5. Задание свойств таблицы

```
// Добавляем шапку
myTable.rows.item(0).rowType = RowTypes.headerRow;

// Присваиваем цвет ячейкам
myTable.rows[0].fillColor = myDocument.swatches.item("MyColor1");
myTable.rows[0].fillTint = 40;
myTable.rows[1].fillColor = myDocument.swatches.item("MyColor1");
myTable.rows[1].fillTint = 40;
myTable.rows[2].fillColor = myDocument.swatches.item("MyColor1");
myTable.rows[2].fillTint = 20;
myTable.rows[3].fillColor = myDocument.swatches.item("MyColor1");
myTable.rows[3].fillTint = 40;

// Для форматирования диапазона ячеек удобно использовать
// метод everyItem()
with(myTable.cells.everyItem())
{
  topEdgeStrokeColor = myDocument.swatches.item("MyColor2");
  topEdgeStrokeWeight = 1;
  bottomEdgeStrokeColor = myDocument.swatches.item("MyColor2");
  bottomEdgeStrokeWeight = 1;

  // Если вы задаете ячейке толщину окантовки, задайте и цвет:
  // иначе будет цвет по умолчанию (черный)
  leftEdgeStrokeColor = myDocument.swatches.item("None");
  leftEdgeStrokeWeight = 0;
  rightEdgeStrokeColor = myDocument.swatches.item("None");
  rightEdgeStrokeWeight = 0;
}
```

8.1.5. Определение положения курсора в таблице

Для того чтобы иметь возможность работать не только с каким-то заранее известным элементом таблицы, а вообще с любым (например, выделенным фрагментом — как вы помните, положение курсора также считается выделе-

нием), первое, что нужно выполнить, — определить тип выделения, чтобы использовать только те методы и свойства, которые позволяет для него объектная модель. Собственно, этот процесс очень похож на тот, который был использован нами для текста — разумеется, со своей спецификой. Посмотрите, как InDesign воспринимает тот или иной фрагмент таблицы (листинг 8.6).

Листинг 8.6. Определение типа выделенного объекта

```
if(app.documents.length != 0){
  if(app.selection.length != 0){
    switch(app.selection[0].constructor.name){
      // Если выделена либо строка, либо колонка, либо диапазон ячеек —
      // их тип определяется как "Cell"
      case "Cell":
        alert("Курсор в ячейке");
        break;
      case "Table":
        alert("Выделена таблица");
        break;
      case "InsertionPoint":
      case "Character":
      case "Word":
      case "TextStyleRange":
      case "Line":
      case "Paragraph":
      case "TextColumn":
      case "Text":
        if(app.selection[0].parent.constructor.name == "Cell"){
          alert("Курсор в ячейке");
        }
        break;
    }
  }
}
```

В более сложных случаях, возникающих при дополнительной вложенности элементов друг в друга, можно воспользоваться вариантом из листинга 8.7. В некоторых случаях приходится использовать конструкцию типа `parent.parent` и даже более длинную.

Листинг 8.7. Использование конструкции `parent.parent`

```
case "Rectangle":
case "Oval":
```

```
case "Polygon":
case "GraphicLine":
  if(app.selection[0].parent.parent.constructor.name == "Cell"){
    alert("Выделенный объект находится в ячейке таблицы");
  }
  break;
```

Иногда приходится опускаться по иерархии еще глубже. Вот, например, как происходит доступ к иллюстрациям, размещенным в таблице (листинг 8.8).

Листинг 8.8. Доступ к иллюстрациям в таблице

```
case "Image":
case "PDF":
case "EPS":
  if(app.selection[0].parent.parent.parent.constructor.name == "Cell"){
    alert("Выделенное изображение находится в ячейке таблицы");
  }
  break;
default:
  alert("Курсор не в таблице");
  break;
}
```

8.2. "Резиновые" таблицы

Основные действия с таблицами мы рассмотрели ранее, теперь давайте перейдем к решению практических задач и напишем сценарий, который обязательно пригодится вам в повседневной работе.

Если в публикациях, с которыми вы работаете, достаточно много таблиц, то часто возникает ситуация, когда нижний край таблицы немного не доходит до конца колонки набора — начинать новый абзац смысла нет, а оставлять "висельника" тоже не хорошо. Оптимальный вариант — увеличить высоту таблицы за счет добавления интервалов между ячейками так, чтобы она четко вписывалась в оставшееся до низа колонки место (рис. 8.2).

Скрипт приведен в листинге 8.9.

Листинг 8.9. Увеличение высоты таблицы

```
myTable = mySelection.tables[0]
requredHeight = myTable.geometricBounds[0] - myTable.geometricBounds[2]
```

```
realHeight = myTable.height
rowsNumber = myTable.rows.length
delta = (requredHeight — realHeigth) / rowsNumber
for (i=0; i<myTable.rows.length; i++) {
  if (delta > thresholdValue) delta = thresholdValue
  myTable.rows[i].topInset += delta / 2;
  myTable.rows[i].bottomInset += delta / 2;
}
```

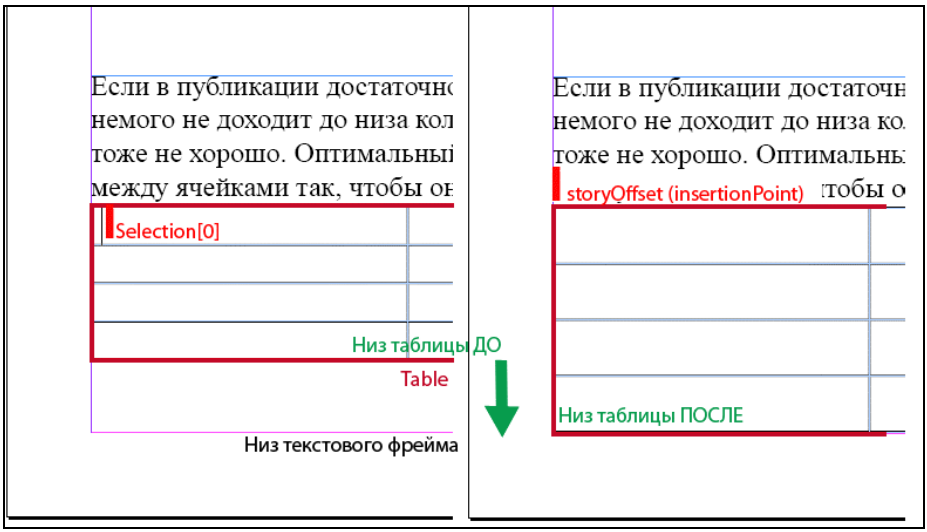


Рис. 8.2. Увеличение высоты таблицы

После традиционного задания ссылок определяем реальную высоту таблицы (`realHeight`) и требуемую (`requredHeight`), разницу делим на количество строк — это и будет то приращение, на которое нужно увеличить высоту каждой строки. Дальнейшие действия могут происходить по двум направлениям:

- изменять высоту ячейки, поскольку величина приращения нам уже известна;
- изменять отступ текста от верхней и нижней границ ячейки.

Оба дадут требуемый результат, только разными способами. На каком же остановиться? Предлагаю выбрать последний вариант как наиболее универсальный, поскольку он не приведет к изменению размещения текста в ячейке по вертикали (представьте себе ситуацию, если текст был распределен по высоте ячейки: при увеличении высоты ячейки межстрочное расстояние увели-

чится еще больше, а вот второй вариант приемлемое расположение сохранит). При этом величина приращения для каждого отступа должна быть в два раза меньше, чтобы в итоге мы набрали ту же самую высоту.

Осталось предусмотреть вариант, когда строк не так уже и много, а расстояние до края колонки набора достаточно большое. В таком случае зададим порог (`thresholdValue`):

```
if (delta > thresholdValue)
{
  delta = thresholdValue
}
```

8.3. Быстрый перенос таблиц из Word

Идущий в стандартной поставке InDesign фильтр, открывающий файлы с расширениями doc и rtf, позволяет напрямую работать с офисными документами — корректно обрабатывая в том числе таблицы, сноски и т. п.

Если вы верстаете публикацию на одном языке, вопросов не возникает. Однако довольно часто приходится, кроме одного варианта, создавать еще и версию на другом языке. Выходов, как поступить, всего два — либо полностью переверстывать публикацию, либо заменить только текст, а саму верстку не трогать. Как правило, выбирают второй вариант как наименее трудозатратный, при этом с обычным текстом проблем не возникает (он переносится через системный буфер, для упрощения процесса можно воспользоваться утилитой Punto Switcher. Ее основное предназначение — конвертировать ошибочно набранные в другой раскладке буквы, но есть и еще одно ценное — она позволяет хранить в памяти до 15 фрагментов скопированного текста. Иными словами, можно в Word скопировать до 15 фрагментов, переключиться в InDesign и вставлять их в нужные места до тех пор, пока лимит не исчерпается, потом снова вернуться в Word и т. д., очень удобно). Все это прекрасно подходит к работе с текстом, открытым остается вопрос с таблицами.

Поскольку переводчики переводят только текст (цифры заново не набирают), встает вопрос — каким образом они должны форматировать текст, чтобы содержимое разных ячеек осталось разграниченным? Как вариант — в качестве разграничителя по горизонтали использовать любой не встречающийся в тексте символ, например символ табуляции, а каждый ряд начинать с новой строки. Однако если таблицы большие, то лимита в 15 фрагментов явно мало и придется часто переключаться туда-сюда. Чтобы упростить процесс, были написаны два крошечных скрипта, которые позволяют вставить как всю таблицу сразу, так только выделенные ячейки.

Первый написан на Visual Basic for Applications (редактор другой язык просто не понимает) — листинг 8.10. Основное предназначение скрипта — обход ограничения механизма скриптинга InDesign, связанного с отсутствием доступа к системному буферу. Именно системному, поскольку доступ к буферу обмена самого InDesign реализован через методы `copy()` и `paste()`. Вопрос решается просто — выбранный текст копируется в пустой документ, который сохраняется в определенном месте, после чего он становится доступным скрипту через метод `File.open()`.

Конструкция `\1` в Visual Basic for Applications соответствует `$1` в JavaScript, остальное должно быть понятно. Все операции можно записать в автоматическом режиме (вы выбираете те или иные действия, а скрипт их переводит на VBA и записывает в виде макроса).

Листинг 8.10. Скрипт на VBA для Word

```
Sub txtToIDD_AI()
  // Отключение режима отслеживания изменений
  If ActiveDocument.TrackRevisions = True Then
    ActiveDocument.TrackRevisions = False
  End If

  // Делаем поправку на человеческий фактор — учитываем, что, кроме
  // символа табуляции, люди ошибаются и часто добавляют пробелы.
  // Поэтому ищем табулятор с пробелом и пробел удаляем.
  Selection.Find.ClearFormatting
    With Selection.Find
      .Text = "(^t) "
      .Replacement.Text = "\1"
      .Forward = True
      .Wrap = wdFindNoAsk
      .Format = False
      .MatchWildcards = True
      .MatchSoundsLike = False
      .MatchAllWordForms = False
    End With
  Selection.Find.Execute Replace:=wdReplaceAll

  // Следующий шаг — замена символов абзаца на табуляторы. В результате
  // мы получим единственную строку текста, с которой будем потом
  // работать. Почему нельзя было изначально, в Word, записать все
  // в виде одной строки? Причина проста — так удобнее переводчикам
  // отслеживать, что уже переведено, а что нет, особенно это касается
  // объемных таблиц.
```

```
  Selection.Find.ClearFormatting
    With Selection.Find
      .Text = "^13([A-Z0-9])"
      .Replacement.Text = "^t\1"
      .Forward = True
      .Wrap = wdFindNoAsk
      .Format = False
      .MatchWildcards = True
      .MatchSoundsLike = False
      .MatchAllWordForms = False
    End With
  Selection.Find.Execute Replace:=wdReplaceAll

  // Удаление лишних табуляторов. Должна выполняться четкая
  // регламентация: между ячейками в строках только один табулятор,
  // иначе скрипт-парсер некорректно их отработает.
  With Selection.Find
    .Text = "^t{1;5}([A-Za-z0-9])"
    .Replacement.Text = "^t\1"
    .Forward = True
    .Wrap = wdFindNoAsk
    .Format = False
    .MatchWildcards = True
    .MatchSoundsLike = False
    .MatchAllWordForms = False
  End With
  Selection.Find.Execute Replace:=wdReplaceAll

  // Сохранение результата в виде файла
  Selection.Copy
  Documents.Add DocumentType:=wdNewBlankDocument
  Selection.PasteAndFormat (wdPasteDefault)
  ChangeFileOpenDirectory "D:\"
  ActiveDocument.SaveAs FileName:="temp.txt", _
    FileFormat:=wdFormatText, LockComments:=False, Password:="", _
    AddToRecentFiles:=True, WritePassword:="", _
    ReadOnlyRecommended:=False, EmbedTrueTypeFonts:=False, _
    SaveNativePictureFormat:=False, SaveFormsData:=False, _
    SaveAsAOCELetter:= False, Encoding:=1251, InsertLineBreaks:=False,  _
    AllowSubstitutions:=False, LineEnding:=wdCRLF
  ActiveWindow.Close
End Sub
```

Следующий шаг выполняется на стороне InDesign: скрипт открывает только что сохраненный файл, разбивает его по заранее предопределенным разделителям и каждый получившийся фрагмент вставляет по очереди во все выделенные ячейки в таблице из публикации (листинг 8.11). При этом все происходит как нужно. Допустим, мы меняем текст в шапке таблицы. В таком случае первый фрагмент переносится в первую выделенную ячейку, второй — в следующую колонку, и т. д. Если был выделен, допустим, столбец, то опять-таки первый фрагмент занимает место в первой строке, второй — в следующей строке, и т. д. Наконец, если были выделены несколько строк, InDesign пробегает сначала по горизонтали (по строкам), в конце строки переходит на следующую и все повторяется.

Листинг 8.11. Скрипт для InDesign

```
mySelection = app.selection[0];

// Поскольку Word в конце файла ставит знак абзаца (такова его
// особенность), нам его придется удалять для корректности
find0 = /^ |(\n )|\n$/g;
replace0 = "";

// Параллельно удаляем лишние пробелы, чем так часто грешат переводчики
find1 = / +/g;
replace1 = " "

// Считываем файл, сохраненный Word
fName = File("D:/temp.txt");
fName.open("r");
myLine = fName.read()

// Получаем массив с содержимым ячеек
arr = myLine.split("\t");

// Если переводчик по какой-то причине решил использовать
// не табуляцию, а оформил текст в виде таблицы — ничего страшного,
// напишем еще пару строк.
// Если массив не получился, то пробуем разные варианты.
if (arr.length == 1)
  arr1 = myLine.split("\r");
if (arr1.length == 1)
  arr = myLine.split("\n");
```

```
// После предыдущих операций мы в любом случае получим массив
// из содержимого ячеек
for (i=0; i< mySelection.cells.length; i++) {
  // Проверяем, чтобы не было ячеек, объединенных по горизонтали —
  // иначе возникнут проблемы
  if (columnSpan==1){
    // Проводим замену
    r = arr[i].replace(find1, replace1);
    r = r.replace(find0, replace0)

    // Первая буква всегда должна быть заглавной
    res = r.substring(0,1).toUpperCase() + r.substring(1)
    mySelection.cells[i].contents = res;

    // Поскольку текст на разных языках может существенно отличаться
    // по количеству знаков (например, английский, как правило,
    // короче, а значит, высота, оставшаяся от русского варианта, будет
    // излишней). Обновляем высоту строк таблицы путем обнуления —
    // при этом InDesign самостоятельно рассчитывает необходимую
    // высоту ячеек
    with (mySelection.cells[i]) {
      if ((rowSpan==1) && (lines.length>0)) {
        height = 0
      }
    }
  }
  fName.close();
} else {
  alert("Среди выделенных ячеек есть объединенные по горизонтали —
с такими скрипт работает некорректно. Действие отменено.")
}
```

Таким образом, творчески подходя к решению задачи, можно обойти некоторые ограничения, накладываемые текущей реализацией механизма скриптинга в InDesign, как это было продемонстрировано в предыдущем примере.

8.4. Форматирование таблицы

Продолжим изучать более глубокие настройки форматирования: будем использовать назначение цветов, стили, уделим внимание вопросам оптимизации, без которой работа с объемными таблицами малоэффективна, после чего сможем гибко управлять таблицами любых размеров. Рассмотрим пример, который полностью заменяет труд верстальщика в данном вопросе.

Как известно, при импорте документа Word в InDesign параметры ячеек теряются, типичный результат показан на рис. 8.3, вверху. Наша задача — из этого полуфабриката создать полноценную таблицу, результат представлен на рис. 8.3, внизу.

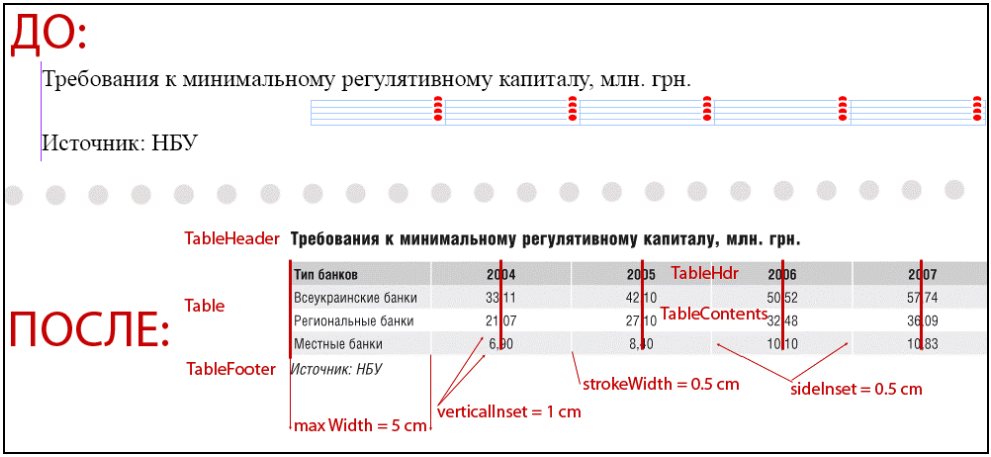


Рис. 8.3. Стили, используемые при оформлении таблицы

Предположим, что верстка двухколоночная. Предусмотрим, чтобы таблицы были двух видов: узкие — в одну колонку, и широкие — занимающие обе полосы набора. Далее для заливки строк будем использовать два чередующихся цвета, цвет окантовки у всех ячеек выберем белым. Внутренние отступы с боков зададим в 1 мм, по вертикали — 0,5 мм (см. рис. 8.2, внизу). Для текста в шапке таблицы будем использовать стиль абзаца `MyTable Hdr`, для остального текста — `Table contents`. Кроме того, обеспечим выключку во всех ячейках, кроме крайней правой колонки, по центру и отцентрировано по вертикали. Что касается самой левой колонки, то поставим задачу, чтобы ее ширина была такой, чтобы не было переноса текста (в случае, если таблица занимает две колонки текста), ну а в случае слишком длинного текста, ограничим ее ширину каким-то логичным значением. Что касается форматирования текста, то абзац перед таблицей будет нести стиль `Table Header`, абзац с таблицей — `Table`, а следующий абзац под таблицей — `Table Footer`. Таким образом, после выполнения скрипта мы получим полностью отформатированную по нашим правилам таблицу, что невозможно даже в самом современном InDesign, не говоря уже о тех пользователях, которые работают с InDesign CS2.

Скрипт приведен в листинге 8.12.

Листинг 8.12. Форматирование таблицы, импортированной из Word

```
Obj = {w:{narrow:120, wide:174}, flag:[false,false,false,false], ↵
  alterFill:"K 8%", headerFill:"K 15%"};
step = 2;
DEF_SIDE_INSET = DEF_STROKE_W = 0.5;
VERT_INSET = 1;

aD = app.activeDocument;
sel = aD.selection[0];
tbl = sel.parent.parent;
with(tbl.cells)
{
  cellsTbl = itemByRange(firstItem(), lastItem());
}
with(tbl.rows)
{
rowsTbl = itemByRange(firstItem(), lastItem());
}
tF = tbl.parent;
pS = aD.paragraphStyles;
pars = sel.paragraphs;
curr_para = pars[0];
startP = pars.previousItem(curr_para);
endP = pars.nextItem(curr_para);

value = prompt("Table width (Narrow/Wide) [1/2]: ", "1")
table_W = (value==1) ? Obj.w.narrow : Obj.w.wide;
pSA = [];
for (i=2; i<pS.length; i++)
{
  switch(pS[i].name)
  {
    case 'Table Header':
      pSA['Table Header'] = i;   Obj.flag[0] = true;
      break;
    case 'Table Footer':
      pSA['Table Footer'] = i;   Obj.flag[1] = true;
      break;
    case 'Table contents':
      pSA['Table contents'] = i; Obj.flag[2] = true
      break;
```

```
    case 'Table':
      pSA['Table'] = i;          Obj.flag[3] = true;
      break;
    case 'Tbl Hdr':
      pSA['Tbl Hdr'] = i;        Obj.flag[4] = true;
      break;
  }
}
with(Obj)
  if(flag[0] && flag[1] && flag[2] && flag[3] && flag[4])
  {
    if(sel.constructor.name == "InsertionPoint" || ↳
       sel.constructor.name == "Paragraph")
    {
      with(cellsTbl)
      {
        texts[0].applyStyle(pS [pSA ['Table contents']], true)
        verticalJustification = VerticalJustification.centerAlign
        if (rowSpan==1 && lines.length>0)
        {
          height = 0
        }

        topInset = bottomInset = VERT_INSET + " mm"
        leftInset = rightInset = DEF_SIDE_INSET + " mm"
      }

      with(tbl)
      {
        alternatingFills = AlternatingFillsTypes.alternatingRows;
        startRowFillCount = 1;
        startRowFillColor = Obj.alterFill;
        endRowFillColor = "None";
        startRowFillTint = 100;
        skipFirstAlternatingFillRows = 1
        width = (columns.length<5 || table_W==Obj.w.narrow) ? ↳
          Obj.w.narrow : Obj.w.wide
      }

      with(rowsTbl)
      {
        topEdgeStrokeWeight = 0;
        bottomEdgeStrokeWeight = 0;
```

```
  cells[0].texts[0].justification = 1818584692
}

for (i=0; i<tbl.rows.length; i++)
{
  with(tbl.rows[i].cells[0]) while(lines.length>1 &&  width<50)
  width += step
}

with(tbl) {
  for (i=0; i<columns.length; i++) {
    with (columns[i]) {
      leftEdgeStrokeColor = aD.swatches[1];
      rightEdgeStrokeColor = aD.swatches[1];
      leftEdgeStrokeWeight = DEF_STROKE_W + " pt";
      rightEdgeStrokeWeight = DEF_STROKE_W + " pt";
      if(i>0)
      {
        width = (table_W-parent.columns[0].width)/ ↳
                (parent.columns.length-1)
      }
    }
  }
}
myCellName = mySelection.parent.name
myCellRow = myCellName.split(":")[1]

myTable.rows[0].select
for(k=1; k< myCellRow; k++)
{
  myTable.rows[k].select(SelectionOptions.addTo)
}
with(myDocument.selection[0])
{
  rowType = RowTypes.headerRow
  fillColor = Obj.headerFill
  texts[0].applyStyle(pS[pSA['Tbl Hdr']],true)
}
aD.select(NothingEnum.nothing)

curr_para.applyStyle(pS[pSA['Table']], true);
pars.previousItem(curr_para).applyStyle(pS[pSA['Table ↳
  Header']], true)
```

```
    pars.nextItem(curr_para).applyStyle(pS[pSA['Table Footer']],true)
    if (table_W==Obj.w.wide)
      pars.itemByRange(startP, endP).leftIndent = 0
    alert("Готово!")
  }
  else
  {
    alert("Курсор не в таблице!")}
  }else{
    alert("Нет нужных стилей абзаца!")}
```

Перечисляем необходимые параметры: отступы от краев, ширину окантовки, задаем ширину колонок:

```
DEFAULT_SIDE_INSET = DEFAULT_STROKE_WIDTH = "0.5 mm" ;
VERTICAL_INSET = "1 mm";
narrow_column_width = "120 mm";
wide_column_width= "174 mm";
```

Используем нестандартный метод задания параметров — в явном виде — в миллиметрах, что зачастую более наглядно. Поскольку в таком случае они будут восприняты уже не в виде чисел, а как текстовые строки, оформим их соответствующим образом, заключив с обеих сторон в кавычки. Интересно, что InDesign понимает такую запись без каких-либо дополнительных действий с нашей стороны — правильно считывает единицы измерения и устанавливает требуемые значения.

Также нам придется использовать цвета для заливки ячеек таблицы. В отличие от названия стиля, который непосредственно в скрипте задать нельзя (как мы его видим в палитре **Paragraph Styles**, например, "Table"), с цветом ситуация значительно проще. InDesign понимает их названия такими, какими мы видим их в окне **Swatches**. Таким образом, для цвета содержимого таблицы можно записать:

```
alterFill = "alterFillColor";
```

и для шапки:

```
headerFill = "headerFillColor";
```

Разумеется, эти цвета перед запуском скрипта должны присутствовать в публикации. Устанавливаем ссылки на объекты, с которыми будем работать:

```
myDocument = app.activeDocument;
```

Стандартная строка для работы с любым выделенным объектом:

```
mySelection = myDocument.selection[0];
```

Для упрощения требований к начальным условиям договоримся, что таблицу будем обозначать установкой в нее курсора (необходимо, если на странице несколько таблиц). При этом для обращения ко всей таблице получим:

```
myTable = mySelection.parent.parent;
```

Прямой родитель у точки вставки — ячейка, а родитель у ячейки — таблица. У таблицы родительским объектом будет текстовый фрейм, в котором она находится.

```
myTextFrame = myTable.parent;
```

Для использования стилей публикации создаем ссылки на них:

```
parStyles = myDocument.paragraphStyles;
```

Следующий шаг — проверка публикации на наличие требуемых стилей — это позволит избежать возникновения ошибки в случае, если какого-то необходимого стиля не существует. Перебираем по очереди все стили и в случае нахождения нужного устанавливаем соответствующий признак. InDesign хранит стили не по названиям, как цвета, а по их индексам, поэтому присвоить стиль непосредственно по его названию не получится. Соответственно, запоминать будем индексы стилей, по которым в дальнейшем и станем обращаться.

Создадим временное хранилище для них:

```
paragraphStylesStorage  = new Array();// Хранилище для индексов стилей
flag = new Array();   // Здесь будем хранить признаки наличия
                      // требуемых стилей.
```

Запоминаем индексы используемых стилей (листинг 8.13).

Листинг 8.13. Сохранение индексов используемых стилей

```
for (i=0; i< parStyles.length; i++){
  switch(parStyles[i].name)
  {
    case 'Table Header':
      paragraphStylesStorage ['Table Header'] = i;  flag [0] = true;
      break;
    case 'Table Footer':
      paragraphStylesStorage['Table Footer'] = i;  flag [1] = true;
      break;
    case 'Table contents':
      paragraphStylesStorage['Table contents'] = i;  flag [2] = true;
      break;
```

```
    case 'Table':
      paragraphStylesStorage['Table'] = i;    flag [3] = true;
      break;
    case 'Tbl Hdr':
      paragraphStylesStorage['Tbl Hdr'] = i;    flag [4] = true;
      break;
  }
}
```

Операция `break` позволяет оптимизировать производительность скрипта: ведь если какой-то из искомых стилей найден, то продолжать его дальнейший поиск смысла нет, можно переходить к поиску следующего стиля. Проверяем, все ли используемые стили присутствуют в публикации:

```
if(flag[0] && flag[1] && flag[2] && flag[3] && flag[4])
```

По умолчанию булевы выражения всегда сравниваются с `true`, поэтому отпадает необходимость каждый раз использовать конструкцию типа `if(flag[...]==true)`.

Если все стили на месте, переходим к проверке типа выделенной области, что нужно для корректной работы скрипта. Ведь нам необходимо гарантировать, чтобы он был либо точкой вставки, либо абзацем, иначе (например, если выделена ячейка) ранее использовавшееся выражение

```
myTable = mySelection.parent.parent;
```

потеряет смысл, т. к. оно жестко привязано к конкретному типу выделенной области.

Удостоверимся, что выделение допустимое:

```
if (mySelection.constructor.name == "InsertionPoint" || ↳
  mySelection.constructor.name == "Paragraph")
```

Приступаем непосредственно к форматированию таблицы: сначала присваиваем тексту в ячейках стиль `Table contents`:

```
for (i=0; i< myTable.cells.length; i++)...
```

Однако такой метод достаточно медленный. Эксперименты, проведенные с таблицами, которые занимают одну или несколько страниц, показали, что нужно искать другое, более эффективное решение. Одна из причин такой медлительности — неоптимизированный подход при установке параметров форматирования ячеек. В самом деле, мы по очереди перебираем все ячейки и к каждой по отдельности применяем несколько параметров форматирования. В таблицах с несколькими сотнями ячеек работа скрипта может занять ощутимое время даже на относительно мощной машине. Попробуем исправить ситуацию.

При переборе ячеек мы пользуемся исключительно JavaScript — задаем цикл, начиная с самой первой ячейки и заканчивая последней. Это универсальный подход, а такой редко оптимален. Попробуем отыскать среди возможностей самого InDesign такие методы, которые бы смогли ускорить данный процесс. Диапазон методов, применимых к ячейке (`Cell`), разнообразием не блещет: типичный набор. А что, если посмотреть на методы, существующие для набора ячеек (`Cells`)? Тут нас ждет находка: ключом к решению проблемы будет метод `itemByRange()`. Он имеет два параметра, задающих диапазон ячеек, к которым InDesign обращается через свои внутренние механизмы (первый задает начальный объект, второй — конечный). Анализ производительности обоими методами (перебор средствами JavaScript и InDesign) дал убедительный результат: в среднем скорость выполнения форматирования поднялась на порядок, что является прекрасным результатом и отлично подходит для работы с таблицами любой степени сложности.

Одна из причин повышения эффективности — InDesign в таком случае передает параметры в виде массива, а не каждый по отдельности, отсюда и скачок в производительности.

Изменим часть скрипта, относящуюся к форматированию ячеек. Введем две переменные — для доступа ко всем ячейкам и ко всем строкам:

```
with(myTable.cells)
{
  cellsTbl = itemByRange(firstItem(), lastItem());
}
with(myTable.rows)
{
  rowsTbl = itemByRange(firstItem(), lastItem());
}
```

Почему мы задаем отдельно две переменные? Ведь, по логике, через строку можно перейти к содержащимся в ней ячейкам? Однако на самом деле это не так. Как уже упоминалось, InDesign при использовании метода `itemByRange()` заданный диапазон трактует как массив, поэтому проще использовать именно раздельное обращение.

Таким образом, для форматирования текста в таблице можем записать:

```
with(cellsTbl)
{
  texts[0].applyStyle(pS [pSA ['Table contents']], true);
```

Конструкция `texts[0]` используется для обращения к текстовому содержимому ячейки, поскольку применить метод `applyStyle()` к ячейке невозможно. Первый параметр — индекс стиля (это индекс мы определили раньше), вто-

рой параметр (true) говорит о том, что текущее форматирование текста будет очищено. Горизонтальное выравнивание в ячейке — по центру.

```
  verticalJustification = 1667591796;
```

Следующий шаг — установка необходимой высоты для каждой ячейки соответственно ее содержимому. В InDesign для строк и ячеек существует свойство autoGrow, что избавляет нас от решения этой задачи собственными методами:

```
  autoGrow = true;
```

Наконец, задаем параметры оформления:

```
  topInset = VERT_INSET;
  bottomInset = VERT_INSET;
  leftInset = DEF_SIDE_INSET;
  rightInset = DEF_SIDE_INSET;
}
```

На данном этапе все основные операции на уровне отдельных ячеек выполнены. Осталось заняться объектами более высокого уровня. Выполняем ряд операций над таблицей:

```
with(myTable)
{
  alternatingFills = AlternatingFillsTypes.alternatingRows; // чередовать
                                                    // заливку по строкам
  startRowFillCount = 1;
  startRowFillColor = alterFill;    // первый используемый цвет
  endRowFillColor = "None";         // второй используемый цвет
  startRowFillTint = 100;           // плотность цвета
  skipFirstAlternatingFillRows = 1; // начальная строка
```

Следующий шаг — определение необходимой ширины таблицы. Исходя из практических требований, было выбрано, что если количество колонок больше 5, таблица однозначно должна быть широкой. Если колонок меньше и пользователь не указал явно в диалоговом окне, что таблица должна быть широкой, делаем ее узкой:

```
width = (columns.length < 5 || table_W == narrow_column_width) ? ↳
  narrow_column_width : wide;
```

Настало время задать выключку влево для крайнего левого столбца. Как уже упоминалось, метод itemByRange() воздействует только на те элементы, к которым он применяется, поэтому раньше (когда работали на уровне всех ячеек таблицы) мы не могли получить доступ к столбцам. Поэтому используем переменную rowsTbl, дающую доступ к строкам, и в каждом ряду воздействуем

только на самую первую ячейку (порядок ячеек в InDesign идет слева направо, поэтому для рядов `cells[0]` означает крайнюю левую ячейку):

```
with(rowsTbl)
{
  cells[0].texts[0].justification = 1818584692;  // выключка влево
  topEdgeStrokeWeight = bottomEdgeStrokeWeight = 0;
}
```

Причина перебора всех строк вместо всего одной колонки связана с необходимостью задать верхний и нижний отступы для строк, иначе, конечно же, проще было бы взять только одну колонку.

Наш следующий шаг — установка ширины первой колонки. Автору очень часто приходится иметь дело с ситуацией, когда в таблице фигурируют одни цифры, и лишь первая колонка предназначена для текста — для работы именно с такими таблицами скрипт в первую очередь и создавался. Чтобы повысить их читабельность, было решено постепенно увеличивать ширину первой колонки — до тех пор, пока текст не займет одну строку. В случае, если это невозможно (текста слишком много), ограничиваем максимальную ширину колонки значением 5 см (подобрано опытным путем). Оптимальная величина шага — 1 мм, что вполне достаточно для большинства случаев.

Использовать конструкцию `tbl.columns[0].cells.itemByRange()` не получится, поскольку в данном случае нам нужна работа не с массивом данных, а непосредственно с каждой ячейкой — ведь количество символов в каждой из них может быть совершенно разным.

```
step = 0.1  // Единица измерения (сантиметры) установлена заранее
for (i = 0; i <  myTable.rows.length; i++)
{
  with(myTable.rows[i].cells[0])
  {
    while(lines.length > 1 && width < 5)
    {
      width += step;
    }
  }
}
```

Наконец, последний аккорд: оформляем разделительную сетку. Она должна быть только вертикальной, поэтому:

```
with(myTable) {
for (i = 0; i < columns.length; i++)
{
  with (columns[i])
```

```
  {
    leftEdgeStrokeColor = aD.swatches[1];
    rightEdgeStrokeColor = aD.swatches[1];
    leftEdgeStrokeWeight = DEF_STROKE_W + " pt";
```

Явно указываем единицу измерения, иначе будут использоваться миллиметры:

```
    rightEdgeStrokeWeight = DEF_STROKE_W + " pt";
```

Изменяем ширину всех колонок с учетом новой ширины у первой:

```
if(i>0)
{
  width=(table_W-parent.columns[0].width)/(parent.columns.length-1)
}
}}
```

Тут также мы отказались от использования метода `itemByRange(firstItem(), lastItem())`. Дело в том, что нам нужно сделать исключение для крайнего левого столбца (который с текстом) — хотя, конечно же, можно было вместо `firstItem()` использовать `nextItem(firstItem())`. Присваиваем стили абзацам, окружающим таблицу:

```
myParagraphs = mySelection.paragraphs;
curr_para = myParagraphs[0];
curr_para.applyStyle(pS [pSA ['Table']], true);
if (table_W == Obj.w.wide)
{
  if (table_W==Obj.w.wide)
  pars.itemByRange(startP, endP).leftIndent = 0
}
with(myParagraphs)
{
  itemByRange(previousItem(curr_para), ↳
    nextItem(curr_para)).leftIndent = 0;
}
```

Остался завершающий штрих: оформить шапку. Если использовать метод `convertToHeader()` напрямую, и если в шапке есть слитые ячейки, то мы поймем, что не все вопросы одинаково скрупулезно проработаны разработчиками: InDesign аварийно завершает свою работу без каких-либо предупреждений. Альтернативный вариант, найденный путем экспериментов, состоит в повторении всех тех действий, которые выполняются ручным способом: сначала выделение необходимого количества строк с дальнейшей конвертацией их в шапку.

Встает вопрос: как определить, сколько в шапке строк? Автоматически определить это невозможно (шапка может быть сколь угодно сложной), поэтому, чтобы от чего-то отталкиваться, будем считать, что положение курсора в момент запуска скрипта как раз и определяет последнюю строку шапки. Это достаточно удобно: вы же видите, какая структура таблицы, ставите в последнюю строку шапки курсор и запускаете скрипт.

У ячеек есть полезное свойство `name`, в котором записана информация о том, в какой колонке и строке данная ячейка находится:

```
name(columnIndex : rowIndex)
```

Воспользуемся этим для определения текущей строки:

```
myCellName = mySelection.parent.name

myCellRow = myCellName.split(":")[1] // Строка идет второй по списку
```

Выделяем все строки, включая ту, в которой стоит курсор:

```
myTable.rows[0].select
for(k=1; k< myCellRow; k++)
{
  myTable.rows[k].select(SelectionOptions.addTo)
```

Теперь у нас выделены все строки, принадлежащие шапке, при этом объект `selection` у нас все равно один:

```
with(myDocument.selection[0])
{
  rowType = RowTypes.headerRow
  fillColor = Obj.headerFill
```

Присваиваем тексту в шапке свой стиль:

```
  texts[0].applyStyle(myParagraphStyles[parStylesTempArray ↳
    ['MyTable Hdr']], true)
}
```

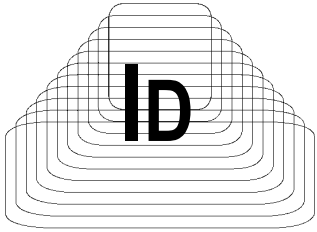
Выделение снимаем:

```
myDocument.select(NothingEnum.nothing)
```

В случае, если возникли проблемы несоответствия выделения ожидаемому типу, выдаем предупреждения и останавливаем скрипт:

```
else{
  alert("Курсор установлен не в таблице!")
}
} else{
  alert("Отсутствуют необходимые стили абзацев!")
}
```

ГЛАВА 9



Работа с изображениями

Обработка и размещение изображений — неотъемлемая часть работы над публикацией, в той или иной мере она присутствует в любом макете. Однако с учетом того, что именно грамотно подобранные и расположенные изображения придают изданию художественную ценность, то в данном вопросе сфера применения автоматизации крайне невелика. Совсем иная ситуация с технической, финансовой документацией — вообще любой, где должны присутствовать строгость форм и четкий порядок расположения иллюстраций, в том числе с привязкой к тексту.

В таких случаях, как показывает практика, автоматизация верстки дает превосходный результат. Из своего опыта скажу, что удается подготавливать публикацию нажатием всего одной клавиши — конечно же, предварительно изображения должны быть подготовлены в растровом или векторном формате для размещения в публикации. Приятнейшее ощущение — пьешь кофе и смотришь, как в InDesign мелькают диалоговые окна, страницы — процесс идет. И к тому моменту, когда кофе уже выпит, окончена и верстка. Итак, если вы работаете с подобными изданиями — считайте, что вам повезло.

Изображения в публикации, точно так же, как и текст, вставляются обязательно в контейнер (родитель). Для текста он имеет текстовый тип, а вот для изображения конкретный тип не предусмотрен — чаще всего в качестве контейнера используется прямоугольник (`rectangle`), хотя объект может быть любой воспроизводимой формы, в том числе полученной посредством логических операций между несколькими объектами (объединение, пересечение и т. п.). Процедура вставки изображения через скриптинг аналогична традиционному способу:

1. Создание объекта-контейнера.
2. Расположение его в нужном месте.

3. Вставка в него изображения.
4. При необходимости — дополнительные операции: подгонка размеров контейнера под размеры изображения, центровка изображения в контейнере.

Создание прямоугольника происходит аналогично добавлению любого другого объекта, только в данном случае нужно использовать коллекцию `rectangles`:

```
newRectangle = app.activeDocument.rectangles.add()
```

Разумеется, можно сразу же задать его габаритные размеры — или же отложить это до помещения в него изображения. При создании таким образом прямоугольник помещается на первую страницу публикации в левый угол с размерами по умолчанию.

Расположение в конкретном месте не должно вызвать никаких проблем — в этом поможет метод `move()`, а как определить необходимое место — уже рассматривалось ранее.

Вставка изображения выполняется методом `place()`, которому в качестве параметра указывается ссылка на файл.

Среди основных параметров изображения — его габаритные размеры (доступ — через родителя `rectangle`), цветовая модель, тип, а также текущее состояние связи с оригиналом (`status`). Доступ ко всем связям изображений в публикации происходит через свойство `activeDocument.links`, соответственно, все параметры связи конкретного изображения хранятся в объекте `Link`.

Импортированное изображение может как сохранять связь с оригиналом на диске, так и быть автономным (свойство `linkEmbedded=true`). В случае, если по каким-то причинам оригинал был перемещен в другое место и InDesign не может его найти, автоматически устанавливается флаг `linkMissing=true`.

Рассмотрим пример, в котором изображение импортируется в публикацию с размещением в абзаце, где находится курсор (листинг 9.1). Если размеры изображения меньше некоторых минимальных значений по ширине и высоте, центрируем его по размеру контейнера (`FitOptions.centerContent`), если же больше — расширяем контейнер таким образом, чтобы все его содержимое стало видимым (`FitOptions.frameToContent`).

Листинг 9.1. Импорт изображения

```
mySelection = app.selection[0];
if(mySelection != 'undefined')
{
  myParagraph = mySelection.paragraphs[0];
```

```
  myParagraph.rectangles.add({fillColor: ↳
    app.activeDocument.swatches[0], strokeWeigth:0})
  myParagraph.recompose();

  with (myParagraph.rectangles[0])
  {
    place(File(fname);
    with (allGraphics[0]){
    myWidth = geometricBounds[3] - geometricBounds[1];
    myHeight = geometricBounds[2] - geometricBounds[0];
  }
  if( (myWidth < min_width) && (myHeight < min_height) )
  {
    geometricBounds = [geometricBounds[0], ↳
      geometricBounds[1], geometricBounds[0] + myHeight, ↳
      geometricBounds[1] + minWidth];
    fit (FitOptions.centerContent);

  } else {
    fit(FitOptions.frameToContent);
  }
}
```

В данном случае для доступа к изображению использовалась конструкция `allGraphics`, поскольку мы заранее не знаем, какого типа изображение было импортировано (в InDesign существуют отдельные коллекции для каждого типа векторных изображений, для растровых — `images`, а наиболее универсальный вариант — `graphics` или `allGraphics`). Кроме того, при создании контейнера мы изменили его свойства: присвоили заливку белым цветом, а окантовку убрали, чтобы изменить значения, которые устанавливает InDesign по умолчанию.

`fname` — ссылка на импортируемый файл, которая может быть получена через диалоговое окно.

9.1. Управление связями

Рассмотрим, как определяется связь изображений с оригиналами на диске.

Как правило, в публикациях некоторое количество рекламы повторяется из номера в номер, а после сдачи текущего номера вся использованная реклама архивируется в отдельную папку. Соответственно для сбора всей использованной рекламы потребуется, во-первых, просмотр папки Links, полученной

после выполнения операции **Collect For Output**, и ручной отбор только рекламы, и, во-вторых, дополнительный поиск рекламных макетов из других номеров. Попробуем переложить задачу на "плечи" машины, создав аналог операции **Collect For Output** и расширив ее нашими потребностями.

Для того чтобы скрипт мог отбирать только рекламу, название рекламных файлов должно как-то отличаться от других изображений, либо просто находиться в отдельной папке, например, Реклама. В таком случае скрипт будет просматривать все связи и, если найдет в пути к файлу слово "Реклама" или же какой-то другой флаг (ключевое слово, позволяющее однозначно идентифицировать файл рекламного содержания), будет копировать данный файл в новое место (листинг 9.2).

Листинг 9.2. Отбор рекламы для публикации

```
aD = app.activeDocument
myFolder = Folder.selectDialog("Select picture folder", aD.filePath)
for (i = 0; i < aD.links.length; i++) {
  l = aD.links[i];

  if (l.status == LinkStatus.linkMissing)
    continue;
  if ((l.status == LinkStatus.linkEmbedded))
    l.unembed();

  myFolder = Folder(aD.links[i].filePath)
  if(myFolder.toString().search(/\/Реклама\// )!=-1) || ↳
    (myFolder.toString().search(/myFlag/)!=-1)
  {
    myFile = new File (aD.links[i].filePath);
    myNewFile = new File (myFolder + "/" + myFile.name);
    myFile.rename(myNewFile)
  }
}
```

Скрипт очень компактен и прост. В самом начале пользователю предлагается ввести название папки, в которую будет выполняться копирование найденных изображений, после чего просматриваются все изображения в публикации (вернее, не сами изображения, а только существующие связи с изображениями) и проверяются их свойства.

Если связь с оригиналом по каким-то причинам потеряна (`LinkStatus.linkMissing`), то пропускаем соответствующее этой связи изображение и продолжаем просмотр; если изображение было внедрено в пуб-

ликацию (`LinkStatus.linkEmbedded`), его из публикации исключаем, устанавливая связь с оригиналом (`unembed()`).

Затем проверяем наличие в названии полного пути (`myFolder`) либо ключевого слова "/Реклама/" (с наклонными слэшами — признак папки), либо вымышленного флага `myFlag`. В случае успеха получаем название файла (`links[i].filePath`), после чего создаем новый путь к будущему месторасположению изображения и производим копирование. Обратите внимание, как производится операция копирования: фактически она сводится к переименованию файла.

В случае, если связи потребуется обновить, достаточно заменить последнюю строку на следующий фрагмент:

```
if (myFile.rename(myNewFile))
  l.relink(myNewFile)
```

9.2. Поиск изображений с разрешением менее заданного

При верстке публикаций со значительным количеством иллюстраций достаточно часто встречается ситуация, когда в результате всех трансформаций некоторые изображения оказываются настолько увеличены в размере по сравнению с оригиналами, что при печати, если не предпринять корректирующих действий, они окажутся в плохом качестве. Особенно остро такие вопросы стоят в случае, когда некоторые изображения доносят с большим опозданием по срокам, когда уже не остается времени на проверку их разрешения. Решить проблему иллюстраций с низким конечным разрешением (часто его называют еще эффективным, что точно отображает его специфику) можно несколькими способами.

Способ первый. Использовать возможности, заложенные в средства предпечатной проверки (preflight). Классический вариант — задействование Adobe Acrobat, альтернативные варианты — Flight Check и прочие решают вопрос аналогичным способом. В процессе проверки ПО отображает объекты, которые имеют параметры, отличающиеся от заданных пользователем, в виде списка. Вроде бы удобно, да не совсем. Давайте подсчитаем временные затраты на исправление ситуации на примере наиболее распространенного Acrobat.

Во-первых, сначала необходимо сформировать PDF: в зависимости от количества и качества изображений в публикации процесс может занять четверть часа и более. Во-вторых, сам preflight-анализ тоже не происходит мгновенно. Наконец, третий шаг — получение информации о проблемных изображениях для принятия корректирующих мер. Приходится возвращаться в публикацию, переходить на обозначенную страницу, на ней переходить на проблемное

изображение, и лишь только после этого начинаются корректирующие действия. После снова надо возвращаться в preflight-пакет, смотреть, на какой странице находится следующее проблемное изображение, и т. д. Подобную последовательность нужно пройти для каждого потенциально опасного изображения, что эффективным рабочим процессом язык назвать никак не поворачивается.

Второй способ состоит в использовании возможностей скриптинга. Получение результирующего разрешения лежит на поверхности, даже не нужно определять масштаб изображения: в InDesign все необходимые значения доступны пользователю в явном виде. Встает вопрос другого плана: как достичь эффективного взаимодействия с пользователем, чтобы ему было максимально удобно переходить к найденным изображениям (после этого следующее действие в InDesign — запуск операции **Edit in...**). В результате нескольких проб выкристаллизовался такой вариант: результаты поиска отображаются в виде списка с названиями проблемных файлов, в котором для оценки степени проблематичности также указаны их эффективные разрешения. Поскольку на одной странице может быть до десятка изображений, при выборе конкретного изображения из списка предусмотрен автоматический переход на выбранное изображение, с его выделением (реализовать следующую за этим операцию **Edit in Photoshop** можно через механизм, заложенный в Bridge).

В таком случае поиск изображений с разрешением ниже заданного сводится к запуску скрипта (листинг 9.3), результатом чего будет переход только к одному выбранному изображению. Последующий запуск приведет к переходу на следующее изображение и т. д.

Кроме экономии времени, затрачиваемого на поиск проблемных файлов данным скриптом, нельзя пропустить побочный эффект от использования вообще любого самостоятельно написанного скрипта — общее повышение настроения: это как раз один из тех случаев, когда, говоря словами известной песни, не мы, а "мир прогнулся под нас". Заметьте: вы не просто работаете в программе, а управляете ею — а это две ощутимые разницы.

Листинг 9.3. Поиск изображений с разрешением ниже заданного

```
document = app.activeDocument;
imageList = new Array ();
imagePageList = new Array ();
imageInfoList = new Array ();
numImagesFound = 0;

// Задание диалога
dlg = app.dialogs.add ↳
  ({name:"Поиск изображений с разрешением ниже заданного"});
```

```
column = dlg.dialogColumns.add ();

// Первая колонка
row = column.dialogRows.add ();
row.staticTexts.add ↳
  ({staticLabel:"Найти все изображения с разрешением ниже"});

// Вторая колонка
row = column.dialogRows.add ();
editbox = row.integerEditboxes.add ↳
  ({editContents:"300", smallNudge:1, largeNudge:10, ↳
    minimumValue:1, maximumValue:10000});

dialogCanceled = (dlg.show() == false);
dpiValue = editbox.editContents;
dlg.destroy ();

if (dialogCanceled)  {
  exit ();
}

for (i=0; i < document.pages.length; i++)
{
  currPage = document.pages[i];
  for (j=0; j < currPage.allPageItems.length; j++)
  {
    currPageItem = currPage.allPageItems[j];
    if (typeof currPageItem.images != 'undefined')
    {
      for (k=0; k < currPageItem.images.length; k++)
      {
        currImage = currPageItem.images[k];
        if (currImage.actualPpi[0] < dpiValue || ↳
            currImage.actualPpi[1] < dpiValue || ↳
            currImage.effectivePpi[0] < dpiValue || ↳
            currImage.effectivePpi[1] < dpiValue)
        {
          imageList[numImagesFound] = currImage;
          imagePageList[numImagesFound] = currPage;

          imageInfoList[numImagesFound] = "Page " + ↳
            currPage.name +":  actual DPI=[" + currImage.actualPpi[0] + ↳
            "," + currImage.actualPpi[1] + "]" +" effective DPI=[" + ↳
```

```
            currImage.effectivePpi[0] + "," + ↳
            currImage.effectivePpi[1] + "]" +" filename='" + ↳
            currImage.itemLink.filePath + "'";
          numImagesFound ++;
        }
      }
    }
  }
}

if (numImagesFound == 0)
{
  alert ("Изображений с разрешением ниже " + dpiValue + ↳
         " DPI не найдено.", "Поиск изображений по их разрешению");
  exit ();
}

// Действия после просмотра всей публикации: если проблемные
// изображения найдены, вывод их в виде списка
dlg = app.dialogs.add ({name:" Поиск изображений по их разрешению ", ↳
  canCancel:true});

column = dlg.dialogColumns.add ();
row = column.dialogRows.add ();
row.staticTexts.add ({staticLabel:"Выберите изображение и нажмите OK ↳
  для перехода на него"});
row = column.dialogRows.add ();
dropdown = row.dropdowns.add ({minWidth:400, ↳
  stringList:imageInfoList, selectedIndex:0});

if (dlg.show () == true)
{
  document.layoutWindows[0].activePage = ↳
    imagePageList[dropdown.selectedIndex];

  document.select (imageList[dropdown.selectedIndex]);
}
dlg.destroy ();
```

Рассмотрим шаги, которые скрипт выполняет. После вывода диалогового окна, в котором пользователь задает требуемые параметры, выполняем просмотр всех страниц с одновременным считыванием всех объектов на них:

```
for (i=0; i < document.pages.length; i++){
  currPage = document.pages[i];
  for (j=0; j < currPage.allPageItems.length; j++)
  {
    currPageItem = currPage.allPageItems[j];
```

Далее идет проверка того, что на странице есть изображения — она вызвана тем, что если на странице нет ни одного изображения, коллекция `pageItem.images` становится недоступной (у нее все свойства становятся в данном случае `undefined` — недочет программистов) и, соответственно, скрипт останавливается.

```
    if (typeof currPageItem.images != 'undefined')
    {
      for (k=0; k < currPageItem.images.length; k++)
      {
        currImage = currPageItem.images[k];
```

Затем для каждого изображения проверяем оба разрешения: заданное в изображении (`actualPpi`) и фактическое (`effectivePpi`), которое получается в результате применения всех трансформаций над изображением (масштабирования), и заносим текущие значения в список:

```
        if (currImage.actualPpi[0] < dpiValue || ↳
            currImage.actualPpi[1] < dpiValue || ↳
            currImage.effectivePpi[0] < dpiValue || ↳
            currImage.effectivePpi[1] < dpiValue)
        {
          imageList[numImagesFound] = currImage;
          imagePageList[numImagesFound] = currPage;
```

Кроме того, записываем в него дополнительную информацию: о номере страницы, пути к файлу изображения — они пригодятся при выводе информации в отчете

```
          imageInfoList[numImagesFound] = "Page " + ↳
            currPage.name +":  actual DPI=[" + currImage.actualPpi[0] + ↳
            "," + currImage.actualPpi[1] + "]" +" effective DPI=[" + ↳
            currImage.effectivePpi[0] + "," + ↳
            currImage.effectivePpi[1] + "]" +" filename='" + ↳
            currImage.itemLink.filePath + "'";
          numImagesFound ++;
```

Если искомые изображения найдены, отображаем окно с результатами поиска. В нем отображаются в списке все найденные изображения. При щелчке на

любом из них (dropdown.selectedIndex) InDesign переходит на соответствующую страницу, и, более того, изображение становится выделенным — это пригодится в случае, если публикация богата на изображения и на странице находятся несколько изображений:

```
document.layoutWindows[0].activePage = ↳
  imagePageList[dropdown.selectedIndex];
document.select (imageList[dropdown.selectedIndex]);
```

9.3. Импорт графики

Вопросы экспорта уже достаточно широко были освещены в предыдущих примерах, теперь рассмотрим вопросы импорта. Процесс импорта принципиально ничем от экспорта не отличается, в чем можно убедиться на следующем примере, в котором автоматизирован импорт многостраничного PDF-файла (листинг 9.4).

При этом предусмотрены следующие опции импорта документа:

- размещение на всю страницу, включая поля;
- на всю страницу (без полей);
- в центр страницы;
- в левый верхний угол страницы.

Листинг 9.4. Импорт многостраничного PDF-файла

```
with (app)
{
  if (documents.length < 1) {
    alert("Нет открытых документов");
    exit();
  }
  var myDoc = activeDocument;

  var myPDFFile = File.openDialog('Выберите файл', 'Файлы PDF: *.pdf');
  if(myPDFFile == null) {
    exit();
  }
  var myDlg = dialogs.add({name: "Импорт многостраничного PDF"})
  myLabelColumn = myDlg.dialogColumns.add()
  with (myLabelColumn) {
    staticTexts.add({staticLabel:"Диапазон страниц:"});
    staticTexts.add({staticLabel:"Начальная страница для помещения:"});
```

```
  staticTexts.add()
  staticTexts.add({staticLabel:"Опции размещения:"});
  staticTexts.add({staticLabel:"Границы:"});
}

myControlsColumn = myDlg.dialogColumns.add()
with(myControlsColumn){
  var myRangeField = integerEditboxes.add({editValue: 1-, ⮷
    minWidth:60});
  var myFirstPage = integerEditboxes.add({editValue: 1, minWidth:60});
  staticTexts.add()

  var myPlaceOptions = dropdowns.add({stringList:["На всю ⮷
    страницу (включая поля)", "На всю страницу (без полей)", ⮷
    "В центр страницы", "В левый верхний угол страницы"], ⮷
    selectedIndex:2});
  var myCropOptions = dropdowns.add({stringList:['Bounding Box', ⮷
    'Artwork', 'Bleed', 'Media', 'Crop', 'Trim'], selectedIndex:3});
}

var myResult = myDlg.show();
if(!myResult) exit();

if(myRangeField.editValue.split("-")[0] == 0) {
  alert("Начальная страница равна 0");
  exit();
}

if(myFirstPage.editValue == 0) {
  alert("Значение стартовой страницы для размещения PDF в ⮷
    публикации равно 0");
  exit();
}

if(myRangeField.editValue.split("-")[1] == '') {
  alert("Конечная страница PDF не указана");
  exit();
}

// Конец формирования диалога
var myPlacePref = pdfPlacePreferences;

// Считывание введенных значений
with(myPlacePref){
```

```
  switch (myCropOptions.selectedIndex) {
    case 0:
      pdfCrop = PDFCrop.cropContent;
      break;
    case 1:
      pdfCrop = PDFCrop.cropArt;
      break;
    case 2:
      pdfCrop = PDFCrop.cropBleed;
      break;
    case 3:
      pdfCrop = PDFCrop.cropMedia;
      break;
    case 4:
      pdfCrop = PDFCrop.cropPDF;
      break;
    case 5:
      pdfCrop = PDFCrop.cropTrim;
      break;
  }
  transparentBackground = true
}

myRange = myRangeField.editValue.split("-");
myStartPage = myRange[0]
myEndPage = myRange[1]

// Определение конечного числа страниц
newPageRange = (myEndPage - myStartPage) + myFirstPage

// Добавление требуемых страниц
while(newPageRange > myDoc.pages.length) {
  myDoc.pages.add(LocationOptions.atEnd);
}

for (i = myFirstPage-1; i < newPageRange + 1; i++)
{
  myPlacePref.pageNumber = i;
  var myMap = myDoc.pages[i];
  var myFrame = myMap.rectangles.add();

  // Вставка страницы
  myFrame.place(myPDFFile);
  myFrame.fit(FitOptions.frameToContent);
```

```
    var myBounds = myFrame.geometricBounds;
    var myMapBound = myMap.bounds;
    var myMargins = myMap.marginPreferences;
    PDF_Height = myBounds[2] - myBounds[0];
    PDF_Width = myBounds[3] - myBounds[1];

    pageHeight = myDoc.documentPreferences.pageHeight;
    pageWidth = myDoc.documentPreferences.pageWidth;

    // Установка необходимых параметров в зависимости от значений,
    // введенных пользователем
    switch (myPlaceOptions.selectedIndex)
    {
      // Расположение по размерам страницы с учетом полей
      case 0:
        myFrame.geometricBounds = myMapBound;
        myFrame.fit(FitOptions.proportionally);
        break;

      // То же самое, только без полей
      case 1:
        myFrame.geometricBounds = Array(myMapBound[0] + ↵
          myMargins.top, myMapBound[1] + myMargins.left, ↵
          myMapBound[2] - myMargins.bottom, ↵
          myMapBound[3] - myMargins.right);
        myFrame.fit(FitOptions.proportionally);
        break;

      // Помещение в центр страницы
      case 2:
        deltaY = (pageHeight - PDF_Height)/2;
        deltaX = (pageWidth - PDF_Width)/2;
        myFrame.move(undefined,[deltaY, deltaX]);
        break;

      // В левый верхний угол
      case 3:
        myFrame.move(undefined,[0, 0]);
        break;
    }
  }

  myDlg.destroy();
}
```

Начало — традиционное: задание основных переменных. В окне импорта отображаем только файлы формата PDF. В пользовательском окне задаем опции: диапазон импортируемых страниц, страницу публикации, с которой начнется размещение страниц PDF, опции размещения (на всю страницу (включая поля), на всю страницу (без полей), в центр страницы, в левый верхний угол страницы).

В принципе, для создания диалоговых окон можно использовать разные методы, в данном случае был применен несколько отличающийся от рассматривавшихся раньше. Некоторым он покажется более простым в силу большей наглядности.

Для дальнейшей корректной работы желательно проверить введенные пользователем данные. Проверяем диапазон начальной и конечной страниц, затем подготавливаем переменные, значения которых будут использоваться (`pdfPlacePreferences.pdfCrop`).

Далее определяем количество страниц, которые при необходимости потребуется добавить. Диапазон страниц задан через тире. Соответственно, оно же и будет выступать разделителем для получения стартовой и конечной страниц:

```
myStartPage = myRangeField.editValue.split("-")[0];
myEndPage = myRangeField.editValue.split("-")[1];
```

После чего, начиная с указанной страницы, помещаем постранично содержимое PDF. Содержимое PDF-файла, как и любой графики, как известно, помещается в контейнер (объект типа `rectangle`, `oval` и т. п.). В нашем случае выберем `rectangle`.

```
for (i = myFirstPage-1; i < newPageRange + 1; i++)
{
  myPlacePref.pageNumber = i;
  var myMap = myDoc.pages[i];
  var myFrame = myMap.rectangles.add();
```

Наконец, размещаем страницу PDF и масштабируем контейнер по размерам содержимого:

```
  myFrame.place(myPDFFile);
  myFrame.fit(FitOptions.frameToContent);
```

В зависимости от установок пользователя выполняем необходимые преобразования.

Если выбрано размещение PDF на всю страницу (включая поля):

```
case 0:
  myFrame.geometricBounds = myMapBound;
```

```
  myFrame.fit(FitOptions.proportionally);
  break;
case 1:
```

Если выбрано размещение на всю страницу, но без полей — с учетом направления осей: следите, где минус, где плюс:

```
  myFrame.geometricBounds = Array(↵
    myMapBound[0] + myMargins.top, ↵
    myMapBound[1] + myMargins.left, ↵
    myMapBound[2] - myMargins.bottom, ↵
    myMapBound[3] - myMargins.right);
  myFrame.fit(FitOptions.proportionally);
  break;
case 2:
```

Если пользователем выбрана опция помещения PDF в центре страницы, то сначала определяем величину смещения, исходя из размеров листа и страницы PDF. Используем метод `move()`, который имеет следующие параметры:

```
  deltaY = (pageHeight - PDF_Height)/2;
  deltaX = (pageWidth - PDF_Width)/2;

  myFrame.move(undefined, [deltaY, deltaX]);
```

Первый параметр (`undefined`) указывать необходимо, чтобы InDesign воспринял наши значения как относительное смещение (идет вторым параметром), иначе они будут трактоваться как абсолютные координаты.

Если выбрана привязка к левому верхнему углу страницы (как вы, наверно, помните, точка отсчета привязана именно к нему), то:

```
case 3:
  myFrame.move([0, 0]);
  break;
```

9.4. Создание каталога изображений

Еще один пример работы с изображениями — создадим публикацию, которая играет роль каталога изображений, находящихся в конкретной папке. Скрипт работает не только под Windows, но и под Mac OS, что будет полезно при создании кроссплатформенных решений.

Под каждым изображением поставим подпись с названием файла (листинг 9.5).

Листинг 9.5. Формирование каталога изображений

```
myExtensions = [".jpg", ".jpeg", ".eps", ".ps", ".pdf", ".tif", ↳
  ".tiff", ".gif", ".psd", ".ai"]
var myFolder = Folder.selectDialog("Выберите папку с изображениями", "");
if(myFolder != null){
  if(File.fs == "Macintosh") {
    // Предварительная фильтрация изображений,
    // соответствующих заданному типу
    var myFilteredFiles = myMacOSFileFilter(myFolder);
  }
  else{
    // То же самое, только под Windows
    myFilteredFiles = myWinOSFileFilter(myFolder);
  }
  if(myFilteredFiles.length != 0) {
    myDisplayDialog(myFilteredFiles, myFolder);
    alert("Done!");
  }
}

function myWinOSFileFilter(myFolder){
  var myFiles = new Array;
  var myFilteredFiles = new Array;
  for(i = 0; i < myExtensions.length; i++){
    myExtension = myExtensions[i];
    myFiles = myFolder.getFiles("*"+ myExtension);
    if(myFiles.length != 0){
      for(var j = 0; j < myFiles.length; j++){
        myFilteredFiles.push(myFiles[j]);
      }
    }
  }
  return myFilteredFiles;
}

function myMacOSFileFilter(myFolder){
  var myFilteredFiles = myFolder.getFiles(myFileFilter);
  return myFilteredFiles;
}

function myFileFilter(myFile){
  var myFileType = myFile.type;
```

```
  switch (myFileType){
    case "JPEG":
    case "EPSF":
    case "PICT":
    case "TIFF":
    case "GIFf":
    case "PDF ":
      return true;
      break;
   }
  return false;
}

function myDisplayDialog(myFiles, myFolder){
  var myLabelWidth = 130;
  var myDialog = app.dialogs.add({name:"Создание каталога"});
  with(myDialog.dialogColumns.add()){
    with(dialogRows.add()){
      with(dialogColumns.add()){
        staticTexts.add({staticLabel:"Информация"});
      }
    }
  }
  with(borderPanels.add()){
    with(dialogColumns.add()){
      with(dialogRows.add()){
        staticTexts.add({staticLabel:"Папка:", minWidth:myLabelWidth});
        staticTexts.add({staticLabel:myFolder.path + "/" + ↳
          myFolder.name});
      }
      with(dialogRows.add()){
        staticTexts.add({staticLabel:"Количество изображений:", ↳
          minWidth:myLabelWidth});
        staticTexts.add({staticLabel:myFiles.length + ""});
      }
    }
  }
  with(dialogRows.add()){
    staticTexts.add({staticLabel:"Опции"});
  }
  with(borderPanels.add()){
    with(dialogColumns.add()){
      with(dialogRows.add()){
```

```
  with(dialogColumns.add()){
    staticTexts.add({staticLabel:"Количество строк:", ⮷
      minWidth:myLabelWidth});
    }
  with(dialogColumns.add()){
    var myNumberOfRowsField = integerEditboxes.add({editValue:3});
  }
}
with(dialogRows.add()){
  staticTexts.add({staticLabel:"Количество столбцов:", ⮷
    minWidth:myLabelWidth});
  myNumberOfColumnsField = integerEditboxes.add({editValue:3});
}
with(dialogRows.add()){
  staticTexts.add({staticLabel:"Horizontal Offset:", ⮷
    minWidth:myLabelWidth});
  myHorizontalOffsetField = measurementEditboxes.add ⮷
   ({editValue:12, editUnits:MeasurementUnits.points});
}
with(dialogRows.add()){
  staticTexts.add({staticLabel:"Смещение для подписи:", ⮷
    minWidth:myLabelWidth});
  myVerticalOffsetField = measurementEditboxes.add({editValue:12, ⮷
    editUnits:MeasurementUnits.points});
}
with (dialogRows.add()){
  with(dialogColumns.add()){
    staticTexts.add({staticLabel:"Label:", minWidth:myLabelWidth});
  }
  with(dialogColumns.add()){
    myLabelsButtons = radiobuttonGroups.add();
    with(myLabelsButtons){
      radiobuttonControls.add({staticLabel:"Ничего", ⮷
        checkedState:true});
      radiobuttonControls.add({staticLabel:"Файл", ⮷
        checkedState:false});
      radiobuttonControls.add({staticLabel:"Полный путь", ⮷
        checkedState:false});
    }
  }
}
with (dialogRows.add()){
  with(dialogColumns.add()){
```

```
        staticTexts.add({staticLabel:"Размещение:", ↳
          minWidth:myLabelWidth});
      }
      with(dialogColumns.add()){
        myFitProportionalCheckbox = ↳
          checkboxControls.add({staticLabel: ↳
          "Пропорциональное масштабирование", checkedState:true});

        myFitCenterContentCheckbox = checkboxControls.add ↳
          ({staticLabel:"Центровка содержимого", checkedState:true});

        myFitFrameToContentCheckbox = ↳
          checkboxControls.add({staticLabel: ↳
          "Подогнать контейнер по размеру содержимого ", ↳
          checkedState:true});
      }
    }
  }
}
// Конец диалога

// Считывание значений и удаление диалога из памяти
myResult = myDialog.show();
if(myResult == true){
  myNumberOfRows = myNumberOfRowsField.editValue;
  myNumberOfColumns = myNumberOfColumnsField.editValue;
  myRemoveEmptyFrames = myRemoveEmptyFramesCheckbox.checkedState;
  myLabels = myLabelsButtons.selectedButton;
  myFitProportional = myFitProportionalCheckbox.checkedState;
  myFitCenterContent = myFitCenterContentCheckbox.checkedState;
  myFitFrameToContent = myFitFrameToContentCheckbox.checkedState;
  myHorizontalOffset = myHorizontalOffsetField.editValue;
  myVerticalOffset = myVerticalOffsetField.editValue;

  // Основная функция
  myMakeImageCatalog(myFiles, myNumberOfRows, myNumberOfColumns, ↳
    myLabels, myRemoveEmptyFrames, myFitProportional, ↳
    myFitCenterContent, myFitFrameToContent, myHorizontalOffset, ↳
    myVerticalOffset);
}
myDialog.destroy();
}
```

```
function myMakeImageCatalog(myFiles, myNumberOfRows, ↳
  myNumberOfColumns, myLabels, myRemoveEmptyFrames, ↳
  myFitProportional, myFitCenterContent, myFitFrameToContent, ↳
  myHorizontalOffset, myVerticalOffset){
  var myMap, myFile, i, myX1, myY1, myX2, myY2, myRectangle, ↳
    myLabelStyle, myLabelLayer;
  var myFramesPerPage = myNumberOfRows * myNumberOfColumns;
  var myDocument = app.documents.add();
  with(myDocument.viewPreferences){
    horizontalMeasurementUnits = MeasurementUnits.points;
    verticalMeasurementUnits = MeasurementUnits.points;
  }
  var myNumberOfFrames = myFiles.length;
  var myNumberOfPages = Math.round(myNumberOfFrames / myFramesPerPage);

  // Если количество размещаемых изображений не помещается в публикации,
  // добавляем нужное количество страниц
  if ((myNumberOfPages * myFramesPerPage) < myNumberOfFrames){
    myNumberOfPages++;
  }

  // Определяем размеры допустимой области на листе
  // для размещения каталога
  var myMap = myDocument.pages[0];
  with(myMap.marginPreferences){
    var myLeftMargin =left;
    var myTopMargin =top;
    var myRightMargin =right;
    var myBottomMargin =bottom;
  }
  with( myDocument.documentPreferences){
    pagesPerDocument = myNumberOfPages;
    facingPages = false;
    var myLiveWidth = (pageWidth - (myLeftMargin + myRightMargin)) + ↳
      myHorizontalOffset
    var myLiveHeight = pageHeight - (myTopMargin + myBottomMargin)
  }

  // По ним определяем количество строк и столбцов,
  // а также их физические размеры
  myColumnWidth = myLiveWidth / myNumberOfColumns
  myFrameWidth = myColumnWidth - myHorizontalOffset
  myRowHeight = (myLiveHeight / myNumberOfRows)
```

```
myFrameHeight = myRowHeight - myVerticalOffset
yMaps = myDocument.pages;

// Если пользователь выбрал добавление названий файлов,
// создаем для этого отдельный слой
if(myLabels >0){
  myLabelStyle = myDocument.paragraphStyles.add({name:"labels"});
  myLabelStyle.pointSize = 8;
  myLabelLayer = myDocument.layers.add({name:"labels"});
}

// Собственно размещение.
// Идем по всем страницам
for (i = myDocument.pages.length-1; i >= 0; i--){
  myMap = myMaps.item(i);

  // Заполняем по очереди каждый ряд
  for (myRowCounter = myNumberOfRows; myRowCounter >= 1; ↳
       myRowCounter--){
    myY1 = myTopMargin + (myRowHeight * (myRowCounter-1));
    myY2 = myY1 + myFrameHeight;

    // В каждую ячейку вставляем изображение
    for (myColumnCounter = myNumberOfColumns; myColumnCounter >= 1; ↳
         myColumnCounter--){
      myX1 = myLeftMargin + (myColumnWidth * (myColumnCounter-1));
      myX2 = myX1 + myFrameWidth;

      myRectangle = myMap.rectangles.add(myDocument.layers.item(-1), ↳
        undefined, undefined, {geometricBounds:[myY1, myX1, myY2, ↳
        myX2], strokeWeight:0, ↳
        strokeColor:myDocument.swatches.item("None")});
    }
  }
}

// Пробегаем по всем изображениям и решаем вопрос
// с расположением содержимого в контейнере
for (i = 0; i < myNumberOfFrames; i++){
  myFile = myFiles[i];
  myRectangle = myDocument.rectangles.item(i);
  myRectangle.place(File(myFile));
  myRectangle.label = myFile.fsName.toString();
```

```
    if(myFitProportional){
      myRectangle.fit(FitOptions.proportionally);
    }
    if(myFitCenterContent){
      myRectangle.fit(FitOptions.centerContent);
    }
    if(myFitFrameToContent){
      myRectangle.fit(FitOptions.frameToContent);
    }

    // Если нужна подпись для изображения, делаем это
    if(myLabels >0){
      myX1 = myRectangle.geometricBounds[1];
      myY1 = myRectangle.geometricBounds[2];
      myX2 = myRectangle.geometricBounds[3];
      myY2 = myRectangle.geometricBounds[2]+myVerticalOffset;

      switch(myLabels){
        case 1:
          myString = myFile.name;
          break;
        case 2:
          myString = myFile.fsName.toString();
          break;
      }

      myTextFrame = myRectangle.parent.textFrames.add(myLabelLayer, ↵
        undefined, undefined,{geometricBounds:[myY1, myX1, myY2, myX2], ↵
        contents:myString});
      myTextFrame.textFramePreferences.firstBaselineOffset = ↵
        FirstBaseline.leadingOffset;
      myTextFrame.paragraphs[0].appliedParagraphStyle = myLabelStyle;
    }
  }
}
```

Сначала создаем список типов файлов, которые будут включены в каталог. Ограничимся десятью наиболее востребованными:

```
myExtensions = [".jpg", ".jpeg", ".eps", ".ps", ".pdf", ".tif", ".tiff",
".gif", ".psd", ".ai"]
```

Получаем путь к искомой папке и список всех файлов по заданной маске в ней. Показаны два варианта — при работе в Mac OS и Windows. Далее идет вывод диалогового окна функцией `myDisplayDialog(myFiles, myFolder)`.

Основная функция — `myMakeImageCatalog()`.

Сначала определяем размеры рабочей зоны на листе и количество изображений на нем, что позволяет нам определить, сколько страниц `myNumberOfPages` займет весь каталог, затем создаем отдельный слой для подписей с названиями изображений. Следующий шаг — определяем положение на листе для каждого импортируемого изображения и начинаем импорт. При необходимости выполняем операцию `Fit`, после чего занимаемся подрисуночной подписью.

Как видите, львиную долю всего кода занимают создание диалогового окна и считывание из него значений — именно поэтому автор во многих случаях предпочитает пользоваться стандартным методом JavaScript — `prompt()`. Если параметров много, их можно распределить между двумя отдельными вызовами `prompt()` — в таком случае и читабельность кода повышается, и размер интерфейсной части сокращается в несколько раз.

9.5. Автомат для создания фреймов

На данном этапе мы научились пользоваться большинством свойств текстовых и графических объектов, создали даже несколько сценариев, имеющих реальную практическую пользу. Теперь предлагаю решить еще одну задачу, с которой наверняка верстальщики сталкиваются достаточно часто. Предположим, перед нами стоит задача выполнить верстку из файла Word, изобилующего графиками, таблицами. Каждый график должен быть помещен в отдельный фрейм, имеющий определенный стиль; шапка и подпись к нему должны иметь свои стили.

Скрипт будет располагать графику точно в том же месте в тексте, где она стоит в документе Word. Единственное — поскольку фрейм становится заякоренным, его можно сдвигать относительно точки вставки.

Проще всего решить задачу импортом оригинала в формате Word, после чего автоматически заменять вложенную графику на заранее подготовленную, а подрисуночные подписи также автоматически вставлять во фреймы. Таким образом, для корректной работы скрипта:

- графические файлы должны быть подготовлены и помещены в определенную папку;
- подрисуночные надписи должны всегда точно располагаться на своих местах.

Порядок следования должен быть таким: сначала название диаграммы, затем в следующем абзаце — сама диаграмма, в следующем абзаце — подпись к диаграмме. Далее снова должен идти текст и так до конца материала (рис. 9.1).

- Крупные банки Западной Европы
- Крупные Казахские и Прибалтийские банки
- Крупные Российские банки.

КРУПНЫЕ БАНКИ ЗАПАДНОЙ ЕВРОПЫ

Активная деятельность по расширению своего присутствия в странах Восточной Европы, которая началась в середине 1990-х, была вызвана двумя основными факторами:

- более высокой нормой рентабельности в банковском секторе Восточной Европы (рентабельность активов банков Западной Европы в среднем меньше 1 %, в то время как рентабельность активов банков в Восточной Европе колеблется от 1 до 6 %);
- насыщенностью рынка банковских услуг Западноевропейских стран (уровень проникновения банковских услуг в Западной Европе составляет 200-300 %, тогда как в странах Восточной Европы, на момент начала экспансии западноевропейских банков, - 30-50 %).

Объемы сделок по слиянию и поглощения в банковском секторе в странах Восточной Европы (2005-2006), млн.долл.США

Источник: Bloomberg, расчеты City Finance

Западноевропейские банки относятся к группе крупных банков, имеющих низкие показатели рентабельности. Наиболее перспективными рынками для них являются рынки банковских услуг Турции, Румынии, России и Украины как стран с высокими темпами роста экономики, ненасыщенными рынками, высокими темпами роста доходов населения и большим населением. Общая сумма, потраченная на слияния и поглощения в Восточной Европе за 2005-2006 гг. составляет около 38,0 млрд. долл. США, из которых более 16,0 млрд. долл. США – в Турции, 5,6 млрд. долл. США – в Румынии, около 4,3 и 4,0 млрд. долл. США – в России и Украине соответственно.

Детальный анализ потенциальных покупателей по странам Восточной и Западной Европы приведен в Отраслевом отчете в полном объеме по M&A в банковской системе Украины.

КРУПНЫЕ БАНКИ КАЗАХСТАНА И ПРИБАЛТИКИ

Банки Казахстана и Прибалтики имеют опыт работы в рискованной экономической и политической среде. Крупнейшие казахстанские банки имеют значительные ресурсы, аккумулированные в нефтегазовом секторе, а прибалтийские банки, имеют почти стопроцентный иностранный капитал, что позволяет им

Рис. 9.1. Пример работы скрипта. Диаграмма была предварительно отформатирована в Illustrator, после чего в формате EPS помещается в публикацию. Публикация выполнена одноколоночной, свободное место полосы набора отведено под иллюстративный материал

А вот результат работы скрипта в случае, если диаграммы идут одна за другой, а текст, который бы разделял их, отсутствует — в таком случае автоматически создается таблица (рис. 9.2).

Вы можете возразить: почему нельзя использовать возможности библиотек (Library) — ведь, создав шаблон, можно легко его использовать в публикации? Причин несколько.

Во-первых, скрипт может сразу же наполнить фрейм необходимым содержанием (вставить название диаграммы, саму диаграмму из файла на диске, подпись).

Во-вторых, если диаграммы имеют разные размеры, скрипт будет автоматически изменять рамку для них и сам текстовый фрейм таким образом, чтобы

отображалось все содержимое — с использованием библиотек это придется выполнять вручную.

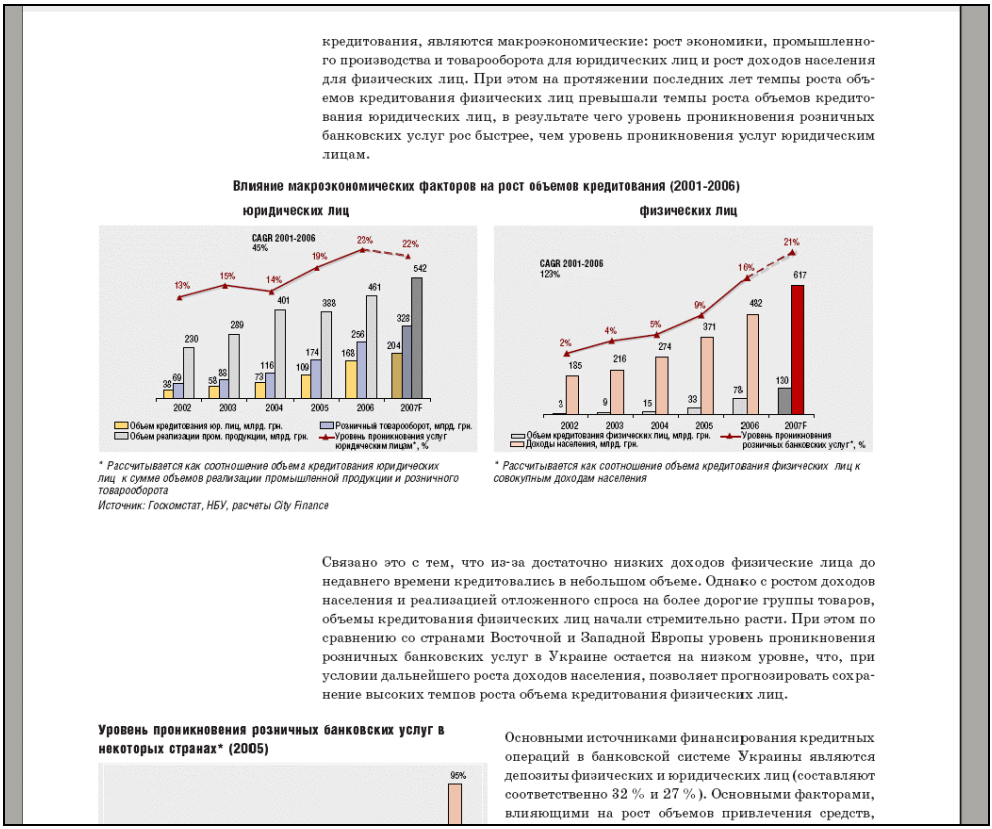


Рис. 9.2. Результат работы скрипта в случае, если диаграммы идут одна за другой

В-третьих, на скрипт можно возложить разные дополнительные функции: например, если висячая строка короткая, применять трекинг, чтобы она втянулась на предыдущую строку, если же трекинг эффекта не дал (висячая строка слишком длинная), трекинг отменять. Вроде мелочь, секундное дело, но когда перед вами публикация с несколькими сотнями таких диаграмм, то секунды превратятся в часы механической, малотворческой работы.

Наконец, через скрипт можно легко добавить подписи в формате "Таблица 1. Название таблицы", "Таблица 2. Название таблицы", "Диаграмма 1. Название диаграммы" и т. д., чтобы нумерация возобновлялась в каждом начальном фрейме из цепочки.

Скрипт, в несколько модифицированном виде, можно использовать для решения похожих задач: например, для заключения объявлений в рамку и т. п.

Напишем универсальный скрипт, который создает как свободные текстовые блоки, так и заякоренные фреймы (листинг 9.6).

Листинг 9.6. Автоматическое создание фреймов для иллюстраций

```
_WIDTH = 87
PREF_HIGHT = 51
home = prompt("/F/!_EconomicReview/!_2006/year/Links")
aD = app.activeDocument
mySelection = app.selection[0]
if(mySelection!='undefined'){
  textF = mySelection.parentTextFrames[0]
  curr_Page = textF.parent.name - 1
  free_tF = aD.pages[curr_Page].textFrames
  pars = mySelection.paragraphs
  inline_tF = pars[0].textFrames
  startP = pars.previousItem(pars[0]);
  endP = pars.nextItem(pars[0])
  pS = aD.paragraphStyles
  l = aD.links
  l_curr=0
  pSA = [];
  aS = ['Diagram Header', 'Diagram', 'Diagram Footer', 'Spacer'];

  for(i=0; i<l.length; i++){
    // Определяем максимальный номер в названии файла,
    // встроенного в публикацию, и продолжаем со следующего
    if (l[i].name.charCodeAt(0) < 58 && parseInt(l[i].name) > l_curr){
      l_curr = parseInt(l[i].name)
    }
  }
  next = l_curr + 1

  // Запрос на тип контейнера
  ask = prompt("In-line Frame? [1/0]", "0")
  listFolder = Folder.selectDialog("Select picture folder", home)

  // Сначала проверяем наличие файлов с расширением emf
  // (такова специфика подготовки публикации), если нет — ищем eps
  fname = listFolder + '/' + next + '.emf'
  if (!File(fname).exists){
    fname = listFolder + '/' + next + '.eps'
  }
```

```
if (File(fname).exists) {
  for (var i=2; i<pS.length; i++){
    switch(pS[i].name){
      case 'Diagram Header':  pSA[ aS[0] ] = i;   break;
      case 'Diagram':         pSA[ aS[1] ] = i;   break;
      case 'Diagram Footer':  pSA[ aS[2] ] = i;   break;
      case 'Spacer':          pSA[ aS[3] ] = i;   break;
    }
  }

  // В зависимости от того или иного выбора пользователя
  // в дальнейшем будем использовать ссылки на разные объекты
  target = (ask==0) ? free_tF : inline_tF

  // Не привязанный к тексту контейнер
  if(target==free_tF){
    // Позицию для контейнера выбираем так, чтобы он находился
    // по вертикали на том же уровне, что и курсор.
    // Фактически это эквивалент in-line-графики,
    // но только без привязки к тексту.
    top = (mySelection.baseline < textF.geometricBounds[2] - ↳
      PREF_HIGHT)? mySelection.baseline : (textF.geometricBounds[2] - ↳
      PREF_HIGHT)
    target.add({geometricBounds: Array(top, ↳
      mySelection.parentTextFrames[0].geometricBounds[1], ↳
      top + PREF_HIGHT+10, _WIDTH + ↳
      mySelection.parentTextFrames[0].geometricBounds[1])})
  }else{
    // Контейнер, встроенный в текст
    target.add()
    with(target[0]) {
      // Обновляем содержимое родительского объекта
      parent.recompose();

      // Устанавливаем минимально необходимые размеры для фрейма
      geometricBounds = ↳
        [geometricBounds[0],geometricBounds[1], ↳
         geometricBounds[0]+PREF_HIGHT, geometricBounds[1]+_WIDTH] }
  }

  // Вставляем в контейнер заголовок и подрисуночную подпись,
  // оставляем пустой абзац для вставки в него иллюстрации
```

```
with(target[0]){
  contents = startP.contents +'\r'+ endP.contents.substr(0, ↳
    endP.contents.length-1)

  // Сразу же форматируем текст
  paragraphs[0].applyStyle(pS [pSA [ aS[0]]], true)

  // В пустой абзац вставляем контейнер для иллюстрации
  paragraphs[1].rectangles.add({fillColor: aD.swatches[0], ↳
    strokeWeight: 0})

  // Стандартная процедура при внедрении в текст графических объектов
  paragraphs[1].recompose()

  // Вставляем иллюстрацию
  with(paragraphs[1].rectangles[0]){
    place(File(fname))
    w = allGraphics[0].geometricBounds[3]- ↳
        allGraphics[0].geometricBounds[1]
    h = allGraphics[0].geometricBounds[2]- ↳
        allGraphics[0].geometricBounds[0]

    // В зависимости от размеров иллюстрации принимаем решение:
    // если они меньше, чем у контейнера, то содержимое центрируем;
    // если больше — расширяем размеры контейнера до размеров
    // иллюстрации, чтобы ничего не осталось скрытым
    if (w < _WIDTH-1 || h < PREF_HIGHT-1) {
      geometricBounds = [geometricBounds[0], ↳
        geometricBounds[1], geometricBounds[0]+PREF_HIGHT, ↳
        geometricBounds[1]+_WIDTH]
      fit(FitOptions.centerContent)
    } else fit(FitOptions.frameToContent)

    // Присваиваем стиль контейнеру иллюстрации.
    // InDesign поддерживает стили для объектов
    applyObjectStyle(aD.objectStyles[1], true)
  }

  // Если контейнер был увеличен в размерах, необходимо увеличить
  // размеры текстового фрейма — иначе останется переполнение
  if(overflows) fit(FitOptions.frameToContent)
  if(geometricBounds[3] < rectangles[0].geometricBounds[3])
```

```
    geometricBounds = [geometricBounds[0], ↳
      geometricBounds[1],geometricBounds[2], ↳
      rectangles[0].geometricBounds[3]]

// Задаем форматирование для абзаца с иллюстрацией и подписи к ней
paragraphs[1].applyStyle(pS [pSA [ aS[1]]], true)
paragraphs[2].applyStyle(pS [pSA [ aS[2]]], true)

// Корректируем размеры текстового фрейма с учетом форматирования
// заключенного в нем текста
fit(FitOptions.frameToContent)
```

Почему нельзя было совместить операции `fit()` — вместо двух использовать всего одну? Дело в том, что InDesign "видит" лишь тот текст, который видим. Предположим, мы уберем первую операцию — что получится? Если контейнер с иллюстрацией превысил свои первоначальные размеры, оба абзаца — содержащий контейнер для публикации и следующие — в контейнер не поместятся, таким образом, они выпадают из поля зрения InDesign. В результате никаких операций с ними проделать уже нельзя — этого-то мы и избегаем.

Пытаемся избежать ситуации, когда на следующую строчку переносится всего несколько символов — уменьшаем трекинг текста до –25 (найдено эмпирическим путем). Если количество строк уменьшилось, значит, нам удалось избежать "висячки", трекинг оставляем. Если же количество строк осталось неизменным, значит, данная мера не помогла, текста слишком много, а больше сжимать уже нельзя, поэтому отпускаем трекинг до прежнего значения:

```
with(paragraphs[0]) {
  tracking -= 25;
  if (lines.length>1) tracking += 25 }
applyObjectStyle(aD.objectStyles[5], true)
}

// Последние, дополнительные шаги — зачистка:
// удаляем из импортированного текста те строчки,
// которые были перенесены в созданные нами фреймы.
if(target==inline_tF) {
  target[0].parent.applyStyle(pS [pSA [aS[3]]], true)
  endP.remove();
  startP.remove()
  pars[0].spaceBefore = 0
} else {
  pars.itemByRange(startP, endP).remove()
}
```

```
} else {
  alert ("Требуемый файл не найден") }
}else {
  alert ("Поместите курсор в место, где должна быть размещена ↵
    иллюстрация")
}
```

Задаем требуемую ширину и высоту фрейма, а также указываем путь `home` к диаграммам по умолчанию. Поскольку этот скрипт писался для собственного использования, в нем было решено отказаться от диалогов, размеры кода которых зачастую превышают размеры самой работающей части, а открыть скрипт раз в месяц и заменить в нем одну строчку — минутное дело. Также не делается проверка на тип выделения — предполагается, что курсор стоит в том месте, куда будет помещена диаграмма.

Дальнейшие действия выполняются только, если существует выделение. Среди них: ссылки на абзац, в котором стоит курсор, задаются стили, которые будут использоваться для форматирования текста: `Diagram Header`, `Diagram`, `Diagram Footer`, `Spacer`.

Работа скрипта построена на следующем допущении. Все внедряемые иллюстрации должны быть собраны в одной папке и, кроме того, пронумерованы — соответственно, первым будет импортирован файл с названием "1", следующим — с названием "2" и т. д. Скрипт сначала определяет наибольшее значение в названиях графических файлов из выбранной папки и продолжает работу со следующего. Это удобно, если работа над публикацией занимает больше одного дня. В таком случае корректная работа будет также обеспечена, поскольку скрипт продолжает импорт с последнего внедренного изображения.

```
for( i=0; i<l.length; i++){
  if (l[i].name.charCodeAt(0)<58 && parseInt(l[i].name) > l_curr)
  l_curr = parseInt(l[i].name)
}
```

9.6. Автомат по раскладке рекламы на листе

Как правило, раскладка рекламы в специализированных изданиях — задача не самая творческая: перед вами не менее 4 полос формата A3, которые сплошь испещрены рекламой — крупной, мелкой, ее наберется несколько сотен. И так каждую неделю, а если еще есть региональные выпуски — то и чаще... Тут поневоле задумаешься о том, как бы облегчить себе работу. Для таких задач скриптинг подойдет как нельзя лучше. Поэтому и был создан эдакий "автомат" по раскладке рекламы. Как любая более-менее серьезная задача, она нуждается в техническом задании — в нем нужно описать весь круг задач, которые скрипт должен выполнять.

Самое главное, конечно же, — расстановка рекламы. При этом нужно учитывать определенный порядок — достаточно часто рекламодатели заказывают определенное место, оплачивая это отдельно. Для остальных реклам порядок не столь важен — тут действует правило: кто подал рекламу первым — у того она появится на листе раньше (выше) остальных.

Чтобы задача могла быть автоматизирована, названия файлов должны идти в конкретном порядке — не важно, в каком именно, главное, чтобы присутствовала определенная закономерность, например, названия либо увеличивались, либо уменьшались. Выбор в качестве отправной точки названия файла очень удачен: во-первых, этот критерий очень наглядный, во-вторых, в таком случае ничего не понадобится переделывать в производственном процессе — название файла, как правило, соответствует номеру заказа, а они, естественно, идут в возрастающем порядке.

Далее. Необходимо точно знать размер листа — не только по физическим характеристикам (см или мм), но и в логических (в рекламных модулях), поскольку мы будем работать именно на уровне этих модулей. При этом положение на листе каждого отдельного рекламного модуля будет однозначно определяться его порядковым номером.

Итак, определяем количество рядов и столбцов рекламы — нашу макетную сетку. На одном листе помещается 9 рядов, в каждом по 4 колонки. Учтем, что реклама может быть многомодульной: минимальный размер соответствует одному модулю, встречаются двух- и более модульные рекламные блоки — соответственно, нужно так продумать процесс раскладки, чтобы площадь листа использовалась максимально — ведь может случиться, что после размещения мелкой рекламы оставшееся место окажется слишком фрагментированным — при этом более крупные блоки просто не поместятся в эти фрагменты, хотя суммарная свободная площадь это позволяет. Поэтому начинаем раскладку с самой крупноразмерной рекламы и далее по убывающей, при этом мы не переживаем по поводу фрагментации — в остающиеся фрагменты всегда поместится одномодульная реклама.

Продумаем, какие могут возникнуть проблемы при таком варианте раскладки. Во-первых, теоретически возможна ситуация, когда на первом листе соберутся блоки только крупной рекламы, на другом — более мелкие и т. д., что не очень хорошо. Однако на практике каждая страница издания отведена под определенную рубрику ("Дом", "Окна", "Недвижимость") — соответственно, описанная ситуация исключается.

В целях упорядочения размещения рекламы и для удобства читателей при просмотре рекламные блоки одного размера будем группировать — в таком случае мы сможем избежать ситуации, когда, например, 8-модульная реклама соседствует с одномодульной или тому подобные варианты. Поскольку коли-

чество рекламы того или иного размера — число достаточно случайное, поэтому в каждой рубрике получим относительно ровную ситуацию — рекламные блоки будут идти сверху вниз, постепенно увеличиваясь в размере, что, кстати, воспринимается глазом гораздо лучше, чем, скажем, если бы ситуация была противоположной — когда крупные блоки стоят в самом верху полосы набора.

Важнейший вопрос: как определить, какие модули уже заняты, а какие еще нет? Самый простой способ — создать массив по размерам модульной сетки (9×4) и после очередного размещения рекламного блока отмечать в массиве соответствующую ячейку (или несколько ячеек — в случае рекламы, занимающей более одного модуля) как занятую. В результате мы получим карту занятых и свободных областей, что даст нам информацию о том, куда нужно ставить следующую рекламу. Переход от порядкового номера свободного модуля до получения его координат — дело техники.

Для наглядности порядок заполнения страницы показан на рис. 9.3

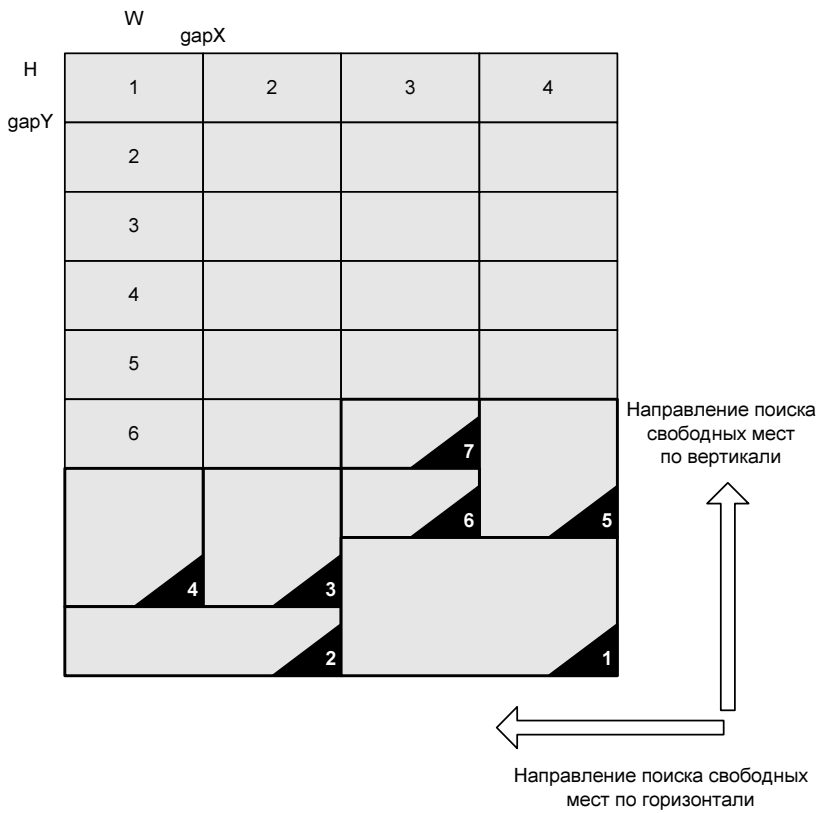


Рис. 9.3. Принцип расположения рекламных блоков в страничных модулях

Точки вставки показаны черными треугольниками. Первая такая точка — крайний правый нижний модуль. Предположим, что у нас существует только одна реклама размером 2×2 модуля. Тогда она займет 4 модуля (размеры рекламы показаны толстой линией). После ее вставки скрипт найдет следующий пустой модуль — он обозначен черным треугольником под номером 2. Пусть у нас есть одна горизонтальная реклама размером 1×2 модуля (ее размеры также показаны жирной линией). Следующий свободный модуль обозначен треугольником с номером 3, в него вставляется реклама вертикальная. Пусть у нас три таких рекламы, которые займут соответственно места 3, 4 и 5 (см. стрелки, показывающие направление поиска свободных мест), после чего пойдет реклама одномодульная — (места 6, 7 и т. д.)

Описательную часть будем считать законченной. Осталось воплотить все это на языке JavaScript. Составим алгоритм действий:

1. Устанавливаем модульную сетку.
2. Задаем папку, в которой находится реклама. Предполагается, что на каждую рубрику отведена отдельная папка. В данном примере ограничимся раскладкой только на одну полосу, но распространить работу скрипта на все разделы (папки) не представляет никакой проблемы — в таком случае можно получить полностью автоматизированный рабочий процесс.
3. Получаем список всех файлов из папки и переставляем его в обратном порядке, поскольку заполнение листа будет начинаться с низа полосы, т. е. туда будут помещены файлы с самыми большими номерами в названиях.
4. Сортируем файлы: в одну группу — рекламу размером в 1 модуль, в другую — размером в 2 модуля, при этом не забываем о разбивке по высоте и ширине. Например, реклама, занимающая 2 модуля по высоте (соответственно 1 модуль по ширине), и реклама, занимающая 2 модуля по ширине (и 1 по высоте), окажутся в разных группах. Для чего это нужно? А как нам определить, какие именно блоки заняты — ведь два блока можно положить горизонтально, а можно поставить вертикально, при этом карта свободных областей совершенно изменится! Побочный положительный эффект более глубокой сортировки — более логичное заполнение полосы.

   Поскольку в InDesign отсутствует механизм определения размеров файла до его размещения в публикации (вопросы, связанные с Bridge, выходят за рамки данной книги), поэтому каждое изображение временно помещаем в публикацию, считываем размеры и тут же его заменяем следующим по списку. В результате получим полное представление о качественном и количественном составе рекламы на полосе — нам открывается прямая дорога к расстановке макетов.
5. Начинаем с самой крупной рекламы. Выбираем самый последний модуль и вставляем туда рекламу. После того как изображение размещено, сво-

бодная площадь полосы сократится на размер этой рекламы. Поэтому на карте распределения рекламы область, где стоит реклама, помечаем как занятую. Размер области задан в модулях, поэтому с этим проблем никаких не возникает. Кроме того, поскольку известны пропорции рекламного макета, мы можем точно сказать, какие именно модули он занял, что даст нам точную картину распределения свободной площади.

6. Перед размещением следующей рекламы в оставшейся свободной области находим первый свободный участок (опять же идем с самой нижнего доступного модуля) и занимаем его — и так до последнего объявления. Как определить координаты места, куда ставить следующую рекламу? Для этого мы пронумеруем модули особым способом: первым числом в его номере будет идти номер ряда, в котором он находится, а вторым — номер колонки (поскольку ни то ни другое число не превышает 9, проблем, возможных при использовании двузначных чисел, не возникнет). То есть, если модуль имеет номер 46, это значит, что он расположен в четвертом ряду и шестой колонке. Зная размеры модуля (они фиксированы), определить его координаты не представляет никакой проблемы.
7. Собственно говоря, вот и весь процесс. При необходимости, можно пойти дальше и указывать, какие блоки на странице отводятся под рекламу без оговоренного положения, а в какие будет вставлена реклама с конкретным местом на странице (для этого достаточно использовать систему пометок в названиях файлов). И вообще, рассматриваемый сценарий — лишь демонстрация возможностей скриптинга, свободно модифицируя его, добавляя те или иные функции, можно решить многие вопросы конкретного технологического процесса.

Итак, скрипт представлен в листинге 9.7.

Листинг 9.7. Автоматическое размещение рекламы в публикации

```
// Задаем модульную сетку
rows = 9;
cols = 4;
startR =0;
startC = 0;

// Задаем папку с изображениями
myFolder = Folder(myFolder)
myPath = myFolder.toString()+'/'

// Получаем список файлов с рекламой и меняем его порядок
// на противоположный
myFiles= myFolder.getFiles().reverse()
```

```
// Вставляем контейнер для изображений, временно размещаемых на листе —
// исключительно для получения их размеров
myRec = app.activeDocument.rectangles.add()

// Задаем размеры одного модуля — по ним будет определяться положение
// рекламных блоков
stepX = 67; stepY = 43

// Зазоры между блоками
gapX = 5; gapY = 5

// Создаем массивы для внесения туда ссылок на файлы.
// Каждая ссылка заносится только в свой массив (с учетом размера
// и пропорций рекламного макета).
for (i=1; i<5; i++) {
  for (j=1; j<5; j++)
    eval('Arr'+i+''+j+'= new Array()')
}

// Массив для хранения положения всех возможных рекламных модулей
positionA = new Array();

// Массив для хранения размеров рекламы
gBA = new Array();

// Массив для хранения свободных и занятых областей
myMap = new Array()

for (i=0; i<rows; i++) {
  positionA[i] = new Array()

  // Создаем карту свободных областей
  myMap[i] = new Array()
  for (j=0; j<cols; j++) {
    // Задаем координаты каждого рекламного модуля.
    // Используем особую нумерацию модулей, позволяющую точно
    // определить, в каком ряду и какой колонке он находится.
    // Первым записываем положение по оси y, потом — по оси x
    positionA[i][j] = i * stepY + gapY + ';' + parseInt(j * stepX + gapX)

    // Заполняем карту нулями (начальное состояние).
    // Ноль означает, что модуль свободен.
    myMap[i][j] = 0
  }
}
```

```
// Выполняем временный импорт всех файлов
for (i in myFiles){
  pic = File(myFiles[i]).name
  myRec.place(File(myPath + pic));
  img = myRec.images[0]

  // И записываем размеры каждого изображения

  gBA[i] = [parseInt (img.geometricBounds[2]-img.geometricBounds[0]), ↳
    parseInt(img.geometricBounds[3]-img.geometricBounds[1])]

  // Сортируем рекламу по размерам.
  // Минимальный размер — 1x1, максимальный выбран как 4x4 модуля
```

Создаем столько массивов, сколько имеется типоразмеров рекламы (разных размеров с учетом расположения — горизонтальная/вертикальная) и в каждый заносим соответствующие названия файлов. В название каждого массива добавляем числовой индекс, соответствующий размерам изображений, ссылки на которые добавляются в массив. Например, в массив `Arr42` заносим ссылки на рекламу размером 4 модуля по высоте и 2 по ширине, в `Arr12` — размером 1 модуль по высоте и 2 в ширину, и т. д.

```
  for (j=1; j<=4; j++) {
    for (k=1; k<=4; k++){
      if (gBA[i][0] < k* stepY){
        eval('Arr' + parseInt(j + "" + k) + '.push(pic)'));
        continue;
      }
    }
  }
}
// После получения размеров рекламных макетов из папки
// временный контейнер удаляем
app.activeDocument.rectangles[0].remove()

// Собственно начало размещения.
// Находим положение рекламы, размещаемой самой первой.
findCurrPos()

// В зависимости от размера и пропорций рекламного макета
// ищем рекламу размером 4x2 модуля, расположение вертикальное
for (i in Arr42){
  // Импортируем его в публикацию
  addPicture(Arr42, i)
```

```
  // Устанавливаем в нужную позицию.
  // Внимание: учитываются размер и пропорции рекламы!
  myRec.move([X-stepX, Y-3*stepY])

  // Помечаем соответствующие модули на странице как занятые
  myMap[myRow-1][myColumn] = 1
  myMap[myRow-1][myColumn-1] = 1
  myMap[myRow-2][myColumn] = 1
  myMap[myRow-2][myColumn-1] = 1
  myMap[myRow-3][myColumn] = 1
  myMap[myRow-3][myColumn-1] = 1

  // Определяем следующий свободный модуль
  findCurrPos()
}

// Таким образом, просматриваем содержимое каждого массива.
// При этом начинаем с тех, в которых заключены ссылки
// на самую крупную рекламу, и продолжаем до самой мелкой.
```

Как только мы закончим просмотр всех массивов, это будет означать, что вся реклама из указанной папки уже размещена — в самом начале скрипта мы ее разбрасывали по этим массивам, т. е. количество элементов в массивах в точности соответствует количеству файлов с рекламой.

```
// Ищем рекламу такого же размера, но горизонтальную
for (i in Arr24){
  addPicture(Arr24, i)
  myRec.move([X-3*stepX, Y-stepY])

  myMap[myRow][myColumn-1] = 1
  myMap[myRow-1][myColumn-1] = 1
  myMap[myRow][myColumn-2] = 1
  myMap[myRow-1][myColumn-2] = 1
  myMap[myRow][myColumn-3] = 1
  myMap[myRow-1][myColumn-3] = 1

  findCurrPos()
}

// Ищем рекламу размером 3x1 блок
for (i in Arr31){
  addPicture(Arr31, i)
  myRec.move([X, Y-2*stepY])
```

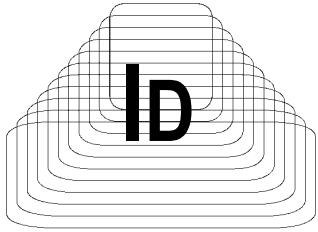
```
  myMap[myRow-1][myColumn] = 1
  myMap[myRow-2][myColumn] = 1

  findCurrPos()
}

// И так далее для всех возможных комбинаций
for (i in Arr32){
  addPicture(Arr32, i)
  myRec.move([X-stepX, Y-2*stepY])

  myMap[myRow-1][myColumn] = 1
  myMap[myRow-1][myColumn-1] = 1
  myMap[myRow-2][myColumn] = 1
  myMap[myRow-2][myColumn-1] = 1

  findCurrPos()
}
for (i in Arr33){
  addPicture(Arr33, i)
  myRec.move([X-2*stepX, Y-2*stepY])

  myMap[myRow-1][myColumn] = 1
  myMap[myRow-1][myColumn-1] = 1
  myMap[myRow-1][myColumn-2] = 1
  myMap[myRow-2][myColumn] = 1
  myMap[myRow-2][myColumn-1] = 1
  myMap[myRow-2][myColumn-2] = 1

  findCurrPos()
}
for (i in Arr13){
  addPicture(Arr13, i)
  myRec.move([X-2*stepX, Y])

  myMap[myRow][myColumn-1] = 1
  myMap[myRow][myColumn-2] = 1

  findCurrPos()
}
for (i in Arr23){
  addPicture(Arr23, i)
  myRec.move([X-2*stepX, Y-stepY])
```

```
  myMap[myRow][myColumn-1] = 1
  myMap[myRow-1][myColumn-1] = 1
  myMap[myRow][myColumn-2] = 1
  myMap[myRow-1][myColumn-2] = 1

  findCurrPos()
}
for (i in Arr22){
  addPicture(Arr22, i)
  myRec.move([X-stepX, Y-stepY])

  myMap[myRow][myColumn-1] = 1
  myMap[myRow-1][myColumn] = 1
  myMap[myRow-1][myColumn-1] = 1

  findCurrPos()
}
for (i in Arr12){
  addPicture(Arr12, i)
  myRec.move([X-stepX, Y])
  myMap[myRow][myColumn-1] = 1

  findCurrPos()
}
for (i in Arr21){
  addPicture(Arr21, i)
  myRec.move([X, Y-stepY])
  myMap[myRow-1][myColumn] = 1

  findCurrPos()
}
for (i in Arr11){
  addPicture(Arr11, i)
  myRec.move([X, Y])
  findCurrPos()
}
// В принципе, перебор всех возможных типоразмеров рекламы
// можно оформить в виде цикла — это даст гораздо большую
// компактность кода

// Функция определения следующей свободной позиции.
// Возвращает координаты для размещения модуля, а также строку и колонку
function findCurrPos(){
```

```
  for (ii=startR; ii<rows; ii++) {
    for (j=startC; j<cols; j++) {
     // Как только найден первый свободный модуль,
     // запоминаем его положение
     if (myMap[ii][j] == 0) {
       myRow = ii
       myColumn = j
     }
   }
  }
  Y = parseInt( positionA[myRow][myColumn].split(";")[0] )
  X = parseInt( positionA[myRow][myColumn].split(";")[1] )

  return [X, Y, myRow, myColumn]
}

// Размещение файла на листе в заданном месте
function addPicture(Arr, i){
  myRec = app.activeDocument.rectangles.add()
  myRec.place(File(myPath + Arr[i]))
  myRec.fit(FitOptions.frameToContent)

  // Пометка на карте свободных и занятых областей
  myMap[myRow][myColumn] = 1
  return myRec
}
```

Кстати говоря, ничего не стоит несколько модифицировать скрипт, например, если нужно предусмотреть место для объявлений чисто текстового свойства — часто они служат островками жизни среди бескрайнего моря разномастной рекламы. В таком случае в начале скрипта нужно указать те модули, которые должны быть сразу же исключены из карты свободных областей — проще всего делать через массив, в котором задавать номера этих модулей.

ГЛАВА 10



Работа с контурами

Процесс верстки, если он действительно творческий, не может обойтись только типографикой и расположением иллюстративного материала на полосах. Обязательный атрибут по-настоящему интересного издания — дополнительные элементы оформления, которые помогают подчеркнуть специфику размещенного материала. Не случайно уже в первых версиях InDesign появились инструменты для создания векторных путей. Собственно говоря, для разработчиков данное направление не стало каким-то особым прорывом — они просто адаптировали движок, существующий в Illustrator, для потребностей InDesign. Именно поэтому процедура работы с ним абсолютно идентична работе в векторном редакторе, насколько это удобно — уже другой вопрос, выходящий за рамки обсуждения в данной книге.

Поскольку автоматизировать творческий процесс невозможно (а добавление графических элементов — именно такой процесс), то одно из направлений применения автоматизации при работе с векторными путями — создание интересных геометрических форм, которые трудно получить, используя стандартный инструментарий InDesign. В самом деле, в редакторе предусмотрен лишь базовый набор примитивов, при помощи которых, даже используя известные математические операции (объединение, пересечение, нахождение общего), получить оригинальные фигуры можно, но на это придется потратить значительное время. Собственно говоря, именно сложность построения красивых и, в особенности, симметричных узоров и является тем препятствием, которое отбивает желание верстальщиков широко применять оригинальные графические элементы в дизайне. В данном случае скриптинг — весьма эффективное решение проблемы. С его помощью можно реализовать, например, создание гильоширных узоров (используются на банкнотах и прочих документах для защиты от несанкционированного копирования), либо создание более простых, но не менее интересных оригинальных узоров для графи-

ческих фреймов. Безусловно, при работе с путями потребуется знание математики, но, как говорится, если желание есть, а тем более, если есть с чего начать — полдела уже сделано. Далее приведен пример скрипта, в котором как раз и поднимаются вопросы, связанные с работой на уровне точек и направляющих — он станет хорошим подспорьем для желающих попробовать свои силы в этом интересном направлении.

Форма кривой определяется взаимным расположением ее точек и соединяющими линиями. В самом простейшем случае, если точки соединены прямыми линиями, достаточно задать обе координаты (*X*, *Y*) для каждой точки (`anchor[0]`, `anchor[1]`). Несколько сложнее ситуация, если вместо прямых линий — кривые. В таком случае потребуется задать координаты и обеих касательных — левой и правой (`leftDirection[0]`, `leftDirection[1]`, `rightDirection[0]`, `rightDirection[1]`) — рис. 10.1.

Трансформации можно проводить над любыми точками прямой (рис. 10.2).

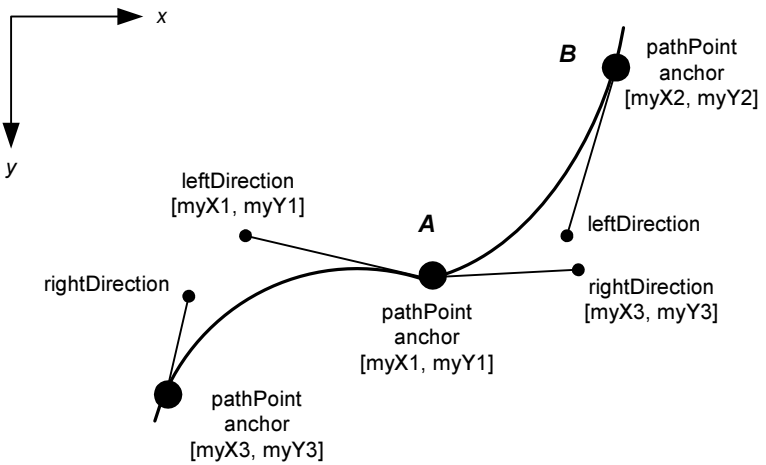


Рис. 10.1. Параметры, задающие форму кривой

Итак, скрипт представлен в листинге 10.1.

Листинг 10.1. Создание эффектов в углах объекта

```
var myObjectList = new Array;
if (app.documents.length != 0){
  if (app.selection.length != 0){
    for(var i = 0;i < app.selection.length; i++){
      switch (app.selection[i].constructor.name){
        case "Rectangle":
```

```
        case "Oval":
        case "Polygon":
        case "GraphicLine":
        case "TextFrame":
          // Если выделенный объект допустимого типа, запоминаем его
          myObjectList.push(app.selection[i]);
          break;
      }
    }
    if (myObjectList.length != 0){
      // Если есть, с чем работать, отображаем диалог
      myDisplayDialog(myObjectList); }
    else{
      alert ("Выберите любой прямоугольник, овал, многоугольник,
текстовый фрейм либо просто линию и попробуйте снова");
    }
  }
  else{
    alert ("Нет ни одного выделенного объекта!");
  }
}

function myDisplayDialog(myObjectList){
  var myStringList = ["все точки", "первая точка", "последняя точка", ↵
    "вторая точка", "третья  точка", "четвертая  точка", ↵
    "нечетные точки", "четные точки"]

  // Сохранение размерностей и перевод их в "правильный" вид
  with(app.activeDocument.viewPreferences){
    var myOldXUnits = horizontalMeasurementUnits;
    var myOldYUnits = verticalMeasurementUnits;

    horizontalMeasurementUnits = MeasurementUnits.points;
    verticalMeasurementUnits = MeasurementUnits.points;
  }
  myDialog = app.dialogs.add();

  // Собственно диалог
  with(myDialog){
    with(dialogColumns.add()){
      with(borderPanels.add()){
        staticTexts.add({staticLabel:"Тип углов:"});
        var myCornerEffectButtons = radiobuttonGroups.add();
```

```
      with(myCornerEffectButtons){
        radiobuttonControls.add({staticLabel:"Скругленные", ↳
          checkedState:true});
        radiobuttonControls.add({staticLabel:"Втянутые"});
        radiobuttonControls.add({staticLabel:"Срезанные"});
        radiobuttonControls.add({staticLabel:"Вырезанные"});
        radiobuttonControls.add({staticLabel:"Оригинальные"});
      }
    }
    with(borderPanels.add()){
      with (dialogColumns.add()){
        staticTexts.add({staticLabel:"Настройки:"});
      }
      with(dialogColumns.add()){
        with(dialogRows.add()){
          with(dialogColumns.add()){
            staticTexts.add({staticLabel:"Величина смещения:", ↳
              minWidth:50});
            staticTexts.add({staticLabel:"Воздействуемые точки:", ↳
              minWidth:50});
          }
          with (dialogColumns.add()){
            var myOffsetEditbox = ↳
              measurementEditboxes.add({editValue:12});
            var myPatternDropdown = ↳
              dropdowns.add({stringList:myStringList, ↳
              selectedIndex:0});
          }
        }
      }
    }
  }
}
// Конец диалога

var myReturn = myDialog.show();
if (myReturn == true){
  // Получение значений из диалога
  var myCornerType = myCornerEffectButtons.selectedButton;
  var myOffset = myOffsetEditbox.editValue;
  var myPattern = myStringList[myPatternDropdown.selectedIndex];
  myDialog.destroy();
```

```
    for(i = 0; i<myObjectList.length; i++){
      myChangeCorners(myObjectList[i], myCornerType, ↳
        myOffset, myPattern);
    }
    // Возврат к предыдущим настройкам единиц измерения
    app.activeDocument.viewPreferences.horizontalMeasurementUnits = ↳
      myOldXUnits;
    app.activeDocument.viewPreferences.verticalMeasurementUnits = ↳
      myOldYUnits;
  }
  else{
    myDialog.destroy();
  }
}

function myChangeCorners(myObject, myCornerType, myOffset, myPattern){
  // Функция создает список всех координат опорных точек,
  // а также координат касательных к ним (в случае наличия кривых
  // сегментов) и записывает их в массив. После того как необходимые
  // координаты вычислены, можно сразу же создать кривые нужной формы
  // используя свойство entirePath. Это гораздо производительнее,
  // чем создавать кривую точка за точкой.

  var myPathPoint, myPoint, myPointA, myPointB, myPointC, myAnchor, ↳
      myX, myY, myX1, myY1, myX2, myY2, myX3, myY3;
  var myNewX1, myNewY1, myNewX2, myNewY2, myXOffset, myYOffset, ↳
      myPoint, myPathPoint;

  for(var myPathCounter = 0; myPathCounter < myObject.paths.length; ↳
      myPathCounter ++){
    var myPath = myObject.paths[myPathCounter];

    // Массив, в который будут заноситься все координаты точек
    // и их направляющих
    myPointArray = new Array;

    for (var myPathPointCounter = 0; myPathPointCounter < ↳
         myPath.pathPoints.length; myPathPointCounter ++){
      // Считывание параметров каждой точки.
      // Определяем, должна ли точка подвергаться преобразованиям.
      if(myPointTest(myPathPointCounter, myPath, myPattern) == false){
        // Если точка не принадлежит ни одной из списка myPattern,
        // сохраняем положение опорной точки и координаты
        // обеих касательных
```

```
    with(myPath.pathPoints[myPathPointCounter]){
      myX1 = leftDirection[0];
      myY1 = leftDirection[1];
      myX2 = anchor[0];
      myY2 = anchor[1];
      myX3 = rightDirection[0];
      myY3 = rightDirection[1];
    }
    // Запись параметров текущей точки без изменений
    myPoint = [[myX1, myY1], [myX2, myY2], [myX3, myY3]];

    // Запоминание параметров в массиве с координатами всех точек
    myPointArray.push(myPoint);
  }
  else{
    // Если точка будет подвергаться трансформациям.
    // Основные преобразования (см. рис. 10.1).
    // Текущая точка — точка A
    myPointA = myPath.pathPoints[myPathPointCounter];
    myAnchor = myPointA.anchor;
    myX1 = myAnchor[0];
    myY1 = myAnchor[1];

    // Для корректных преобразований нужно рассматривать
    // взаимное расположение каждой точки с ее двумя ближайшими
    // соседями. myPointB — следующая точка на прямой.
    // Учитываем особенности работы InDesign с путями.
    // Если myPathPoint — последняя точка прямой, то,
    // соответственно, myPointB - начальная точка.
    if (myPathPointCounter == (myPath.pathPoints.length - 1)){
      myPointB = myPath.pathPoints[0];
    }
    else{
      myPointB = myPath.pathPoints[myPathPointCounter + 1];
    }

    // Запоминание параметров следующей точки
    myAnchor = myPointB.anchor;
    myX2 = myAnchor[0];
    myY2 = myAnchor[1];

    // myPointC — предыдущая точка на прямой.
    // Если myPathPoint — начальная точка прямой, то,
    // соответственно, myPointC - последняя точка.
```

```
if (myPathPointCounter == 0){
  myPointC = myPath.pathPoints[myPath.pathPoints.length — 1];
}
else{
  myPointC = myPath.pathPoints[(myPathPointCounter - 1);
}

// Запоминание параметров предыдущей точки
myAnchor = myPointC.anchor;
myX3 = myAnchor[0];
myY3 = myAnchor[1];

// Получение новых координат
myPoints = myAddPoints(myX1, myY1, myX2, myY2, ↵
  myX3, myY3, myOffset);

myNewX1 = myPoints[0];
myNewY1 = myPoints[1];
myNewX2 = myPoints[2];
myNewY2 = myPoints[3];

// Расчет положения новых точек в зависимости от выбранного
// пользователем типа кривой.
switch (myCornerType){
  case 0:
    // Скругление углов

    // Сначала — координаты
    myPoint = [[myNewX2, myNewY2], [myNewX2, myNewY2], ↵
      [myX1, myY1]];
    myPointArray.push(myPoint);
    myPoint = [[myNewX1, myNewY1], [myNewX1, myNewY1], ↵
      [myNewX1, myNewY1]];
    myPointArray.push(myPoint);
    break;
  case 1:
    // Втянутые углы
    myPoint = [[myNewX2, myNewY2], [myNewX2, myNewY2], ↵
      [(myNewX2 + myNewX1 - myX1), (myNewY2 + myNewY1 - myY1)]];
    myPointArray.push(myPoint);
    myPoint = [[myNewX1, myNewY1], [myNewX1, myNewY1], ↵
      [myNewX1, myNewY1]];
```

```
    myPointArray.push(myPoint);
    break;
  case 2:
    // Срезанные углы
    myPoint = [[myNewX2, myNewY2], [myNewX2, myNewY2], ↵
      [myNewX2, myNewY2]];
    myPointArray.push(myPoint);
    myPoint = [[myNewX1, myNewY1], [myNewX1, myNewY1], ↵
      [myNewX1, myNewY1]];
    myPointArray.push(myPoint);
    break;
  case 3:
    // Обрезанные углы
    myPoint = [[myNewX2, myNewY2], [myNewX2, myNewY2], ↵
      [myNewX2, myNewY2]];
    myPointArray.push(myPoint);
    myPoint = [[(myNewX2 + myNewX1 - myX1), ↵
      (myNewY2 + myNewY1 - myY1)],[(myNewX2 + myNewX1 - myX1), ↵
      (myNewY2 + myNewY1 - myY1)],[(myNewX2 + myNewX1 - myX1), ↵
      (myNewY2 + myNewY1 - myY1)]];
    myPointArray.push(myPoint);
    myPoint = [[myNewX1, myNewY1], [myNewX1, myNewY1], ↵
      [myNewX1, myNewY1]];
    myPointArray.push(myPoint);
    break;
  case 4:
    // Оригинальные углы.
    // Каждая угловая точка ставится на расстоянии 1/3
    // величины смещения, т. е. первая на 1/3,
    // вторая 2/3 для каждого угла.
    var myOneThird = 0.33;
    var myTwoThirds = 0.67
    var myPointZX = myNewX2 + myNewX1 - myX1;
    var myPointZY = myNewY2 + myNewY1 - myY1;
    var myTemp1X = (myX1 - myNewX2) * myTwoThirds;
    var myTemp1Y = (myY1 - myNewY2) * myTwoThirds;
    var myTemp2X = (myX1 - myNewX1) * myTwoThirds;
    var myTemp2Y = (myY1 - myNewY1) * myTwoThirds;
    var myPointDX = myPointZX + myOneThird * (myNewX1 - ↵
      myPointZX);
    var myPointDY = myPointZY + myOneThird * (myNewY1 - ↵
      myPointZY);
```

```
            var myPointEX = myPointZX + myOneThird * (myNewX2 - ↵
              myPointZX);
            var myPointEY = myPointZY + myOneThird * (myNewY2 - ↵
              myPointZY);
            var myPointFX = myPointDX + myTwoThirds * (myX1 - ↵
              myTemp1X - myPointDX);
            var myPointFY = myPointDY + myTwoThirds * (myY1 - ↵
              myTemp1Y - myPointDY);
            var myPointGX = myPointEX + myTwoThirds * (myX1 - ↵
              myTemp2X - myPointEX);
            var myPointGY = myPointEY + myTwoThirds * (myY1 - ↵
              myTemp2Y - myPointEY);
            var myPointHX = myPointZX + myTemp1X + myTemp2X;
            var myPointHY = myPointZY + myTemp1Y + myTemp2Y;

            myPoint = [[myNewX2, myNewY2],[myNewX2, myNewY2], ↵
              [myNewX2, myNewY2]];
            myPointArray.push(myPoint);
            myPoint = [[myPointEX, myPointEY],[myPointEX, ↵
              myPointEY],[myPointEX, myPointEY]];
            myPointArray.push(myPoint);
            myPoint = [[myPointGX, myPointGY],[myPointGX, ↵
              myPointGY],[myPointGX, myPointGY]];
            myPointArray.push(myPoint);
            myPoint = [[myPointHX, myPointHY],[myPointHX, ↵
              myPointHY],[myPointHX, myPointHY]];
            myPointArray.push(myPoint);
            myPoint = [[myPointFX, myPointFY],[myPointFX, ↵
              myPointFY],[myPointFX, myPointFY]];
            myPointArray.push(myPoint);
            myPoint = [[myPointDX, myPointDY],[myPointDX, ↵
              myPointDY],[myPointDX, myPointDY]];
            myPointArray.push(myPoint);

            myPoint = [[myNewX1, myNewY1],[myNewX1, myNewY1], ↵
              [myNewX1, myNewY1]];
            myPointArray.push(myPoint);
            break;
        }
      }
    }
    // Самая основная операция — создание кривой по имеющимся
    // координатам опорных точек и касательных
```

```
    myPath.entirePath = myPointArray;
  }
}

//
function myAddPoints(myX1, myY1, myX2, myY2, myX3, myY3, myOffset){
  var myXAdjust, myYAdjust, myNewX1, myNewY1, myNewX2, myNewY2, ↳
    myHypotenuse;
  myHypotenuse = Math.sqrt(Math.pow((myX1 - myX2),2) + ↳
    Math.pow((myY1 - myY2),2));
  if (myY1 != myY2) {
    // Коэффициенты, учитывающие пропорции сторон фигуры
    myXAdjust = ((myX1 - myX2) / myHypotenuse) * myOffset;
    myYAdjust = ((myY1 - myY2) / myHypotenuse) * myOffset;
    myNewX1 = myX1 - myXAdjust;
    myNewY1 = myY1 - myYAdjust;
  }
  else {
    myXAdjust = myOffset;
    myYAdjust = 0;
    if (myX1 < myX2) {
      myNewX1 = myX1 + myXAdjust;
      myNewY1 = myY1 + myYAdjust;
    }
    else{
      myNewX1 = myX1 - myXAdjust;
      myNewY1 = myY1 - myYAdjust;
    }
  }

  myHypotenuse = Math.sqrt(Math.pow((myX1 - myX3),2) + ↳
    Math.pow((myY1 - myY3),2));
  if (myY1 != myY3) {
    myXAdjust = ((myX1 - myX3) / myHypotenuse) * myOffset;
    myYAdjust = ((myY1 - myY3) / myHypotenuse) * myOffset;
    myNewX2 = myX1 - myXAdjust;
    myNewY2 = myY1 - myYAdjust;
  }
  else{
    myXAdjust = myOffset;
    myYAdjust = 0;
    if (myX1 < myX3) {
      myNewX2 = myX1 + myXAdjust;
```

```
      myNewY2 = myY1 + myYAdjust;
    }
    else{
      myNewX2 = myX1 - myXAdjust;
      myNewY2 = myY1 - myYAdjust;
    }
  }
  return [myNewX1, myNewY1, myNewX2, myNewY2];
}

function myPointTest(myPathPointCounter, myPath, myPattern){
  // Из-за ограничений InDesign при работе с кривыми мы должны
  // исключить из трансформаций первую и последнюю точки кривой
  if((myPath.pathType == PathType.openPath)&& ↳
    ((myPathPointCounter ==0)|| ↳
     (myPathPointCounter == myPath.pathPoints.length-1))){
    return false;
  }
  else{
    switch(myPattern){
      case "all points":
        return true;
      case "first point":
        if(myPathPointCounter == 0){
          return true;
        }
        else{
          return false;
        }
      case "last point":
        if(myPathPointCounter == myPath.pathPoints.length-1){
          return true;
        }
        else{
          return false;
        }
      case "second point":
        if(myPathPointCounter == 1){
          return true;
        }
        else{
          return false;
        }
```

```
      case "third point":
        if(myPathPointCounter == 2){
          return true;
        }
        else{
          return false;
        }
      case "fourth point":
        if(myPathPointCounter == 3){
          return true;
        }
        else{
          return false;
        }

      // Поскольку myPathPointCounter начинается с 0,
      // индексы всех четных точек будут делиться на 2 без остатка.
      // Поэтому мы можем использовать конструкцию % (mod) 2 != 0
      // для доступа к четным точкам.
      case "even points":
        if(myPathPointCounter % 2 != 0){
          return true;
        }
        else{
          return false;
        }
      case "odd points":
        if(myPathPointCounter % 2 == 0){
          return true;
        }
        else{
          return false;
        }
    }
  }
}
```

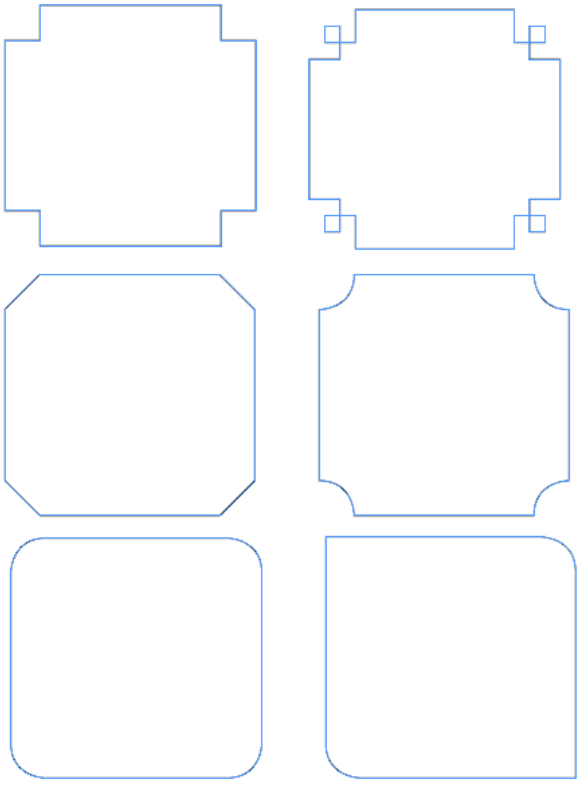
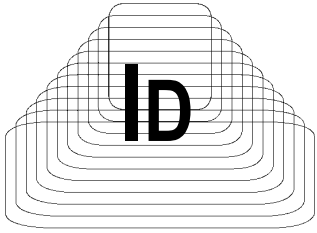


Рис. 10.2. Результаты работы скрипта

ГЛАВА 11



Межпрограммное взаимодействие в Creative Suite 2

11.1. Автоматическая проверка публикации

Конечный результат верстки — создание полностью готовой к печати публикации. В ней задействовано множество ресурсов: текст, растровые изображения, векторные иллюстрации, собственные объекты верстального пакета. Все это превратится в красочное издание, однако при одном условии: если в ней нет ошибок, иначе будут неприятности. Соответственно задача номер один — проверка каждого ресурса. Если публикация представляет собой более чем 200-страничный журнал с сотнями иллюстраций, вероятность пропуска ошибки достаточно велика, особенно с учетом различных непредвиденных обстоятельств, время от времени возникающих перед сдачей издания. Ведь над номером работают несколько человек, а роль человечекого фактора еще никто не отменял. Как обезопасить себя от возможных проблем при печати?

Рассмотрим пример, который является в своем роде квинтэссенцией книги, поскольку проверка макета на соблюдение техническим требованиям, принятым в препресс-процессе, — одна из наиболее эффективных областей использования скриптинга. В примере особый упор делается на проверку макетов рекламы, хотя никаких проблем с распространением его использования в обычной верстке нет. Ответ на вопрос, почему упор был сделан на проверку именно рекламы, прост — дизайнеры, собирающие издание, имеют достаточно времени на тщательную проверку своей собственной работы, а вот изготовители рекламы бывают разные. Кроме того, в глянцевом журнале общее число размещаемой рекламы — около 100 полос. Если бы макеты поступали в течение подготовки номера равномерно — каждый можно было бы тща-

тельно проверить, благо времени хватает, однако на практике все обстоит несколько иначе. Как правило, львиная доля рекламы поступает лишь в последнюю неделю перед сдачей номера, причем типична ситуация, когда в одном макете нужно переделать/доделать одно, в другом — другое, и т. д., исправленный вариант отправить на утверждение заказчику, при необходимости снова внести изменения... Макеты подаются в TIFF, EPS, PDF, AI. Хорошо еще, если сделаны в InDesign, а бывает что передается сборочный файл в XPress — в общем, скучать не приходится.

Поскольку в условиях цейтнота вероятность пропуска макета, который вызовет на печати проблемы, резко возрастает, родилась идея переложить предпечатную проверку максимально на "плечи" машины. Описываемый случай как нельзя лучше демонстрирует практическую пользу от скриптинга, поскольку компьютер беспристрастно и с огромной скоростью проверяет объект за объектом, не оставляя ошибкам ни шанса, в то время как при ручной проверке, во-первых, времени потребуется затратить значительно больше, а во-вторых, пресловутый человеческий фактор не позволяет гарантировать 100%-е отсутствие ошибок. Поскольку реклама — неотъемлемая часть каждого издания, данный вопрос представляет значительный интерес для множества людей, работа которых сродни работе диспетчера в аэропорту: вроде бы — чего проще: прием/передача файлов (бортов), однако последствия ошибки велики.

Конечно, в качестве альтернативного варианта можно было бы макет сохранять в формате PDF, после чего проверять хорошо известным Acrobat Preflight. Однако он имеет несколько недостатков. Во-первых, невысокая оперативность. Если ошибка обнаружена, нужно сначала определить, в каком именно файле она находится (как правило, реклама состоит из нескольких сборочных файлов), после чего загружать соответствующее приложение, отыскивать в этом файле проблемный элемент и принимать меры по исправлению ситуации. И если с редактированием EPS ситуация более-менее приемлемая, то с PDF работать гораздо сложнее — достаточно сказать, что, к примеру, окантовка превращается в отдельный объект с заливкой и т. п., в общем, работа не для слабонервных. Кроме того, как это всегда бывает, в некоторые макеты необходимо внести изменения уже после начала печати. Именно поэтому их стараются оставлять редактируемыми вплоть до самого последнего момента, в результате чего возможности Acrobat Preflight нам пригодятся слабо.

Более того, даже после исправления ошибки придется снова тратить время на создание PDF либо вручную переподставлять страницы в Acrobat, что также занимает время.

В данном примере описывается метод, которой активно используется в нескольких глянцевых журналах. Проверка публикации использует возможности всего ПО из коллекции Creative Suite и происходит в три этапа:

1. Проверка корректности родных объектов InDesign. Частичный анализ связанной графики: если она растровая, узнать ее свойства можно встроенными средствами верстального пакета. Если же графика векторная, то единственный выход из положения — задействовать Illustrator.
2. Проверка связанных EPS- и PDF-файлов в Illustrator.
3. Возврат в публикацию и завершающие шаги: сбор информации о проверенных файлах и, если все благополучно, создание файла предпросмотра для утверждения заказчику (PDF с низким разрешением) и вывод макета на печать. Если же найдена ошибка на любом из этапов, выдается сообщение с описанием ошибки и все дальнейшие действия прекращаются.

Разберем подробнее каждый из этапов.

11.2. Проверка самой верстки

На первом этапе проверяются объекты, созданные в самом InDesign.

Список вопросов для проверки в самой верстке таков.

- ❒ Толщина линий — не менее 0,25 пунктов.
- ❒ Отсутствие RGB-цветов.
- ❒ Проверка плотности цвета на обширных участках. Если на них используется всего одна составляющая (например, Black = 100%), то на печати обязательно возникнет полошение, связанное с неравномерностью прокраски бумаги. Решение состоит в использовании как минимум еще одного цвета, например, Cyan = 50% — в таком случае вероятность проявления полошения в два раза меньше, поскольку неравномерность в черном будет частично компенсироваться неравномерностью в голубом. Оптимально же использование всей триады — Cyan = 55%, Magenta = 35%, Yellow = 35%, что в сумме (Total Ink) дает 225% и полностью укладывается в допустимые пределы при ролевой печати.
- ❒ Наличие кроющего цвета (overprint).
- ❒ Количество красок у мелких деталей, в частности мелкого текста — во избежание возможного несовмещения не рекомендуется использование густой краски (более 2-х составляющих).
- ❒ Использование смесевых красок (Pantone) и spot-цветов.
- ❒ Удаление неиспользуемых образцов цвета (swatches).

- Наличие прозрачных объектов.
- Использование нестандартных режимов наложения (**Overlay**, **Multiply** и др.), чтобы избежать возможного растрирования нижележащего текста со всеми вытекающими из этого последствиями.
- Проверка связей (links): существуют ли файлы на своих местах, последние ли версии отображаются на экране.
- Отсутствие редактируемого текста (во избежание проблем на компьютере, где вся публикация собирается из глав).
- Проверка цветовых моделей для растровых изображений.
- Для них же — определение фактического разрешения (с учетом трансформаций над оригиналом).
- В принципе можно, кроме того, учитывать трансформации. Масштабирование, повороты растровой графики, добавить замену изображений с низким разрешением на высококачественные с сохранением обтравочных путей и т. д. — каждый, наверное, сможет добавить к списку то, что требуется именно ему.

Если проверка рассмотренных ранее вопросов реализуется достаточно просто, то в случае с векторной графикой одних только возможностей InDesign мало. Вспомним, что такое EPS — это Encapsulated PostScript File, т. е. самодостаточный файл, в котором может быть намешано абсолютно все, что угодно. Поэтому совершенно логично, что InDesign просто не в состоянии на "лету" дать исчерпывающую информацию о его содержимом. Для этого ему нужно задействовать совершенно иные механизмы, которыми обладают программы, понимающие такой формат. Как известно, EPS может быть сгенерирован разными программами, например, Photoshop (Photoshop EPS), а может и Illustrator, поэтому вначале нужно точно определить, в какой именно программе его проверять.

11.3. Bridge и его роль в Creative Suite

Среди редакторов, входящих в Creative Suite, имеется Bridge — один из компонентов программного комплекса, который, в первую очередь, служит для синхронизации цветовых настроек, а также является достаточно хорошо продуманным просмотрщиком изображений. В нашем же случае Bridge интересен в первую очередь тем, что позволяет организовать межпрограммное взаимодействие всех редакторов, входящих в пакет. Так, посредством его можно из InDesign вызывать Illustrator, в котором запустить другой скрипт, а затем снова вернуться в InDesign или же перейти в Photoshop.

К слову: с помощью Bridge достаточно просто организуется проверка растровых изображений, находящихся в EPS — например, если изображение повернуто или его масштаб отличается от 100%, рационально вызвать Photoshop (уже из Illustrator), провести в нем необходимые трансформации, а затем вернуться в Illustrator.

Таким образом, именно благодаря Bridge мы можем в скрипте, запущенном в InDesign, запускать скрипты, написанные для других программ, реализуя любую из необходимых цепочек:

- InDesign — Illustrator — InDesign;
- InDesign — Photoshop — InDesign;
- InDesign — Illustrator — Photoshop — InDesign.

Для активации Bridge для целей скриптинга используют конструкцию `new BridgeTalk` (кто занимался программированием под офисные пакеты в Windows, найдет очень много общего — например, аналогичным образом запускаются объекты ActiveX). Какому именно приложению отдать управление, задается в свойстве `target`, а содержимое скрипта должно быть записано в свойство `body`.

Разумеется, для реализации такого взаимодействия необходимо, чтобы Bridge был установлен. Если при работе на Macintosh такая ситуация типична, поскольку он является идеальным средством для просмотра каталогов изображений, то на платформе PC Bridge — достаточно редкий гость. Связано это с несколькими причинами. Во-первых, в своей непосредственной области — просмотр изображений. Под Windows существует масса аналогичных утилит: ACDSee, Total Commander, которым по тем или иным причинам пользователи традиционно отдают предпочтение. Во-вторых, в *запущенном* состоянии Bridge достаточно требователен к системным ресурсам — во всяком случае с ОЗУ менее 1 Гбайт на компьютере становится работать некомфортно.

Я раньше специально подчеркнул — в запущенном состоянии, поскольку в нашем случае необходимости запускать весь Bridge нет (ведь мы используем всего лишь один его модуль, отвечающий за межпрограммное взаимодействие), а потому затраты памяти минимальны и совершенно не ощущаются даже при памяти 256 Мбайт.

Уместо подчеркнуть еще одну особенность скриптинга — возможность открытия документов без отображения их на экране (так называемый *фоновый режим*). Очевидное преимущество этого — экономия времени на отрисовку изображения, которое напрямую зависит от сложности макета и от установленного разрешения экрана. Проведенные эксперименты дали двух-, а иногда и трехкратный выигрыш во времени (в зависимости от характера выполняемых задач) по сравнению с типичным способом, при котором изображение выводится на экран.

Итак, если в процессе проверки публикации обнаруживается наличие файлов с векторным содержимым, переходим ко второму этапу: через Bridge переключаемся на Illustrator, в котором проводим свои проверки. Какими они могут быть?

11.4. Проверки в Illustrator

Фактически все проверки, проводимые в InDesign, должны быть повторены в Illustrator. Как вариант, можно добавить еще отслеживание трансформаций у внедренной растровой графики и, при необходимости, переключение в Photoshop. Проверка смесевых красок обязательна, поскольку, если они присутствуют в векторной графике, в InDesign удалить их не получится и при печати возможны эксцессы.

Файл за файлом мы просматриваем векторные изображения в Illustrator. Хорошо, если ни в одном из использующихся в публикации проблем не найдено. Но предположим, что в каком-то из них возникла ошибка или же ситуация, требующая повышенного внимания и вмешательства человека. В таком случае скрипт должен остановиться, специалист — оценить ситуацию, внести, при необходимости, коррективы и продолжить проверку. Чтобы скрипт знал, какие файлы осталось проверить, нужно заранее составить список всех проверяемых файлов и по мере проверки его сокращать. В таком случае при повторном запуске скрипт обработку уже продолжит, а не начнет заново, что даст экономию времени.

Список сформируем в InDesign и запишем в виде файла. Для универсальности его можно разместить вне папки, в которой собран макет с рекламой — в результате независимо от того, над чем работаем, пути к проверяемым файлам всегда будут находиться в одном и том же известном месте.

Таким образом, самое первое, что скрипт в Illustrator должен сделать, — это прочитать содержимое файла $$links.txt (путь к каждому изображению в нем записан в отдельной строке) и лишь потом начать проверку этих изображений по очереди.

Наконец, на данной стадии все родные объекты InDesign, а также использованные внешние файлы проверены. Осталась финальная часть — возврат в публикацию и выполнение завершающих шагов. Для упрощения процесса можно вручную создать предустановки как для формирования PDF (**File | Adobe PDF Presets | Define**), так и для печати (**File | Print Presets | Define**), а в скрипте использовать только ссылки на них, что существенно сократит размер скрипта.

Кроме того, технологический процесс, используемый во многих глянцевых журналах, предусматривает копирование файлов с рекламой в отдельную

папку на сервере. Поскольку единовременное копирование по активно используемой сети всей папки может растянуться на несколько часов (размер рекламы часто превышает 10 Гбайт), логично переписывать файлы на диск по мере ее утверждения — таким образом, к утверждению последнего макета вся реклама будет находиться уже на сервере.

Описательную часть считаем завершенной, приступим непосредственно к скрипту, вернее, скриптам. Их у нас будет три: Preflight_Check_ID.jsx (первая часть проверки, выполняется в InDesign), при наличии векторной графики завершается запуском Preflight_Check_IL.jsx (проверка EPS/PDF из Illustrator), который, в свою очередь, вызывает продолжение проверки (Preflight_Check_ID_final.jsx, заключительная часть, InDesign). Если векторной графики нет, то Preflight_Check_ID.jsx сразу же вызывает Preflight_Check_ID_final.jsx.

11.5. Скрипты

11.5.1. Этап 1. Проверка в InDesign

Первый этап можно разбить на несколько шагов. Самый первый — вывод диалогового окна, в котором пользователю предлагается задать требуемые установки. Среди них — правила, согласно которым проводить проверку (записаны в `myColumn1` и `myColumn2`), а также некоторые дополнительные параметры (`myColumn3` и `myColumn4`): флаг, позволяющий включить все проверки сразу, журнал, а также какое действие выполнять после предпечатной проверки: создать PDF для пересылки на утверждение заказчику (если макет только поступил) либо уже утвержденный вариант отправить на печать для получения "белки" выпускающему редактору, после чего вся сборка копируется на сервер, где издание полностью собирается.

Поскольку Creative Suite 2 не позволяет полностью прозрачно запускать сценарии в других приложениях (это выражается в том, что переменные, доступные в одном скрипте, недоступны для другого, а нам это нужно, поскольку параметры задаются в самом первом скрипте, а остальные действия, кроме предпечатной подготовки, выполняются в последнем), пойдем на маленькую хитрость и сохраним введенные пользователем данные в виде файла. В таком случае они останутся доступными для любого скрипта и в любое время, что нам и нужно. Данная операция записана в виде функции `writeINI()` (листинг 11.1). В переменной `iniFilePath` хранится путь к создаваемому в технических целях файлу настроек $$ini.txt, в который записываем название журнала и тип конечного результата: создание PDF либо печать и копирование на сервер. Еще один создаваемый в технических целях файл — `linksF("/MyFolder/$$links.txt")`, куда будем записывать пути к файлам иллюстраций, на которые есть ссылки.

Листинг 11.1. Сохранение настроек в виде файла

```
function writeINI(){
  iniFilePath = "/MyFolder/$$ini.txt"
  iniFile = File(iniFile)
  f1.open('w')
  iniFile.writeln(currMagazine)
  iniFile.writeln(lowResPDF)
  iniFile.close()
}
```

Шаг первый

Листинг 11.2. Вывод окна диалога и считывание из него значений (рис. 11.1)

```
#target 'indesign'
var aD = app.documents[0];
myDialog = app.dialogs.add()
myDialog.name = "Preflight Checking"
myColumn1= myDialog.dialogColumns.add()

with(myColumn1.staticTexts){
  add({staticLabel:"Images with eff. resolution below"})
  add({staticLabel:"SpotColors"})
  add({staticLabel:"Overprint"})
  add({staticLabel:"Hair Lines"})
  add({staticLabel:"Colored small text"})
  add({staticLabel:"Large areas with one colorant"})
  add({staticLabel:"RGB images"})
  add({staticLabel:"JPEG images"})
  add({staticLabel:"Delete unused swatches"})
  add({staticLabel:"Transparent items"})
  add({staticLabel:"Blending modes other than Normal"})
  add({staticLabel:"Editable text"})
  add({staticLabel:"Ghost items"})
}

myColumn2= myDialog.dialogColumns.add()
myMinimalResolution = myColumn2.integerEditboxes.add({editValue:250})
with(myColumn2.checkboxControls){
  mySpotColor = add({checkedState: true})
  myOverprint = add({checkedState: true})
  myHairLine = add({checkedState: true})
```

```
  myColoredSmallSizeText = add({checkedState: true})
  myLargeAreas = add({checkedState: true})
  myRGBimage = add({checkedState: true})
  myJPG = add({checkedState: true})
  myDeleteUnusedSwathes = add({checkedState: true})
  myTransparency = add({checkedState: true})
  myBlending = add({checkedState: true})
  myEditableText = add({checkedState: true})
  myGhosts = add({checkedState: true})
}

myColumn3= myDialog.dialogColumns.add()
with(myColumn3.staticTexts){
  add({staticLabel:"Check all issues"});
  add({staticLabel:"Magazine"});
  add({staticLabel:"Create PDF?"});
  add({staticLabel:"Print"});
}

myColumn4= myDialog.dialogColumns.add();
with(myColumn4){
  checkAll = checkboxControls.add({checkedState: true});
  magazine = dropdowns.add({stringList:["Magazine1", "Magazine2", ↳
    "Magazine3"], selectedIndex:0});
  lowResPDF = checkboxControls.add({checkedState: false});
  doPrint = checkboxControls.add({checkedState: true});
}
if(myDialog.show()){
currMagazine=(magazine.selectedIndex==0) ? true: false;
    lowResPDF =(lowResPDF.checkedState==false) ? false : true;
    doPrint = (doPrint.checkedState==true) ? true: false;
if(checkAll){mySpotColor = true;
myOverprint = true;
myHairLine = true;
myColoredSmall= true;
myLargeAreas = true;
myRGBimage = true;
myJPG = true;
myDeleteUnused = true;
myTransparency = true;
myBlending = true;
myEditableText = true;
myGhosts = true;  }
}
```

```
myDialog.destroy();

linksF = "/MyFolder/$$links.txt";

writeINI();
```

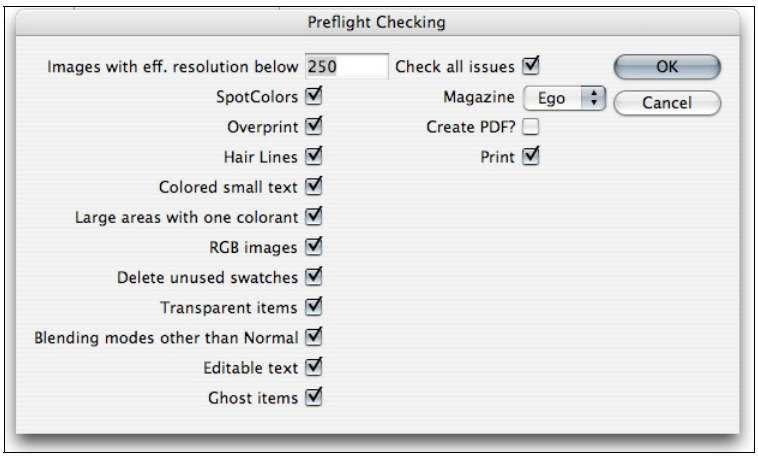


Рис. 11.1. Диалоговое окно не блещет дизайном, но в данном случае дизайну мы уделяем минимум внимания

Итак, первый шаг сделан, переходим к следующему, в котором происходит запуск основного модуля `preflightIDD()`, но перед этим выполняются две проверки. Первая: в публикации на момент запуска не должно быть выделенных объектов, поскольку объекты, требующие дополнительного внимания, скрипт помечает выделением, а без этой проверки может возникнуть недоразумение. Вторая проверка связана с вопросами оптимизации — при повторном запуске скрипта (в случае, если в связанных файлах был обнаружен вопрос, требующий уточнения) нет смысла повторять проверку самой публикации — ведь она уже была выполнена ранее. Признак запуска публикации на проверку — отсутствие содержимого в файле со связями `File(linksF)` (т. е. все связи проверены, что возможно лишь в случае совершенно новой публикации).

В скрипте выполняется последовательная процедура проверки: сначала изучается публикация. Если в ней потенциально опасные объекты не выявлены, начинают рассматриваться связанные векторные файлы (функция `switchToILL()`). В случае обнаружения проблемных участков в публикации дальнейшая проверка останавливается до их ликвидации.

В случае, если в векторных изображениях ошибок нет, но в публикации обнаружены растровые изображения с разрешением менее заданного или выше

300 ppi, ссылки на них записываются в файл linksPSDFile (по полной аналогии, как это выполнялось для векторных иллюстраций), после чего управление передается Photoshop, который этот список файлов обрабатывает. Последний шаг — возврат в InDesign.

Шаг второй

Листинг 11.3. Предпечатная проверка публикации и, при отсутствии ошибок и наличии ссылок на векторные изображения, переход в Illustrator

```
if(aD.selection.length>0)
  alert("Make sure there aren't selected objects in the document!");
else {
  err = false;
  preflightIDD();

  if (err){
    alert("Due to potential problems in IDD file all other operations
terminated!");
    alert("There are some issues:\r" + errArray.join('\r'));
  }else{
    if(File(linksEPSFile).length>0){
      switchTo('AI', 'illustrator')
    }else if(File(linksPSDFile).length>0){
      switchTo('PSD', 'photoshop');
    }else
      returnToIDD()}
```

Собственно проверка публикации выполняется так. Сначала займемся текстом (листинг 11.4).

Листинг 11.4. Начинаем просматривать текст

```
tF = aD.textFrames;
if (tF.characters.length>0){
  for (i=0; i<tF.length; i++){
    currTF  = tF[i];
```

Просматриваем по очереди все символы и определяем цвет их заполнения. Если кегль небольшой, а содержимое красок более 250, высока вероятность несовмещения при печати. Значение цвета объекта (`colorValue`) хранится в виде массива с 4 значениями — для `cyan`, `magnta`, `yellow` и `black` соответственно (листинг 11.5).

Листинг 11.5. Проверка текста

```
for (var j=0; j<currTF.characters.length; j++){
  z = currTF.characters[j].fillColor.colorValue;
  if (currTF.characters[j].pointSize<9 && ↳
     (z[0]+z[1]+z[2]+z[3] > 250))
    LatchObj("Rich color of small-size text", currTF);
```

Функция `LatchObj()` выполняет вспомогательную роль и отвечает за информирование о подозрительных свойствах, которые выдают в конце работы скрипта. Первый ее параметр — содержимое сообщения, например, "С надпечатью!". Оно же дублируется в ярлык объекта (в InDesign — **Window | Automation | Script Label**), помогая определить, какая же именно проблема у объекта ("С надпечатью"). Ярлык представляет собой свойство объекта (`label`), куда можно записывать все, что угодно, в нашем случае мы используем его для хранения типа ошибки объекта.

Второй параметр функции — объект, который нужно выделить. Делается это с той единственной целью, что выделение проблематичных объектов очень помогает при локализации ошибки, особенно когда в публикации много объектов. Если параметр равен `false`, выделение не выполняется — например, если речь идет об отдельных символах текста или образцах цвета палитры **Swatches**. Наконец, сообщение об ошибке сохраняется в массиве `ErrArray`, чтобы по окончании проверки выдать консолидированный отчет (листинг 11.6).

Листинг 11.6. Функция обработки проблемного объекта

```
function LatchObj(label_msg, Obj){
  if(Obj!=false) {
    Obj.label = label_msg;
    Obj.select(SelectionOptions.addTo);
  }
  ErrArray.push(label_msg)
}
```

Кстати говоря, можно ярлыки использовать и более широко. Как вы помните, вначале скрипта мы записывали установки пользователя в файл, чтобы финальный скрипт их смог прочитать, но можно было использовать для этого тот же механизм labels. Дело в том, что InDesign поддерживает два типа ярлыков: первый, содержимое которого отображается в палитре **Script Label**, и второй, содержимое которого на экран не выводится: при этом ярлык рассматривается как хранилище данных, куда можно записывать все, что угодно,

в виде пары значений `key:value`, где `key` — любой параметр (ключ), а `value` — его значение. Применительно к нашему случаю параметром будут переменные, а значениями — собственно введенные пользователем значения. Интересно то, что такой ярлык могут иметь объекты, которые обычным способом выделены быть не могут, например, документ, слой и т. п. Поскольку механизмы при обращении к разным типам labels также разные, для второго способа в InDesign используются методы `insertLabel(key,value)` и `extractLabel(key)`. Поскольку Labels хранятся в документе, они будут доступны в финальном скрипте, т. е. можно отказаться от создания файла $$ini.txt , вместо этого записав:

```
aD.insertLabel("SpotColor", mySpotColor.checkedState.toString())
aD.insertLabel("Overprint", myOverprint.checkedState.toString())
aD.insertLabel("HairLine", myHairLine.checkedState.toString())
aD.insertLabel("ColoredText", myColoredSmall.checkedState.toString())
aD.insertLabel("LargeArea", myLargeAreas.checkedState.toString())
aD.insertLabel("RGB", myRGBimage.checkedState.toString())
aD.insertLabel("JPG", myJPG.checkedState.toString())
aD.insertLabel("UnusedSwatches", myDeleteUnused.checkedState.toString())
aD.insertLabel("Transparency", myTransparency.checkedState.toString())
aD.insertLabel("Blending", myBlending.checkedState.toString())
aD.insertLabel("EditableText", myEditableText.checkedState.toString())
aD.insertLabel("Ghosts", myGhosts.checkedState.toString())
```

Конструкция `toString()` необходима для преообразования значения (в нашем случае им будет либо `true`, либо `false`) в текстовый тип, поскольку в ярлыке, независимо от его типа, может храниться исключительно текст.

Итак, возвращаемся к скрипту и продолжаем проверку свойств текстовых объектов: ищем наличие кроющего цвета и в случае нахождения устанавливаем соответствующий признак:

```
if(currTF.characters[j].overprintFill)
  LatchObj("Item with Fill overprint", currTF);
```

С текстом закончили. Переходим к остальным элементам публикации. Поскольку реклама может занимать целый разворот, работаем с целым разворотом:

```
for(i=0;i<aD.spreads[0].rectangles.length;i++){
  rect = aD.spreads[0].rectangles[i];
```

Проверяем отсутствие надпечати у белых объектов — иначе они на печати просто пропадут. Учитываем, что если объект не имеет никакого цвета, попытка считать его цвет (`fillColor.colorValue`) приведет к ошибке. Отстутст-

вие у объекта цвета в InDesign соответствует значению `Swatch` его свойства `fillColor` (листинг 11.7).

Листинг 11.7. Проверка установленной надпечати

```
if(o && rect.fillColor.constructor.name != 'Swatch'){
  fC = rect.fillColor.colorValue;
  if (fC.toString() != "0,0,0,0" && rect.overprintFill)
    LatchObj("Fill Overprint", rect);
```

Заодно проверям, чтобы обширные участки публикации (превышающие площадь квадрата со стороной 5 см) имели более чем одну цветовую составляющую — это обезопасит нас от полошения на печати (неравномерности прокраса). В случае, если используется только черный, заменяем его на черный густой с компонентами `C=55%`, `M=35%`, `Y=35%`, `K=100%` (голубого больше для корректного обеспечения уровня серого) — листинг 11.8.

Листинг 11.8. Проверка цвета заливки у обширных участков страницы

```
w = rect.geometricBounds[3] - rect.geometricBounds[1];
h = rect.geometricBounds[2] - rect.geometricBounds[0];
if((w*h > 2500) && fC[0]<10 && fC[1]<10 && fC[2]<10 && fC[3]>90)
  rect.fillColor.colorValue = [55,35,35,100];
```

Тут же проверяем наличие "мусорных" объектов — зачастую в процессе макетирования создаются пустые рамки, которые потом иногда забывают удалить. В случае, если вы используете такие элементы, как выталкиватели текста под иллюстрациями (вместо использования обтравочных контуров), эту операцию нужно исключить.

```
if(gh && rect.strokeColor.constructor.name != 'Swatch' && ↳
    rect.strokeColor.colorValue.toString() == "0,0,0,100")
  LatchObj("Maybe ghost item", rect)
```

По ходу дела удаляем неиспользуемые образцы цвета. Помним о том, что любые операции удаления/добавления приводят к изменению их порядковых номеров (индексов), поэтому удаление проводим, начиная с самого последнего объекта. Если воспользоваться традиционным перебором

```
for (i=0; i<aD.unusedSwatches.length; i++)
```

мы пропустим некоторые объекты. Так, при удалении самого верхнего объекта (с индексом 0) самым верхним станет следующий объект (после удаления он получит индекс 0). В конце цикла счетчик `i` увеличится на единицу и укажет на объект с индексом 1 (`unusedSwatches[1]`), т. е. на объект, который из-

начально был третьим, в результате второй окажется пропущенным. Чтобы этого не было, начинаем с "хвоста":

```
for (i=aD.unusedSwatches.length-1; i>-1; i-){
  aD.unusedSwatches[i].remove();
}
```

Следующий вопрос — тип используемых красок: они должны быть цветоделимыми (process color). В случае неудачи перевода их в CMYK (это произойдет при попытке изменения цвета, содержащегося в присоединеннном EPS-файле), используем уже известную функцию `LatchObj()`, только в данном случае второй параметр (что выделять) установим в `false`, поскольку в InDesign выделять можно только физические объекты (листинг 11.9).

Листинг 11.9. Проверка используемых красок

```
if(mySpotColor) {
  for (i=0; i<aD.inks.length; i++){
    if (!aD.inks[i].isProcessInk){
      if (!aD.inks[i].convertToProcess)
        LatchObj("Spot colors", false);
    }
  }
}
```

Настала очередь иллюстраций — как растровых, так и векторных. В первом случае (тип `Image`) мы непосредственно в InDesign сможем определить их основные параметры — цветовую модель и разрешение. Как известно, иллюстрации существуют исключительно внутри собственных объектов InDesign — будь то прямоугольники, овалы или что-либо еще. Свойство `effectivePpi` определяет реальное разрешение растровой графики (с учетом всех проведенных трансформаций). Если же они проведены без сохранения пропорций, разрешения по горизонтали и вертикали будут разными, поэтому `effectivePpi` представляет собой массив из двух значений (разрешение по обоим направлениям). Пороговое значение `myMinimalResolution` выбрано равным 250 ppi, что позволяет оставлять сюжетно интересные изображения, даже не обладающими идеальным качеством.

В случае, если разрешение картинки выше 300 ppi, путь к ней записывается в массив `ResolutionArray` для последующей записи в файл, который будет обрабатывать Photoshop (листинг 11.10).

Один нюанс связан с выделением изображений. Дело в том, что в отличие от собственных векторных объектов, никакие другие InDesign выделять не по-

зволяет (в том числе изображения), поэтому нам придется обращаться к их родителям (rectangle).

Листинг 11.10. Проверка связанной графики

```
EPS_flag = false
for(i=0; i<aD.allGraphics.length; i++){
  var myGraphic = aD.allGraphics[i];
  if(myGraphic.name=='Image'){
    if(myGraphic.space!="CMYK")
      LatchObj("NOT CMYK", myGraphic.parent);
    if (myGraphic.effectivePpi[0]<myMinimalResolution ||
myGraphic.effectivePpi[1]<myMinimalResolution)
      LatchObj("LOW RESOLUTION", myGraphic.parent);
if ((myGraphic.effectivePpi[0]>300 && myGraphic.effectivePpi[1]>300))
       ResolutionArray.push(File(myGraphic.itemLink.filePath))
  }
```

Далее идут редко используемые каверзные "финты", но, тем не менее, об их присутствии также лучше знать (листинг 11.11).

Листинг 11.11. Проверка экзотики

```
  if(myGraphic.blendMode!=BlendMode.NORMAL)
    LatchObj("NOT NORMAL BLENDING MODE", myGraphic.parent);
  if (myGraphic.opacity!=100)
    LatchObj("TRANSPARENCY", myGraphic.parent);
```

Шаг третий

Наконец, самое интересное — проверка иллюстраций в форматах EPS и PDF. Если таковые в публикации имеются (EPS_flag=true), устанавливаем соответствующий признак (нужен запуск Illustrator) и записываем все пути к ним в текстовый файл. Параллельно проверяем содержимое массива ErrArray, куда на протяжении работы скрипта собиралась информация о всех потенциально опасных моментах публикации, и выдаем ее в виде предупреждения. Если массив ErrArray не пуст, значит, в публикации существуют подозрительные элементы, и все дальнейшие операции прекращаются. Если же массив пуст, значит, все нормально, можно спокойно продолжать.

Один тонкий момент связан с обработкой объектов, вставленных в InDesign через системный буфер. Дело в том, что при определенных условиях (например, фрагмент слишком сложный, чтобы его оставлять в виде кривой) InDesign автоматически конвертирует объект в формат EPS и в таком виде

помещает его в публикацию. При этом возникает ситуация, когда, с одной стороны, формат файла действительно EPS, а с другой — он не имеет ссылки на файл на диске (`myGraphic.itemLink==null`), что нам также приходится учитывать. Наконец, запись ссылок к внешним иллюстрациям в соответствующие файлы (листинг 11.12).

Листинг 11.12. Проверка связанных иллюстраций

```
if (myGraphic.constructor.name=='EPS'|| ↳
    myGraphic.constructor.name=='PDF'){
  linkPath = decodeURI(File(myGraphic.itemLink.filePath));
  if(myGraphic.itemLink==null){
    alert("Some vector graphics were pasted from Clipboard")}
    EPS_Storage.push(linkPath);
  }else EPSArray.push(linkPath);
}
// Запись в файлы
if(EPSArray.length>0 && (File(linksEPSFile).length>0)
  writeLinks(EPSArray, linksEPSFile);
if(ResolutionArray.length>0 & File(linksPSDFile).length>0)
  writeLinks(ResolutionArray, linksPSDFile);}

function writeLinks(arrName, fileName){
  if(arrName.length>0 && (!File(fileName).exists ||
     File(fileName).length==0)) {
    txtFile = new File(fileName);
    txtFile.open('w');
    txtFile.write(arrName.join('\r'));
    txtFile.close();
    delete txtFile;
  }
}
```

Итак, оба шага, составляющие первый этап проверки, завершены. На текущий момент выполнены следующие задачи:

- проведена предпечатная проверка сборочного файла публикации;
- проверены все задействованные растровые изображения;
- пути к векторным иллюстрациям, использованным в публикации, записаны в файле /myFolder/$$links.txt;
- подготовлена база для выполнения следующих этапов;
- управление передано Illustrator, который проверит векторные иллюстрации.

На данный момент первая часть фактически завершена, осталось совсем немного — проверить признак наличия векторной графики и при необходимости запустить в Illustrator другой скрипт, который прочитает содержимое созданного файла отчета и по очереди запустит все перечисленные в нем иллюстрации на аналогичную проверку, но только с учетом иной объектной модели. Несмотря на то, что книга посвящена программированию в InDesign, некоторые вопросы программирования в другом редакторе пакета Creative Suite в необходимом для решения конкретной задачи минимуме также освещены.

Ранее выполненная проверка (`File(linksF).length==0`) необходима для того, чтобы InDesign знал, нужно ли в данный момент записывать новую информацию о связях в файл отчета. В самом деле, если в каком-либо проверяемом файле нашлась ошибка, скрипт безоговорочно останавливается. При последующем запуске скрипта InDesign уже не должен записывать пути в файл, поскольку иначе ему придется снова проводить проверку тех файлов, которые ранее уже были проверены, что не рационально.

Будем считать, что скрипт находится в папке myFolder — требований к его месторасположению не накладывается никаких, поскольку он не вызывается через операцию в меню **File | Scripts**. Напомню, для возможности запуска скриптов в Illustrator через операцию **Scripts** они должны находиться на Macintosh в папке /Volumes/VolumeName/Applications/Adobe Illustrator CS2/Presets.localized/Scripts.localized/, в Windows — в /Program Files/Adobe Illustrator CS2/Presets/Scripts/.

Итак, используем Bridge, задаем приложение, указываем какой скрипт использовать. Запуск выполняется методом `send()`. В свойстве `body` указывается текст скрипта, но поскольку он занимает много строк, проще прочитать содержимое файла скрипта (IL_ScriptContent) и передать его в Bridge в виде одной строки (листинг 11.13).

Листинг 11.13. Запуск скрипта обработки векторных иллюстраций

```
function switchToILL(){
  f = File('/myFolder/Preflight_Check_IL.jsx")
  f.open('r')
  scriptContent = f.read();
  f.close();

  bt = new BridgeTalk;
  bt.target = 'illustrator';
  bt.body = scriptContent;
  bt.send();
}
```

Вот теперь первый этап завершен полностью.

11.5.2. Этап 2. Проверка векторных иллюстраций в Illustrator

Перейдем к предпечатной проверке средствами Illustrator (скрипт Preflight_Check_IL.jsx). Собственно говоря, ничего сложного в этом нет, его объектная модель значительно проще, чем у InDesign, поэтому проблем не должно возникнуть никаких.

Вначале несколько подготовительных операций: открываем файл с путями для редактирования (поскольку каждый раз после успешной проверки документа в Illustrator будем изменять его содержимое), считываем первую строку и передаем ее как параметр методу `open()`, что приведет к открытию заданного файла (листинг 11.14).

Листинг 11.14. Открытие файла с путями

```
linksFile = File('myFolder/$$links.txt')
do{
  linksFile.open('r')
  myFilePath = linksFile.readln()
  otherLines = linksFile.read()
  linksFile.close()
  app.open(myFilePath)
```

В случае, если в предыдущей сессии в файле была обнаружена и исправлена ошибка, но по какой-то причине документ не был закрыт (а значит, гарантированно сохранен), возможна коллизия. Чтобы быть уверенным в том, что изменения были сохранены, мы принудительно закрываем файл. Если он сохранен не был, Illustrator предложит сделать это, что нам и нужно.

```
if(app.documents.length>0)
  app.documents[0].close()
```

Отключаем отображение диалоговых окон (часто Illustrator выдает предупреждения, которые в нашем случае не принципиальны) и открываем первый файл из списка в файле $$links.txt, после чего сразу же выдачу сообщений разрешаем — а как же иначе мы узнаем о ошибках, если их обнаружит скрипт (листинг 11.15)?

Листинг 11.15. Открытие файла из списка в файле $$links.txt

```
  app.userInteractionLevel= UserInteractionLevel.DONTDISPLAYALERTS;
  app.open(File(myFilePath))
  app.userInteractionLevel= UserInteractionLevel.DISPLAYALERTS;
  var aD = app.documents[0];
```

Далее идет собственно проверка текущего документа (листинг 11.16). После успешного ее прохождения иллюстрацией (`err==false`) из файла со связями удаляем самую первую строку. Это происходит после успешной проверки каждой иллюстрации. Удаление реализовано как перезапись текущего содержимого файла без первой строки (`otherLines`). В результате, если ошибок не возникло, в конце проверки содержимое файла станет пустым (`File(linksF).length==0`).

Листинг 11.16. Проверка текущего документа

```
  AI_preflight();
  if(!err){
    f.open('w')
    f.write(otherLines)
    f.close()
    aD.close();
  }
} while (File(linksF).length!=0 && !err)
```

В отсутствие ошибок последний шаг на стороне Illustrator — передача управления Bridge для возврата в InDesign (листинг 11.17).

Листинг 11.17. Передача управления Bridge для возврата в InDesign

```
if (!err) {
  f = File('myFolder/Preflight_Check_ID_final.jsx')
  f.open('r')
  scriptContent = f.read();
  f.close();

  bt = new BridgeTalk;
  bt.target = 'indesign';
  bt.body = scriptContent;
  bt.send();
}
```

Общая картина достаточно проста, рассмотрим собственно проверку, выполняемую в Illustrator (листинг 11.18).

Листинг 11.18. Проверка в Illustrator

```
function AI_preflight(){
  // Если остался редактируемый текст
  if (aD.textFrames.length>0){
```

```
  for (var i=0; i<aD.textFrames.length; i++){
    aD.textFrames[i].selected=true;
    errObj.et = 1;
  }
}

for(i=0; i<aD.pathItems.length; i++){
  var pgI = aD.pathItems[i];

  // Проверяем наличие надпечати у заливки и окантовки
  if ((pgI.filled  && pgI.fillOverprint) || ↳
      (pgI.stroked && pgI.strokeOverprint))  {
    errObj.op = 1;
    pgI.selected = true;
  }

  // Проверяем режимы наложения — возможно, понадобится
  // сведение (flattening)
  if(pgI.blendingMode!=BlendModes.NORMAL){
    errObj.bm=1;
    pgI.selected = true;
  }

  // Аналогично - для прозрачности
  if (pgI.opacity!=100) {
    errObj.o=1;
    pgI.selected = true;
  }

  // Проверка минимальной толщины окантовки
  if (pgI.stroked && pgI.strokeWidth<0.25){
    errObj.sw= 1;
    pgI.selected = true
  }

  // Наличие заказных цветов. Одной проверки typename =="SpotColor"
  // мало, поскольку стандартная триада также воспринимается как
  // SpotColor (это логично, ведь каждая из них генерирует свою
  // плашку), поэтому проверям еще и тип заказного цвета
  if(pgI.fillColor.typename =="SpotColor"){
    if(pgI.fillColor.spot.colorType!=ColorModel.PROCESS){
    errObj.spotC = 1;
    pgI.selected = true;
  }
}
```

```
  // Проверка окантовки с густым черным
  // (чтобы не было заметно возможное несовмещение)
  var fC = pgI.strokeColor;
  if(fC.black>75){
    if (fC.cyan>50 && fC.magenta>50 && fC.yellow>50){
      errObj.ps=1;
      pgI.selected=true;
    }
  }
}

// Проверка растровых объектов.
// Все проверки уже хорошо известны
for(i=0; i<aD.rasterItems.length; i++){
  var rI = aD.rasterItems[i];
  if (rI.blendingMode!=BlendModes.NORMAL){
    errObj.bm=1;
    rI.selected = true;
  }
  if (rI.opacity!=100) {
    errObj.o=1;
    rI.selected = true;
  }
  if (!rI.embedded){
    errObj.em = 1;
    rI.seleted = true;
  }

  // Проверка цветовой модели
  if(rI.imageColorSpace!=ImageColorSpace.CMYK){
    errObj.iRGB=1;
    rI.selected = true;
  }

  // Проверка разрешения. Информация хранится в свойстве matrix,
  // причем стандартным для экрана 72 dpi соответствует 1.
  // Мы устанавливаем предел - 250 точек, что сооответствует
  // значению 0.29
  if ((rI.matrix.mValueA > 0.29) || (rI.matrix.mValueD ↳> 0.29)){
    errObj.res = 1;
    rI.selected = true;
  }
}
```

```
// Если есть непечатаемые, но видимые слои с объектами
for (i=0; i<aD.layers.length; i++){
  var l= aD.layers[i];
  if (!l.printable && l.visible && l.pageItems.length>0)
    errObj.pv = 1;
  }

  // Если файл представляет собой матрешку -
  // в нем есть встроенные EPS
  for (i=0; i<aD.placedItems.length; i++){
    aD.placedItems[i].selected = true;
    errObj.p =1;
  }

  // Проверка заказных цветов, не использующихся в объектах,
  // но присутствующих в образцах цветов (swatches)
  if(aD.inkList.length>4)
    errObj.spotC = 1;

  // Проверка цветовой модели документа
  if(app.activeDocument.documentColorSpace!=DocumentColorSpace.CMYK)
    errObj.notCMYK=1;
}
```

И вот уже второй этап полностью завершен. Наступает очередь следующего, заверщающего этапа.

11.5.3. Этап 3. Конечные штрихи

Собственно говоря, третий этап — самый простой: на основании настроек, заданных в диалоговом окне на первом этапе, происходят создание PDF, печать и копирование на сервер.

К этим шагам следует добавить еще один, необходимость в котором была продиктована использованием скрипта в реальной работе. Часто рекламу, идущую на разворот, размещают на втором и третьем листах документа, очевидно не зная, как можно сделать публикацию всего из двух страниц, составляющих разворот. Нам приходится выполнять это за них (листинг 11.19).

Листинг 11.19. Удаление первой страницы в 3-страничных рекламных макетах

```
if(aD.pages>2){
  aD.documentPreference.allowPageShuffle = false;
  aD.pages[0].remove();
}
```

Остальные действия — в листинге 11.20.

Листинг 11.20. Финальные шаги скрипта

```
resF = File("C/$$ini.txt");
resF.open('r');
  EgoM = resF.readln();
  HR = resF.readln();
  pageSize = resF.readln();
  toPrint = resF.writeln();
resF.close();

exportToPDF();

if (HR)  postToServer();
alert(copiedFilesArr.toString() + "\rwere successfully copied");
```

Разберем шаги подробнее.

Сначала считываем содержимое файла настроек $$ini.txt и теперь знаем, что ввел пользователь в диалоговом окне (в самом начале). Общая для нескольких процессов функция — `exportToPDF()`. Она использует заранее созданный набор предустановок для формирования PDF. Удобно, что InDesign понимает название набора в текстовом виде — для наших целей достаточно создать два набора: `LoRes_A4` для создания PDF на страницу и `LoRes_A3` на целый разворот соответственно. Поскольку эти PDF-файлы нужны исключительно для утверждения, в настройках наборов установлено ухудшение качества растровых изображений до 100 dpi, чего вполне достаточно для подобной задачи и, с другой стороны, не нагружает электронную почту.

Имя файла PDF формируется из названия публикации, т. е. если имя у последней myPublication.indd, то будет сформирован файл myPublication_preview.pdf. Необходимость в окончании "preview" продиктована самой жизнью: чтобы случайно кто-то не подставил этот PDF вместо файла с высоким разрешением.

В случае, если задан вывод на печать, снова-таки используется заранее предопределенный набор установок, на этот раз для печати. В отличие от метода `exportFile()` метод `print()` не понимает текстовое название предустановки, приходится использовать индексы. Их можно получить, написав дополнительный мини-скрипт либо подбором, что будет быстрее: в нашем случае ими будут `printerPresets[3]` (распечатывает на листе A3) и `printerPresets[4]` (на листе A4) соответственно (листинг 11.21).

Листинг 11.21. Вывод в PDF

```
function exportToPDF(){
  var newLRFile = new File(myNameNoExt+'_preview.pdf');
  if (HR=='false') {
    if (aD.exportFile(ExportFormat.pdfType, newLRFile, false, ↳
        'LoRes' + pageSize))
  }else{
    if (toPrint) {
      if(pageSize=='_A4') aD.print(false, app.printerPresets[3]);
      else aD.print(false, app.printerPresets[4]);
    }
  }
  aD.save();
}
```

Если вариант заказчиком утвержден, все используемые файлы после этого сразу же копируются на сервер в папку, называющуюся точно так же, как и на компьютере, где реклама готовилась (листинг 11.22).

Процесс переноса файлов в ExtendedScript реализован методом `copy()`: сначала указывается исходный файл, затем — новое его расположение. Сначала формируем пути — к папке со сборочным макетом (`Magazine1Folder`) и к папке на сервере (`serverFolder`). Если последняя не существует, создаем ее. Также задаем два пути для файла с публикацией — на локальном компьютере (`iddAtHome`) и на сервере (`iddAtServer`). Затем в цикле просматриваем все имеющиеся в публикации связи и получаем для каждой путь, где она расположена (`linkAtHome`), и формируем путь, который она должна иметь на сервере (`linkAtServer`).

В случае, если такого файла на сервере нет либо дата его создания ранее даты внесения изенений в копируемом файле (например, файл уже после отправки на сервер снова редактировался на локальной машине), файл переписывается по новому адресу. Параллельно информация о скопированных файлах записывается в переменную `copiedFilesArr` с тем, чтобы после окончания процесса выдать консолидированный отчет: сколько и каких файлов было скопировано. Поскольку операция ответственная, проверяем результат каждого копирования: если оно закончилось неудачей, устанавливаем признак проблемы `copyOK=false`.

Функция `decodeURI()` выполняет сразу две задачи: во-первых, она корректно обрабатывает специальные символы, знаки пробела и т. п., поскольку синтаксис путей к файлам имеет определенные ограничения. Во-вторых, она вычленяет из объекта `File` его путь (если использовать конструкцию

File.toString(), то потом бы пришлось производить замену обратных слэшей на двоеточия, принятые для обозначения пути на Macintosh (replace(// \/g, ':')).

Листинг 11.22. Копирование ресурсов на сервер

```
function postToServer(){
  Magazine1Folder = decodeURI(aD.filePath.name);
  serverFolder = Magazine1Folder + issueNumberFolder+'/'

  iddAtHome = decodeURI( File(aD.fullName))
  iddAtServer = serverFolder+decodeURI( aD.name)

  if (!Folder(serverFolder).exists)
    Folder(serverFolder).create();

  for (myLinks = 0; myLinks < aD.links.length; myLinks++) {
    myl = aD.links[myLinks];
    if(myl.status != LinkStatus.linkMissing){
      linkAtHome = decodeURI( File(myl.filePath))
      linkAtServer = serverFolder + decodeURI( File(myl.filePath).name);

      if (!File(linkAtServer).exists || ↳
         (File(linkAtServer).created.getTime() < ↳
          File(linkAtHome).modified.getTime() ) || ( ↳
          File(linkAtServer).created.getTime() < ↳
          File(linkAtHome).created.getTime() ) ){
        f_src = File(linkAtHome);
        f_new = new File(linkAtServer);
        if (!File(f_src).copy(f_new))
          copyOK = false;
        copiedFilesArr.push(decodeURI(File(f_new).name))
      }
    }
  }

  f_src = File(iddAtHome);
  f_new = new File(iddAtServer);
  if(!File(f_src).copy(f_new))
    copyOK = false;
}
```

В самом начале книги мы обсуждали выбор того или иного языка для скриптинга и остановились на JavaScript как идеальном кандидате на эту роль. Од-

нако традиционно издательские процессы строились на платформе Macintosh, в которой особую популярность имеет AppleScript — как полноценный язык программирования. Поэтому, хотя автор с данным языком не знаком, из уважения к достаточно многочисленному Mac-сообществу напишем несколько вставок, которые демонстрируют одно из многочисленных преимуществ (автор в этом твердо уверен!) альтернативной операционной.

Психологически человек устроен так, что все ответственные операции он склонен перепроверять. Все это в полной мере относится и к формированию конечного PDF для передачи в типографию. Если макетов рекламы много, человеку требуется дополнительный сигнал, что макет проверен и перепроверен и его можно включать в итоговый документ. Обычно для таких целей папки с содержимым рекламных макетов красят в зеленый цвет (`Color label`, есть такая достаточно интересная возможность в Mac OS). Чтобы максимально автоматизировать процесс обработки рекламы, эту последнюю операцию также реализуем с помощью скрипта.

Листинг 11.23. Установка цветной метки на папку

```
applescriptContent = 'tell application "Finder"\rset label index of
parent of file ' & f_src & ' to 6\rend tell'
app.doScript(applescriptContent, ScriptLanguage.applescriptLanguage)
```

Первая строка — не что иное, как скрипт на AppleScript (простите за тавталогию). В ней управление передается программе Finder (аналог Проводника на платформе Windows), после чего выполняется единственная операция — присваивается необходимый цвет папке, содержащей файл InDesign (6 — порядковый номер зеленого цвета из всего списка цветов для `label`).

Для запуска скриптов, написанных на других языках (в том числе и на Visual Basic), в InDesign предусмотрен метод `doScript()`, имеющий два параметра: первый — собственно текст скрипта либо ссылка на файл скрипта, второй — значение, указывающее на использованный в данном скрипте язык, в нашем случае это `applescriptLanguage`.

Теперь решим заачу несколько посложнее. Допустим, обрабатывается реклама, поступающая в несколько изданий (это ситуация, кстати, отражена в скриптинге). В случае утверждения макета заказчиком соответствующий макет распечатывается. Как правило, для этого используется один и тот же принтер, в таком случае менеджерам по рекламе, разбросанным по разным комнатам, нужно знать, когда уже можно идти и забирать распечатки — иначе неминуемы постоянные звонки: "А такой-то и такой-то макеты уже распечатаны? А когда можно их будет забрать?" и т. п. Было решено написать скрипт, который бы после распечатки макета отсылал соответствующему ме-

неджеру уведомление о том, что макет распечатан. Поскольку автор работает на Mac, а скрипт использует межпрограммные соединения, код приведен на applescript (листинг 11.24).

Листинг 11.24. Отправление извещения о готовности распечатки

```
str1 = 'tell application "Mail"\rset n to make new outgoing message\rtell
n\rset subject to "'
subj = Название_рекламы уже распечатана на принтере Название_принтера'
str2 = '"\rmake new to recipient with properties {address:"'
app.doScript(str1+subj+str2+addr+'"}\rend tell\rsend n\rend tell',
ScriptLanguage.applescriptLanguage)
```

Поскольку скрипт занимает несколько строчек, было решено разбить его на несколько читабельных фрагментов, которые потом собираются в одно целое.

Адрес при этом берется из поля Отправитель e-mail сообщения (они также обрабатываются applescript). Или, как вариант, можно создать список менеджеров с их электронными адресами и выводить его на экран перед отсылкой.

Обратите внимание на легкость чтения и понятность выполняемых операций — в этом заключена особая философия, которая свойственна вообще платформе Macintosh.

Удачи в освоении скриптинга!

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

Краткое сравнение синтаксиса AppleScript, JavaScript и VBScript

Данное приложение посвящено обзору наиболее общих сведений о структурах, используемых в программировании (более подробное изложение возможностей одного из языков — JavaScript — приведено в *приложении 2*). Здесь также приведено краткое сравнение синтаксиса языков, которые понимает InDesign. Подобная организация позволит в кратчайшие сроки непрограммистам освоиться с новой для них средой, а программистам, практикующим какой-то один язык (как правило, это JavaScript), в сжатые сроки перейти на другой (как правило, на поддерживаемый Macintosh AppleScript).

П1.1. Комментарии

Комментарии позволяют вносить в код пометки, что полезно в случаях, когда выполняемое действие не очевидно или код длинный. Комментарии дают возможность отслеживать логику работы скрипта и, кроме того, могут использоваться для документирования кода другим пользователям.

В момент выполнения скрипта исполняющая система комментарии игнорирует.

Для комментирования целой строки комментарий размещают в самом ее начале. Кроме того, комментарии могут быть размещены в одной строке с кодом — в таком случае они также распознаются (табл. П1.1).

Таблица П1.1. *Синтаксис комментариев*

Язык	Синтаксис
AppleScript	`* это комментарий` `*) это также комментарий`

Таблица П1.1 *(окончание)*

Язык	Синтаксис

JavaScript	`// это комментарий`
	`/* это также комментарий */`
	`var i=1 // это тоже комментарий`

VBScript	`Rem это комментарий` `' (одинарная кавычка) это также комментарий`

П1.2. Переменные

Переменные используются для хранения значений и упрощают управление данными, позволяя обращаться к ним по именам. Поэтому название, отражающее смысл хранимого значения, значительно облегчает их использование, особенно если код большой. Например, сравните:

```
X_yZ123 = 1
```

и

```
start = 1
```

Разница заметна, не так ли?

Вид данных, хранящихся в переменной, определяет ее тип (табл. П1.2). В первую очередь, такое разделение необходимо для оптимизации выделения памяти, что играет заметную роль в сложных системах. Например, если в переменной хранится слово, тип данных для нее — строка (`string`), а тип данных для переменной, хранящей размер шрифта, — число (`number`). Это наиболее часто используемые типы данных. Более всего лоялен к типу данных язык JavaScript. Другие языки, например VBScript, четко отслеживают соответствие заявленного типа данных реально хранящейся информации, что при миграции скрипта с одного языка на другой часто приводит к возникновению ошибок.

Таблица П1.2. *Типы переменных*

Тип	Значения	Пример
Булево значение	Логические: `true` или `false`	`True`
Целое	Целые числа (больше и меньше нуля). В VBScript можно использовать тип `long`	`14 as integer`

Таблица П1.2 *(окончание)*

Тип	Значения	Пример
	В AppleScript также используются типы `fixed` и `long` — как для целых чисел, так и с плавающей точкой	`541 as long`
Двойной точности	`Double` (VBScript), `fixed` или `real` (AppleScript), с плавающей точкой (JavaScript)	`13.99724 as double`
Строка	Набор символов, заключенных в кавычки	`"Это строка"`
Массив	Набор значений (значения могут быть совершенно любыми)	• AppleScript: `{"0p0", "0p0", "16p4", "20p6"}` • VBScript: `Array("0p0", "0p0", "16p4", "20p6")` • JavaScript: `["0p0", "0p0", "16p4", "20p6"]`

П1.3. Преобразование из одного типа в другой

Необходимость преобразования из одного типа в другой проиллюстрируем на таком примере: допустим, пользователь в окне диалога `prompt()` ввел значение "2" и оно используется как множитель для межстрочного расстояния. Поскольку все значения из окна `prompt()` воспринимаются как строка, фактически "2" будет воспринято как текст "два", соответственно операция `"два"*leading space` приведет к ошибке — поскольку умножение текста на число смысла не имеет. Наиболее часто используемые операции конвертирования — из строки в число и, наоборот, из числа в строку (табл. П1.3).

Таблица П1.3. *Преобразование данных из одного типа в другой*

Язык	Из строки в число	Если строка содержит десятичную точку

AppleScript	`set myNumber to 2`
	`set myString to (myNumber as string)`
	
	`set myString to "2"`
	`set myNumber to (myString as integer)`
		`myNumber to (myString as real)`

Таблица П1.3 *(окончание)*

Язык	Из строки в число	Если строка содержит десятичную точку

JavaScript	`myNumber = 2;`
	`myString = myNumber + "";`
	
	`myString = "2";`
	`myNumber=parseInt(myString)`
	
	`myNumber = +myString;`
		`myNumber=parseFloat(myString);`

VBScript	`myNumber = 2`
	`myString = cStr(myNumber)`
	
	`myString = "2"`
	`myNumber = cInt(myString)`
		`myNumber = cDbl(myString)`

П1.4. Присвоение значений

Различные способы присвоения значений представлены в табл. П1.4.

Таблица П1.4. *Способы присвоения значений в разных языках*

Язык	Пример
AppleScript	`Set myNumber to 10` `Set myString to 'Hello, World!'`
VBScript	`myNumber = 10` `myString = 'Hello, World!'`
JavaScript	`var myNumber = 10` `var myString = 'Hello, World!'`

П1.5. Сравнение значений

Сравнение значений в языках AppleScript, VBScript и JavaScript выполняются по-разному (табл. П1.5).

Таблица П1.5. *Отличия в сравнении значений в разных языках*

Язык	Знак
AppleScript	=
VBScript	=
JavaScript	== (двойное равенство)

П1.6. Массивы

Массивы представляют собой список значений, доступ к которым возможен (в наиболее универсальном варианте) через задание индекса элемента (его порядкового номера в массиве). Способы задания массивов представлены в табл. П1.6.

Таблица П1.6. *Задание массивов в разных языках*

Язык	Пример	Индекс первого элемента
AppleScript	`{"0p0", "0p0", "16p4", "20p6"}`	1
VBScript	`Array("0p0", "0p0", "16p4", "20p6")`	0 (1 — при установке параметра `OptionBase = 1`)
JavaScript	`["0p0", "0p0", "16p4", "20p6"]`	0

П1.6.1. Вложенные массивы

Вложенные массивы используются для дополнительной группировки наборов значений. Отличия в языковых реализациях достаточно незначительны (табл. П1.7).

Таблица П1.7. *Вложенные массивы*

Язык	Пример
AppleScript	`myArray = Array(Array(0,0), Array(72, 72)) NewDouble (0,0))`
JavaScript	`var myArray = [[0,0], [72,72]];`
VBScript	`myArray = New Array(New Double(0,0))`

П1.7. Определение типа переменной

Часто для определенной работы скрипта необходимо знать тип конкретного объекта. Например, если в InDesign выделен фрейм, присвоение ему кегля вызовет ошибку. В каждом языке существует инструмент для определения типа объекта (табл. П1.8).

Таблица П1.8. *Определение типа переменной*

Язык	Пример
AppleScript	`set myType to class of myMysteryVariable`
JavaScript	`myType = myMysteryVariable.constructor.name;` `// myType будет строкой, соответствующей типу элемента`
VBScript	`myType = TypeName(myMysteryVariable)`

П1.8. Объединение строк

Объединение строк (конкатенация) — одна из наиболее распространенных операций при работе со строками. При этом синтаксис AppleScript и VBScript полностью идентичен, несколько отличается только JavaScript (табл. П1.9).

Таблица П1.9. *Объединение строк в разных языках*

Язык	Знак	Пример	Результат
AppleScript	`&` (амперсанд)	`"Pride " & "and Prejudice"`	`"Pride and Prejudice"`
JavaScript	`+`	`"Pride " + "and Prejudice"`	`"Pride and Prejudice"`
VBScript	`&`	`"Pride " & "and Prejudice"`	`"Pride and Prejudice"`

П1.9. Проверка условий

Ни один мало-мальски функциональный скрипт не может обойтись без проверки корректности какого-либо условия с тем, чтобы гарантировать безошибочное выполнение дальнейших операций. Например, в коммерческих скриптах перед любым действием, если только есть малейшее подозрение на возможность возникновения ошибки, сначала проверяются все необходимые для его работы условия, и только потом выполнение программы продолжается. Фактически, именно задание правильных условий в нужных местах — половина успешного написания программы (табл. П1.10).

Таблица П1.10. *Проверка условий в разных языках*

Язык	Пример
AppleScript	**if** (*условие*)
VBScript	**if** (*условие*)
JavaScript	**if** (*условие*)

П1.10. Циклы

Циклы используются для выполнения повторяющихся действий (табл. П1.11). Например, можно вставить в публикацию 10 пустых страниц — это пример простейшего цикла. Как правило, циклы используются в составе других управляющих структур, позволяя задействованным объектам менять (или не менять) свои свойства. Циклы всегда должны иметь определенное количество шагов — иначе возможно "зацикливание": когда он бесконечно повторяет заключенные внутри него действия.

Таблица П1.11. *Организация циклов*

Язык	Пример
AppleScript	**repeat** (*начальное_значение*, *конечное_значение*, *шаг*)
VBScript	**for** (*начальное_значение*, *конечное_значение*, *шаг*)
JavaScript	**for** (*начальное_значение*, *конечное_значение*, *шаг*)

П1.11. Функции

Функции (в VBScript и JavaScript известны как `Functions`, в AppleScript — как `handlers`) предназначены для повышения читабельности кода и представляют собой фрагменты кода, которые вызываются необходимое количество раз из основного скрипта. Как правило, им на обработку передаются определенные данные (параметры), а по окончании выполнения функции полученный результат возвращается в вызвавший модуль (табл. П1.12).

Таблица П1.12. *Задание функций*

Язык	Пример
AppleScript	**on** *название_функции* (*параметры*)
VBScript	**function** *название_функции* (*параметры*)
JavaScript	**function** *название_функции* (*параметры*)

П1.12. Пример

Далее приведен пример реализации простейшего скрипта на трех языках для тех, кто еще не имеет опыта программирования — возможно, какой-то из этих языков покажется более удобным. В данной книге основной упор сделан

на JavaScript, как наиболее распространенный язык в связи с его использованием в интернет-браузерах. Описание его возможностей, достаточное для решения большинства задач в InDesign, приведено в *приложении 2*.

При программировании вы постоянно будете создавать переменные — в них временно хранятся значения, результаты вычислений, ссылки на объекты и т. п. Для удобства им присваивают имена, раскрывающие их смысл: например, `textFrame` — для текстового фрейма, `page` — для страницы и т. д. Для того чтобы переменная, которую вы создаете, случайно не совпала с одним из зарегистрированных методов и свойств InDesign (их названия тоже выбраны так, чтобы легко запоминались), хорошим тоном может быть добавление приставки `my`: например, `myTextFrame`, `myPage` и т. п.

Рассмотрим скрипты, выводящие текст с приветствием в новом документе (листинги П1.1—П1.3).

Для этого потребуется выполнить следующие шаги:

1. Установить связь с InDesign.
2. Создать новый документ.
3. Создать на первой странице текстовый фрейм.
4. Добавить во фрейм текст.

П1.12.1. AppleScript

Листинг П1.1

```
tell application "Adobe InDesign CS3"
set myDocument to make document
tell page 1 of myDocument
set myTextFrame to make text frame
set geometric bounds of myTextFrame to {"10mm", "10mm", "24mm", "24mm"}
set contents of myTextFrame to "Hello, World!"
end tell
end tell
```

П1.12.2. JavaScript

Листинг П1.2

```
myDocument = app.documents.add();
myTextFrame = myDocument.pages[0].textFrames.add();
myTextFrame.geometricBounds = {"10mm", "10mm", "24mm", "24mm"];
myTextFrame.contents = "Hello, World!";
```

П1.12.3. VBScript

Листинг П1.3

```
Set myInDesign = CreateObject("InDesign.Application.CS3")
Set myDocument = myInDesign.Documents.Add
Set myTextFrame = myDocument.Pages(1).TextFrames.Add
myTextFrame.GeometricBounds = Array{"10mm", "10mm", "24mm", "24mm")
myTextFrame.Contents = "Hello, World!"
```

ПРИЛОЖЕНИЕ 2

JavaScript

П2.1. Переменные

В качестве первого символа в имени переменной могут использоваться латинские символы и знаки подчеркивания (_). Цифры в начале имени не допустимы, хотя в любом другом месте ограничений нет.

Например:

```
option_Ex8;
```

создаст переменную, а варианты

```
8_optEx;
Ы_param;
```

ее не создадут и вызовут ошибку, поскольку в первом случае имя переменной начинается с цифры, а во втором случае имя переменной содержит недопустимый символ.

Также имена переменных не могут совпадать с зарезервированными словами *(см. приложение 3)*.

JavaScript различает переменные по регистру, поэтому `Xyz` и `xyz` — это совершенно *разные* переменные.

П2.1.1. Задание переменных

Если переменная задается вне функции, то она называется *глобальной*, поскольку доступна в любом месте документа. Если же она объявлена внутри функции, то называется *локальной*, поскольку доступна только внутри функции.

Использование `var` при объявлении глобальной переменной не требуется. Однако при объявлении переменной внутри функции использование `var` необходимо.

П2.1.2. Типы переменных

В JavaScript переменные объявляются при помощи специального слова `var` (от англ. *variable* — переменная). Например,

```
var myNum;
```

Переменную можно объявить и одновременно присвоить ей значение:

```
var myNum = 12;
```

Несколько переменных можно объявить в одной строке, разделив их имена запятыми, например:

```
var myNum, myNum1, myNum2 = 16, myNum3;
```

Но, с точки зрения удобочитаемости скрипта, лучше применять запись:

```
var myNum;
var myNum1;
var myNum2 = 16;
var myNum3;
```

Слово `var` обязательным не является, хотя применение его рекомендуется, опять же из соображений удобочитаемости.

Переменным в JavaScript можно присваивать значения перечисленных далее типов.

Целочисленные значения

Целочисленные значения могут быть в трех системах счисления: десятичной, шестнадцатеричной и восьмеричной.

Десятичное счисление — традиционное:

```
var myNum = 10;
```

Если же значение переменной начинается с `0x`, это означает, что переменной присваивается значение в шестнадцатеричной системе счисления.

```
var myNum = 0x10;
```

Переменной `myNum` присвоено значение, равное `0x10` в шестнадцатеричной системе счисления, что соответствует 16 в десятичной.

Когда значение переменной начинается с `0` и содержит только цифры от 0 до 7, то такая переменная воспринимается JavaScript, как число в восьмеричной системе счисления:

```
var myNum = 010;
```

Переменной myNum присвоено значение, равное 010 в восьмеричной системе счисления, что соответствует 7 в десятичной.

Значения с плавающей точкой

С такой же легкостью, как и целочисленные значения, переменные в JavaScript принимают и значения с плавающей точкой.

```
var myRealNum = 12.756;
var myRealNum2 = 12.322e-16;
```

Логические значения

Логических значений всего два — true (истина) и false (ложь). Присвоение этих значений переменным имеет вид:

```
var myTrue = true;
var myFalse = false;
```

Значения true и false всегда записываются строчными буквами.

Строковые значения

Строковое значение переменной должно быть заключено либо в двойные, либо в одинарные кавычки:

```
var myString1 = "Строка в двойных кавычках";
var myString2 = 'Строка в одинарных кавычках';
```

Если в значении строковой переменной должны быть знаки кавычек, то используются так называемые escape-последовательности, начинающиеся с обратной косой черты:

```
var myString1 = "\" — двойная кавычка"
var myString2 = '\' — одинарная кавычка';
```

Объекты

Объекты являются основой работы с InDesign в JavaScript. Каждый элемент в иерархической объектной модели InDesign может стать значением переменной JavaScript. Например:

```
var myDocument = app.activeDocument;
```

означает, что переменная myDocument становится ссылкой на объект — на документ, который в настоящее время активизирован в InDesign.

Специальное значение

Специальное значение null для переменной означает отсутствие значения. Это не все равно, что присвоить переменной значение, равное 0 или пустой строке "", поскольку тип значения null является объектом.

Если объявить переменную, но не присвоить ей никакого значения, то тип такой переменной будет восприниматься как `undefined` (отсутствие значения).

П2.1.3. Преобразование типов

В JavaScript нет четкого разделения переменных по типам — переменная может менять свой тип несколько раз на протяжении скрипта, принимая тот тип, какой имеет хранящееся в ней значение. То есть одна и та же переменная может в разные моменты выполнения скрипта содержать целочисленное значение (1, 2, 3, ...), строку ("один", "два" и т. п.) или логическое значение (`true` или `false`).

Например, вполне допустимо:

```
var myNum = 2;
myNum = true
```

В выражениях, содержащих числовые и строковые значения и операцию `+`, JavaScript автоматически конвертирует числа в строки, а вот обратная операция требует применения специальных функций `parseInt()` и `parseFloat()` (перевести в целочисленное и с плавающей точкой).

```
nowWhat = "10" + 1; // результат 101, поскольку первой идет строка
```

Однако

```
nowThen = 1 + parseInt("10") // результат 11
```

или

```
nowThen = 1 + "10" // результат 11, поскольку первым идет число
```

Таким образом, во избежание ошибок следует внимательно относиться к операциям присвоения значений переменным.

П2.2. Операции

Все операции в JavaScript подразделяются на описанные далее типы.

П2.2.1. Операции сравнения

Возможные в JavaScript операции сравнения перечислены в табл. П2.1.

По умолчанию все зарезервированные свойства объектов логического типа имеют значение `true`, поэтому для краткости можно использовать, например, вариант, приведенный в листинге П2.1.

Таблица П2.1. *Операции сравнения*

Операция	Название	Описание
`a < b`	Меньше	Возвращает `true`, если левая часть меньше, чем правая
`a > b`	Больше	Возвращает `true`, если левая часть больше, чем правая
`a <= b`	Не больше	Возвращает `true`, если левая часть меньше или равна правой
`a => b`	Не меньше	Возвращает `true`, если левая часть больше или равна правой
`a == b`	Равно	Возвращает `true`, если левая часть равна правой
`a != b`	Не равно	Возвращает `true`, если левая часть не равна правой

Листинг П2.1

```
if (filled)
...// Выполняется, если объект имеет заливку
```

Вместо

```
if (filled == true){
...
```

Соответственно:

```
if (!filled)
...// Выполняется, если объект заливки не имеет
```

Вместо

```
if (filled != true)...
```

П2.2.2. Арифметические операции

Арифметические операции представлены в табл. П2.2.

Таблица П2.2. *Арифметические операции*

Операция	Название	Описание
`a + b`	Сложение	Возвращает сумму двух значений
`a - b`	Вычитание	Возвращает разность от вычитания правого значения из левого
`a * b`	Умножение	Возвращает произведение двух значений
`a / b`	Деление	Возвращает частное от деления левого значения на правое

Таблица П2.2 *(окончание)*

Операция	Название	Описание
`a % b`	Остаток по модулю (целочисленный остаток)	Возвращает целый остаток от деления левого значения на правое. Числа с плавающей точкой перед операцией округляются до целых
`++`	Инкремент	Увеличивает значение переменной на 1. Если используется как префикс (`++a`), возвращает значение переменной после увеличения ее на 1. Если используется как постфикс (`a++`), возвращает значение переменной перед увеличением ее на 1
`--`	Декремент	Уменьшает значение переменной на 1. Если используется как префикс (`--a`), возвращает значение переменной после уменьшения ее на 1. Если используется как постфикс (`a--`), возвращает значение переменной перед уменьшением ее на 1
`-a`	Смена знака	Возвращает арифметическое отрицание значения

Примеры:

```
x = 3
y = x++   // y = 3; x = 4
y = ++x   // x = 4; y = 4
```

П2.2.3. Логические операции

Логические операции применяются к логическим операндам и возвращают логическое значение, означающее результат операции (табл. П2.3). Если типы операндов различны, то делается попытка преобразовать их к логическому типу.

Таблица П2.3. *Логические операции*

Операция	Название	Описание
`a && b`	Логическое И	Возвращает `true`, если оба выражения истинны. Если первое выражение ложно, то возвращает `false`, не вычисляя значение второго выражения
`a || b`	Логическое ИЛИ	Возвращает `true`, если хотя бы одно выражение истинно. Если первое выражение истинно, то возвращает `true`, не вычисляя значение второго выражения
`!a`	Логическое НЕ	Возвращает `true`, если выражение ложно

Обратите внимание, что для ускорения работы скриптов дальнейшие проверки прекращаются, если текущая операция возвращает `true`.

Пример:

```
(a || !b)
```

Если `a` — ложно, следующее выражение (`!b`) не вычисляется.

П2.2.4. Операции со строками

Объединение (конкатенация) строк обозначается символом `+`. Если хотя бы одно значение является строкой, то результатом операции является слияние строк.

Примеры:

```
text = "In" + "Design";   // k равно "InDesign"
text = "Suite: " + 2;     // text равно "Suite: 2"
```

П2.2.5. Операции присваивания

Возможные операции присваивания перечислены в табл. П2.4.

Таблица П2.4. *Операции присваивания*

Сокращенная запись	Эквивалент
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`

Нужно понимать разницу между присваиванием значения и проведением каких-то манипуляций объектом. Например, если вы осуществляете замену содержимого текста материала публикации

```
a = stories[0].contents
```

то

```
a.replace(/шаблон/, на_что_менять)
```

не приведет к желаемому результату.

Причина в том, что проводится операция замены со строкой, содержание которой соответствует тексту в публикации. Но при этом никаких изменений

с самой публикацией не происходит. Чтобы действительно произошла замена, текст в публикации должен быть заменен результатом замены, который хранится в переменной `a`. Например:

```
stories[0].contents = a.replace(/шаблон/, на_что_менять)
```

Любой язык программирования использует лишь три вида операций: *потоковые вычисления* (присвоение переменных, вычисление значений), *ветвление* (если..., то..., иначе...) и *циклы* (делать, пока не...).

П2.3. Ветвления (операторы условий)

П2.3.1. Оператор *if...else*

Используется для проверки условия и выполнения тех или иных действий в зависимости от результата. Если результат — истина, выполняются *действия_1*, иначе — *действия_2*.

```
if (условие)
  действия_1
[else
  действия_2]
```

Краткая форма:

```
() ? () : ()
```

Пример приведен в листинге П2.2.

Листинг П2.2

```
var price = (price>100) ? "дорого" : "дешево"
```

Запись эквивалентна:

```
if (price>100)
{
  price = "дорого"
} else {
  price = "дешево"
}
```

П2.3.2. Оператор *switch*

Оператор выбора `switch` выполняет ту или иную последовательность операторов в зависимости от значения определенного выражения.

Он имеет вид:

```
switch (выражение) {
  case значение:
    операторы
    break;
  case значение:
    операторы
    break;
 ...
  default:
    операторы
}
```

Здесь *выражение* — это любое выражение, *значение* — это возможное значение выражения, а *операторы* — любые группы операторов JavaScript.

Оператор выбора сначала вычисляет значение выражения, а затем проверяет, нет ли этого значения в одной из меток `case`. Если такая метка есть, то выполняются операторы, принадлежащие ей; если нет, то выполняются операторы, следующие за меткой `default` (если она отсутствует, то управление передается оператору, следующему за `switch`).

Необязательный оператор `break` указывает, что после выполнения операторов управление передается оператору, следующему за `switch`. Используется для ускорения проверки (зачем выполнять остальные проверки, если совпадение уже произошло — результатом этих проверок все равно будет `false`). Если `break` отсутствует, то после выполнения операторов начинают выполняться операторы, стоящие после следующей метки `case`.

Пример использования оператора `switch` представлен в листинге П2.3.

Листинг П2.3

```
switch(app.selection[0].constructor.name){
  case "Cell":
    alert("Курсор в ячейке");
    break;
  case "Table":
    alert("Выделена таблица");
    break;
}
```

П2.3.3. Оператор *try...catch*

Оператор `try...catch` используется в тех фрагментах сценария, где может возникнуть ошибка, для ее обработки.

Он имеет вид:

```
try {
  оператор1
}
catch (исключение) {
  оператор2
}
```

Здесь *исключение* — любое имя переменной, а *оператор1* и *оператор2* — любые группы операторов JavaScript, заключенные в фигурные скобки `{}`.

Оператор1 содержит программный код, в котором возможно возникновение исключения. Если исключение не возникло, то после исполнения *оператора1* управление передается обычным образом оператору, следующему за `try...catch`. Если же оно возникло, то информация об исключении заносится в локальную переменную *исключение*, и управление передается *оператору2*, который должен содержать код обработки этого исключения.

Пример приведен в листинге П2.4.

Листинг П2.4

```
try {
  for (var i in findA)
    pars[j].contents = pars[j].contents.replace(eval('/'+findA[i]+'/g'),
      replaceA[i])
  pars[j].contents.replace(/\(см\. ?.+\)/, f)
  if(ix!=-1)  {
    pars[j].characters.itemByRange(ix, ix+l).applyStyle(cS)
    ix = -1
  }
}
catch (err) {
  alert(pars[j].contents + err.name+"; " + err.description + "; " +
    err.message)
}
```

П2.4. Циклы

Цикл — это последовательность операторов, выполнение которой повторяется до тех пор, пока определенное условие не станет ложным. JavaScript содержит три оператора цикла: `for`, `while` и `do...while`, а также операторы `break` и `continue`, которые используются внутри циклов.

П2.4.1. Оператор *while*

Синтаксис:

while (*условие*) *оператор*

Это простейший из циклов и используется в случаях, когда заранее не известно, сколько раз должен выполняться цикл.

Оператор `while` выполняется следующим образом:

1. Вычисляется значение выражения *условие*. Если оно ложно, то управление передается оператору, следующему за данным оператором.
2. Если значение истинно, выполняются текущие операторы и управление передается этапу 1.

При использовании данного оператора необходимо убедиться, что рано или поздно условие станет ложным, т. к. иначе сценарий войдет в бесконечный цикл, например:

```
while (true)
alert("Привет всем!");
```

Пример использования оператора `while` приведен в листинге П2.5.

Листинг П2.5

```
while(true){
  switch(myRadioButtons.selectedButton){
    case 0:
      c=s[0].contents;
      break;
    case 2:
      c = s[1].contents;
  }
}
```

П2.4.2. Оператор *do...while*

Синтаксис:

do *оператор* **while** (*условие*)

Здесь *условие* — выражение сравнения или логическое выражение, определяющее, сколько раз выполнять цикл, *оператор* — блок операторов, выполняемых в каждом цикле.

Оператор `do...while` выполняется следующим образом:

1. Выполняется *оператор*.

2. Вычисляется значение выражения *условие*. Если оно ложно, то управление передается оператору, следующему за данным оператором.
3. Если оно истинно, управление передается этапу 1.

Этот оператор отличается от оператора `while` тем, что цикл обязательно выполняется хотя бы раз (листинг П2.6).

Листинг П2.6

```
var i = 0;
do
  alert(i++);
while (i < 10);
```

П2.4.3. Оператор *for*

Синтаксис:

```
for (счетчик = начальное_значение; условие; шаг){
  операторы
}
```

Здесь:

- *счетчик* — числовая переменная, являющаяся счетчиком цикла. Определяет, сколько раз будут выполнены операторы цикла;
- *начальное_значение* — первоначальное значение счетчика;
- *условие* — выражение сравнения, определяющее конечное значение счетчика;
- *шаг* — шаг изменения счетчика.

В отличие от `while`, цикл `for` — более сложный способ организации циклов, применяется в тех случаях, когда точно известно количество повторений. Это наиболее часто используемая конструкция.

Оператор `for` выполняется следующим образом:

1. Переменной *счетчик* присваивается *начальное_значение*.
2. Вычисляется значение выражения *условие*. Если оно ложно, то управление передается оператору, следующему за данным оператором.
3. Если выражение истинно, выполняется блок операторов.
4. Изменяется текущее значение счетчика на *шаг* (обычно это выражение увеличивает или уменьшает счетчик или счетчики цикла) и управление передается этапу 2.

Пример использования цикла `for` представлен в листинге П2.7.

Листинг П2.7

```
for(i=0; i< myTable.cells.length; i++){
  with (myTable.cells[i]){
    topInset = "1 mm"
    bottomInset = "1 mm"
  }
}
```

П2.4.4. Оператор *break*

Оператор `break` прерывает выполнение текущего цикла и передает управление оператору, следующему за прерванным.

Используется в тех случаях, когда в цикле возникает непредвиденное значение переменной либо условие, делающее дальнейшее выполнение цикла невозможным (листинг П2.8).

Листинг П2.8

```
function findValue(a, theValue) {
  for (var i = 0; i < a.length; i++) {
    if (a[i] == theValue)
      break;
  }
  return i;
}
```

П2.4.5. Оператор *continue*

Оператор `continue` похож на оператор `break`, только в отличие от него не полностью завершает цикл, а только пропускает оставшиеся операторы цикла и начинает новый цикл.

Переход к следующей итерации цикла происходит следующим образом:

1. Циклы `while` и `do...while` проверяют условие цикла и, если оно истинно, начинают очередное выполнение цикла.
2. Цикл `for` выполняет изменение выражения, проверяет условие цикла и, если оно истинно, начинает очередное выполнение цикла.
3. Цикл `for...in` переходит к следующему полю переменной и начинает очередное выполнение цикла.

В примере из листинга П2.9 к переменной `n` последовательно добавляются значения `1`, `2`, `4` и `5`.

Листинг П2.9

```
var i = 0;
var n = 0;
while (i < 5) {
  i++;
  if (i == 3)
    continue;
  n += i;
}
```

П2.4.6. Оператор *for...in*

Оператор `for...in` выполняет заданные действия для каждого свойства объекта или для каждого элемента массива. Основная область применения — для объектов (в первую очередь массивов), созданных пользователем. Синтаксис:

for (*переменная* in *выражение*) *оператор*

Оператор `for...in` выполняется следующим образом:

1. Переменной присваивается имя очередного свойства объекта или очередного элемента массива.
2. Выполняется *оператор*.
3. Управление передается этапу 1.

При итерации массива переменной последовательно присваивается значение первого, второго, ..., последнего элемента массива. Однако при итерации свойств объекта невозможно предсказать, в каком порядке они будут присваиваться переменной: этот оператор гарантирует только то, что все они будут просмотрены.

Сценарий из листинга П2.10 создает новый объект `city`, а затем последовательно выводит все его свойства на экран.

Листинг П2.10

```
var city = {"а" : " Киев ", "б" : " Москва ", "в" : " Владивосток "};
for (var key in city)
  alert (key + ": " + city [key] + "\n");
```

Результат:

```
а: Киев
б: Москва
в: Владивосток
```

П2.4.7. Оператор *with*

Оператор `with` задает имя объекта по умолчанию.

Этот оператор действует следующим образом. Для каждого оператора в пределах конструкции `with{}` проверяется, не является ли он именем свойства объекта, заданного по умолчанию (задан в *выражение*). Если да, то этот идентификатор считается именем свойства, если же нет, то именем переменной.

Оператор используется для сокращения размера программного кода и ускорения доступа к свойствам объектов. Например, для доступа к математическим функциям мы должны каждый раз указывать имя объекта `Math`:

```
x = Math.cos(Math.PI / 2) + Math.sin(Math.LN10);
y = Math.tan(2 * Math.E);
```

С помощью оператора `with` этот фрагмент сценария можно существенно укоротить (листинг П2.11).

Листинг П2.11

```
with (Math) {
  x = cos(PI / 2) + sin(LN10);
  y = tan(2 * E);
}
```

> ***Примечание***
>
> Оператор `with` может применяться только к существующим свойствам и методам объекта. Попытка создания нового свойства или метода с его помощью вызовет ошибку.

П2.5. Объекты

JavaScript содержит встроенные объекты, перечисленные в табл. П2.5.

Таблица П2.5. *Встроенные объекты JavaScript*

Объект	Описание	Объект	Описание
`String`	Строковые объекты	`RegExp`	Регулярные выражения
`Array`	Массивы	`Number`	Числовые объекты
`Function`	Функции	`Boolean`	Логические объекты
`Math`	Математические функции и константы	Дата и время	Обработка даты и времени

П2.5.1. Строковые объекты *(String)*

Методы

Методы, относящиеся к объекту `String`, перечислены в табл. П2.6.

Таблица П2.6. *Методы объекта* `String`

Метод	Описание
`split`	Разбивает объект на массив строк
`charAt`	Возвращает символ, расположенный в заданной позиции строки
`charCodeAt`	Возвращает Unicode-код символа, расположенного в указанной позиции строки
`concat`	Объединяет несколько строк
`fromCharCode`	Объединяет символы Unicode в строку
`indexOf`	Возвращает первую позицию в строке искомого выражения
`lastIndexOf`	Возвращает последнюю позицию в строке искомого выражения

Метод *split*

Синтаксис:

объект`.`**`split`**`(`*разделитель*`)`

Аргумент: *разделитель* — строковое или регулярное выражение.

Результат: массив строк (`Array`).

Метод `split` разбивает объект на массив строк и возвращает его. Сам разделитель в результат не попадает.

Разделитель может быть задан либо строкой, либо регулярным выражением.

Например,

```
var s = "a1b2c3d".split(/\d/);
```

вернет массив `["a", "b", "c", "d"]`.

Метод *charAt*

Синтаксис:

объект`.`**`charAt`**`(`*позиция*`)`

Аргумент: *позиция* — любое числовое выражение.

Результат: строка.

Метод charAt возвращает символ, расположенный в данной позиции строки. Позиции символов строки нумеруются от 0 до *объект*.length-1. Если позиция лежит вне этого диапазона, то возвращается пустая строка. Например, конструкция

```
alert("Строка".charAt(0))
```

выведет на экран символ С.

Метод *charCodeAt*

Синтаксис:

объект.**charCodeAt**(*позиция*)

Аргумент: *позиция* — любое числовое выражение.

Результат: число.

Метод charCodeAt возвращает код Unicode символа, расположенного в указанной позиции строки. Позиции символов строки нумеруются от 0 до *объект*.length-1. Если позиция лежит вне этого диапазона, то возвращается NaN.

Метод *concat*

Синтаксис:

объект.**concat**(*строка0*, *строка1*, ..., *строкаN*)

Аргументы: *строка0*, *строка1*, ..., *строкаN* — любые строковые выражения.

Результат: строка.

Метод concat возвращает новую строку, являющуюся объединением исходной строки с заданными строками. Этот метод эквивалентен операции

объект + *строка0* + *строка1* + ... + *строкаN*

Например, оператор

```
alert("Я вас любил. ".concat("Чего же боле?"))
```

выведет на экран строку

```
"Я вас любил. Чего же боле?"
```

Метод *fromCharCode*

Синтаксис:

String.**fromCharCode**(*код1*, *код2*, ..., *кодN*)

Аргументы: *код1*, *код2*, ..., *кодN* — числовые выражения.

Результат: строка.

Метод fromCharCode создает новую строку, которая является объединением символов Unicode с кодами *код1*, *код2*, ..., *кодN*.

Пример:

```
var s = String.fromCharCode(65, 66, 67);  // s равно "ABC"
```

Метод *indexOf*

Синтаксис:

объект.**indexOf**(*подстрока*[, *начало*])

Аргументы: *подстрока* — любая строка, *начало* — любое число.

Результат: число.

Метод `indexOf` возвращает первую позицию в строке искомой подстроки. Позиции символов строки нумеруются от 0 до *объект*.`length-1`. Если задан необязательный аргумент *начало*, то поиск ведется, начиная с позиции *начало*; если нет, то с позиции 0, т. е. с первого символа строки. Если начало отрицательно, то оно принимается равным нулю; если начало больше, чем *объект*.`length-1`, то оно принимается равным *объект*.`length-1`. Если объект не содержит данной подстроки, то возвращается значение –1.

Поиск ведется слева направо. В остальном этом метод идентичен методу `lastIndexOf`.

Пример из листинга П2.12 подсчитывает количество вхождений подстроки `pattern` в строку `str`.

Листинг П2.12

```
function occur(str, pattern) {
 var pos = str.indexOf(pattern);
 for (var count = 0; pos != -1; count++)
   pos = str.indexOf(pattern, pos + pattern.length);
 return count;
}
```

Метод *lastIndexOf*

Синтаксис:

объект.**lastIndexOf**(*подстрока*[, *начало*])

Аргументы: *подстрока* — любая строка, *начало* — любое число.

Результат: число.

Метод `lastIndexOf` возвращает последнюю позицию в строке искомой подстроки. Позиции символов строки нумеруются от 0 до *объект*.`length-1`. Если задан необязательный аргумент *начало*, то поиск ведется, начиная с позиции *начало*; если нет, то с позиции 0, т. е. с первого символа строки. Если начало отрицательно, то оно принимается равным нулю; если начало больше, чем

объект length-1, то оно принимается равным *объект*.length-1. Если объект не содержит данной подстроки, то возвращается значение –1.

Поиск ведется справа налево. В остальном этом метод идентичен методу indexOf. Пример:

```
var n = "Розовый слон".lastIndexOf("слон");
```

Свойства

У строк, как и у массивов, есть единственное свойство — length — количество элементов в строке.

П2.5.2. Массивы

Объект Array используется для создания массивов, т. е. упорядоченных наборов элементов любого типа. Доступ к элементу массива производится по его номеру в массиве, называемому *индексом элемента*; обозначается i-й элемент массива a как a[i]. Элементы массива нумеруются с нуля, т. е. массив a, состоящий из N элементов, содержит элементы a[0], a[1], ..., a[N-1].

Для создания массивов используются следующие конструкции:

```
new Array()
new Array(размер)
new Array(элемент0, элемент1, ..., элементN)
```

Здесь *размер* — любое числовое выражение, задающее количество элементов в массиве; *элемент0*, *элемент1*, ..., *элементN* — любые выражения.

Первый конструктор создает пустой массив, второй — массив из *размер* элементов, третий создает массив из N+1 элементов и присваивает им соответствующие значения. Если размер не является числом без знака, то создается массив с единственным элементом, имеющим это значение.

Кроме того, массив может быть создан и так:

```
[элемент0, элемент1, ..., элементN]
```

Примеры:

```
var a = new Array(5);          // массив из 5 элементов
var b = new Array("строка");   // массив из 1 элемента "строка"
var c = new Array(1, 2, 3);    // массив из 3 элементов: 1, 2 и 3
var d = ["1", "2", "3"];       // то же самое
```

Мы можем неявно увеличить размер массива, присвоив значение элементу с несуществующим индексом, например:

```
var colors = new Array();      // пустой массив
colors[100] = "пурпурный";     // размер массива стал равен 100
```

Свойства

Свойство у массивов, как и у строк, — всего одно.

Свойство *length*

Значением свойства `length` является размер массива, т. е. количество элементов в нем, например:

```
var x = new Array();
x[0] = "Строка";
x[5] = "Еще строка";
var l = x.length;  // l равно 6
```

Мы можем явно задать новый размер массива, изменяя значение свойства `length`. Если при этом новый размер массива меньше текущего, то лишние элементы массива будут удалены. Если же новый размер массива больше текущего, то к массиву будут добавлены новые элементы со значением `undefined`.

Методы

Методы массивов перечислены в табл. П2.7.

Таблица П2.7. *Методы массивов*

Метод	Описание
`concat`	Объединяет два массива в один новый и возвращает его
`join`	Объединяет все элементы массива в текстовую строку
`pop`	Удаляет последний элемент массива
`push`	Добавляет элементы в конец массива
`reverse`	Изменяет порядок элементов массива на противоположный
`shift`	Удаляет первый элемент массива и возвращает его
`slice`	Извлекает часть массива и возвращает новый массив
`sort`	Сортирует элементы массива
`splice`	Заменяет часть массива
`toLocaleString`	Преобразует массив в строку с учетом формата операционной системы
`toString`	Преобразует массив в строку
`unshift`	Добавляет элементы в начало массива
`valueOf`	Возвращает примитивное значение массива

Метод *concat*

Синтаксис:

массив.**concat**(*массив1*)

Аргумент: *массив1* — выражение, возвращающее массив.

Результат: новый массив.

Метод concat объединяет *массив* и *массив1* в новый массив и возвращает его. При этом все элементы из *массив1* добавляются в конец *массива*.

Пример:

```
var x = new Array(1, 2, 3);
var y = new Array(4, 5, 6);
currentTextFrame.contents = x.concat(y)
```

Результат:

```
1,2,3,4,5,6
```

Метод *join*

Синтаксис:

массив.**join**(*разделитель*)

Аргумент: *разделитель* — любое строковое выражение.

Результат: строковое значение.

Метод join преобразует массив в строковое значение. Для этого все элементы массива преобразуются в строки, и эти строки объединяются в одну строку. Разделителем между ними в результирующей строке является заданный *разделитель*; если он опущен, то разделителем служит пустая строка.

Пример:

```
var x = new Array(1, 2, 3);
currentTextFrame.contents = x.join("-")
```

Результат:

```
1-2-3
```

Метод *pop*

Синтаксис:

массив.**pop**()

Результат: значение последнего элемента массива.

Метод pop удаляет последний элемент массива и возвращает его в качестве результата. Размер массива при этом уменьшается на 1.

Пример:

```
var x = ["a", "b", "c", "d"]
x.pop();
```

Результат:

```
a,b,c
```

Метод *push*

Синтаксис:

массив.**push**(*элемент1*, ..., *элементN*)

Аргументы: *элемент1*, ..., *элементN* — любые выражения.

Результат: новая длина массива.

Метод `push` добавляет значения аргументов в конец массива и возвращает в качестве результата новый размер массива, который при этом увеличивается на `N`.

Пример:

```
var x = ["a", "b", "c", "d"]
x.push("e")
```

Результат:

```
a,b,c,d,e
```

Метод *reverse*

Синтаксис:

массив.**reverse**()

Результат: массив.

Метод `reverse` изменяет порядок элементов в массиве на противоположный. При этом новый объект `Array` не создается, перестановка элементов производится в исходном массиве. Если не все элементы массива были определены, то им присваивается значение `undefined`.

Пример:

```
var x = new Array();
x[0] = 0;
x[2] = 2;
x[4] = 4;
x.reverse()
```

Результат:

```
4,,2,,0
```

Метод *shift*

Синтаксис:

массив.**shift**()

Результат: значение первого элемента массива.

Метод shift удаляет первый элемент массива и возвращает его в качестве результата. Размер массива при этом уменьшается на 1.

Пример:

```
var x = ["a", "b", "c", "d"]
x.shift();
```

Результат:

```
b,c,d
```

Метод *slice*

Синтаксис:

массив.**slice**(*начало* [,*конец*])

Аргументы: *начало* и *конец* — любые числовые выражения.

Результат: новый массив.

Метод slice возвращает новый массив, содержащий указанную часть исходного массива. Аргумент *начало* задает индекс первого элемента копируемой части, необязательный аргумент *конец* — индекс ее последнего элемента. При этом:

- если значение *конец* неотрицательно, то копируются элементы *массив*[*начало*], *массив*[*начало*+1], ..., *массив*[*конец*-1];
- если значение *конец* отрицательно, то копируются элементы *массив*[*начало*], *массив*[*начало*+1], ..., *массив*[*массив*.length-*конец*-1], т. е. *конец* означает смещение от конца массива;
- если аргумент *конец* отсутствует, то копируются элементы *массив*[*начало*], *массив*[*начало*+1], ..., *массив*[*массив*.length-1], т. е. копируются все элементы до конца массива.

Пример:

```
var x = new Array(10);
for (i = 0; i < 10; i++)
  x[i] = i;
x.slice(5,-1)
```

Результат:

```
5,6,7,8
```

Метод *sort*

Синтаксис:

массив.**sort**(*функция*)

Аргумент: *функция* — функция сортировки, описанная далее.

Результат: массив.

Метод `sort` сортирует элементы массива. При этом новый массив не создается, перестановка элементов производится в исходном массиве. Способ сортировки задается необязательным аргументом *функция*. Если аргумента нет, то производится сортировка в алфавитном порядке по возрастанию значений элементов, которые предварительно преобразуются в строки.

Функция должна иметь вид:

```
function compare(a, b) {
  if (a меньше b по критерию сортировки)
    return -1;
  if (a больше b по критерию сортировки)
    return 1;
 return 0;  // a равно b
}
```

Пример сортировки массива по убыванию значений элементов:

```
function cmp(a, b) {
  if (String(a) > String(b))
    return -1;
  if (String(a) < String(b))
    return 1;
  return 0;
}
var flowers = ["белый", "красный", "зеленый", "синий"];
flowers.sort(cmp)
```

Результат:

```
белый, зеленый, красный, синий
```

Метод *splice*

Синтаксис:

массив.**splice**(*начало*, *счетчик*[, *элементы*])

Аргументы: *начало* и *счетчик* — любые числовые выражения, *элементы* — список любых выражений через запятую.

Результат: новый массив.

Метод splice удаляет часть массива и возвращает ее в качестве результата. Если заданы *элементы*, то они вставляются вместо удаленной части массива. Аргумент *начало* задает индекс первого элемента удаляемой части, аргумент *счетчик* — количество удаляемых элементов.

Пример:

```
var x = new Array(10);
for (i = 0; i < 10; i++)
  x[i] = i;
x.splice(5, 3, -5, -6, -7);
```

Результат:

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

Метод *toString*

Синтаксис:

массив.**toString**()

Результат: строковое значение.

Метод toString преобразует массив в строковое значение. Для этого все элементы массива преобразуются в строки, и эти строки объединяются в одну строку через запятую.

Пример:

```
var x = new Array(1, 2, 3);
x.toString())
```

Результат:

```
1,2,3
```

Метод *unshift*

Синтаксис:

массив.**unshift**(*элемент1*, ..., *элементN*)

Аргументы: *элемент1*, ..., *элементN* — любые выражения.

Результат: новая длина массива.

Метод unshift добавляет значения аргументов в начало массива и возвращает в качестве результата новый размер массива, который при этом увеличивается на N. Пример:

```
var x = ["a", "b", "c", "d"]
x.unshift("e")
```

Результат:

```
5
```

П2.5.3. Функции

Функция в JavaScript — это набор операторов, выполняющих определенную задачу.

Для того чтобы пользоваться функцией, мы должны сначала ее определить. Декларация функции имеет вид:

```
function имя(аргументы) {
  операторы
}
```

Здесь *имя* — идентификатор, задающий имя функции, *аргументы* — необязательный список идентификаторов, разделенных запятыми, который содержит имена формальных аргументов функции, а *операторы* — любой набор операторов, который называется *телом функции* и исполняется при ее вызове.

Рассмотрим следующий пример:

```
function cube(number) {
  return number * number * number;
}
```

Эта функция называется `cube` и имеет аргумент `number`. При вызове этой функции вместо формального аргумента подставляется его фактическое значение, функция выполняет возведение этого значения в куб и возвращает полученное число оператором `return`.

Переменные, декларированные в теле функции, являются локальными, т. е. недоступны вне ее тела.

Вызов функции

Важно понимать, что появление декларации функции в тексте сценария не означает ее немедленного выполнения; тело функции будет выполняться только тогда, когда какой-либо оператор будет содержать вызов этой функции. Например, функция из предыдущего примера может быть вызвана так:

```
var x = cube(5);
```

В результате переменная `x` получит значение `125`.

Рекурсивные функции

Важной особенностью языка JavaScript является то, что функция может вызывать не только другие функции, но и саму себя. Такие функции называются *рекурсивными*; во многих случаях использование рекурсии позволяет писать краткий код вместо сложных вложенных циклов. Следует, однако, учитывать, что рекурсия работает медленнее, чем обычный цикл, и пользоваться ею целесообразно только в тех случаях, когда это действительно оправдано.

Пример функции, вычисляющей факториал числа (факториал числа *n* равен 1×2×...×*n*):

```
function factorial(n) {
  if (n <= 1)
    return 1;
  else
    return (n * factorial(n-1));
}
```

Неаккуратно написанная рекурсивная функция может войти в бесконечный цикл и никогда не вернуть результата. Например, попытка вычислить факториал отрицательного числа с помощью приведенной функции приведет именно к такому результату.

Оператор *return*

Функции могут (но не обязаны) возвращать значение. Для указания этого значения используется оператор `return`, который имеет две формы:

return *выражение*
return

Первая форма оператора завершает выполнение функции и возвращает значение выражения. Функция, содержащая такой оператор `return`, должна вызываться как часть выражения присваивания, например:

```
x = 2 * cube(a)
```

Вторая форма оператора завершает выполнение функции и возвращает значение `undefined`. Функция, содержащая такой оператор `return`, должна вызываться как оператор, например `setBrowser(myBrowser)`.

Если тело функции не содержит оператора `return`, то ее выполнение завершается с выполнением последнего оператора тела и возвращается значение `undefined`.

Оператор всегда возвращает единственное значение. Это значит, что в случае, если нужно передать несколько значений, их потребуется заносить либо в массив, либо создавать собственный объект, свойствами которого объявлять необходимые переменные.

```
temp = new Array()
temp.push[var1]
temp.push[var2]
return temp
```

или

```
return myObj = [var1 : value1, var2 : value2]
```

Второй способ более предпочтителен, поскольку вы сразу же задаете переменным осмысленные значения, по которым впоследствии будете их вызывать (`myObj.var1`, `myObj.var2`). В первом случае придется либо потом задавать осмысленные значения, что приведет к увеличению кода, либо пользоваться малозначащими `temp[0]`, `temp[1]` и т. п.

П2.5.4. Объект *Math*

Объект `Math` обеспечивает доступ к различным математическим константам и функциям. Свойства объекта `Math` перечислены в табл. П2.8, а методы — в табл. П2.9.

Таблица П2.8. *Свойства объекта* `Math`

Свойство	Описание
`E`	Основание натуральных логарифмов e
`LN10`	Число *ln* 10
`LN2`	Число *ln* 2
`LOG10E`	Число *lg* e
`LOG2E`	Число $log_2 e$
`PI`	Число π
`SQRT1_2`	Квадратный корень из 1/2
`SQRT2`	Квадратный корень из 2

Таблица П2.9. *Методы объекта* `Math`

Метод	Описание
`abs`	Возвращает абсолютную величину аргумента
`acos`	Возвращает арккосинус аргумента
`asin`	Возвращает арксинус аргумента
`atan`	Возвращает арктангенс аргумента
`atan2`	Возвращает арктангенс частного от деления аргументов
`ceil`	Возвращает наименьшее целое число, большее или равное аргументу
`cos`	Возвращает косинус аргумента

Таблица П2.9 *(окончание)*

Метод	Описание
exp	Возвращает экспоненту аргумента
floor	Возвращает наибольшее целое число, меньшее или равное аргументу
log	Возвращает натуральный логарифм аргумента
max	Возвращает наибольший из аргументов
min	Возвращает наименьший из аргументов
pow	Возводит первый аргумент в степень, заданную вторым
random	Генерирует случайное число в диапазоне от 0 до 1
round	Округляет аргумент до ближайшего целого числа
sin	Возвращает синус аргумента
sqrt	Возвращает квадратный корень из аргумента
tan	Возвращает тангенс аргумента

П2.6. Регулярные выражения

Регулярные выражения представляют собой шаблоны для поиска заданных комбинаций символов в текстовых строках (такой поиск называется *сопоставлением с образцом*). В отличие от прямого поиска, использование регулярных выражений дает гораздо большую гибкость, поскольку достаточно указать лишь какое-то общее правило (например, строка должна начинаться с "начало" и/или заканчиваться "конец"), при этом все фрагменты, удовлетворяющие этому правилу, будут найдены. Существуют два способа присваивания переменным регулярных выражений:

```
var re = /pattern/switch
var re = new RegExp("pattern"[,"switch"])
```

Здесь *pattern* — регулярное выражение, а *switch* — необязательные опции поиска.

Инициализаторы объекта, например, `var re = /ab+c/`, применяют в тех случаях, когда значение регулярного выражения остается неизменным во время работы сценария. Такие регулярные выражения выполняются быстрее.

П2.6.1. Специальные символы в регулярных выражениях

Специальные символы описаны в табл. П2.10.

Таблица П2.10. *Символы в регулярных выражениях*

Символ	Описание
\	Для символов, которые обычно трактуются буквально, означает, что следующий символ является специальным. Например, /n/ соответствует букве n, а /\n/ соответствует символу перевода строки. Для символов, которые обычно трактуются как специальные, означает, что символ должен пониматься буквально. Например, /^/ означает начало строки, а /\^/ соответствует просто символу ^. /\\/ соответствует обратной косой черте \
^	Соответствует началу строки
$	Соответствует концу строки
*	Соответствует повторению предыдущего символа ноль или более раз
+	Соответствует повторению предыдущего символа один или более раз
?	Соответствует повторению предыдущего символа ноль или один раз
.	Соответствует любому символу, кроме символа новой строки
(*pattern*)	Соответствует строке *pattern* и запоминает найденное соответствие
(?:*pattern*)	Соответствует строке *pattern*, но не запоминает найденное соответствие. Используется для группировки частей образца, например, /ко(?:т|шка)/ — это краткая запись выражения /кот|кошка/
(?=*pattern*)	Соответствие с "заглядыванием вперед", происходит при соответствии строки *pattern* без запоминания найденного соответствия. Например, /Windows (?=95|98|NT|2000)/ соответствует "Windows" в строке "Windows 98", но не соответствует в строке "Windows 3.1". После сопоставления поиск продолжается с позиции, следующей за найденным соответствием, без учета заглядывания вперед
(?!*pattern*)	Соответствие с "заглядыванием вперед", происходит при несоответствии строки *pattern* без запоминания найденного соответствия. Например, /Windows (?!95|98|NT|2000)/ соответствует "Windows" в строке "Windows 3.1", но не соответствует в строке "Windows 98". После сопоставления поиск продолжается с позиции, следующей за найденным соответствием, без учета заглядывания вперед
x|y	Соответствует x или y
{n}	n — неотрицательное число. Соответствует ровно n вхождениям предыдущего символа
{n,}	n — неотрицательное число. Соответствует n или более вхождениям предыдущего символа. /x{1,}/ эквивалентно /x+/. /x{0,}/ эквивалентно /x*/

Таблица П2.10 *(продолжение)*

Символ	Описание
{n,m}	n и m — неотрицательное число. Соответствует не менее чем n и не более чем m вхождениям предыдущего символа. /x{0,1}/ эквивалентно /x?/
[xyz]	Соответствует любому символу из заключенных в квадратные скобки
[^xyz]	Соответствует любому символу, кроме заключенных в квадратные скобки
[a-z]	Соответствует любому символу в указанном диапазоне
[^a-z]	Соответствует любому символу, кроме лежащих в указанном диапазоне
\b	Соответствует границе слова, т. е. позиции между словом и пробелом или переводом строки
\B	Соответствует любой позиции, кроме границы слова
\cX	Соответствует символу, генерируемому нажатием комбинации клавиш <Ctrl>+<X>. Например, /\cI/ эквивалентно /\t/
\d	Соответствует цифре. Эквивалентно [0-9]
\D	Соответствует нецифровому символу. Эквивалентно [^0-9]
\f	Соответствует символу перевода формата (FF)
\n	Соответствует символу перевода строки (LF)
\r	Соответствует символу возврата каретки (CR)
\s	Соответствует символу пробела. Эквивалентно /[\f\n\r\t\v]/
\S	Соответствует любому непробельному символу. Эквивалентно /[^ \f\n\r\t\v]/
\t	Соответствует символу табуляции
\v	Соответствует символу вертикальной табуляции (VT)
\w	Соответствует латинской букве, цифре или подчеркиванию. Эквивалентно /[A-Za-z0-9_] /
\W	Соответствует любому символу, кроме латинской буквы, цифры или подчеркивания. Эквивалентно /[^A-Za-z0-9_] /
\n	n — положительное число. Соответствует n-й запомненной подстроке. Вычисляется путем подсчета левых круглых скобок. Если левых скобок до этого символа меньше, чем n, то эквивалентно \0n
\0n	n — восьмеричное число, не большее 377. Соответствует символу с восьмеричным кодом n. Например, /\011/ эквивалентно /\t/
\xn	n — шестнадцатеричное число, состоящее из двух цифр. Соответствует символу с шестнадцатеричным кодом n. Например, /\x31/ эквивалентно /1/

Таблица П2.10 *(окончание)*

Символ	Описание
\un	n — шестнадцатеричное число, состоящее из четырех цифр. Соответствует символу Unicode с шестнадцатеричным кодом n. Например, /\u00A9/ эквивалентно /©/

Регулярные выражения вычисляются аналогично остальным выражениям JavaScript, т. е. с учетом приоритета операций: операции, имеющие больший приоритет, выполняются первыми. Если операции имеют равный приоритет, то они выполняются слева направо. Далее приведен список операций регулярных выражений в порядке убывания их приоритетов; операции, расположенные в одной строке, имеют равный приоритет:

1. \
2. () (?:) (?=) (?!) []
3. * + ? . {n} {n,} {n,m}
4. ^ $ \метасимвол
5. |

П2.6.2. Опции поиска

При создании регулярного выражения можно указать дополнительные опции поиска:

- i (ignore case) — не различать строчные и прописные буквы;
- g (global search) — глобальный поиск всех вхождений образца;
- m (multiline) — многострочный поиск;
- любые комбинации этих трех опций, например ig или gim.

При создании регулярного выражения следует учитывать, что заключение его в кавычки влечет за собой необходимость использовать escape-последовательности, как и в любой другой строке.

Если нам нужно просто проверить, содержит ли данная строка подстроку, соответствующую образцу, то используются методы test или search.

Если же нам необходимо извлечь подстроку (или подстроки), соответствующие образцу, то нам придется воспользоваться методами exec или match.

Метод replace обеспечивает поиск заданной подстроки и замены ее на другую строку, а метод split позволяет разбить строку на несколько подстрок, основываясь на регулярном выражении или обычной текстовой строке.

П2.6.3. Свойства

Свойства *$1*, ..., *$9*

Синтаксис:

`RegExp.$n`

Если часть регулярного выражения заключена в круглые скобки, то соответствующая ей подстрока запоминается для последующего использования. Значениями свойств `$1`, ..., `$9` являются подстроки исходной строки, которые были запомнены в процессе последнего сопоставления с образцом. Регулярное выражение может содержать любое количество выражений в круглых скобках, но в объекте `RegExp` запоминаются только последние девять найденных соответствий.

Пример:

```
var re = new RegExp("(\\d*)\\s*(\\d*)","ig");
var arr = re.exec("111 2222 33333");
var s = "$1 = '" + RegExp.$1 + "' ";
s += "$2 = '" + RegExp.$2 + "' ";
s += "$3 = '" + RegExp.$3 + "'";
```

Результат:

```
$1 = '111' $2 = '2222' $3 = ''
```

Эти свойства объекта `RegExp` изменяются при каждой операции сопоставления с регулярным выражением. В методе `string.replace()` они употребляются без имени объекта `RegExp`.

Свойство *input*

Синтаксис:

`RegExp.input`

Значением свойства `input` является последняя исходная строка, к которой применялось сопоставление с образцом. Это свойство объекта `RegExp` изменяется при каждой операции сопоставления с регулярным выражением.

Пример:

```
var re = new RegExp("\\d+", "g");
var arr = re.exec("111 2222 33333");
currentTextFrame.contents = RegExp.input;
```

Результат:

```
111 2222 33333
```

Свойство *lastIndex*

Синтаксис:

```
RegExp.lastIndex
```

Значением свойства `lastIndex` является целое число, содержащее номер элемента строки, с которого начнется следующее сопоставление с образцом. При создании объекта `RegExp` этому свойству присваивается значение 0. Оно используется только в тех случаях, когда включена опция глобального поиска (т. е. свойство `global` имеет значение `true`).

Пример:

```
var re = /(aha)/g;
var a = re.exec("aha");  // a равно ["aha", "aha"],
re.lastIndex = 3
var a = re.exec("aha");  // a равно [""],
re.lastIndex = 3
re.lastIndex = 0;        // повторить поиск
var a = re.exec("aha");  // a равно ["aha", "aha"],
re.lastIndex = 3
```

Свойство *lastMatch*

Синтаксис:

```
RegExp.lastMatch
```

Значением свойства `lastMatch` является последняя найденная подстрока исходной строки. Это свойство объекта `RegExp` изменяется при каждой операции сопоставления с регулярным выражением.

Пример:

```
var re = new RegExp("\\d+", "g");
var arr = re.exec("111 2222 33333");
currentTextFrame.contents = RegExp.lastMatch
```

Результат:

```
111
```

Свойство *lastParen*

Синтаксис:

```
RegExp.lastParen
```

Значением свойства `lastParen` является последняя запомненная подстрока исходной строки, соответствующая подвыражению регулярного выражения, заключенному в круглые скобки.

Пример:

```
var re = new RegExp("(\\d+) (\\d+)", "g");
var arr = re.exec("111 2222 33333");
currentTextFrame.contents = RegExp.lastParen
```

Результат:

```
2222
```

Свойство *leftContext*

Синтаксис:

```
RegExp.leftContext
```

Значением свойства `leftContext` является подстрока исходной строки, предшествующая последней найденной подстроке.

Пример:

```
var arr = / (\d+)/.exec("111 2222 33333");
currentTextFrame.contents = RegExp.leftContext + "|" + RegExp.lastMatch +
"|" + RegExp.rightContext
```

Результат:

```
111| 2222| 33333
```

Свойство *rightContext*

Синтаксис:

```
RegExp.rightContext
```

Значением свойства `rightContext` является подстрока исходной строки, следующая за последней найденной подстрокой.

Пример:

```
var arr = / (\d+)/.exec("111 2222 33333");
currentTextFrame.contents = RegExp.leftContext + "|" + RegExp.lastMatch +
"|" + RegExp.rightContext
```

Результат:

```
111| 2222| 33333
```

П2.6.4. Методы

Метод *match*

Возвращает массив, состоящий из символов, соответствующих регулярному выражению `RegExp`. В случае, если искомый текст найден не был, возвращает `null`.

Пример:

```
var txt = "скрипты значительно облегчают верстку в InDesign"
result = txt.match(/InDesign/)
```

Результат:

```
result = true
```

Метод *search*

Возвращает позицию первой подстроки, соответствующей регулярному выражению `RegExp`. Возвращаемые значения — от –1 (если соответствие не найдено) до длины строки минус 1.

Пример:

```
var txt = "скрипты значительно облегчают верстку в InDesign"
result = txt.search(/InDesign/)
```

Результат:

```
result = 40
```

Пример:

```
txt.search(/Adobe/)
```

Результат:

```
"-1"
```

Таким образом, для поиска теста с одинаковым успехом можно использовать либо

```
match(/'искомое выражение'/) != null
```

либо

```
search(/"искомое выражение"/) != -1
```

Метод *replace*

Синтаксис:

объект.**replace**(*RegExp*,*строка*)
объект.**replace**(*RegExp*,*функция*)

Аргументы: *RegExp* — регулярное выражение, *строка* — строка, *функция* — имя функции.

Результат: новая строка.

Метод `replace()` сопоставляет регулярное выражение *RegExp* со строкой и заменяет найденные подстроки другими подстроками. Результатом является новая строка, которая является копией исходной строки с проведенными за-

менами. Способ замены определяется опцией глобального поиска в *RegExp* и типом второго аргумента.

Если *RegExp* не содержит опцию глобального поиска, то выполняется поиск первой подстроки, соответствующей *RegExp* и производится ее замена. Если *RegExp* содержит опцию глобального поиска, то выполняется поиск всех подстрок, соответствующих *RegExp*, и производится их замена.

Если вторым аргументом является строка, то замена каждой найденной подстроки производится на нее. При этом строка может содержать такие свойства объекта RegExp, как $1, ..., $9, lastMatch, lastParen, leftContext и rightContext.

Пример:

```
("Adobe Indesign, Adobe Photoshop").replace(/Adobe/g, "Quark")
```

Результат:

```
"Quark Indesign, Quark Photoshop"
```

Если вторым аргументом является функция, то замена каждой найденной подстроки производится вызовом этой функции. Функция имеет следующие аргументы. Первый аргумент — это найденная подстрока, затем следуют аргументы, соответствующие всем подвыражениям *RegExp*, заключенным в круглые скобки, предпоследний аргумент — это позиция найденной подстроки в исходной строке, считая с нуля, и последний аргумент — это сама исходная строка.

Следующий пример показывает, как с помощью метода replace можно написать функцию преобразования градусов Фаренгейта в градусы Цельсия. Приведенный сценарий:

```
function myfunc($0,$1) {
  return (($1-32) * 5 / 9) + "C";
}
function FahrToCelsium(x) {
  var s = String(x);
  return s.replace(/(\d+(\.\d*)?)F\b/, myfunc);
}
currentTextFrame.contents = FahrToCelsium("212F")
```

Результат:

```
100C
```

Если часть регулярного выражения заключена в круглые скобки, то соответствующая ей подстрока будет запомнена для последующего использования. Для доступа к запомненным подстрокам используются свойства $1, ..., $9 объекта RegExp.

Метод *exec*

Синтаксис:

`RegExp.`**`exec`**`(`*`строка`*`)`

Аргументы: *строка* — любая строка.

Результат: массив результатов или `null`.

Метод `exec` выполняет сопоставление строки с образцом, заданным `RegExp`. Если сопоставление с образцом закончилось неудачей, то возвращается значение `null`. В противном случае результатом является массив подстрок, соответствующих заданному образцу.

Результирующий массив имеет следующие свойства:

- свойство `input` содержит исходную строку;
- свойство `index` содержит позицию найденной подстроки в исходной строке;
- свойство `length` равно $n + 1$, где n — количество подвыражений регулярного выражения, заключенных в круглые скобки;
- элемент 0 содержит найденную подстроку;
- элементы 1, ..., n содержат подстроки, соответствующие подвыражениям регулярного выражения в круглых скобках.

Пример:

```
var arr = /(\d+)\.(\d+)\.(\d+)/.exec("Я родился 21.05.1958");
alert("Дата рождения: ", arr[0], "<br>");
alert ("День рождения: ", arr[1], "<br>");
alert ("Месяц рождения: ", arr[2], "<br>");
alert ("Год рождения: ", arr[3], "<br>");
```

Результат:

```
Дата рождения: 21.05.1958
День рождения: 21
Месяц рождения: 05
Год рождения: 1958
```

Включение в регулярное выражение опции глобального поиска позволяет многократно применять этот метод к исходной строке для последовательного выделения всех подстрок, соответствующих данному образцу.

Пример:

```
var re = /\d+/g;
var s = "123 abc 456 def 789 xyz";
var result;
```

```
while (result = re.exec(s))
  alert (result[0] + " ");
```

Результат: `123 456 789`.

Метод *test*

Синтаксис:

`RegExp.`**`test`**`(`*`строка`*`)`

Аргумент: *строка* — любая строка.

Результат: логическое значение.

Метод `test` выполняет сопоставление строки с образцом, заданным `RegExp`, и возвращает `true`, если сопоставление прошло успешно, и `false` — в противном случае. Этот метод эквивалентен выражению `RegExp.exec(строка) != null`.

Пример:

```
if (!/(\d+)\.(\d+)\.(\d+)/.test(str))
  alert("Неверное значение строки!");
```

П2.7. Комментарии

Комментарии служат для документирования тех или иных действий непосредственно в теле скрипта. Комментарий длиной в одну строчку предваряется двойным слэшем (`//`), если их длина больше — заключается в комбинацию символов: `/*` перед и `*/` после него. Такие комментарии не могут быть вложенными (т. е., например, `/* .../*.../*` недопустимо).

Пример комментария в одну строку:

```
// Таким образом оформленный комментарий не может превышать одной строки
```

Пример многострочного комментария:

```
/* Текст в многострочном комментарии может быть сколь угодно длинным
   и занимать хоть несколько экранов */
```

ПРИЛОЖЕНИЕ 3

Зарезервированные слова

В JavaScript существует определенный набор слов, которые являются специальными, а потому их применение в качестве названий переменных, функций, методов не допускается:

abstract	double	instanceof	static
boolean	else	int	super
break	enum	interface	switch
byte	export	label	synchronized
case	extends	long	this
catch	false	native	throw
char	finally	new	transient
class	float	null	true
comment	for	package	try
const	function	private	typeof
continue	goto	protected	var
debugger	if	public	void
default	implements	return	while
delete	import	short	with
do	in		

ПРИЛОЖЕНИЕ 4

ExtendScript Editor

ExtendScript Editor — приложение, предназначенное для отладки скриптов (пошагового выполнения инструкций), которые написаны для программ, разработанных Adobe. В нем можно редактировать скрипты как под Photoshop, так и под Illustrator, InDesign. В процессе отладки отслеживаются значения переменных, прохождение циклов и выполнение заданных условий, чтобы обеспечить корректность выполнения скрипта.

П4.1. Установка режима отладки

Для управления глобальной активностью отладчика предназначены специальные команды, которые вставляются непосредственно в скрипт:

```
// Отключение отладчика в случае возникновения ошибок
$.level = 0;

// Активизация отладчика перед выполнением скрипта
$.level = 1;

// Активизация отладчика только в случае появления ошибки
$.level = 2;
```

Для задания точки останова (breakpoint) служат инструкции `debugger` и `$.bp()`. При достижении такой строки выполнение скрипта приостанавливается. Расширяет функциональность данных инструкций указание условий, при каких пауза должна наступать.

Например:

```
$.bp( someValue == 0 );
```

В этой точке только в том случае произойдет останов, если `someValue` на момент прохождения этой строки будет равно `0`. При наступлении события за-

пустится отладчик, и вы увидите три панели, предназначенные для тестирования скрипта.

П4.2. Интерфейс

В InDesign Creative Suite 2 возможности ExtendScript Editor значительно выше, чем в предыдущей версии пакета, поэтому настоятельно рекомендуется использовать как минимум InDesign CS2 (а еще лучше — отладчик из Creative Suite 3). К сожалению, простая установка отладчика из Creative Suite 3 (без установки всего пакета) не позволяет пользоваться преимуществами связки InDesign CS2 и ExtendScript Editor CS3.

Интерфейс программы можно условно разделить на три функциональные зоны (рис. П4.1):

- управляющая зона (самая верхняя строчка). Состоит из окна выбора движка (под каждое приложение он свой, для InDesign нужно выбрать **InDesign**) — крайний левый раскрывающийся список — и кнопок для управления прохождением скрипта;
- главное окно, в котором отображается код с подсветкой синтаксиса;
- информационная зона (крайний слева столбец с панелями), состоит из нескольких панелей, предназначенных собственно для отладки.

Порядок функций, которые были вызваны до точки останова (breakpoint), отображается на вкладке **Call Stack** в левом верхнем углу окна отладчика. Это помогает определить, какие модули (функции) были задействованы, и при хорошо продуманном коде, разбитом на относительно короткие функции, позволяет быстрее обнаружить проблему в логике исполнения.

В панели **Data Browser** отображается вся доступная информация не только об объектах InDesign, но также и пользовательских переменных, что позволяет в любой момент определить значение того или иного свойства у выбранного объекта или переменной. Панель чрезвычайно полезна при отладке.

Панель **JavaScript Console** расширяет возможности **Data Browser**, позволяя выполнять "на лету" некоторые преобразования, а также получать значения свойств конструкций, которые не были объявлены в теле скрипта как переменные (`var`). Кроме того, через нее удобнее вызывать переменные — особенно в том случае, если дерево всех доступных объектов и переменных слишком велико, и приходится тратить значительное количество времени на прокрутку при поиске необходимой информации.

В InDesign существует 6 кнопок, предназначенных для управления исполнением скрипта.

- **Resume** — эта кнопка восстанавливает продолжение исполнения скрипта после достижения точки останова. Кнопка становится активной, когда скрипт приостановлен (**Pause**) или же остановлен совсем (**Stop**).

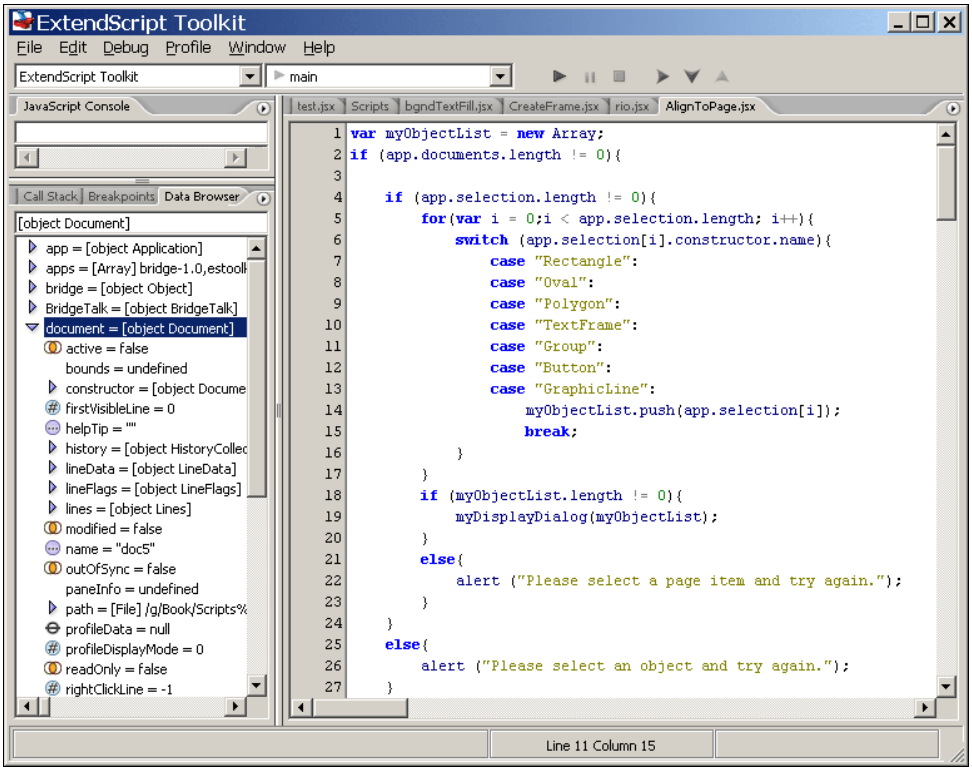


Рис. П4.1. Основное окно отладчика

- **Pause** — эта кнопка приостанавливает на время исполнение текущего скрипта и активизирует окно отладчика. Кнопка активна, когда скрипт выполняется.
- **Stop** — кнопка останавливает дальнейшее выполнение скрипта. Активна, когда скрипт работает.
- **Step into** — кнопка приостанавливает выполнение скрипта после выполнения следующей строчки, в том числе и внутри функций.
- **Step over** — основное отличие данной кнопки от предыдущей в том, что приостановка скрипта происходит только после выполнения всей функции, т. е. отладить с ее помощью скрипт, содержащий функции, как вариант — использовать ее совместно со **Step into**.
- **Step out** — если отладчик был приостановлен в момент выполнения функции, эта кнопка продолжит исполнение скрипта вплоть до конца функции. Если останов произошел вне тела функции, кнопка позволяет выполнять оставшийся код.

ПРИЛОЖЕНИЕ 5

Работа с файловой системой

П5.1. Объект *Path*

При создании объектов типа `File` (файл) или `Folder` (папка) можно использовать как платформенно-независимый способ задания путей, так и адаптированный под конкретную платформу. В последнем случае допускается задание абсолютного пути либо относительного. При задании пути вручную обязательно учитывайте регистр символов, поскольку `myFile` и `MyFile` (как и любые другие переменные в JavaScript) являются ссылками на совершенно разные файлы.

Абсолютный путь — это полное перечисление всего пути, начиная с корневого каталога, при этом папки разделяются (в Windows — символами `/` либо `\`, в Mac OS — символами `/` либо `:`). Относительный путь — это путь относительно текущего каталога (свойство `Folder.current`). Наиболее оптимальный вариант — первоначальное задание абсолютного пути для текущего каталога, а в последующем использование только относительных путей. При этом повышается читабельность кода и в случае изменения расположения скрипта достаточно будет внести изменения только в одном месте.

Несколько примеров задания путей представлено в табл. П5.1.

Таблица П5.1. *Примеры использования разных типов задания путей*

Пример	Описание
myFile.jsx ./myFile.jsx	В текущем каталоге
../myFile.jsx	В каталоге на предыдущем уровне
../../myFile.jsx	В каталоге на два уровня выше
../dir1/myFile.jsx	В папке dir1 на текущем уровне

Для получения названия папки или файла требуется конвертировать соответствующий объект в строку посредством метода `toString()`.

Для удобства можно использовать возможности личных папок. В Windows по умолчанию — C:\Documents and Settings*username_folder*, в MacOS — /Users/*username*). В таком случае в названии первым символом должен стоять знак ~ (тильда).

В названиях путей или файлов можно использовать специальные символы (`/`, `-`, `.`, `!`, `~`, `*`, `'`, `(`, `)`), предваряя их особым знаком (`%`) и задавая их код в шестнадцатеричном виде.

П5.2. Объект *File*

Создать в InDesign новый файл (объект `File`) можно двумя вариантами:

```
File ([path])
new File ([path])
```

В любом случае возвращается ссылка на папку, содержащую файл.

Общие замечания: если помечено R, то свойство открыто только для чтения, если R/W — также и для записи

Параметры в квадратных скобках (`[]`) — не обязательные.

Свойства объекта `File` перечислены в табл. П5.2, а методы — в табл. П5.3.

Таблица П5.2. *Свойства объекта* `File`

Свойство	Тип	Атрибут	Описание
`alias`	`Boolean`	R	Если `true`, объект ссылается на ярлык
`created`	`Date`	R	Дата создания файла
`encoding`	`String`	R	Кодировка
`eof`	`Boolean`	R	Конец файла
`error`	`String`	R	Сообщение, описывающее ошибку файловой системы
`exists`	`Boolean`	R	Если `true`, объект существует
`fsName`	`String`	R	Платформенно-зависимый полный путь к файлу
`hidden`	`Boolean`	R/W	Файл скрыт для отображения в системном браузере
`length`	`Number`	R	Размер файла в байтах
`lineFeed`	`String`	R/W	Тип символа переноса строки: `windows` для Windows, `mac` для Mac OS
`modified`	`Date`	R	Дата модификации файла

Таблица П5.2 *(окончание)*

Свойство	Тип	Атрибут	Описание
`name`	`String`	R	Имя файла, без пути
`parent`	`Folder`	R	Папка, в которой расположен файл
`path`	`String`	R	Путь к файлу без его имени
`readonly`	`Boolean`	R/W	Если `true`, файл не может быть изменен или удален
`type`	`String`	R	Только в Mac OS: тип файла

Таблица П5.3. *Методы объекта* `File`

Метод	Описание
`close()`	Закрыть файл. В случае успеха возвращает `true`, иначе — `false`
`copy(`*`target`*`)`	Копировать файл в другое место. Если файл с таким именем существует, он перезаписывается. В случае успеха возвращает `true`, иначе — `false`. Здесь *`target`* — путь к файлу или сам файл
`createAlias(`*`toFile`*`)`	Создать ярлык
`execute()`	Открыть файл в соответствующем приложении (аналог двойного щелчка в Проводнике)
`open(`*`mode`*`[, `*`type`*`] [, `*`creator`*`])`	Открыть файл в определенном режиме. Допустимые режимы: • `r` (read) — только для чтения; • `w` (write) — для записи; • `e` (edit) — для чтения и записи. Необязательные параметры — только для Mac OS: • *`type`* — тип созданного файла; • *`creator`* — программа-создатель файла
`openDlg([`*`prompt`*`] [, `*`select`*`])`	Отобразить диалоговое окно для открытия файла. Необязательные параметры: • *`prompt`* — текст, который будет отображаться в диалоговом окне; • *`select`* — разделенный точкой с запятой список типов файлов, которые будут отображаться в диалоговом окне
`read([`*`char`*`])`	Прочитать содержимое файла. Необязательный параметр `char` — количество символов, которые будут прочитаны

Таблица П5.3 *(окончание)*

Метод	Описание
`readln()`	Прочитать строку файла
`remove()`	Удалить файл. Происходит в обход Корзины. В случае успеха возвращает `true`. В случае удаления ярлыка удаляет именно ярлык, а не связанный с ним файл
`rename(`*`newName`*`)`	Переименовать файл на *newName*. Здесь *newName* — это новое имя файла или папки, без пути. В случае успеха возвращает `true`
`resolve()`	Определить реальный файл по его ярлыку. Возвращает `null`, если файл не найден
`saveDlg([`*`prompt`*`][, `*`select`*`])`	Отобразить диалоговое окно для сохранения файла. Необязательные параметры: • *prompt* — текст, который будет отображаться в диалоговом окне; • *select* — разделенный точкой с запятой список типов файлов, которые будут отображаться в диалоговом окне
`seek(`*`pos`*`, `*`mode`*`)`	Переместить указатель в нужную позицию в файле
`write(`*`text`*`[, `*`text`*`, `*`text`*`])`	Записать текст в файл. В случае успеха возвращает `true`
`writeln(`*`text`*`[, `*`text`*`, `*`text`*`])`	Записать текст в файл и в конце добавить символ перевода строки. В случае успеха возвращает `true`

Работа с этими объектами строится таким образом. Сначала создается объект требуемого типа и через его свойства ему устанавливается ссылка на физический файл на диске, после чего с ним возможны остальные операции (открытие, чтение, изменение содержимого, закрытие и т. п.). При открытии файла одним объектом операционная система блокирует доступ к нему для других объектов.

П5.3. Объект *Folder*

Создать в InDesign новую папку (объект `Folder`) можно двумя вариантами:

```
Folder([path])
new Folder([path])
```

В любом случае возвращается ссылка на папку, содержащую файл.

Свойства объекта `Folder` перечислены в табл. П5.4, а методы — в табл. П5.5.

Таблица П5.4. *Свойства объекта* `Folder`

Свойство	Тип	Атрибут	Описание
`alias`	`Boolean`	R	Если `true`, объект ссылается на ярлык
`created`	`Date`	R	Дата создания файла
`errror`	`String`	R	Сообщение, описывающее ошибку файловой системы
`exists`	`Boolean`	R	Если `true`, объект существует
`fsName`	`String`	R	Платформенно-зависимый полный путь к файлу
`modified`	`Date`	R	Дата модификации файла
`name`	`String`	R	Имя файла, без пути
`parent`	`Folder`	R	Папка, в которой расположен файл
`path`	`String`	R	Путь к файлу без его имени
`appData`	`Folder`	R	В Windows — расположение папки *%APPDATA%* (по умолчанию, C:\Documents and Settings \All Users\Application Data). В Mac OS — /Library/Application Support
`commonFiles`	`Folder`	R	В Windows — расположение папки *%CommonProgramFiles%* (по умолчанию, C:\Program Files\Common Files). В Mac OS — /Library/Application Support
`current`	`Folder`	R	Текущая папка
`myDocuments`	`Folder`	R	Расположение папки по умолчанию для документов. В Windows — C:\Documents and Settings *username*\My Documents. В Mac OS — ~/Documents
`startup`	`Folder`	R	Расположение папки, из которой запущено приложение
`system`	`Folder`	R	Расположение системной папки. В Windows — *%windir%* (по умолчанию, C:\Windows). В Mac OS — /System
`temp`	`Folder`	R	Папка для хранения временных файлов
`userData`	`Folder`	R	В Windows — расположение папки *%APPDATA%* (по умолчанию, C:\Documents and Settings\All Users\Application Data). В Mac OS — /Library/Application Support

Таблица П5.5. *Методы объекта* `Folder`

Метод	Описание
`createAlias(`*`toFile`*`)`	Создать ярлык
`execute()`	Открыть файл в соответствующем приложении (аналог двойного щелчка в Проводнике)
`getFiles([`*`mask`*`])`	Возвратить файлы и папки, находящиеся в выбранной папке. При указании *`mask`* отображает только соответствующие типы файлов
`remove()`	Удалить папку. Происходит в обход Корзины. В случае успеха возвращает `true`. В случае удаления ярлыка удаляет именно ярлык, а не связанную с ним папку
`rename(`*`newName`*`)`	Сменить текущее название папки на *`newName`*, которое должно быть именем файла или папки, без пути. В случае успеха возвращает `true`
`selectDlg([`*`prompt`*`, `*`preset`*`])`	Открыть диалоговое окно для выбора файла. Если пользователь нажимает кнопку **OK**, возвращает выбранную папку. Если нажата кнопка **Отмена**, возвращает `null`

Пример записи в новый файл пути, по которому он находится, приведен в листинге П5.1.

Листинг П5.1. Запись в файл его пути

```
aD = app.activeDocument
listFolder = Folder.selectDialog("Select folder", aD.filePath)

// listFolder — ссылка на объект Folder. Непосредственно использовать
// его название нельзя, поэтому для превращения его в текст название
// необходимо преобразовать в строку

INIfile = new File(aD.filePath + "/test.txt");
INIfile.open("w");    // Открываем для записи
INIfile.write(listFolder.toString());

// Закрываем файл
INIfile.close();
```

Пример чтения из файла приведен в листинге П5.2.

Листинг П5.2. Чтение из файла

```
INIfile = File(aD.filePath + "/test.txt");  // Указатель на файл

// Считывание файла и передача ссылки на его содержимое переменной
// listFolder для дальнейшего использования
INIfile.open("r");   // Открываем файл для чтения
listFolder = INIfile.read()

// Закрываем файл
INIfile.close();
```

Еще пример чтения из файла (листинг П5.3).

Листинг П5.3. Чтение из файла построчно

```
INIfile = File(aD.filePath + "/test.txt");  // Указатель на файл
INIfile.open("r");    // Открываем файл для чтения

// Считывание содержимого файла построчно и передача его переменной
// listFolder для дальнейшего использования
while (!INIfile.eof){
  listFolder = INIfile.readln()
}

INIfile.close();      // Закрываем файл
```

П5.4. Получение ссылки на скрипт

В случае, если в вашей работе достаточно простора для программирования, и вы полны решимости переложить рутинную работу на "плечи" машин, есть смысл создать собственную библиотеку наиболее часто повторяющихся функций и по мере необходимости подключаться к ним. Это позволит вам повысить читабельность кода и, кроме того, сократит время отладки, поскольку не придется вспоминать, в каком из ранее созданных скриптов вы уже выполняли необходимую операцию и искать ее.

InDesign поддерживает включение кода внешнего скрипта непосредственно в текущий (`include`), при этом используемый язык должен быть одинаковым. То есть если используется JavaScript, то включаемый таким образом скрипт тоже должен быть написан на данном языке. Второй вариант — более универсальный — заключается в подгрузке скрипта, написанного на любом из поддерживаемых InDesign языков — JavaScript, AppleScript, Visual Basic. Это

дает гораздо более широкие возможности по организации автоматизированных рабочих мест, поскольку позволяет использовать в полной мере всю мощь технологии ActiveX (взаимодействие с пакетами MS Office, доступ к Интернету (MSXML2.XMLHTTP), доступ к двоичным базам данных на основе ADODB.Stream и т. д.)

Поскольку, как правило, скрипты хранятся в одном месте, в InDesign предусмотрены возможности, облегчающие поиск из скрипта пути ко всей библиотеке, что необходимо для указания относительного пути к другим скриптам. По аналогии с документом (`activeDocument`) используется свойство `activeScript`.

```
myScript = app.activeScript;
alert("Название скрипта: "+ myScript)

myParentFolder = File(myScript).parent
alert("Расположен в папке: "+ myParentFolder)
```

Свойство имеет смысл только при запуске скрипта с таким кодом из палитры **Scripts**, в ExtendScript обращение к нему выдаст ошибку. Поэтому рекомендуется использовать традиционную для JavaScript конструкцию для обработки ошибок:

```
try {app.activeScript}
catch (err) {File (err.fileName)}
```

Для того чтобы скрипты выполняли возложенные на них задачи, они должны тесно между собой взаимодействовать. Это происходит — сначала — путем передачи параметров из вызывающего скрипта вызываемому, а по окончании работы вызванного — передачей результата вызывающему.

П5.5. Запуск связанного скрипта

Ведомый скрипт может быть запущен из основного через метод `doScript()`.

`doScript(`*непосредственно код скрипта*`) [, `*язык*`][, `*параметры*`])`

Синтаксис использования разных языков:

```
ScriptLanguage.applescript
ScriptLanguage.javascript
ScriptLanguage.visualBasic
```

Пример, иллюстрирующий запуск скрипта из родительского скрипта:

```
var myScriptParameters = new Array("Первый переданный параметр", "Второй
переданный параметр")
var myScript = "alert(\"Первый параметр: \" + arguments[0] + \"Второй
параметр: \" + arguments[1])"
app.doScript(myScript, ScriptLanguage.applescript, myScriptParameters)
```

Правила использования кавычек таковы: кавычки не могут быть вложенными, т. е. выражение

```
"первая + "вторая" "
```

недопустимо. Нужно использовать либо одинарные кавычки (''), что обычно применяется, если вложенность не превышает двух уровней:

```
"первая + 'вторая' "
```

либо использовать символ escape-последовательности (\), что является универсальным вариантом для вложенности любой глубины:

```
"первая + \"вторая\" "
```

Чтобы запустить связанный файл скрипта, предусмотрен метод `doScriptFile()`.

П5.6. Получение результата работы скрипта

Возвращаемый результат хранится в объекте `scriptArgs`. При этом вызванный скрипт сначала записывает в него данные через метод `setValue()`, а потом принимает через `getValue()`.

ПРИЛОЖЕНИЕ 6

Описание компакт-диска

Перечень файлов, размещенных на компакт-диске, прилагаемом к этой книге

Файл	Описание
Материалы к главе 3	
dialog.jsx	Создание диалогового окна
Материалы к главе 4	
Print.jsx	Создание пользовательского набора предустановок для печати
export_sel.jsx	Экспорт выделенных объектов
Transform.jsx	Трансформации объектов
Материалы к главе 5	
Colontitle.jsx	Расстановка скользящих колонтитулов
Материалы к главе 6	
Tabulation.jsx	Установка позиций табуляции
Материалы к главе 7	
Grep.jsx	Автоматический корректор
Материалы к главе 8	
Word.jsx	Форматирование таблицы, импортированной из Word
Материалы к главе 9	
Resolution.jsx	Поиск изображений с разрешением ниже заданного
Import.jsx	Импорт многостраничного PDF-файла
Catalog.jsx	Формирование каталога изображений
Auto.jsx	Автоматическое создание фреймов для иллюстраций

(окончание)

Файл	**Описание**
Материалы к главе 10	
Corners.jsx	Создание угловых эффектов у объектов
Материалы к главе 11	
Idd_1.jsx, ai.jsx	Проверка публикации

Кроме того, методы и свойства основных объектов приведены в одноименном PDF-файле.

Предметный указатель

L

M

O

P

R

S

T

U

V

А

Г

Д

З

И

К

М

О

П

Р

С

Т

Ф

Ц

Ш

Э

Я