Before you start

This file is not for reading, but just for acquaintance with the book you are going to buy. Do not ruin your eyes reading it from a monitor or pda. Just decide is the book worth your money. Buy hardcovered books, support authors and publishers.

Like the book? Buy it!

Перед тем, как вы начнете

Этот файл предназначен не для чтения, а лишь для ознакомления с книгой, которую вы собираетесь приобрести. Не портите своё зрение, читая его с экрана монитора или PDA. Вам дана возможность решить, стоит ли книга ваших денег. Покупайте реальные книги, поддержите авторов и издательства.

Понравилась книга? Купите её!

Данный файл представлен исключительно в ознакомительных целях. После ознакомления с содержанием данного файла Вам следует его незамедлительно удалить. Сохраняя данный файл, вы несете ответственность в соответствии с законодательством.
Любое коммерческое и иное использование кроме предварительного ознакомления запрещено. Публикация данного документа не преследует за собой никакой коммерческой выгоды.
Эта книга способствует профессиональному росту читателей и является рекламой бумажных изданий.
Все авторские права принадлежат их уважаемым владельцам.
Если вы являетесь автором данной книги и её распространение ущемляет Ваши авторские права или если Вы хотите внести изменения в данный документ или опубликовать новую книгу, свяжитесь с нами по e-mail на нашем сайте:

http://web-bookcase.ru/

Вернуться назад
Оглавление диска

Кен Арнольд
Джеймс Гослинг
ЯЗЫК
ПРОГРАММИРОВАНИЯ
JAVA

Книга, выпускаемая по лицензии издательства Addison-Wesley, является каноническим описанием языка программирования Java. Структура книги и стиль изложения напоминают «библию» программистов на C — работу «Язык программирования С» Б. Кернигана и Д. Ричи. Эта аналогия тем более оправдана, что один из авторов «Языка программирования Java» (Д. Гослинг) известен как основной разработчик этого языка. Книга в равной степени может служить учебником и справочником по Java, а многочисленные упражнения позволят читателю попрактиковаться в использовании популярного языка программирования.

ВВЕДЕНИЕ
ГЛАВА 1 ПЕРВОЕ ЗНАКОМСТВО С JAVA
ГЛАВА 2 КЛАССЫ И ОБЪЕКТЫ
ГЛАВА 3 РАСШИРЕНИЕ КЛАССОВ
ГЛАВА 4 ИНТЕРФЕЙСЫ
ГЛАВА 5 ЛЕКСЕМЫ, ОПЕРАТОРЫ И ВЫРАЖЕНИЯ
ГЛАВА 6 ПОРЯДОК ВЫПОЛНЕНИЯ
ГЛАВА 7 ИСКЛЮЧЕНИЯ
ГЛАВА 8 СТРОКИ
ГЛАВА 9 ПОТОКИ
ГЛАВА 10 ПАКЕТЫ
ГЛАВА 11 ПАКЕТ ВВОДА/ВЫВОДА
ГЛАВА 12 СТАНДАРТНЫЕ ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА
ГЛАВА 13 ПРИМЕНЕНИЕ ТИПОВ В ПРОГРАММИРОВАНИИ
ГЛАВА 14 СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ
ПРИЛОЖЕНИЕ А Родные методы
ПРИЛОЖЕНИЕ Б Runtime-исключения в Java
ПРИЛОЖЕНИЕ В Полезные таблицы

Далее

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Книга посвящается команде Java,
чей упорный труд и умение предвидеть будущее
вырастили могучее дерево.
Посвящается Сьюзен - К. А.
Посвящается Джуди и Кейт - Дж. Г.

Об авторах этой книги

Кен Арнольд, ведущий инженер Sun Microsystems Laboratories, является одним из экспертов в области объектно-ориентированных технологий. Он много пишет о программировании на C и C++ для UNIX Review. Кен Арнольд - автор книги «A C User's Guide to ANSI C».

Джеймс Гослинг, член совета директоров и вице-президент Sun Microsystems, является создателем языка Java и одним из самых известных программистов в современном компьютерном мире. В 1996 году ему была присуждена премия «Programming Excellence Award». Ранее он участвовал в разработке NeWS (Network-extensible Window System, сетевой расширяемой оконной системы компании Sun), а также в проекте Andrew в университете Карнеги-Мэллон, где получил докторскую степень.

ПРЕДИСЛОВИЕ

Красивые здания - нечто большее, чем создание науки. Это настоящие живые организмы, постигаемые на духовном уровне; это произведения искусства, в которых современная технология подчиняется вдохновению более, нежели вопросам вкуса и соображениям житейской пользы.
Фрэнк Ллойд Райт

Язык программирования Java (в дальнейшем именуемый просто Java) был тепло встречен мировым сообществом программистов и провайдеров Internet-услуг. Его преимущество для пользователей Internet и World Wide Web заключается в появлении безопасных, платформонезависимых приложений, которые могут использоваться в любом узле Internet. Программисты, создающие приложения на Java, пишут свой код всего один раз - им не приходится «переносить» свои приложения на все возможные программные и аппаратные платформы.

Многие рассматривают Java в первую очередь как средство создания *аплетов* для World Wide Web. Термином «аплет» в Java обозначается мини-приложение, работающее внутри Web-страницы. После того как аплет загружен на компьютер пользователя, он может выполнять определенные задачи и взаимодействовать с пользователем через броузер, не требуя ресурсов Web-сервера. Разумеется, некоторые аплеты могут взаимодействовать с сервером для решения своих внутренних задач, но это их личное дело.

Java является действительно ценным инструментом разработки в распределенных сетевых средах, к которым относится World Wide Web. Тем не менее Java не ограничивается этими рамками и представляет собой мощный универсальный язык программирования, подходящий для создания самых разнообразных приложений, которые либо не зависят от сетевых средств, либо используют их нестандартным образом. Способность Java безопасно выполнять код на удаленных хост-машинах оказалась очень кстати во многих крупных организациях.

Кроме того, Java используется как язык программирования общего назначения для реализации тех проектов, в которых платформенная независимость не так важна. Простота программирования и средства безопасности позволяют быстро создавать отлаженный код. Некоторые распространенные в других языках ошибки в программах на Java вообще не встречаются благодаря таким средствам, как сборка мусора и ссылки, безопасные по отношению к типам. Современные приложения, работающие в сетевых условиях и применяющие графический пользовательский интерфейс, которым приходится одновременно обслуживать несколько задач, используют поддержку многопоточности в Java, а *механизм исключений* облегчает обработку ошибок. Несмотря на всю мощь своих встроенных средств, Java - это *простой* язык, который быстро осваивается программистами.

Язык Java проектировался с расчетом на максимальную переносимость и на минимальную зависимость от конкретной системы. Например, во всех реализациях Java тип int представляет собой 32-разрядное число со знаком, дополняемое по модулю 2, независимо от архитектуры процессора, на котором выполняется Java-программа. Определение всех стандартов, касающихся языка и его runtime-среды, позволяет выполнять скомпилированный код в каком угодно месте и переносить его на любую машину, где присутствует среда Java.

Java обладает многими языковыми средствами, присущими большинству современных языков программирования. Тем не менее, в отличие от C и C++, Java автоматизирует хранение переменных и обработку исключений, дополняя их поддержкой многопоточности.

Об этой книге

Книга обучает программированию на Java и рассчитана на читателей, знакомых с основными концепциями программирования. Язык Java рассматривается в ней без претензий на формальность или полноту описания. Книга не является пособием по объектно-ориентированному программированию, хотя некоторые его аспекты рассматриваются для выработки общей терминологии.

Программистам на C и C++ язык Java должен показаться знакомым, поскольку в нем используются многие конструкции этих языков. Другие книги данной серии, а также большая часть онлайновой документации посвящены программированию аплетов (см. ссылки, приведенные в разделе «Библиография»).

Глава 1 - «Первое знакомство с Java» - содержит краткий обзор Java. Программистам, не владеющим концепциями объектно-ориентированного программирования, следует внимательно прочитать эту главу, а для остальных она станет полезным введением в объектно-ориентированные возможности Java.

В главах 2, 3 и 4 рассматриваются фундаментальные аспекты объектно-ориентированного программирования в Java, а именно объявления классов, их реализация и создание объектов на основе определений классов. Глава 2 - «Классы и объекты» - описывает основы языка Java. Глава 3 - «Расширение классов» - посвящена расширению, или *субклассированию*, существующих классов, в результате которого появляются новые классы со своими данными и другим поведением. Глава 4 - «Интерфейсы» - рассказывает о том, как объявляются интерфейсы, представляющие собой абстрактные описания поведения объектов, обеспечивающие максимальную гибкость для проектировщиков и программистов.

В главах 5 и 6 рассматриваются стандартные языковые конструкции, общие для большинства языков программирования. Глава 5 - «Лексемы, операторы и выражения» - посвящена лексемам языка, его операторам, их использованию для построения выражений и процессу вычислений. Глава 6 - «Порядок выполнения» - показывает, каким образом управляющие операторы изменяют последовательность выполнения операторов в программе.

Глава 7 - «Исключения» - посвящена очень мощному средству Java - обработке исключений. Глава 8 - «Строки» - описывает встроенные языковые и runtime-средства для работы с объектами String.

В главе 9 - «Потоки» - объясняется, как в Java реализована многопоточность. Многие приложения (в частности, те, что основаны на графическом интерфейсе пользователя) должны одновременно обслуживать несколько задач. Чтобы обеспечить их правильное функционирование, необходимо организовать надлежащее взаимодействие между ними. Потоки Java отвечают таким требованиям.

Глава 10 - «Пакеты» - описывает механизм группировки классов Java в отдельные пакеты.

В главах с 11 по 14 рассматриваются основные пакеты, входящие в библиотеку классов Java. Глава 11 - «Пакет ввода/вывода» - описывает систему ввода/вывода. Глава 12 - «Стандартные вспомогательные средства» - посвящена *вспомогательным классам (utility classes)*, к которым относятся векторы и хеш-таблицы. В главе 13 - «Применение типов в программировании» - рассматриваются классы Java, относящиеся к таким типам, которые представляют собой как отдельные объекты, описывающие класс или интерфейс, так и классы, служащие «оболочками» для примитивных типов данных (в частности, целых и вещественных значений). Глава 14 - «Системное программирование» - объясняет, как получить доступ к системным средствам.

Приложение А показывает, каким образом в Java поддерживаются *родные методы (native methods)* - средства для работы с кодом, написанным на «родном» языке программирования базовой платформы.

В приложении Б перечисляются все runtime-исключения и ошибки, возбуждаемые самой системой Java.

Приложение В содержит ряд полезных таблиц, которые пригодятся для справочных целей.

Наконец, в разделе «Библиография» приведены ссылки, которые могут представлять интерес при дальнейшем знакомстве с объектно-ориентированным программированием, многопоточностью и другими темами.

Примеры и документация

Все примеры были откомпилированы и запущены с помощью последней версии языка (FCS-версия Java 1.0.2), существовавшей на момент написания книги. Вообще говоря, нами рассматривается язык Java 1.0.2. Мы также уделили внимание некоторым аспектам, выходящим за пределы написания безошибочно компилируемых программ, - просто пользоваться языком недостаточно, нужно делать это *правильным образом*. Мы попытались объяснить, что такое хороший стиль программирования.

В ряде мест встречаются ссылки на онлайновую документацию. Среда разработки Java позволяет автоматически генерировать файл справки (обычно в формате HTML) по откомпилированному классу с помощью документирующих комментариев. Для просмотра таких файлов обычно используется Web-броузер.

Ни одна техническая книга не пишется «на необитаемом острове», а в нашем случае уместней было бы говорить о целом континенте. Множество людей помогало авторам полезными советами, точными рецензиями, ценной информацией и рекомендациями.

Редактор Генри Мак-Гилтон (Henry McGilton) из Trilithon Software содействовал разрешению всех проблем и внес немалый вклад в создание книги. Редактору серии Лайзе Френдли (Lisa Friendly) мы обязаны беззаветной настойчивостью и поддержкой.

Многие рецензенты выделили свое драгоценное время на чтение, редактирование, пересмотр и удаление материала, и все это ради улучшения книги. Кевин Койл (Kevin Coyle) написал одну из самых подробных рецензий. Карен Беннет (Karen Bennet), Майк Бурати (Mike Burati), Патриция Гинке (Patricia Giencke), Стив Гильяр (Steve Gilliard), Билл Джой (Bill Joy), Розанна Ли (Rosanna Lee), Джон Мэдисон (Jon Madison), Брайан О'Нейл (Brian O'Neill), Сью Палмер (Sue Palmer), Стивен Перелгат (Stephen Perelgut), Р. Андерс Шнайдерман (R. Anders Schneidermann), Сьюзен Сим (Susan Sim), Боб Спраул (Bob Sproull), Гай Стил (Guy Steele), Артур Ван Хофф (Arthur Van Hoff), Джим Уолдо (Jim Waldo), Грег Уилсон (Greg Wilson) и Энн Уолрат (Ann Wollrath) представили нам свои содержательные рецензии. Джефф Арнольд (Geoff Arnold), Том Карджилл (Tom Kargill), Крис Дэрк (Chris Darke), Пэт Финнеган (Pat Finnegan), Майк Джордан (Mike Jordan), Дуг Ли (Doug Lea), Рэндалл Мюррей (Randall Murray), Роджер Риггс (Roger Riggs), Джимми Торрес (Jimmy Torres), Артур Ван Хофф (Arthur Van Hoff) и Фрэнк Йеллин (Frank Yellin) предоставили полезные комментарии и техническую информацию.

Алка Дешпанд (Alka Deshpande), Шерон Фланк (Sharon Flank), Нассим Фотухи (Nassim Fotouhi), доктор К. Калаянасундарам (Dr. K. Kalayanasundaram), Патрик Мартин (Patrick Martin), Пол Романья (Paul Romagna), Сьюзен Снайдер (Susan Snyder) и Николь Янкелович (Nicole Yankelovich) своими совместными усилиями сделали возможными пять слов на стр. , не относящихся к кодировке ISO-Latin-1. Джим Арнольд (Jim Arnold) предоставил информацию о правильном написании, использовании и этимологии слов smoog и moorge. Эд Муни (Ed Mooney) помог подготовить документацию. Херб (Herb) и Джой Кайзер (Joy Kaiser) были нашими консультантами по хорватскому языку. Куки Каллахан (Cookie Callahan), Роберт Пирс (Robert Pierce) и Рита Тавилла (Rita Tavilla) помогли сдвинуть проект с мертвой точки, когда он собирался остановиться всерьез и надолго.

Благодарим Ким Полез (Kim Polese) за краткую формулировку причин, по которым язык Java важен как для пользователей, так и для программистов.

В критические моменты нам помогали своими советами и поддержкой Сьюзен Уолдо (Susan Waldo), Боб Спраул (Bob Sproull), Джим Уолдо (Jim Waldo) и Энн Уолрат (Ann Wollrath). Спасибо нашим семьям, которые не только дарили нас своей любовью, но и не дали нам зачахнуть за работой, за что мы им глубоко благодарны.

Выражаем искреннюю признательность персоналу фирмы Peets Coffee and Tea, который снабжал нас лучшим кофе Java на всей планете.

Любые ошибки или недочеты, оставшиеся в этой книге (несмотря на все усилия перечисленных выше лиц), лежат исключительно на совести авторов.

Содержание | Далее

Глава 1
ПЕРВОЕ ЗНАКОМСТВО С JAVA

Посмотрите Европу! Десять стран за семнадцать дней!
Реклама в туристическом агентстве

В этой главе представлен краткий обзор языка программирования Java. После его прочтения вы сможете написать свое первое Java-приложение. Здесь мы рассмотрим только основные возможности языка, не задерживаясь на деталях. Конкретные свойства Java подробно изучаются в последующих главах.

1.1. С самого начала

Программы на языке Java строятся на основе *классов*. Руководствуясь определением класса, разработчик создает произвольное количество объектов, или *экземпляров*, данного класса. Класс и его объекты можно сравнить, соответственно, с чертежом и деталями — имея чертеж, не составляет труда произвести необходимое количество деталей.

Класс содержит в себе *члены* двух видов: *поля* и *методы*. Полями называются данные, принадлежащие либо самому классу, либо его объектам; значения полей определяют *состояние* объекта или класса. Методами называются последовательности *операторов*, выполняющих какие-либо действия с полями для изменения состояния объекта.

По сложившейся традиции первая программа на изучаемом языке программирования должна выводить строку **Hello, world**. Текст такой программы на Java выглядит следующим образом:

```
class HelloWorld {
        public static void main(String[] args) {
                System.out.println(“Hello, world”);
        }
}
```

Воспользуйтесь своим любимым редактором и введите исходный текст программы в файл. Затем запустите компилятор Java, чтобы преобразовать исходный текст в байт-код Java, “машинный язык” виртуальной абстрактной машины Java. Набор текста программы и ее компиляция в разных системах могут производиться по-разному и потому здесь не описываются — за информацией следует обратиться к соответствующей документации. Если запустить программу, на экране появится:

```
Hello, world
```

Наше маленькое приложение на языке Java что-то делает — но, собственно, как это происходит?

В приведенной выше программе объявляется класс с именем HelloWorld, который содержит всего один метод main. Члены класса перечисляются внутри фигурных скобок { и }, следующих за именем класса. HelloWorld содержит один метод и не имеет полей.

Единственным *параметром* метода main является массив объектов String, которые представляют собой аргументы программы из командной строки, использованной для запуска. Массивы и строки, а также значение args для метода main рассматриваются ниже.

Метод main объявлен с ключевым словом void, поскольку он не возвращает никакого значения. В Java этот метод имеет особое значение; метод main класса, объявленный так, как показано выше, выполняется, если запустить класс как приложение. При запуске метод main может создавать объекты, вычислять значения выражений, вызывать другие методы и делать все то, что заложил в него программист.

В приведенном выше примере main содержит всего один оператор, вызывающий метод println объекта out класса System. Для вызова метода необходимо указать объект и название метода, разделив их точкой (.). Метод println объекта out выводит в стандартный выходной поток строку текста и символ перехода на новую строку.

Упражнение 1.1

Наберите, откомпилируйте и запустите программу HelloWorld на вашем компьютере.

Упражнение 1.2

Попробуйте изменить различные части программы HelloWorld и ознакомьтесь с полученными сообщениями об ошибках.

1.2. Переменные

Следующий пример выводит числа Фибоначчи — бесконечную последовательность, первые члены которой таковы:

1

1
2
3
5
8
13
21
34

Ряд чисел Фибоначчи начинается с 1 и 1, а каждый последующий его элемент представляет собой сумму двух предыдущих. Программа для печати чисел Фибоначчи несложна, но она демонстрирует объявление переменных, работу простейшего цикла и выполнение арифметических операций:

```
class Fibonacci {
        /** Вывод чисел Фибоначчи < 50 */
        public static void main(String[] args) {
                int lo = 1;
                int hi = 1;

                System.out.println(lo);
                while (hi < 50) {
                        System.out.println(hi);
                        hi = lo + hi;   // Изменение значения hi
                        lo = hi — lo;   /* Новое значение lo равно
                                   старому hi, то есть сумме
                                   за вычетом старого lo */
                }
        }
}
```

В этом примере объявляется класс Fibonacci, который, как и Hello World, содержит метод main. В первых строках метода main объявляются и инициализируются две переменные, hi и lo. Перед именем переменной должен быть указан ее *тип*. Переменные hi и lo относятся к типу int — то есть являются 32-разрядными целыми числами со знаком, лежащими в диапазоне от –232 до 232–1.

В языке Java имеется несколько встроенных, “примитивных” типов данных для работы с целыми, вещественными, логическими и символьными значениями. Java может непосредственно оперировать со значениями, относящимися к примитивным типам, — в отличие от объектов, определяемых программистом. Типы, принимаемые “по умолчанию”, в Java отсутствуют; тип каждой переменной должен быть указан в программе. В Java имеются следующие примитивные типы данных:

boolean одно из двух значений: true или false

char 16-разрядный символ в кодировке Unicode 1.1

byte 8-разрядное целое (со знаком)

short 16-разрядное целое (со знаком)

int 32-разрядное целое (со знаком)

long 64-разрядное целое (со знаком)

float 32-разрядное с плавающей точкой (IEEE 754-1985)

double 64-разрядное с плавающей точкой (IEEE 754-1985)

В программе для вывода чисел Фибоначчи переменным hi и lo было присвоено значение 1. Начальные значения переменных можно задавать при их объявлении с помощью оператора = (это называется инициализацией). Переменной, находящейся слева от оператора =, присваивается значение выражения справа от него. В нашей программе переменная hi содержит последнее число ряда, a lo — предыдущее число.

До инициализации переменная имеет *неопределенное* значение. Если вы попробуете воспользоваться переменной до того, как ей было присвоено значение, компилятор Java откажется компилировать программу до тех пор, пока ошибка не будет исправлена.

Оператор while в предыдущем примере демонстрирует один из вариантов циклов в Java. Программа вычисляет выражение, находящееся в скобках после while, — если оно истинно, то выполняется тело цикла, после чего выражение проверяется снова. Цикл while выполняется до тех пор, пока выражение не станет ложным. Если оно всегда остается истинным, программа будет работать бесконечно, пока какое-либо обстоятельство не приведет к выходу из цикла — скажем, встретится оператор break или возникнет исключение.

Условие, проверяемое в цикле while, является логическим выражением, принимающим значение true или false. Логическое выражение, приведенное в тексте программы, проверяет, не превысило ли текущее число ряда значение 50. Если большее число ряда (hi) меньше 50, то оно выводится, а программа вычисляет следующее число Фибоначчи. Если же оно больше или равно 50, то управление передается в строку программы, находящуюся после тела цикла while. В нашем примере такой строкой оказывается конец метода main, так что работа программы на этом завершается.

Обратите внимание на то, что в приведенном выше примере методу println передается целочисленный аргумент, тогда как в HelloWorld его аргументом была строка. Метод println является одним из многих методов, которые *перегружаются (overloaded)*, чтобы их можно было вызывать с аргументами различных типов.

Упражнение 1.3

Выведите заголовок перед списком чисел Фибоначчи.

Упражнение 1.4

Напишите программу, которая генерирует другой числовой ряд, — например, таблицу квадратов (умножение выполняется с помощью оператора * — например, i * i).

1.3. Комментарии

Текст на русском языке в нашей программе представляет собой *комментарий*. В Java предусмотрены комментарии трех видов — все они встречаются в нашем примере.

Текст, следующий за символами // вплоть до конца строки, игнорируется компилятором; то же самое относится и к тексту, заключенному между символами /* и */.

Комментарии позволяют добавлять описания и пояснения для тех программистов, которым в будущем придется разбираться в вашей программе. Вполне возможно, что на их месте окажетесь *вы сами* через несколько месяцев или даже лет. Комментируя свою программу, вы экономите собственное время. Кроме того, при написании комментариев нередко обнаруживаются ошибки в программе — когда приходится объяснять кому-то, что происходит, то поневоле задумываешься над этим сам.

Комментарий третьего типа встречается в самом начале программы, между символами /** и */. Комментарий, начинающийся с двух звездочек, является *документирующим*. Документирующие комментарии используются для описания назначения следующего за ними фрагмента программы; в нашем примере характеризуется метод main. Специальная программа, которая называется javadoc, извлекает документирующие комментарии и генерирует по ним справочный файл в формате HTML.

1.4. Именованные константы

Константами называются фиксированные значения — например, 12, 17.9 или “String like this”. С их помощью можно работать с величинами, которые не вычисляются заново, а остаются постоянными во всем жизненном цикле программы.

Программисты предпочитают иметь дело с *именованными константами* по двум причинам. Первая из них заключается в том, что имя константы представляет собой некоторую форму документации. Оно может (и должно!) описывать, для чего используется то или иное значение.

Другая причина в том, что именованная константа определяется всего в одном месте программы. Когда ее значение понадобится изменить, это достаточно будет сделать в одном месте, что заметно упрощает модификацию программы. Чтобы создать именованную константу в Java, следует указать в ее объявлении ключевые слова static и final и задать начальное значение:

```
class CircleStuff {
        static final double p = 3.1416;
}
```

Если вдруг окажется, что точности в четыре цифры после десятичной точки недостаточно, значение p легко изменить. Мы объявили p как переменную типа double — 64-разрядное число с плавающей точкой с двойной точностью, так что p можно задать и поточнее — скажем, 3.14159265358979323846.

Взаимосвязанные константы можно группировать в рамках класса. Например, в карточной игре могут пригодиться следующие константы:

```
class Suit {
        final static int CLUBS  = 1;
        final static int DIAMONDS = 2;
        final static int HEARTS  = 3;
        final static int SPADES  = 4;
};
```

При такой группировке на масти можно ссылаться как на Suit.HEARTS, Suit.SPADES и т. д. — все названия мастей сосредоточены в пределах одного класса Suit.

1.4.1. Символы Unicode

Давайте ненадолго остановимся и обратим внимание на символ p, использованный в качестве имени константы в предыдущем примере. В большинстве языков программирования идентификаторы обычно ограничиваются буквами и цифрами, входящими в набор символов ASCII.

Java вводит вас в современный мир интернационального программного обеспечения; в Java используется кодировка *Unicode* — международный стандарт для набора символов. Символы Unicode состоят из 16 бит, благодаря чему обеспечивается поддержка букв, входящих в большинство языков мира. По-этому мы и смогли использовать p как имя константы в приведенном выше примере. Символ p входит в греческую секцию Unicode и, следовательно, может присутствовать в исходном тексте программы. В настоящее время при программировании, как правило, используется набор символов ASCII, 7-битная кодировка или ISO-Latin-1 — 8-битный стандарт, который обычно называется Latin-1. Однако перед обработкой эти символы переводятся в Unicode, так что в Java *всегда* используется кодировка Unicode.

Упражнение 1.5

Измените приложение HelloWorld так, чтобы выводимая строка была представлена именованной константой.

Упражнение 1.6

Измените приложение Fibonacci так, чтобы вместо константы-литерала (50) в условии цикла фигурировала именованная константа.

1.5. Порядок выполнения

“Порядком выполнения” называется очередность, в которой выполняются операторы программы. Один из примеров воздействия на порядок выполнения — цикл while в программе для вывода чисел Фибоначчи. К числу других операторов, изменяющих порядок выполнения, относятся конструкции if/else, for, switch, do/while и *блоки* — несколько операторов, сгруппированных внутри фигурных скобок { и }. Давайте усовершенствуем программу для вывода чисел Фибоначчи — перенумеруем все члены последовательности и пометим нечетные числа звездочкой:

```
        /** Вывести несколько первых чисел Фибоначчи,
        помечая нечетные числа символом ‘*’ */
        static final int MAX_INDEX = 10;

        public static void main (String[] args) {
                int lo = 1;
                int hi = 1;
                String mark;

                System.out.println(“1: ” + lo);
                for (int i = 2; i < MAX_INDEX; i++) {
                        if (hi % 2) == 0)
                                mark = “ *”;
                        else
                                mark = “”;
                        System.out.println(i + “: ” + hi + mark);
                        hi = lo + hi;   // Изменение значения hi
                        lo = hi — lo;   /* Новое значение lo равно
                                старому hi, то есть сумме
                                за вычетом старого lo */

                }
        }
}
```

Вот как выглядит результат работы программы:

```
1: 1

2: 1
3: 2 *
4: 3
5: 5
6: 8 *
7: 13
8: 21
9: 34 *
```

Для упрощения нумерации ряда вместо while используется цикл for. Цикл for является частным случаем while с добавлением инициализации и приращения переменной цикла. Приведенный выше цикл for эквивалентен следующему циклу while:

```
{
        int i = 2;
        while (i < MAX_INDEX) {
                // .. вывод
                i++;
        }
}
```

Оператор ++ в этом фрагменте может показаться непонятным тому, кто не знаком с языками программирования, восходящими к C. Этот оператор увеличивает на единицу значение переменной, к которой он применяется, — в данном случае, i. Оператор ++ является *префиксным*, если он стоит перед операндом, и *постфиксным*, если он стоит после него. Аналогично, оператор — уменьшает на единицу значение переменной, к которой он применяется, и также может быть префиксным или постфиксным. Операторы ++ и — ведут свое происхождение из языка программирования C. В приведенном выше примере оператор

i++;

может быть заменен выражением

i = i + 1;

Помимо простого присваивания, в Java имеются и другие операторы присваивания, которые применяют арифметические действия к значению в их левой части. Например, еще одна возможность представить i++ в цикле for может быть такой:

i += 1;

Значение в правой части оператора += (то есть 1) прибавляется к значению переменной в левой части (то есть i), и результат записывается в ту же переменную. Большинство бинарных операторов в Java (другими словами, операторов с двумя операндами) может аналогичным образом объединяться с оператором =.

Внутри цикла for используется конструкция if/else, проверяющая текущее значение hi на четность. Оператор if анализирует значение выражения в скобках. Если оно равно true, то выполняется первый оператор или блок внутри оператора if. Если же значение равно false, то выполняется оператор или блок, следующий за ключевым словом else. Наличие else не требуется; если else отсутствует и условие равно false, то блок if пропускается. После выполнения одной из двух возможных ветвей конструкции if/else, управление передается оператору, следующему за оператором if.

В нашем примере проверка hi на четность осуществляется с помощью оператора %. Он вычисляет остаток от деления левого операнда на правый. Если значение слева четно, то остаток будет равен 0, и следующий оператор присвоит переменной marker звездочку — индикатор для пометки четного числа. Для нечетных чисел выполняется условие else, присваивающее marker пустую строку.

Метод println выполняется несколько сложнее — оператор + используется для конкатенации следующих строк: i, разделитель, строка для значения hi и строка-индикатор. В случае применения оператора + к строкам он выполняет их конкатенацию, тогда как в арифметических выражениях он занимается сложением.

Упражнение 1.7

Модифицируйте цикл так, чтобы значение переменной i изменялось не в прямом, а в обратном направлении.

1.6. Классы и объекты

Java, как и любой другой объектно-ориентированный язык программирования, располагает средствами построения классов и объектов. Каждый объект в Java имеет *тип*; им является тот класс, к которому принадлежит данный объект. В каждом классе есть члены двух видов: поля и методы.

- Полями называются переменные, содержащие данные класса и его объектов. В них хранятся результаты вычислений, выполняемых методами данного класса.
- Методы содержат исполняемый код класса. Методы состоят из операторов; эти операторы, а также способ вызова методов в конечном счете определяют процесс выполнения программы.

Так может выглядеть объявление простого класса, представляющего точку на плоскости:

```
class Point {
        public double x, y;
}
```

Класс Point содержит два поля с координатами x и y точки, и в нем нет ни одного метода (конечно же, в текущей реализации). Подобное объявление класса определяет, как будут выглядеть объекты, созданные на его основе, а также задает поведение объектов с помощью ряда инструкций. Чертеж приобретает наибольшую ценность после того, как к нему добавляются технические задания и инструкции.

Члены класса могут обладать различными правами доступа. Объявление полей x и y класса Point с ключевым словом public означает, что любой метод программы, получивший доступ к объекту Point, сможет прочитать или изменить эти поля. Разрешается ограничить доступ к данным и предоставлять его лишь методам самого класса или связанных с ним классов.

1.6.1. Создание объектов

Объекты создаются посредством выражений, в которых используется ключевое слово new. Созданные на основе определения класса объекты часто называют *экземплярами* данного класса.

В языке Java создаваемые объекты размещаются в области системной памяти, которая называется *кучей (heap)*. Доступ к любому объекту осуществляется с помощью *ссылки на объект* — вместо самого объекта в переменных содержится лишь ссылка на него. Когда ссылка не относится ни к какому объекту, она равна null.

Обычно между самим объектом и ссылкой на него не делается особых различий — можно сказать “передать методу объект”, на самом деле имея в виду “передать методу ссылку на объект”. В книге мы будем различать эти два понятия лишь там, где необходимо, но чаще всего термины “объект” и “ссылка на объект” будут употребляться как эквивалентные.

Возвращаясь к определенному выше классу Point, давайте предположим, что мы разрабатываем графическое приложение, в котором приходится следить за множеством точек. Каждая точка представляется отдельным объектом Point. Вот как может выглядеть создание и инициализация объектов Point:

```
Point lowerLeft = new Point();
Point upperRight = new Point();
Point middlePoint = new Point();

lowerLeft.x = 0.0;
lowerLeft.y = 0.0;

upperRight.x = 1280.0;
upperRight.y = 1024.0;

middlePoint.x = 640.0;
middlePoint.y = 512.0;
```

Каждый объект класса Point обладает собственной копией полей x и y. Например, изменение поля x объекта lowerLeft никак не влияет на значение x объекта upperRight. Поля объектов иногда называют *переменными экземпляра (instance variables)*, поскольку в каждом объекте (экземпляре) класса содержится отдельная копия этих полей.

1.6.2. Статические поля

Чаще всего бывает нужно, чтобы значение поля одного объекта отличалось от значений одноименных полей во всех остальных объектах того же класса.

Тем не менее иногда возникает необходимость совместного использования поля всеми объектами класса. Такие совместные поля также называются *переменными класса* — то есть переменными, относящимися ко всему классу, в отличие от переменных, относящихся к его отдельным объектам.

Для чего нужны переменные класса? Давайте представим себе фабрику, производящую плееры Sony (Sony Walkman). Каждому плееру присваивается уникальный серийный номер. В наших терминах это означает, что в каждом объекте имеется уникальное поле, в котором хранится значение номера. Однако фабрика должна знать значение номера, который должен быть присвоен следующему плееру. Дублировать эту информацию в каждом объекте-плеере было бы неразумно — нужна всего одна копия номера, которая хранится на самой фабрике, другими словами — в переменной класса.

Чтобы использовать поле для хранения информации, относящейся ко всему классу, следует объявить его с ключевым словом static, поэтому такие поля иногда называют *статическими*. Например, объект Point, представляющий начало координат, может встречаться достаточно часто, поэтому имеет смысл выделить ему отдельное статическое поле в классе Point:

public static Point origin = new Point();

Если это объявление встретится внутри объявления класса Point, то появится ровно один экземпляр данных с именем Point.origin, который всегда будет ссылаться на объект (0,0). Поле static будет присутствовать всегда, независимо от того, сколько существует объектов Point (даже если не было создано ни одного объекта). Значения x и y равны нулю, потому что числовые поля, которым не было присвоено начального значения, по умолчанию инициализируются нулями.

Мы уже встречались со статическим объектом в нашей первой программе. Класс System — стандартный класс Java, в котором имеется статическое поле с именем out, предназначенное для направления вывода программы в стандартный выходной поток.

Когда в этой книге встречается термин “поле”, обычно имеется в виду поле, уникальное для каждого объекта, хотя в отдельных случаях для большей ясности может использоваться термин “нестатическое поле”.

1.6.3. Сборщик мусора

Предположим, вы создали объект с помощью вызова new; но как избавиться от этого объекта, когда он окажется ненужным? Ответ простой — никак. Неиспользуемые объекты Java автоматически уничтожаются *сборщиком мусора*. Сборщик мусора работает в фоновом режиме и следит за ссылками на объекты. Когда ссылок на объект больше не остается, появляется возможность убрать его из кучи, где он временно хранился, хотя само удаление может быть отложено до более подходящего момента.

1.7. Методы и параметры

Объекты определенного выше класса Point могут быть изменены в любом фрагменте программы, в котором имеется ссылка на объект Point, поскольку поля этого класса объявлены с ключевым словом public. Класс Point представляет собой простейший пример класса. На самом деле иногда можно обойтись и простыми классами — например, при выполнении чисто внутренних задач пакета или когда для ваших целей хватает простейших типов данных.

Тем не менее настоящие преимущества объектно-ориентированного программирования проявляются в возможности спрятать реализацию класса. В языке Java операции над классом осуществляются с помощью методов класса — инструкций, которые выполняются над данными объекта, чтобы получить нужный результат. В методах часто используются такие детали реализации класса, которые должны быть скрыты от всех остальных объектов. Данные скрываются в методах и становятся недоступными для всех прочих объектов — в этом заключается основной смысл инкапсуляции данных.

Каждый метод имеет ноль или более параметров. Метод может возвращать значение или объявляться с ключевым словом void, которое означает, что метод ничего не возвращает. Операторы метода содержатся в блоке между фигурными скобками { и }, которые следуют за именем метода и объявлением его *сигнатуры*. Сигнатурой называется имя метода, сопровождаемое числом и типом его параметров. Можно усовершенствовать класс Point и добавить в него простой метод clear, который выглядит так:

```
public void clear() {
        x = 0;
        y = 0;
}
```

Метод clear не имеет параметров, поскольку в скобках (и) после его имени ничего нет; кроме того, этот метод объявляется с ключевым словом void, поскольку он не возвращает никакого значения. Внутри метода разрешается прямое именование полей и методов класса — можно просто написать x и y, без ссылки на конкретный объект.

1.7.1. Вызов метода

Объекты обычно не работают непосредственно с данными других объектов, хотя, как мы видели на примере класса Point, класс *может* сделать свои поля общедоступными. И все же в хорошо спроектированном классе данные обычно скрываются, чтобы они могли изменяться только методами этого класса. Чтобы *вызвать* метод, необходимо указать имя объекта и имя метода и разделить их точкой (.). Параметры передаются методу в виде заключенного в скобки списка значений, разделяемых запятыми. Даже если метод вызывается без параметров, все равно необходимо указать пустые скобки. Объект, для которого вызывается метод (объект, получающий запрос на вызов метода) носит название *объекта-получателя*, или просто *получателя*.

В качестве результата работы метода может возвращаться только одно значение. Чтобы метод возвращал несколько значений, следует создать специальный объект, единственное назначение которого — хранение возвращаемых значений, и вернуть этот объект.

Ниже приводится метод с именем distance, который входит в класс Point, использованный в предыдущих примерах. Метод distance принимает в качестве параметра еще один объект Point, вычисляет евклидово расстояние между двумя точками и возвращает результат в виде вещественного значения с двойной точностью:

```
public double distance(Point that) {
        double xdiff, ydiff;
        xdiff = x — that.x;
        ydiff = y — that.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

Для объектов lowerLeft и upperRight, которые были определены в разделе, посвященном созданию экземпляров объектов, вызов метода distance может выглядеть так:

double d = lowerLeft.distance(upperRight);

После выполнения этого оператора переменная d будет содержать евклидово расстояние между точками lowerLeft и upperRight.

1.7.2. Ссылка this

Иногда объекту-получателю бывает необходимо знать ссылку на самого себя. Например, объект-получатель может захотеть внести себя в какой-нибудь список объектов. В каждом методе может использоваться this — ссылка на текущий объект (объект-получатель). Следующее определение clear эквивалентно приведенному выше:

```
public void clear() {
        this.x = 0;
        this.y = 0;
}
```

Ссылка this часто используется в качестве параметра для тех методов, которым нужна ссылка на объект. Кроме того, this также может применяться для именования членов текущего объекта. Вот еще один из методов Point, который называется move и служит для присвоения полям x и y определенных значений:

```
public void move(double x, double y) {
        this.x = x;
        this.y = y;
}
```

В методе move ссылка this помогает разобраться, о каких x и y идет речь. Присвоить аргументам move имена x и y вполне разумно, поскольку в этих параметрах методу передаются координаты x и y точки. Но тогда получается, что имена параметров совпадают с именами полей Point, и имена параметров *скрывают* имена полей. Если бы мы просто написали x = x, то значение параметра x было бы присвоено самому параметру, а не полю x, как мы хотели. Выражение this.x определяет поле x объекта, а не параметр x метода move.

1.7.3. Статические методы

Мы уже знаем, что в классе могут присутствовать статические поля, относящиеся к классу в целом, а не к его конкретным экземплярам. По аналогии с ними могут существовать и статические методы, также относящиеся ко всему классу, которые часто называют *методами класса*. Статические методы обычно предназначаются для выполнения операций, специфичных для данного класса, и работают со статическими полями, а не с конкретными экземплярами класса. Методы класса объявляются с ключевым словом static и называются статическими методами.

Когда в этой книге встречается термин “метод”, он (как и термин “поле”) означает метод, специфичный для каждого объекта, хотя в отдельных случаях, для большей ясности, может использоваться термин “нестатический метод”.

Для чего нужны статические методы? Давайте вернемся к примеру с фабрикой, производящей плееры. Следующий серийный номер для нового изделия должен храниться на фабрике, а не в каждом объекте-плеере. Соответственно и метод, который работает с этим номером, должен быть статическим, а не методом, работающим с конкретными объектами-плеерами.

В реализации метода distance из предыдущего примера использован статический метод Math.sqrt для вычисления квадратного корня. Класс Math содержит множество методов для часто встречающихся математических операций. Эти методы объявлены статическими, так как они работают не с каким-то определенным объектом, но составляют внутри класса группу со сходными функциями.

Статический метод не может напрямую обращаться к нестатическим членам. При вызове статического метода не существует ссылки на конкретный объект, для которого вызывается данный метод. Впрочем, это ограничение можно обойти, передавая ссылку на конкретный объект в качестве параметра статического метода. Тем не менее в общем случае статические методы выполняют свои функции на уровне всего класса, а нестатические методы работают с конкретными объектами. Статический метод, модифицирующий поля объектов, — примерно то же самое, что и фабрика из нашего примера, которая пытается изменить серийный номер давно проданного плеера.

1.8. Массивы

Простые переменные, содержащие всего одно значение, полезны, но для многих приложений их недостаточно. Скажем, для разработки программы игры в карты требуется множество объектов Card, с которыми можно было бы оперировать как с единым целым. Для таких случаев в языке Java предусмотрены *массивы*.

Массивом называется набор переменных, относящихся к одному типу. Доступ к элементам массива осуществляется посредством простых целочисленных индексов. В карточной игре объект Deck (колода) может выглядеть так:

```
class Deck {
        final int DECK_SIZE = 52;
        Card[] cards = new Card[DECK_SIZE];

        public void print() {
                for (int i = 0; i < cards.length; i++)
                        System.out.println(cards[i]);
        }
}
```

Сначала мы объявляем константу с именем DECK_SIZE, содержащую количество кард в колоде. Затем поле cards объявляется в виде массива типа Card — для этого после имени типа в объявлении необходимо поставить квадратные скобки [и]. Размер массива определяется при его создании и не может быть изменен в будущем.

Вызов метода print показывает, как производится доступ к элементам массива: индекс нужного элемента заключается в квадратные скобки [и], следующие за именем массива.

Как нетрудно догадаться по тексту программы, в объекте-массиве имеется поле length, в котором хранится количество элементов в массиве. *Границами* массива являются целые числа 0 и length-1. Если попытаться обратиться к элементу массива, индекс которого выходит за эти пределы, то возбуждается исключение IndexOutOfBounds.

В этом примере также демонстрируется новый механизм объявления переменных — переменная цикла объявлена в секции инициализации цикла for. Объявление переменной в секции инициализации — удобный и наглядный способ объявления простой переменной цикла. Такая конструкция допускается лишь при инициализации цикла for; вы не сможете объявить переменную при проверке условия в операторе if или while.

Переменная цикла i существует лишь внутри оператора for. Переменная цикла, объявленная подобным образом, исчезает сразу же после его завершения — это означает, что ее имя может использоваться в качестве имени переменной в последующих операторах цикла.

Упражнение 1.8

Измените приложение Fibonacci так, чтобы найденные числа Фибоначчи сохранялись в массиве и выводились в виде списка значений в конце программы.

Упражнение 1.9

Измените приложение Fibonacci так, чтобы числа Фибоначчи сохранялись в массиве. Для этого создайте новый класс для хранения самого числа и логического значения, являющегося признаком четности, после чего создайте массив для ссылок на объекты этого класса.

1.9. Строковые объекты

Для работы с последовательностями символов в Java предусмотрены тип объектов String и языковая поддержка при их инициализации. Класс String предоставляет разнообразные методы для работы с объектами String.

Примеры литералов типа String уже встречались нам в примерах — в частности, в программе HelloWorld. Когда в программе появляется оператор следующего вида:

System.out.println(“Hello, world”);

компилятор Java на самом деле создает объект String, присваивает ему значение указанного литерала и передает его в качестве параметра методу println.

Объекты типа String отличаются от массивов тем, что при их создании не нужно указывать размер. Создание нового объекта String и его инициализация выполняются всего одним оператором, как показывает следующий пример:

```
class StringDemo {
        static public void main(String args[]) {
                String myName = “Petronius”;

                myName = myName + “ Arbiter”;
                System.out.println(“Name = ” + myName);
        }
}
```

Мы создаем объект String с именем myName и инициализируем его строковым литералом. Выражение с оператором конкатенации +, следующее за инициализацией, создает новый объект String с новым значением. Наконец, значение myName выводится в стандартный выходной поток. Результат работы приведенной выше программы будет таким:

```
Name = Petronius Arbiter
```

Кроме знака +, в качестве оператора конкатенации можно использовать оператор += как сокращенную форму, в которой название переменной размещается в левой части оператора. Усовершенствованная версия приведенного выше примера выглядит так:

```
class BetterStringDemo {
        static public void main(String args[]) {
                String myName = “Petronius”;
                String occupation = “Reorganization Specialist”;

                myName = myName + “ Arbiter”;
                myName += “ ”;
                myName += “(” + occupation + “)”;
                System.out.println(“Name = ” + myName);
        }
}
```

Теперь при запуске программы будет выведена следующая строка:

```
Name = Petronius Arbiter (Reorganization Specialist)
```

Объекты String содержат метод length, который возвращает количество символов в строке. Символы имеют индексы от 0 до length()-1.

Объекты String являются *неизменяемыми*, или *доступными только для чтения*; содержимое объекта String никогда не меняется. Когда в программе встречаются операторы следующего вида:

```
str = “redwood”;
// ... сделать что-нибудь со str ...
str = “oak”;
```

второй оператор присваивания задает новое значение *ссылки на объект*, а не *содержимого* строки. При каждом выполнении операции, которая на первый взгляд изменяет содержимое объекта (например, использование выше +=), на самом деле возникает новый объект String, также доступный только для чтения, — тогда как содержимое исходного объекта String остается неизменным. Класс StringBuffer позволяет создавать строки с изменяющимся содержимым; этот класс описывается в главе 8, в которой подробно рассматривается и класс String.

Самый простой способ сравнить два объекта String и выяснить, совпадает ли их содержимое, заключается в использовании метода equals:

```
if (oneStr.equals(twoStr))
        foundDuplicate(oneStr, twoStr);
```

Другие методы, в которых сравниваются части строки или игнорируется регистр символов, также рассматриваются в главе 8.

Упражнение 1.10

Измените приложение StringsDemo так, чтобы в нем использовались разные строки.

Упражнение 1.11

Измените приложение ImprovedFibonacci так, чтобы создаваемые в нем объекты String сначала сохранялись в массиве, а не выводились бы сразу же на печать методом println.

1.10. Расширение класса

Одним из самых больших достоинств объектно-ориентированного программирования является возможность такого *расширения*, или создания *подкласса*, существующего класса, при котором можно использовать код, написанный для исходного класса.

При расширении класса на его основе создается новый класс, *наследующий* все поля и методы расширяемого класса. Исходный класс, для которого проводилось расширение, называется *суперклассом*.

Если подкласс не *переопределяет* (override) поведение суперкласса, то он наследует все свойства суперкласса, поскольку, как уже говорилось, расширенный класс наследует поля и методы суперкласса.

Примером с плеерами Walkman можно воспользоваться и здесь. В последних моделях плееров устанавливаются два разъема для наушников, чтобы одну и ту же кассету могли слушать сразу двое. В объектно-ориентированном мире модель с двумя разъемами расширяет базовую модель. Эта модель наследует все характеристики и поведение базовой модели и добавляет к ним свои собственные.

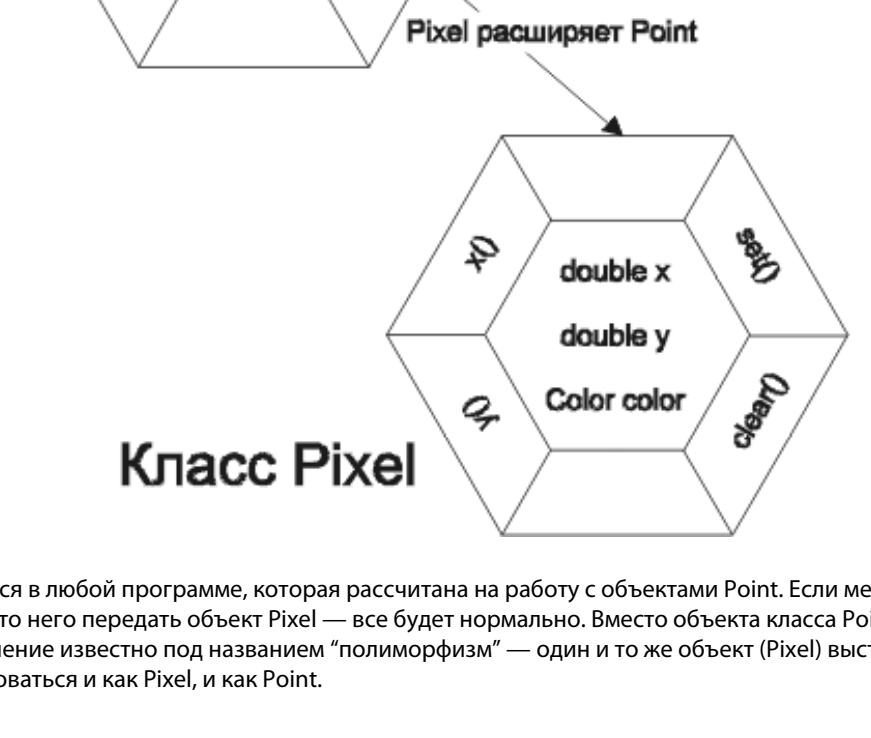
Покупатели сообщали в корпорацию Sony, что они хотели бы иметь возможность разговаривать друг с другом во время прослушивания кассеты. Sony усовершенствовала свою модель с двумя разъемами, чтобы люди могли поговорить под музыку. Модель с двумя разъемами и с возможностью ведения переговоров является подклассом модели с двумя разъемами, наследует все ее свойства и добавляет к ним свои собственные.

У Sony имеются и другие модели плееров. Более поздние серии расширяют возможности базовой модели — они создают подклассы на ее основе и наследуют от нее свойства и поведение.

Давайте посмотрим, как происходит наследование в Java. Расширим наш класс Point, чтобы он представлял пиксель на экране монитора. В новом классе Pixel к координатам x и y добавляется информация о цвете пикселя:

```
class Pixel extends Point {
        Color color;

        public void clear() {
                super.clear();
                color = null;
        }
}
```

Класс Pixel расширяет как *данные*, так и *поведение* своего суперкласса Point. Для данных это означает, что в классе Pixel появляется дополнительное поле color. Pixel также расширяет поведение Point, переопределяя метод clear класса Point. Эта концепция наглядно изображена на рисунке:



Объект Pixel может использоваться в любой программе, которая рассчитана на работу с объектами Point. Если методу необходимо передать параметр типа Point, можно вместо него передать объект Pixel — все будет нормально. Вместо объекта класса Point можно пользоваться объектом подкласса Pixel; это явление известно под названием “полиморфизм” — один и то же объект (Pixel) выступает в нескольких (*поли-*) формах (*-морф*) и может использоваться и как Pixel, и как Point.

Поведение Pixel расширяет поведение Point. Оно может совершенно преобразиться (например, работа с цветами в нашем примере) или будет представлять собой некоторое ограничение старого поведения, удовлетворяющее всем исходным требованиям. Примером последнего может служить объект класса Pixel, принадлежащий некоторому объекту Screen (экран), в котором значения координат x и y ограничиваются размерами экрана. В исходном классе Point значения координат могли быть произвольными, поэтому ограниченные значения координат все равно лежат в исходном (неограниченном) диапазоне.

Расширенный класс часто *переопределяет* поведение своего *суперкласса* (то есть класса, на основе которого он был создан), по-новому реализуя один или несколько унаследованных методов. В приведенном выше примере мы переопределили метод clear, чтобы он вел себя так, как того требует объект Pixel, — метод clear, унаследованный от Point, знает лишь о существовании полей Point, но, разумеется, не догадывается о присутствии поля color, объявленного в подклассе Pixel.

Упражнение 1.12

Напишите набор классов, отражающих структуру семейства плееров Sony Walkman. Воспользуйтесь методами, чтобы скрыть все данные, объявите последние с ключевым словом private, а методы — public. Какие методы должны принадлежать базовому классу Walkman? Какие методы добавятся в расширенных классах?

1.10.1. Класс Object

Классы, для которых не указан расширяемый класс, являются неявным расширением класса Object. Все ссылки на объекты полиморфно относятся к классу Object, который является базовым классом для всех ссылок, которые могут относиться к объектам любого класса:

```
Object oref = new Pixel();
oref = “Some String”;
```

В этом примере объекту oref вполне законно присваиваются ссылки на объекты Pixel и String, невзирая на то что эти классы не имеют между собой ничего общего — за исключением неявного суперкласса Object.

В классе Object также определяется несколько важных методов, рассмотренных в главе 3.

1.10.2. Вызов методов суперкласса

Чтобы очистка объектов класса Pixel происходила правильно, мы заново реализовали метод clear. Его работа начинается с того, что с помощью ссылки super вызывается метод clear суперкласса. Ссылка super во многих отношениях напоминает уже упоминавшуюся ранее ссылку this, за тем исключением, что super используется для ссылок на члены суперкласса, тогда как this ссылается на члены текущего объекта.

Вызов super.clear() обращается к суперклассу для выполнения метода clear точно так же, как он обращался бы к любому объекту суперкласса — в нашем случае, класса Point. После вызова super.clear() следует новый код, который должен присваивать color некоторое разумное начальное значение. Мы выбрали null — то есть отсутствие ссылки на какой-либо объект.

Что бы случилось, если бы мы не вызвали super.clear()? Метод clear класса Pixel присвоил бы полю цвета значение null, но переменные x и y, унаследованные от класса Point, остались бы без изменений. Вероятно, подобная частичная очистка объекта Pixel, при которой упускаются унаследованные от Point поля, явилась бы ошибкой в программе.

При вызове метода super.method() runtime-система просматривает иерархию классов до первого суперкласса, содержащего method(). Например, если бы метод clear отсутствовал в классе Point, то runtime-система попыталась бы найти такой метод в его суперклассе и (в случае успеха) вызвала бы его.

Во всех остальных ссылках при вызове метода используется тип *объекта*, а не тип *ссылки* на объект. Приведем пример:

```
Point point = new Pixel();

point.clear(); // используется метод clear() класса Pixel
```

В этом примере будет вызван метод clear класса Pixel, несмотря на то что переменная, содержащая объект класса Pixel, объявлена как ссылка на Point.

1.11. Интерфейсы

Иногда бывает необходимо только *объявить* методы, которые должны поддерживаться объектом, без их конкретной реализации. До тех пор пока поведение объектов удовлетворяет некоторым критериям, подробности реализации методов оказываются несущественными. Например, если вы хотите узнать, входит ли значение в то или иное множество, конкретный способ хранения этих объектов вас не интересует. Методы должны одинаково хорошо работать со связным списком, хеш-таблицей или любой другой структурой данных.

Java использует так называемый *интерфейс* — некоторое подобие класса, где методы лишь объявляются, но не определяются. Разработчик интерфейса решает, какие методы должны поддерживаться в классах, *реализующих* данный интерфейс, и что эти методы должны делать. Приведем пример интерфейса Lookup:

```
interface Lookup {
        /** Вернуть значение, ассоциированное с именем, или
        null, если такого значения не окажется */
        Object find(String name);
}
```

В интерфейсе Lookup объявляется всего один метод find, который получает значение (имя) типа String и возвращает значение, ассоциированное с данным именем, или null, если такого значения не найдется. Для объявленного метода не предоставляется никакой конкретной реализации — она полностью возлагается на класс, в котором реализуется данный интерфейс. Во фрагменте программы, где используются ссылки на объекты Lookup (объекты, реализующие интерфейс Lookup), можно вызвать метод find и получить ожидаемый результат независимо от конкретного типа объекта:

```
void processValues(String[] names, Lookup table) {
        for (int i = 0; i < names.length; i++) {
                Object value = table.find(names[i]);
                if (value != null)
                        processValue(names[i], value);
        }
}
```

Класс может реализовать произвольное количество интерфейсов. В следующем примере приводится реализация интерфейса Lookup для простого массива (мы не стали реализовывать методы для добавления или удаления элементов):

```
class SimpleLookup implements Lookup {
        private String[] Names;
        private Object[] Values;

        public Object find(String name) {
                for (int i = 0; i < Names.length; i++) {
                        if (Names[i].equals(name))
                                return Values[i];
                }
                return null;
        }
        // . . .
}
```

Интерфейсы, подобно классам, могут расширяться посредством ключевого слова extends. Интерфейс может расширить один или несколько других интерфейсов, добавить к ним новые константы и методы, которые должны быть реализованы в классе, реализующем расширенный интерфейс.

Супертипами класса называются расширяемый им класс и интерфейсы, которые он реализует, включая все супертипы этих классов и интерфейсов. Следовательно, тип объекта — это не только класс, но и все его супертипы вместе с интерфейсами. Объект может полиморфно использоваться в суперклассе и во всех суперинтерфейсах, включая любой их супертип.

Упражнение 1.13

Напишите расширенный интерфейс Lookup с добавлением методов add и remove. Реализуйте его в новом классе.

1.12. Исключения

Что делать, если в программе произошла ошибка? Во многих языках о ней свидетельствуют необычные значения кодов возврата — например, –1. Программисты нередко не проверяют свои программы на наличие исключительных состояний, так как они полагают, что ошибок “быть не должно”. С другой стороны, поиск опасных мест и восстановление нормальной работы даже в прямолинейно построенной программе может затемнить ее логику до такой степени, что все происходящее в ней станет совершенно непонятным. Такая простейшая задача, как считывание файла в память, требует около семи строк в программе. Обработка ошибок и вывод сообщений о них увеличивает код до 40 строк. Суть программы теряется в проверках как иголка в стоге сена — это, конечно же, нежелательно.

При обработке ошибок в Java используются *проверяемые исключения (checked exceptions)*. Исключение заставляет программиста предпринять какие-то действия при возникновении ошибки. Исключительные ситуации в программе обнаруживаются при их возникновении, а не позже, когда необработанная ошибка приведет к множеству проблем.

Метод, в котором обнаруживается ошибка, *возбуждает (throw)* исключение. Оно может быть *перехвачено (catch)* кодом, находящимся дальше в стеке вызова — благодаря этому первый фрагмент может обработать исключение и продолжить выполнение программы. Неперехваченные исключения передаются стандартному обработчику Java, который может сообщить о возникновении исключительной ситуации и завершить работу потока в программе.

Исключения в Java являются объектами — у них имеется тип, методы и данные. Представление исключения в виде объекта оказывается полезным, поскольку объект-исключение может обладать данными или методами (или и тем и другим), которые позволят справиться с конкретной ситуацией. Объекты-исключения обычно порождаются от класса Exception, в котором содержится строковое поле для описания ошибки. Java требует, чтобы все исключения были расширениями класса с именем Throwable.

Основная парадигма работы с исключениями Java заключена в последовательности try-catch-finally. Сначала программа пытается (try) что-то сделать; если при этом возникает исключение, она его перехватывает (catch); и наконец (finally), программа предпринимает некоторые итоговые действия в стандартном коде или в коде обработчика исключения — в зависимости от того, что произошло.

Ниже приводится метод averageOf, который возвращает среднее арифметическое двух элементов массива. Если какой-либо из индексов выходит за пределы массива, программа запускает исключение, в котором сообщает об ошибке. Прежде всего следует определить новый тип исключения Illegal AverageException для вывода сообщения об ошибке. Затем необходимо указать, что метод averageOf возбуждает это исключение, при помощи ключевого слова throws:

```
class IllegalAverageException extends Exception {
}

class MyUtilities {
        public double averageOf(double[] vals, int i, int j)
                throws IllegalAverageException
        {
                try {
                        return (vals[i] + vals[j]) / 2;
                } catch (IndexOutOfBounds e) {
                        throw new IllegalAverageException();
                }
        }
}
```

Если при определении среднего арифметического оба индекса i и j оказываются в пределах границ массива, вычисление происходит успешно и метод возвращает полученное значение. Однако, если хотя бы один из индексов выходит за границы массива, возбуждается исключение IndexOutOfBounds и выполняется соответствующий оператор catch. Он создает и возбуждает новое исключение IllegalAverageException — в сущности, общее исключение нарушения границ массива превращается в конкретное исключение, более точно описывающее истинную причину. Методы, находящиеся дальше в стеке выполнения, могут перехватить новое исключение и должным образом прореагировать на него.

Если выполнение метода может привести к возникновению проверяемых исключений, последние должны быть объявлены после ключевого слова throws, как показано на примере метода averageOf. Если не считать исключений RuntimeException и Error, а также подклассов этих типов исключений, которые могут возбуждаться в любом месте программы, метод возбуждает лишь объявленные в нем исключения — как прямо, посредством оператора throw, так и косвенно, вызовом других методов, возбуждающих исключения.

Объявление исключений, которые могут возбуждаться в методе, позволяет компилятору убедиться, что метод возбуждает только эти исключения и никакие другие. Подобная проверка предотвращает ошибки в тех случаях, когда метод должен обрабатывать исключения от других методов, однако не делает этого. Кроме того, метод, вызывающий ваш метод, может быть уверен, что это не приведет к возникновению неожиданных исключений. Именно поэтому исключения, которые должны быть объявлены после ключевого слова throws, называются *проверяемыми исключениями*. Исключения, являющиеся расширениями RuntimeException и Error, не нуждаются в объявлении и проверке; они называются *непроверяемыми исключениями*.

Упражнение 1.14

Отредактируйте исключение IllegalAverageException так, чтобы в нем содержался массив и индексы и при перехвате этого исключения можно было узнать подробности ошибки.

1.13. Пакеты

Конфликты имен становятся источником серьезных проблем при разработке повторно используемого кода. Как бы тщательно вы ни подбирали имена для своих классов и методов, кто-нибудь может использовать это же имя для других целей. При использовании простых названий проблема лишь усугубляется — такие имена с большей вероятностью будут задействованы кем-либо еще, кто также захочет пользоваться простыми словами. Такие имена, как *set*, *get*, *clear* и т. д., встречаются очень часто, и конфликты при их использовании оказываются практически неизбежными.

Во многих языках программирования предлагается стандартное решение — использование “префикса пакета” перед каждым именем класса, типа, глобальной функции и так далее. Соглашения о префиксах создают *контекст имен (naming context)*, который предотвращает конфликты имен одного пакета с именами другого. Обычно такие префиксы имеют длину в несколько символов и являются сокращением названия пакета — например, Xt для “X Toolkit” или WIN32 для 32-разрядного Windows API.

Если программа состоит всего из нескольких пакетов, вероятность конфликтов префиксов невелика. Однако, поскольку префиксы являются сокращениями, при увеличении числа пакетов вероятность конфликта имен повышается.

В Java принято более формальное понятие пакета, в котором типы и субпакеты выступают в качестве членов. Пакеты являются именованными и могут импортироваться. Имена пакетов имеют иерархическую структуру, их компоненты разделяются точками. При использовании компонента пакета необходимо либо ввести его полное имя, либо *импортировать* пакет — целиком или частично. Иерархическая структура имен пакетов позволяет работать с более длинными именами. Кроме того, это дает возможность избежать конфликтов имен — если в двух пакетах имеются классы с одинаковыми именами, можно применить для их вызова форму имени, в которую включается имя пакета.

Приведем пример метода, в котором полные имена используются для вывода текущей даты и времени с помощью вспомогательного класса Java с именем Date (о котором рассказано в главе 12):

```
class Date1 {
        public static void main(String[] args) {
                java.util.Date now = new java.util.Date();
                System.out.println(now);
        }
}
```

Теперь сравните этот пример с другим, в котором для объявления типа Date используется ключевое слово import:

```
import java.util.Date;

class Date2 {
        public static void main(String[] args) {
                Date now = new Date();
                System.out.println(now);
        }
}
```

Пакеты Java не до конца разрешают проблему конфликтов имен. Два различных проекта *могут* присвоить своим пакетам одинаковые имена. Эта проблема решается только за счет использования общепринятых соглашений об именах. По наиболее распространенному из таких соглашений в качестве префикса имени пакета используется перевернутое имя домена организации в Internet. Например, если фирма Acme Corporation содержит в Internet домен с именем acme.com, то разработанные ей пакеты будут иметь имена типа COM.acme.package.

Точки, разделяющие компоненты имени пакета, иногда могут привести к недоразумениям, поскольку те же самые точки используются при вызове методов и доступе к полям в ссылках на объекты. Возникает вопрос — что же именно импортируется? Новички часто пытаются импортировать объект System.out, чтобы не вводить его имя перед каждым вызовом println. Такой вариант не проходит, поскольку System является классом, а out — его статическим полем, тип которого поддерживается методом println.

С другой стороны, java.util является пакетом, так что допускается импортирование java.util.Date (или java.util.*, если вы хотите импортировать все содержимое пакета). Если у вас возникают проблемы с импортированием чего-либо, остановитесь и убедитесь в том, что вы импортируете тип.

Классы Java всегда объединяются в пакеты. Имя пакета задается в начале файла:

```
package com.sun.games;

class Card
{
        // ...
}

// ...
```

Если имя пакета не было указано в объявлении package, класс становится частью *безымянного пакета*. Хотя это вполне подходит для приложения (или аплета), которое используется отдельно от другого кода, все классы, которые предназначаются для использования в библиотеках, должны включаться в именованные пакеты.

1.14. Инфраструктура Java

Язык Java разработан так, чтобы обеспечивать максимальную переносимость. Многие аспекты Java определяются сразу для всех возможных реализаций. Например, тип int всегда должен представлять собой 32-разрядное целое со знаком с дополнением по модулю 2. Во многих языках программирования точные определения типов являются уделом конкретной реализации; на уровне языка даются лишь общие гарантии, такие как минимальный диапазон чисел данного типа или возможность системного запроса, позволяющего определить диапазон на данной платформе.

В языке Java такие требования продвинуты вплоть до уровня машинного языка, на который транслируется текст программ. Исходный текст программы на Java компилируется в *байт-код*, выполняемый на *виртуальной машине Java*. Байт-код является универсальным языком, и именно он интерпретируется виртуальной машиной в каждой системе, поддерживающей Java. /Разумеется, виртуальная машина Java может быть реализована и на аппаратном уровне - то есть с использованием специализированных микросхем. Это никак не влияет на переносимость байт-кода и является всего лишь одним из видов реализации виртуальной машины. - *Примеч. перев*/

Виртуальная машина присваивает каждому приложению собственный *контекст времени выполнения (runtime)*, который одновременно изолирует приложения друг от друга и обеспечивает безопасность работы. Менеджер безопасности каждого контекста времени выполнения определяет, какие возможности доступны данному приложению. Например, менеджер безопасности может запретить приложению чтение или запись на локальный диск или ограничить сетевые соединения строго определенными компьютерами.

В совокупности все эти средства делают язык Java полностью платформонезависимым и предоставляют схему безопасности для выполнения переданного по сети кода на различных уровнях доверия (trust levels). Исходный текст Java, скомпилированный в байт-код Java, может выполняться на любом компьютере, где имеется виртуальная машина Java. Код может выполняться на соответствующем уровне защиты, чтобы предотвратить случайное или злонамеренное повреждение системы. Уровень доверия регулируется в зависимости от источника байт-кода — байт-код на локальном диске или в защищенной сети пользуется большим доверием, чем байт-код, полученный с удаленного компьютера, неизвестно даже где расположенного.

1.15. Прочее

Java содержит ряд других возможностей, которые кратко упоминаются здесь и рассматриваются в следующих главах:

- *Потоки:* Java обладает встроенной поддержкой потоков для создания многопоточных приложений. Для синхронизации параллельного доступа к объектам и данным класса используется блокировка на уровне объектов и на уровне классов. Подробности приведены в главе 9.
- *Ввод/вывод:* Java содержит пакет java.io, предназначенный для выполнения разнообразных операций ввода/вывода. Конкретные возможности ввода/вывода описаны в главе 11.
- *Классы общего назначения:* в состав Java входят классы, представляющие многие примитивные типы данных (такие, как Integer, Double и Boolean), а также класс Class для работы с различными типами классов. Программирование с использованием типов рассматривается в главе 13.
- *Вспомогательные классы и интерфейсы:* Java содержит пакет java.util со множеством полезных классов — таких, как BitSet, Vector, Stack и Date. Более подробно о вспомогательных классах рассказывается в главе 12.

Содержание | Далее

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Глава 8
СТРОКИ

Что толку в хорошей цитате, если ее нельзя изменить?
Доктор Who, The Two Doctors

Строки в Java являются стандартными объектами со встроенной языковой поддержкой. Нам уже встречалось множество примеров того, как строковые объекты создаются на основе литералов; кроме того, мы видели, как операторы + и += применяются для конкатенации и создания новых строк. Тем не менее функции класса String этим не ограничиваются. Объекты String доступны только для чтения, поэтому в Java также имеется класс StringBuffer для изменяемых строк. В этой главе описаны классы String и StringBuffer, а также преобразование строк в другие типы — например, целый или логический.

8.1. Основные операции со строками

Класс String позволяет работать со строками, доступными только для чтения, и поддерживает операции с ними. Строки могут создаваться неявным образом при помощи заключенной в кавычки последовательности символов (например, “GrцЯе”) или за счет выполнения оператора + или += над двумя объектами String.

Кроме того, допускается и явное построение объектов String оператором new. В классе String предусмотрены следующие конструкторы:

public **String()**

Конструирует новый объект String, значение которого равно “”.

public **String(String value)**

Конструирует новый объект String, являющийся копией заданного.

При работе с объектами String используются два базовых метода — length и charAt. Метод length возвращает количество символов в строке, а метод charAt — символ в заданной позиции. Приведенный ниже цикл подсчитывает количество вхождений каждого из символов в строку:

```
for (int i = 0; i << str.length(); i++)
    counts[str.charAt(i)]++;
```

При попытке обратиться в методе charAt или любом другом методе String к позиции, номер которой отрицателен либо превышает length()-1, возбуждается исключение IndexOutOfBoundsException. Подобные неверные обращения обычно обусловлены наличием ошибок в программе.

Кроме того, имеются простые методы для поиска в строке первого или последнего вхождения конкретного символа или подстроки. Следующий метод возвращает количество символов между первым и последним вхождением заданного символа в строке:

```
static int countBetween(String str, char ch) {
    int begPos = str.indexOf(ch);
    if (begPos << 0)      // не входит
        return -1;
    int endPos = str.lastIndexOf(ch);
    return endPos - begPos - 1;
}
```

Данный метод находит первую и последнюю позицию символа ch в строке str. Если символ входит в строку менее двух раз, метод возвращает –1. Разность между номерами позиций превышает на единицу количество символов между ними (количество символов между позициями 2 и 3 равно 0).

Существует несколько перегруженных вариантов метода indexOf для поиска в прямом направлении и метода lastIndexOf — для поиска в обратном направлении. Каждый из методов возвращает номер найденной позиции или –1, если поиск оказался безуспешным:

Метод	Возвращаемое значение
indexOf(char ch)	первая позиция ch
indexOf(char ch, int start)	первая позиция ch start
indexOf(String str)	первая позиция str
indexOf(String str, int start)	первая позиция str start
lastIndexOf(char ch)	последняя позиция ch
lastIndexOf(char ch, int start)	последняя позиция ch start
lastIndexOf(String str)	последняя позиция str
lastIndexOf(String str, int start)	последняя позиция str start

Упражнение 8.1

Напишите метод, который подсчитывает количество вхождений данного символа в строку.

Упражнение 8.2

Напишите метод, который подсчитывает количество вхождений некоторой подстроки в данную строку.

8.2. Сравнение строк

В классе String имеется несколько методов для сравнения строк и их отдельных частей. Тем не менее, перед тем как переходить к конкретным методам, необходимо остановиться на некоторых аспектах, касающихся интернациональных и локализованных строк Unicode, которые не учитываются этими методами. Например, при сравнении двух строк и попытке определить, какая из них “больше”, происходит числовое сравнение символов в соответствии с их значениями в кодировке Unicode, а не их порядком в локализованном представлении. Для француза символы c и з — это одинаковые буквы, отличающиеся между собой лишь маленьким диакритическим значком. Упорядочивая набор строк, француз проигнорирует отличия между ними и поставит “аза” перед “acz”. Однако с Unicode дело обстоит иначе — в наборе символов Unicode c (\u0063) идет перед з (\u00e7), так что при сортировке эти строки окажутся расположенными в обратном порядке.

Первая операция сравнения, equals, возвращает true, если ей передается ссылка на объект String с тем же содержимым, что и у текущего объекта, — то есть если строки имеют одинаковую длину и состоят в точности из одинаковых символов Unicode. Если другой объект не относится к типу String или же имеет другое содержимое, то String.equals возвращает false.

Чтобы сравнивать строки без учета регистра, используйте метод equals IgnoreCase. Под выражением “без учета регистра” мы имеем в виду, что символы Л и л считаются одинаковыми, но отличающимися от E и e. Символы, для которых понятие регистра не определено (например, знаки пунктуации) считаются равными только себе самим. В Unicode имеется много интересных аспектов, связанных с регистром символов, в том числе и понятие “заглавного регистра” (title case). Работа с регистром в классе String описывается в терминах регистровых методов класса Character в разделе 13.5.

Для проведения сортировки строк нужно иметь возможность сравнивать их между собой. Метод compareTo возвращает значение int, которое меньше, равно либо больше нуля, если строка, для которой он был вызван, соответственно меньше, равна или больше другой строки. При сравнении строк используется кодировка символов в Unicode.

Метод compareTo полезен при создании внутреннего канонического упорядочения строк. Например, при проведении бинарного поиска необходимо иметь отсортированный список элементов, однако при этом не требуется, чтобы порядок сортировки совпадал с порядком символов в локализованном алфавите. Метод бинарного поиска для класса, в котором имеется отсортированный массив строк, выглядит следующим образом:

```
private String[] table;

public int position(String key) {
    int lo = 0;
    int hi = table.length - 1;

    while (lo <<= hi) {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(table[mid]);
        if (cmp == 0)      // нашли!
            return mid;
        else if (cmp << 0)  // искать в нижней половине
            hi = mid - 1;
        else               // искать в верхней половине
            lo = mid + 1;
    }
    return -1; //
}
```

Так выглядит базовый алгоритм бинарного поиска. Сначала он проверяет среднюю точку исследуемого диапазона и сравнивает значение ключа поиска с элементом в данной позиции. Если значения совпадают, то нужный элемент найден, а поиск закончен. Если значение ключа меньше элемента в проверяемой позиции, то дальше поиск будет вестись в нижней половине диапазона; в противном случае элемент необходимо искать в верхней половине диапазона. В результате работы алгоритма либо будет найден нужный элемент, либо нижняя граница диапазона превысит верхнюю — это означает, что ключ отсутствует в списке.

Сравнивать можно не только целые строки, но и их отдельные части. Для этого применяется метод regionMatches в двух формах: в одной происходит точное сравнение символов, как в методе equals, а в другой — сравнение без учета регистра, как в методе equalsIgnoreCase:

public boolean **regionMatches(int start, String other, int ostart, int len)**

Возвращает true, если указанная подстрока данного объекта String совпадает с указанной подстрокой строки other. Проверка начинается с позиции start в данной строке, и с позиции ostart - в строке other. Сравниваются только первые len символов.

public boolean **regionMatches(boolean ignoreCase, int start, String other, int ostart, int len)**

Данная версия regionMatches ведет себя точно так же, как и предыдущая, за исключением того, что логическая переменная ignoreCase определяет, следует ли игнорировать регистр символов при сравнении.

Приведем пример:

```
class RegionMatch {
    public static void main(String[] args) {
        String str = "Look, look!";
        boolean b1, b2, b3;

        b1 = str.regionMatches(6, "Look," 0, 4);
        b2 = str.regionMatches(true, 6, "Look," 0, 4);
        b3 = str.regionMatches(true, 6, "Look," 0, 5);

        System.out println("b1 = " + b1);
        System.out println("b2 = " + b2);
        System.out println("b3 = " + b3);
    }
}
```

Результаты работы будут выглядеть следующим образом:

b1 = false

b2 = true

b3 = false

Результат первого сравнения равен false, потому что в позиции 6 главной строки находится символ ‘l’, а в позиции 0 второй строки — символ ‘L’. Второе сравнение дает true, поскольку регистр не учитывается. Наконец, результат третьего сравнения оказывается равным false, потому что длина сравниваемой подстроки равна 5, а на протяжение этих 5 символов строки отличаются даже без учета регистра.

Простая проверка на совпадение аргумента с началом или концом строки осуществляется с помощью методов startsWith и endsWith:

public boolean **startsWith(String prefix, int toffset)**

Возвращает true, если строка начинается с подстроки prefix (со смещением toffset).

public boolean **startsWith(String prefix)**

Сокращение для startsWith(prefix, 0).

public boolean **endsWith(String suffix)**

Возвращает true, если строка заканчивается подстрокой suffix.

Вообще говоря, строки не могут сравниваться с использованием оператора ==, как показано ниже:

if (str == “ВРеса?”)

answer(str);

Такая запись не анализирует содержимое двух строк. Она сравнивает только ссылку на один объект (str) со ссылкой на другой объект (неявный строковый объект, представленный константой “ВРеса?”). Даже если оба объекта-строки имеют одинаковое содержимое, ссылки на них могут различаться.

Тем не менее два любых строковых литерала с одинаковым содержимым будут указывать на один и тот же объект класса String. Например, в следующем фрагменте оператор ==, вероятно, сработает правильно:

```
String str = "?Pena?";
// ...
if (str == "?Pena?")
    answer(str);
```

Из-за того, что str изначально был присвоен строковый литерал, сравнение этой переменной с другим строковым литералом равносильно проверке этих строк на одинаковое содержание. И все же необходимо соблюдать осторожность — этот трюк сработает лишь в том случае, если вы уверены в происхождении всех ссылок на строковые литералы. Если str изменится и будет указывать на производный объект String — например, на результат ввода пользователем чего-либо, — оператор == вернет значение false, даже если пользователь наберет строку ВРеса?.

8.3. Вспомогательные методы

Класс String содержит два метода, которые оказываются полезными в специализированных приложениях. Один из них — hashCode, который возвращает хеш-код, основанный на содержимом строки. Любые две строки с одинаковым содержимым будут иметь одинаковое значение хеш-кода, хотя и две разные строки тоже могут иметь одинаковый хеш-код. Хеш-коды нужны для работы с хеш-таблицами, такими, например, как в классе Hashtable из java.util.

Второй вспомогательный метод, intern, возвращает строку, содержимое которой совпадает с содержимым исходной строки. Однако для любых двух строк с одинаковым содержимым intern возвращает ссылку на один и тот же объект String, что позволяет проверять равенство строк посредством сравнения *ссылок* вместо более медленной проверки *содержимого* строк. Рассмотрим пример:

```
int putIn(String key) {
    String uniqe = key.intern();
    int i;
    // проверить, имеется ли такой элемент в таблице
    for ( i = 0; i << tableSize; i++)
        if (table[i] == unique)
            return i;
    // если нет - добавить
    table[i] = unique;
    tableSize++;
    return i;
}
```

Все строки, хранящиеся в массиве table, получены в результате вызова intern. Массив просматривается в поисках строки, содержимое которой совпадает с key. Если строка найдена, то поиск завершается. Если же такой строки нет, в конец массива добавляется строка, содержимое которой совпадает с содержимым key. При работе с результатами вызовов intern сравнение ссылок на объекты эквивалентно сравнению содержимого строк, однако происходит существенно быстрее.

8.4. Создание производных строк

Некоторые из методов класса String возвращают новые строки, которые отчасти напоминают исходные, но подвергшиеся определенным модификациям. Напоминаем, что новые строки должны возвращаться из-за того, что объекты String доступны только для чтения. Например, для извлечения из строки фрагмента, ограниченного заданными символами, может применяться следующий метод:

```
public static String quotedString(
    String from, char start, char end)
{
    int startPos = from.indexOf(start);
    int endPos = from.lastIndexOf(end);
    if (startPos == -1)      // начальный символ не найден
        return null;
    else if (endPos == -1)  // конечный символ не найден
        return from.substring(startPos);
    else                    // найдены оба символа-ограничителя
        return from.substring(startPos, endPos + 1);
}
```

Метод quotedString возвращает новый объект String, который содержит фрагмент строки from, начинающийся с символа start и заканчивающийся символом end. Если найден символ start, но не найден end, то метод возвращает новый объект String, содержащий все символы от начальной позиции до конца строки. В работе quotedString используются две перегруженные формы метода substring. Первая из них получает только начальную позицию в строке и возвращает новую строку, содержащую все символы с заданной позиции, и до конца строки. Вторая форма получает и начальную, и конечную позиции и возвращает новую строку, содержащую все символы между соответствующими позициями исходной строки; при этом начальный символ-ограничитель включается в подстроку, а конечный — нет. Именно из-за этого принципа “до конца, но не включая конец” мы и прибавляем единицу к endPos, чтобы в подстроку вошли оба символа-ограничителя. Например, вызов

quotedString(“Il a dit “Bonjour!””, ‘“’, ‘”’);

возвращает строку

“Bonjour!”

Ниже перечисляются остальные методы для создания производных строк:

public String **replace(char oldChar, char newChar)**

Возвращает новый объект String, в котором все вхождения символа old Char заменяются символом newChar.

public String **toLowerCase()**

Возвращает новый объект String, в котором каждый символ преобразуется в эквивалентный ему символ нижнего регистра (если он имеется).

public String **toUpperCase()**

Возвращает новый объект String, в котором каждый символ преобразуется в эквивалентный ему символ верхнего регистра (если он имеется).

public String **trim()**

Возвращает новый объект String, в котором удалены все пробелы в начале и конце строки.

Метод concat возвращает новую строку, которая эквивалентна применению оператора + к двум строкам. Следующие два оператора являются эквивалентными:

newStr = oldStr.concat(“ (not)”);

newStr = oldStr + “ (not)”;

Упражнение 8.3

Как показано выше, метод quotedString предполагает, что в исходной строке имеется всего один экземпляр подстроки с заданными символами-ограничителями. Напишите версию метода, которая извлекает все такие строки и возвращает массив.

8.5. Преобразование строк

Довольно часто возникает необходимость преобразовать строку в значение другого типа (скажем, целого или логического) или наоборот. Согласно конвенции, принятой в Java, тип, *к которому* преобразуется значение, должен содержать метод, выполняющий преобразование. Например, преобразование из типа String в Integer должно выполняться статическим методом класса Integer. Ниже приводится таблица всех конвертируемых типов, а также способы их преобразования в тип String и обратно:

Тип	В String	Из String
boolean	String.valueOf (boolean)	new Boolean(String).booleanValue ()
int	String.valueOf(int)	Integer.ParseInt(String, int base)
long	String.valueOf(long)	Long.ParseLong(String, int base)
float	String.valueOf(float)	new Float(String).floatValue()
double	String.valueOf(double)	new Double(String).doubleValue()

Для логических значений, а также для значений с плавающей точкой сначала создается объект Float или Double, после чего определяется его численное значение. Для значений с плавающей точкой не существует эквивалента метода parseInt, который напрямую выделяет значение из строки.

Не существует методов, которые переводили бы символы из форм, распознаваемых языком Java (\b, \udddd и т. д.) в переменные типа char или наоборот. Вы можете вызвать метод String.valueOf для отдельного символа, чтобы получить строку, состоящую из одного данного символа.

Также не существует возможности создать или преобразовать числовые строки в формат языка Java, в котором начальный 0 означает восьмеричную запись, а 0x — шестнадцатеричную.

Преобразования в byte и short, а также обратные им производятся через тип int, поскольку соответствующие значения всегда лежат в диапазоне int; к тому же при использовании этих типов в вычисляемых выражениях они все равно преобразуются в int.

Новые классы также могут поддерживать строковые преобразования; для этого в них следует включить метод toString и конструктор, который создает новый объект по строковому описанию. Классы, включающие метод toString, могут использоваться в valueOf. В соответствии с определением метода valueOf(Object obj), он возвращает либо строку “null”, либо obj.to String. Если все классы в вашей программе содержат метод toString, то вы сможете преобразовать любой объект в тип String вызовом valueOf.

8.6. Строки и символьные массивы

Содержимое строки может отображаться на символьный массив и наоборот. Часто в программе бывает необходимо предварительно построить строку в массиве char, после чего создать объект String по содержимому этого массива. Если описанный ниже класс StringBuffer (допускающий запись в строки) в каком-то конкретном случае не подходит, существует несколько методов и конструкторов класса String, помогающих преобразовать строку в массив char или же массив char — в строку.

Например, чтобы удалить из строки все вхождения определенного символа, можно воспользоваться следующим несложным алгоритмом:

```
public static String squeezeOut(String from, char toss) {
    char[] chars = from.toCharArray();
    int len = chars.length;
    for (int i = 0; i << len; i++) {
        if (chars[i] == toss) {
            --len;
            System.arraycopy(chars, i + 1,
                             chars, i, len - i);
            --i;    // рассмотреть повторно
        }
    }
    return new String (chars, 0, len);
}
```

Метод squeezeOut сначала преобразует свою входную строку from в символьный массив при помощи метода toCharArray. Затем он в цикле перебирает элементы массива в поисках символа toss. Когда такой символ находится, длина возвращаемой строки уменьшается на 1, а все следующие символы массива сдвигаются к началу. Значение i уменьшается, чтобы можно было проверить новый символ в позиции i и выяснить, не следует ли удалить и его. Когда метод завершает просмотр массива, он возвращает новый объект String, содержащий “выжатую” строку. Для этого применяется конструктор String, которому в качестве аргументов передается исходный массив, начальная позиция внутри массива и количество символов.

Кроме того, имеется отдельный конструктор String, который получает в качестве параметра только символьный массив и использует его целиком. Оба этих конструктора создают копии массива, так что после создания String можно изменять содержимое массива — на содержимое строки это не повлияет.

При желании вместо конструкторов можно воспользоваться двумя статическими методами String.copyValueOf. Например, метод squeezeOut мог бы заканчиваться следующей строкой:

return String.copyValueOf(chars, 0, len);

Вторая форма copyValueOf получает один аргумент и копирует весь массив. Для полноты было решено сделать два статических метода copy ValueOf эквивалентными двум конструкторам String.

Метод toCharArray прост и достаточен в большинстве случаев. Когда желательно иметь больше возможностей для контроля за процессом копирования фрагментов строки в символьный массив, можно воспользоваться методом getChars:

public void **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**

Копирует символы из строки в массив. Символы заданной подстроки копируются в массив начиная с dst[dstBegin]. Подстрока представляет собой фрагмент исходной строки, который начинается с позиции srcBegin и заканчивается на srcEnd (но не включает ее!). Любая попытка выхода за пределы строки или массива char приводит к возбуждению исключения IndexOutOfBoundsException.

8.7. Строки и массивы byte

Существуют методы, предназначенные для преобразования массивов 8-разрядных символов в объекты String в 16-разрядной кодировке Unicode и наоборот. Эти методы помогают создавать строки Unicode из символов ASCII или ISO-Latin-1, которые являются первыми 256 символами в наборе Unicode. Данные методы аналогичны своим прототипам, предназначенным для работы с символьными массивами:

public **String(byte[] bytes, int hiByte, int offset, int count)**

Конструктор создает новую строку, состоящую из символов заданного подмассива, входящего в массив bytes с позиции offset и состоящего из count символов. Старшие 8 бит каждого символа могут быть заданы в переменной hiByte, значение которой обычно равно 0, так как конструктор чаще всего используется для преобразования символов 8-разрядной кодировки ASCII или ISO-Latin-1 в 16-разрядные строки Unicode. Если значения аргументов offset и coun t заставляют конструктор обратиться к элементам, выходящим за границы массива, возбуждается исключение IndexOutOfBounds.

public **String(byte[] bytes, int hiByte)**

Сокращенная форма для String(bytesm hibyte, 0, bytes.length).

public void **getBytes(int srcBegin, int srcEnd, byte[] dst, •• int dstBegin)**

Создает копию фрагмента строки в массиве dst, начиная с dst[dstBegin]. При этом теряются старшие 8 бит каждого символа. Копирование производится с позиции srcBegin и заканчивается в srcEnd (но не включает ее!). Любая попытка выхода за пределы строки или массива char приводит к возбуждению исключения IndexOutOfBoundsException.

Оба конструктора String, создающих строки по содержимому массива byte, производят копии данных, так что дальнейшие изменения содержимого массива никак не отражаются на содержимом строки.

8.8. Класс StringBuffer

Если бы программисты могли работать исключительно со строками, доступными только для чтения, приходилось бы создавать новый объект String для каждого промежуточного результата при последовательных операциях со строками. Например, давайте задумаемся над тем, как компилятор будет вычислять следующее выражение:

```
public static String guillemete(String quote) {
     return '"' + quote + '"';
}
```

Если бы компилятор ограничивался выражениями типа String, он действовал бы следующим образом:

```
quoted = String.valueOf('"').concat(quote)
             .concat(String.valueOf('"'));
```

При каждом вызове valueOf и concat создается новый объект String; следовательно, в результате подобной операции появятся четыре объекта String, из которых в дальнейшем будет использован только один. Что касается остальных, то их создание, присвоение начальных значений и удаление будет сопряжено с непроизводительными расходами.

Разумеется, компилятор действует значительно эффективнее. Для построения промежуточных строк используются объекты класса StringBuffer, а итоговые объекты String создаются лишь в случае необходимости. Объекты StringBuffer могут изменяться, поэтому для хранения промежуточных результатов не нужны новые объекты. При помощи StringBuffer приведенное выше выражение может быть представлено в следующем виде:

```
quoted = new StringBuffer().append('"').
             .append(quote).append('"').toString();
```

В этом варианте создается всего один объект StringBuffer, который хранит строку, добавляет к ней новые фрагменты и пользуется методом toString для преобразования результатов своей работы в объект String.

Для построения и модификации строки можно воспользоваться классом StringBuffer. Класс StringBuffer содержит следующие конструкторы:

public **StringBuffer()**

Конструирует новый объект StringBuffer, начальное значение которого равно “”.

public **StringBuffer(String str)**

Конструирует новый объект StringBuffer, исходное значение которого совпадает с str.

Класс StringBuffer во многих отношениях напоминает класс String, а многие из содержащихся в нем методов имеют те же имена и контракты, что и методы String. Тем не менее StringBuffer не является расширением String, и наоборот. Оба этих класса являются независимыми расширениями класса Object.

8.8.1. Модификация буфера

Существует несколько возможностей изменить содержимое буфера объекта StringBuffer, в том числе добавить новые символы в его конец или вставить их в середину. Самый простой из таких методов называется setCharAt и служить для замены символа в конкретной позиции. Также имеется метод replace, который делает то же самое, что и String.replace, однако работает с объектом StringBuffer. Метод replace не нуждается в создании нового объекта для хранения результата, поэтому несколько последовательных вызовов replace могут выполняться с одним буфером:

```
public static void
     replace(StringBuffer str, char from, char to)
{
     for (int i = 0; i << str.length(); i++)
          if (str.charAt(i) == from)
               str.setCharAt(i, to);
}
```

Метод setLength обрезает или расширяет строку, хранящуюся в буфере. Если передать setLength величину, меньшую длины текущей строки, то строка будет обрезана до указанного значения. Если же передаваемая длина превышает текущую, то строка расширяется, а новые позиции заполняются нуль-символами (\u0000).

Кроме того, имеются методы append и insert, которые преобразуют данные любого типа в String, после чего присоединяют результат преобразования к концу строки либо вставляют его в середину. При выполнении метода insert существующие символы сдвигаются, чтобы освободить место для вставляемых новых символов. Методы append и insert преобразуют данные следующих типов:

```
Object       String       char[]
boolean      char int     int
long         float        double
```

Также существуют методы append и insert, которым в качестве аргумента передается часть массива char. Например, для создания объекта String Buffer с описанием квадратного корня из целого числа можно написать следующее:

```
String sqrtInt(int i) {
     StringBuffer buf = new StringBuffer();

     buf.append("sqrt(").append(i).append(')');
     buf.append(" = ").append(Math.sqrt(i));
     return buf.toString();
}
```

Методы append и insert возвращают исходный объект String, что в данном случае позволяет нам добавить новые символы к результату предыдущего действия.

Метод insert получает два параметра. Первый из них — позиция, начиная с которой в StringBuffer будут вставляться новые символы. Второй параметр — вставляемое значение, при необходимости преобразованное в String. Приведем метод, который вставляет добавляет дату в начало буфера:

```
public static StringBuffer addDate(StringBuffer buf) {
     String now = new java.util.Date().toString();
     buf.ensureCapacity(buf.length() + now.length() + 2);
     buf.insert(0, now).insert(now.length(), ": ");
     return buf;
}
```

Все начинается с создания строки, содержащей текущую дату. Для этой цели используется класс java.util.Date; его конструктор по умолчанию создает объект, представляющий текущее время на момент создания. Далее необходимо убедиться, что размеров буфера хватит для хранения всех добавляемых символов — увеличение буфера должно происходить только один раз, а не после каждого вызова insert. Затем в буфер вставляется строка с текущим временем, за которой следует простейшая строка-разделитель. Наконец, переданный буфер возвращается обратно, чтобы при вызове данного метода можно было осуществить что-нибудь наподобие той конкатенации, которая пригодилась при работе с собственными методами StringBuffer.

Метод reverse изменяет порядок следования символов в StringBuffer. Например, если буфер содержит строку “good”, то после выполнения reverse в нем окажется строка “doog”.

8.8.2. Извлечение данных

Чтобы создать объект String на основе объекта StringBuffer, следует вызвать метод toString.

В StringBuffer не существует методов, которые бы удаляли часть содержимого буфера — вам придется преобразовать буфер в символьный массив, удалить то, что нужно, и построить новый буфер по массиву с оставшимися символами. Вероятно, для этого следует воспользоваться методом getChars, который аналогичен методу String.getChars.

public void **getChars(int srcBegin, int srcEnd, char[] dst, ••** int dstBegin)

Копирует символы из заданной части буфера (определяемой позициями srcBegin и srcEnd) в массив dst начиная с dst[dstBegin]. Копирование ведется с позиции srcBegin до srcEnd (но не включает ее!). Позиция srcBegin должна представлять собой допустимый индекс для данного буфера, а позиция srcEnd не может превышать длины текущей строки в буфере (которая на единицу больше, чем последний индекс). Если какой-либо из индексов окажется недопустимым, возбуждается исключение Inde x OutOfBoundsException.

Приведем метод, в котором getChars используется для удаления части содержимого буфера:

```
public static StringBuffer
     remove(StringBuffer buf, int pos, int cnt)
{
     if (pos << 0 || cnt << 0 || pos + cnt >> buf.length())
          throw new IndexOutOfBoundsException();

     int leftover = buf.length() - (pos + cnt);
     if (leftover == 0) {     // простое обрезание строки
          buf.setlength(pos);
          return buf;
     }

     char[] chrs = new char[leftover];
     buf.getChars(pos + cnt, buf.Length(), chrs, 0);
     buf.setLength(pos);
     buf.append(chrs);
     return buf;
}
```

Прежде всего необходимо убедиться в том, что значения индексов массива не выходят за пределы допустимых. Обработать исключение можно и позднее, однако немедленная проверка позволяет лучше контролировать ситуацию. Затем мы вычисляем, сколько элементов находится после удаляемой части массива; если таких элементов нет, обрезаем буфер и возвращаемся. В противном случае, перед тем как возвращаться из метода, необходимо выделить эти символы методом getChars, затем обрезать буфер и приписать к нему выделенный ранее остаток.

8.8.3. Работа с емкостью буфера

Буфер объекта StringBuffer обладает определенной емкостью — так называется максимальная длина строки, которая может поместиться в нем перед тем, как придется выделять дополнительное место. Буфер может увеличиваться автоматически по мере добавления новых символов, однако для повышения эффективности желательно установить его размер один раз.

Исходный размер буфера объекта StringBuffer может быть задан с помощью конструктора, получающего всего один параметр типа int:

public **StringBuffer(int capacity)**

Конструирует новый объект StringBuffer с заданной исходной емкостью и начальным значением “”.

public synchronized void **ensureCapacity(int minimum)**

Позволяет убедиться в том, что буфер имеет емкость не менее заданного minimum.

public int **capacity()**

Возвращает текущую емкость буфера.

С помощью этих методов можно избежать многократного увеличения буфера. Ниже приводится новая версия метода sqrtInt, которая обеспечивает не более чем однократное выделение нового пространства для буфера:

```
String sqrtIntFaster(int i) {
     StringBuffer buf = new StringBuffer(50);
     buf.append("sqrt(").append(i).append(')');
     buf.append(" = ").append(Math.sqrt(i));
     return buf.toString();
}
```

Единственное изменение заключается в том, что на этот раз используется конструктор, который создает достаточно большой объект StringBuffer, способный вместить строку с результатом. Значение 50 несколько превышает максимально необходимое; следовательно, буфер никогда не придется увеличивать заново.

Упражнение 8.4

Напишите метод для преобразования строк с десятичными числами, при котором после каждой третьей цифры справа ставится запятая. Например, для исходной строки ”1542729" метод должен возвращать строку ”1,542,729".

Упражнение 8.5

Измените метод из предыдущего упражнения так, чтобы при его вызове можно было указать символ-разделитель и количество цифр между разделителями.

Содержание | Далее

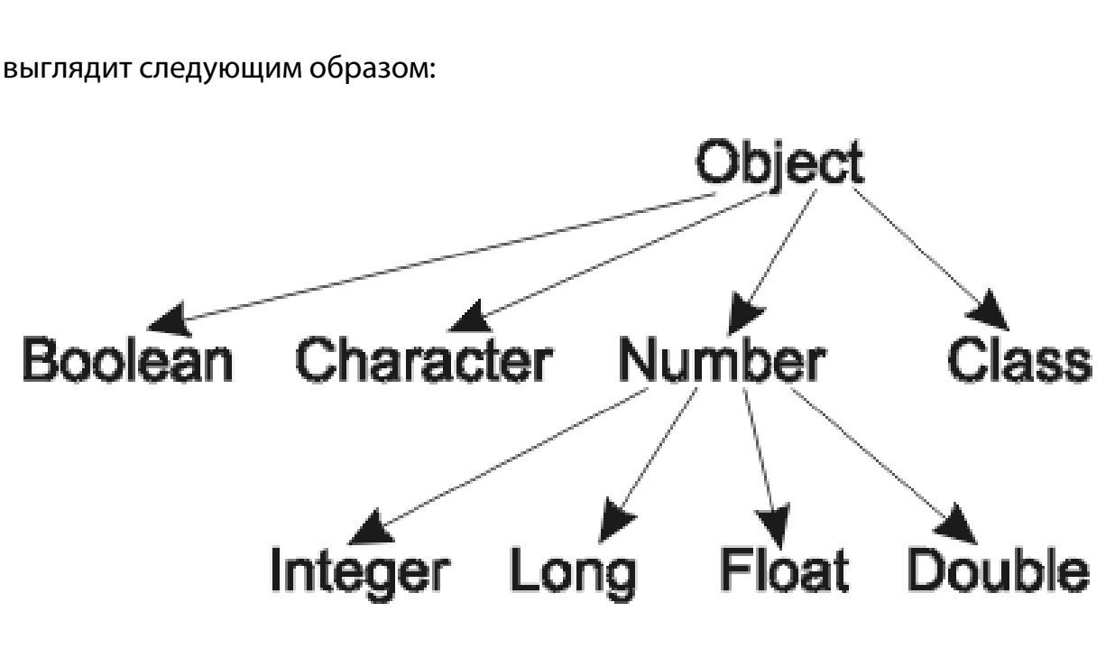
Глава 13
ПРИМЕНЕНИЕ ТИПОВ В ПРОГРАММИРОВАНИИ

Я завернусь в бумагу,
Я намажусь клеем.
Наклейте мне на голову марку!
Я пришлю тебе себя по почте.
Вуди Гатри, Почтовая песня

Типы в языке Java представлены классами. Почти для каждого из примитивных типов (int, boolean и т. д.) существует отдельный класс, известный под названием "оболочки" (wrapper); кроме того, имеется класс Class, представляющий типы классов и интерфейсов. Такие классы обладают следующими преимуществами:

- Полезные статические методы для конкретного типа получают естественное "место жительства". Например, методы для преобразования строки в float являются статическими методами класса Float.
- То же самое справедливо и для описательных методов и полей. В каждом из классов для примитивных числовых типов присутствуют константы MIN_VALUE и MAX_VALUE; с помощью объекта Class можно получить доступ к методам, описывающим супертипы класса.
- Для значений, относящихся к примитивным типам, можно создавать объекты-оболочки. Затем эти объекты используются в любом контексте, где требуется ссылка на класс Object. По этой причине классы для примитивных типов называются *классами-оболочками* .

Иерархия типов для этих классов выглядит следующим образом:



Классы для short или byte отсутствуют, поскольку эти типы используются главным образом из соображений экономии памяти. Все арифметические вычисления для short и byte производятся с выражениями типа int. Чтобы сохранить значение типа short или byte в объекте, пользуйтесь классом Integer. К сожалению, это также означает, что для типов byte и short отсутствуют константы MIN_VALUE и MAX_VALUE.

В данной главе показано, как пользоваться классами-оболочками. В первой части главы рассматриваются объекты Class, которые представляют конкретные классы и интерфейсы. Оставшаяся часть главы посвящена программированию с использованием классов-оболочек для примитивных типов.

13.1. Класс Class

Для каждого класса и интерфейса в системе имеется представляющий его объект Class. Этот объект может использоваться для получения основных сведений о классе или интерфейсе и для создания новых объектов класса.

Класс Class позволяет перемещаться по иерархии типов в программе. Такая иерархия фактически становится частью программной среды, что облегчает процесс отладки и автоматического документирования, а также делает программу более последовательной. Кроме того, это открывает новые возможности работы с классами — в первую очередь для создания объектов (их тип может задаваться в виде строки) и вызова классов с использованием специальных приемов (например, загрузка по сети).

Существует два способа получить объект Class: запросить его у имеющегося объекта методом getClass или искать его по уточненному (включающему все имена пакетов) имени статическим методом Class.forName.

Простейшие методы Class предназначены для перемещения по иерархии типов. Приведенный ниже класс выводит такую иерархию для конкретного объекта Class:

```
public class TypeDesc {
     public static void main(String[] args) {
          TypeDesc desc = new TypeDesc();
          for (int i = 0; i << args.length; i++) {
               try {
                    desc.printType(Class.forName(args[i]), 0);
               } catch (ClassNotFoundException e) {
                    System.err.print(e);  // сообщить об ошибке
          }
     }

     // по умолчанию работать со стандартным выводом
     public java.io.PrintStream out = System.out;
     // используется в printType() для пометки имен типов
     private static String[]
          basic    = { "class",   "interface"   },
          extended = { "extends", "implements"  };

     public void printType(Class type, int depth) {
          if (type == null) // супертип Object равен null
               return;

          // вывести тип
          for (int i = o; i << depth; i++)
               out.print("  ");
          String[] labels = (depth == 0 ? basic : extended);
          out.print(labels[type.isInterface() ? 1 : 0] +  " ");
          out.println(type.getName());

          // вывести интерфейсы, реализуемые классом
          Class[] interfaces = type.getInterfaces();
          for (int i = o; i << interfaces.length; i++)
               printType(interfaces[i], depth + 1);

          // рекурсивный вызов для суперкласса
          printType(type.getSuperclass(), depth + 1);
     }
}
```

Данный пример просматривает имена, введенные в командной строке, и вызывает printType для каждого из них. Делать это необходимо в try-блоке на тот случай, если класс с заданным именем отсутствует. Ниже показан результат работы программы для класса java.util.Hashtable (используется полное уточненное имя, поскольку этого требует метод forName):

```
class java.util.Hashtable
     implements java.lang.Cloneable
          extends java.lang.Object
     extends java.util.Dictionary
          extends java.lang.Object
```

Далее в тексте программы следует объявление выходного потока. Для краткости мы объявили его открытым, но в реальном приложении он должен быть закрытым, а обращения к нему должны осуществляться только через методы доступа. Затем следует описание двух строковых массивов.

Метод printType выводит описание своего типа, а затем рекурсивно вызывает себя для распечатки свойств супертипов. Параметр depth показывает, на сколько уровней мы поднялись в иерархии типов; в зависимости от его значения каждая строка с описанием снабжается соответствующим отступом. С каждым новым уровнем рекурсии это значение увеличивается.

При выводе типа метод isInterface определяет, является ли тип интерфейсом. Результат вызова используется для выбора префикса — пояснительной надписи. В самом низу иерархии типов, где значение depth равно 0, выводятся надписи "class" и "interface"; типы, находящиеся выше в иерархии, расширяют или реализуют свои исходные типы, поэтому используются термины "extends" и "implements". Именно для этого и создаются массивы Basic и Extended. После выбора нужного префикса имя типа выводится методом getName. Конечно, класс Class содержит метод toString, однако в этом методе уже использован префикс "class" или "interface". Мы хотим сами контролировать префикс, и потому пришлось создать собственную реализацию метода.

После вывода описания типа метод printType осуществляет рекурсивный вызов себя самого. Сначала определяются все интерфейсы, реализуемые исходным типом. /Если тип, для которого выводится информация, представляет собой интерфейс, то он расширяет, а не реализует свои интерфейсы-супертипы. Тем не менее, учет таких подробностей привел бы к неоправданному усложнению кода./ Затем выводится расширяемый им суперкласс (если он существует). Постепенно метод доходит до объекта Class класса Object, который не реализует никаких интерфейсов и для которого метод getSuperClass возвращает null; на этом рекурсия завершается.

Упражнение 13.1

Модифицируйте TypeDesc, чтобы избежать вывода информации о классе Object. Эти сведения избыточны, поскольку каждый объект в конечном счете расширяет класс Object. Используйте ссылку на объект Class для типа Object.

Объект Class может воспользоваться методом newInstance для создания нового экземпляра (объекта) представляемого им типа. При этом вызывается безаргументный конструктор класса или возбуждается исключение NoSuch MethodError, если класс не имеет безаргументного конструктора. Если класс или безаргументный конструктор недоступны (не являются открытыми или находятся в другом пакете), возбуждается исключение IllegalAccessException. Если класс является абстрактным, или представляет собой интерфейс, или создание завершилось неудачно по какой-то другой причине, возбуждается исключение InstantiationException. Создавать новые объекты подобным образом оказывается удобно, когда вы хотите написать универсальный код и позволить пользователю задать нужный класс. Например, в программе тестирования алгоритмов сортировки, приведенной в разделе "Проектирование расширяемого класса", пользователь мог ввести имя тестируемого класса и использовать его в качестве параметра для вызова forName. Если введенное имя класса окажется допустимым, можно вызвать метод newInstance для создания объекта этого типа. Метод main для универсального класса SortDouble выглядит следующим образом:

```
static double[] testData = { 0.3, 1.3e-2. 7.9. 3.17, );

public static void main(String[] args) {
     try {
          for (int arg = 0; arg << args.length; arg++) {
               String name = args[arg];
               Class classFor = Class.forName(name);
               SortDouble sorter
                    = (SortDouble)classFor.newInstance();
               SortMetrics metrics
                    = sorter.sort(testData);
               System.out.println(name + ": " + metrics);
               for (int i = 0; i << testData.length; i++)
                    System.out.println("\t" + testData[i]);
          }
     } catch (Exception e) {
          System.err.print(e);     // сообщить об ошибке
     }
}
```

Этот метод почти совпадает с BubbleSortDouble.main, однако из него исключены все имена типов. Он применим к любому типу, объекты которого могут использоваться в качестве объектов SortDouble и который содержит безаргументный конструктор. Теперь нам не придется переписывать метод main для каждого алгоритма сортировки — универсальный main годится для всех случаев. Все, что нужно сделать, — выполнить команду

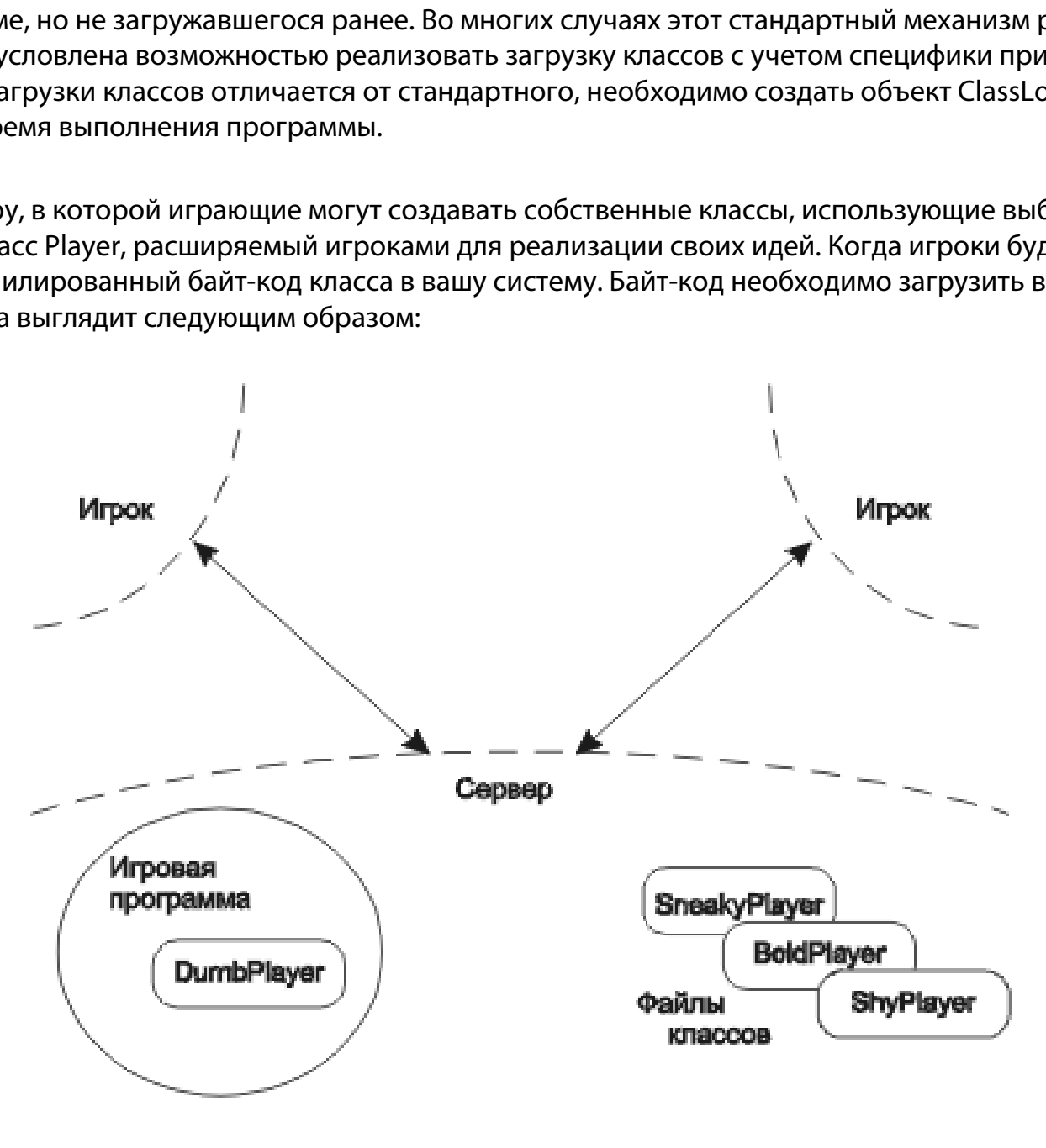
java SortDouble TestClass ...

для любого класса-сортировщика (наподобие BubbleSortDouble); класс будет загружен и выполнен.

13.2. Загрузка классов

Runtime-система Java обращается к классам, когда в этом возникает необходимость. Подробности загрузки классов могут отличаться для различных реализаций Java, однако в большинстве случаев используется механизм "пути класса" для поиска компилированного байт-кода класса, используемого в программе, но не загружавшегося ранее. Во многих случаях этот стандартный механизм работает отлично, однако немалая часть достоинств Java обусловлена возможностью реализовать загрузку классов с учетом специфики приложения. Чтобы написать программу, в которой механизм загрузки классов отличается от стандартного, необходимо создать объект ClassLoader, который получает байт-коды классов и загружает их во время выполнения программы.

Например, вы разрабатываете игру, в которой играющие могут создавать собственные классы, использующие выбранную ими стратегию. Для этого вы создаете абстрактный класс Player, расширяемый игроками для реализации своих идей. Когда игроки будут готовы испытать свою стратегию, они пересылают скомпилированный байт-код класса в вашу систему. Байт-код необходимо загрузить в игру, применить и вернуть игроку его результат (score). Схема выглядит следующим образом:



На сервере игровая программа загружает каждый ожидающий своей очереди класс Player, создает объект нового типа и дает ему возможность противопоставить свою стратегию игровому алгоритму. Когда выясняется результат, он сообщается игроку—разработчику стратегии.

Механизм коммуникаций здесь не рассматривается, однако он может быть упрощен до сообщений электронной почты, с помощью которых игроки посылают свои классы и получают результаты.

Наибольший интерес представляет процесс загрузки игровой программой скомпилированных классов. Здесь всем заправляет *загрузчик классов*. Чтобы создать загрузчик классов, следует расширить абстрактный класс Class Loader и реализовать в подклассе метод loadClass:

protected abstract Class **loadClass(String name, boolean resolve)** throws ClassNotFoundException

Загружает класс с заданным именем name. Если значение resolve равно true, то метод должен вызвать resolveClass, чтобы обеспечить загрузку всех классов, используемых внутри заданного.

В нашем примере мы создадим класс PlayerLoader, предназначенный для чтения байт-кода классов со стратегией игры и подготовки их к работе. Основной цикл будет выглядеть следующим образом:

```
public class Game {
     static public void main(String[] args) {
          String name;    // имя класса
          while ((name = getNextPlayer()) != null) {
               try {
                    PlayerLoader loader = new PlayerLoader();
                    Class
                         classOf = loader.loadClass(name, true);
                    Player
                         player = (Player)classOf.newInstance();
                    Game game = new Game();
                    player.play(game);
                    game.reportScore(name);
               } catch (Exception e) {
                    reportException(name, e);
               }
          }
     }
}
```

Для каждой новой игры нужен свой объект-загрузчик PlayerLoader; следовательно, новый класс Player не будет смешиваться с классами, загруженными ранее. Новый PlayerLoader загружает класс и возвращает представляющий его объект Class, который используется для создания нового объекта класса Player. Затем мы создаем новую игру game и играем в нее. После ее завершения возвращается результат.

Класс PlayerLoader расширяет класс ClassLoader и задает собственную реализацию метода loadClass:

```
class PlayerLoader extends ClassLoader {
     private Hashtable Classes = new Hashtable();

     public Class loadClass(String name, boolean resolve)
          throws ClassNotFoundException
     {
          try {
               Class newClass = (Class)Classes.get(name);
               if (newClass == null) {  // еще не определен
                    try {               // проверить, не является
                                        // ли системным классом
                         newClass = findSystemClass(name);
                         if (newClass != null)
                              return newClass;
                    } catch (ClassNotFoundException e) {
                         ;              // продолжить поиск
                    }

                    // класс не найден - его необходимо загрузить
                    byte[] buf = bytesForClass(name);
                    newClass = defineClass(buf, 0, buf.length);
                    Classes.put(name, newClass);
               }
               if (resolve)
                    resolveClass (newClass);
               return newClass;
          } catch (IOException e) {
               throw new ClassNotFoundException(e.toString());
          }
     }

     // ... bytesForClass() и все прочие методы ...
}
```

Любая реализация loadClass должна загружать класс лишь в том случае, если это не было сделано ранее, поэтому метод содержит хеш-таблицу загруженных классов. Если класс уже присутствует в ней, то возвращается соответствующий объект Class. В противном случае loadClass сначала проверяет, можно ли найти класс в локальной системе, для чего вызывает метод find SystemClass класса ClassLoader; этот метод осуществляет поиск не только среди системных классов (находящихся в пакетах java), но и в пути, заданном для классов. Если при этом будет найден нужный класс, то после выполнения загрузки возвращается соответствующий ему объект Class.

Если поиск в обоих случаях заканчивается безрезультатно, необходимо прочитать байт-код класса, для чего служит метод bytesForClass:

```
protected byte[] bytesForClass(String name)
     throws IOException, ClassNotFoundException
{
     FileInputStream in = streamFor(name);
     int length = in.available();   // получить количество байтов
     if (length == 0)
          throw new ClassNotFoundException(name);
     byte[] buf = new byte[length];
     in.read(buf);                  // прочитать байт-код
     return buf;
}
```

В нем использован метод streamFor (не приводится) для получения потока FileInputStream, содержащего байт-код класса. Затем мы создаем буфер нужного размера, считываем весь байт-код и возвращаем буфер.

Когда loadClass получает байт-код и вызывается метод defineClass класса ClassLoader, в качестве параметров этот метод получает массив byte, начальное смещение и количество байтов — в указанной части массива должен находиться байт-код класса. В данном случае байт-код занимает весь массив. Когда определение класса завершается, он добавляется в хеш-таблицу Classes, чтобы предотвратить его повторную загрузку.

После успешной загрузки класса метод loadClass возвращает новый объект Class.

Невозможно выгрузить класс, когда он перестает быть нужным. Вместо этого вы просто прекращаете его использование, чтобы класс-загрузчик мог быть уничтожен сборщиком мусора.

Чтобы получить объект-загрузчик для заданного объекта Class, следует вызвать его метод getClassLoader. Процесс загрузки классов был рассмотрен выше. Если у данного класса отсутствует класс-загрузчик, метод возвращает null.

Класс-загрузчик применяется лишь на первой стадии подготовки класса к работе. Всего же существует три стадии:

1. Загрузка: получение байт-кода с реализацией класса.

2. Компоновка: поиск супертипов класса и загрузка их в случае необходимости.

3. Инициализация: присвоение начальных значений статическим полям класса и выполнение их инициализаторов, а также всех статических блоков.

Упражнение 13.2

Реализуйте классы Game и Player для какой-нибудь простой игры — например, "крестики-нолики". Проведите оценку различных реализаций Player по данным нескольких запусков.

13.3. Классы-оболочки: общий обзор

Для большинства примитивных типов в языке Java существуют классы, представляющие значения данного типа. Эти *классы-оболочки* обладают двумя основными функциями. Первая — в них находятся методы и переменные, относящиеся к типу (например, методы строковых преобразований и константы для границ диапазона). Следующий пример показывает, как можно проверить, допустимо ли для данной величины выполнение быстрых вычислений типа float или же ее диапазон выходит за границы, разрешенные для float:

if (value >= Float.MIN_VALUE && value <= Float.MAX_VALUE)

return fasterFloatCalc((float)value);

else

return slowerDoubleCalc(value);

Вторая функция заключается в возможности создания объектов, содержащих значения определенного примитивного типа, для универсальных классов, умеющих работать только со ссылками на Object. Например, объекты Hash table могут содержать только ссылки на Object, а не на примитивные типы. Чтобы использовать int в качестве ключа или элемента в объекте Hashtable, необходимо создать объект Integer, содержащий нужное значение:

Integer keyObj = new Integer(key);

map.put(keyObj, value);

В оставшейся части этой главы рассматриваются методы и константы, входящие в классы-оболочки Java. Некоторые из них являются общими для всех классов, поэтому они будут упомянуты до того, как мы перейдем к конкретным классам.

Следующие конструкторы и методы присутствуют во всех классах-оболочках:

- Конструктор, который получает значение примитивного типа и создает объект класса (например, конструктор Character(char)).
- Конструктор, который определяет исходное значение объекта по единственному параметру типа St ring.
- Метод toString, который возвращает строковое представление объекта.
- Метод typeValue, который возвращает значение примитивного типа — например, Character.charValue или Boolean.booleanValue.
- Метод equals, который определяет, равны ли между собой объекты, относящиеся к одному классу.
- Метод hashCode, который возвращает хеш-код, используемый в хеш-таблицах.

Эти методы имеются в каждом из классов-оболочек, поэтому они не приводятся в приведенных ниже описаниях для конкретных классов. Методы выборки и декодирования системных свойств рассматриваются в разделе "Системные свойства" и также отсутствуют в описаниях классов.

Термин "основание", иногда используемый в классах-оболочках, соответствует термину "основание системы счисления". Например, декодирование значения long по основанию 8 означает то же самое, что и перевод в восьмеричную систему счисления.

13.4. Класс Boolean

Класс Boolean служит для представления логического типа boolean. Метод valueOf и конструктор со строковым параметром считают, что строка "true" при любом сочетании символов верхнего и нижнего регистра соответствует true; любая другая строка интерпретируется как false.

13.5. Класс Character

Класс Character служит для представления символьного типа char. Помимо констант MIN_VALUE и MAX_VALUE, он содержит две константы MIN_RADIX и MAX_RADIX, которые равны минимальному и максимальному основанию системы счисления, которые используются методами (описанными ниже) для перевода отдельного цифрового символа в его целочисленный эквивалент и наоборот. Основание должно находиться в диапазоне 2–36; цифры свыше 9 представлены буквами от A до Z или их эквивалентами в нижнем регистре.

public static int **digit(char ch, int radix)**

Возвращает численный эквивалент цифры ch в системе счисления с основанием radix. Если символ не является допустимой цифрой, то возвращается –1.

public static char **forDigit(int digit, int radix)**

Возвращает символьное значение заданной цифры в заданной системе счисления. Если цифра является недопустимой в этой системе, возвращается символ \u0000.

Класс Character также содержит методы для работы с различными атрибутами символов, включая регистр. В Unicode имеется три разновидности регистра: верхний, нижний и заглавный (title case). Верхний и нижний регистры, наверное, хорошо знакомы большинству читателей. Заглавный регистр используется для обозначения символов, которые состоят из нескольких компонентов и написание которых изменяется в заголовках, где первые буквы слов обычно делают прописными. Например, в слове "ljepotica" /Слово "ljepotica" на хорватском языке означает "красавица"./ первая буква является символом lj нижнего регистра (\u01c9, символ из набора Extended Latin, используемого при записи хорватских диграфов). Если это слово встретится в названии книги и вы захотите преобразовать первые буквы всех слов в верхний регистр, то необходимо вызвать для них метод toTitleCase, что в нашем случае даст строку "Ljepotica" (в которой используется символ Lj с кодом \u01c8). Если же по ошибке вызвать метод toUpperCase, будет получена неправильная строка "LJepotica" (символ LJ с кодом \u01c7).

Все проблемы в работе с регистрами решаются в соответствии со стандартом Unicode. Например, в грузинском языке буквы верхнего регистра считаются архаичными, и преобразование в верхний регистр обычно нежелательно. Поэтому метод toUpperCase не переводит грузинские буквы нижнего регистра в верхний, хотя метод toLowerCase переведет буквы из верхнего регистра в нижний. Поэтому нельзя быть уверенным в том, что символы, отличающиеся только регистром, станут одинаковыми после вызова toUpperCase или toLowerCase. Тем не менее выражение

Character.toUpperCase(Character.toLowerCase(ch));

дает символ, который можно сравнить с другим символом, полученным аналогичным образом. Если полученные символы совпадают, то исходные символы также были одинаковыми и могли отличаться только регистром.

public static boolean **isDefined(char ch)**

Возвращает true, если символ ch определен в стандарте Unicode.

public static boolean **isLowerCase(char ch)**

Возвращает true, если символ ch является буквой нижнего регистра.

public static boolean **isUpperCase(char ch)**

Возвращает true, если символ ch является буквой верхнего регистра.

public static boolean **isTitleCase(char ch)**

Возвращает true, если символ ch является буквой заглавного регистра.

public static boolean **isDigit(char ch)**

Возвращает true, если символ ch является цифрой (см. табл. 4 на стр.).

public static boolean **isLetter(char ch)**

Возвращает true, если символ ch является буквой (см. примечание).

public static boolean **isLetterOrDigit(char ch)**

Возвращает true, если символ ch является буквой или цифрой.

public static boolean **isJavaLetter(char ch)**

Возвращает true, если с символа ch может начинаться идентификатор в Java — конкретно, если он является буквой или одним из символов '_' или '$'.

public static boolean **isSpace(char ch)**

Возвращает true, если ch является стандартным символом-разделителем — ' ', '\t', '\n', '\f' или '\r'.

public static char **toLowerCase(char ch)**

Переводит символ ch в нижний регистр. Если эквивалент символа в нижнем регистре отсутствует, возвращается ch.

public static char **toUpperCase(char ch)**

Переводит символ ch в верхний регистр. Если эквивалент символа в верхнем регистре отсутствует, возвращается ch.

public static char **toTitleCase(char ch)**

Переводит символ ch в заглавный регистр. Если эквивалент символа в заглавном регистре отсутствует, возвращается ch.

public static int **digit(char ch, int radix)**

Возвращает численное значение символа ch, рассматриваемого в качестве цифры с заданным основанием. При недопустимом основании возвращается –1. Буквы AZ и az используются для цифр, которые представляют значения 10 и более.

public static char **forDigit(int digit, int radix)**

Возвращает символ, представляющий цифру digit с заданным основанием radix. Если digit превышает radix или значение radix выходит за пределы разрешенного диапазона, возвращается –1.

13.6. Класс Number

Абстрактный класс Number расширяется всеми классами-оболочками, представляющими числовые примитивные типы: Integer, Long, Float и Double. Каждый из этих классов содержит конструкторы, которые присваивают объекту либо значение соответствующего примитивного типа, либо представляющей его строки. В том случае, если строка неверна, конструктор со строковым параметром возбуждает исключение NumberFormatException.

Абстрактные методы Number возвращают значение объекта, преобразованное в один из следующих числовых типов:

public int **intValue()**

public int **longValue()**

public int **floatValue()**

public int **doubleValue()**

Каждый класс, расширяющий Number, переопределяет эти методы для преобразования своего типа в любой другой по тем же правилам, которые применяются при явном приведении типов. Например, для объекта Float со значением 32.87 метод intValue возвращает 32 — столько же, сколько дает выражение (int)32.87.

Кроме того, каждый числовой класс содержит следующие методы и константы:

- Статический метод toString(*тип*), который возвращает объект String для заданного значения примитивного типа. Если строка не подходит к формату данного типа, конструктор возбуждает исключение Number FormatException.
- Статический метод valueOf(String), который возвращает объект числового типа по строке. Все эти методы возбуждают NumberFormatException, если содержимое строки не является допустимым представлением примитивного типа.
- Статические final-константы с именами MIN_VALUE и MAX_VALUE для минимального и максимального значения, принимаемого переменными типа.

13.7. Класс Integer

Класс Integer расширяет Number и служит для представления типа int в виде класса. Для типов short и byte классы-оболочки отсутствуют, поэтому соответствующие значения должны храниться в объектах Integer. Помимо стандартных методов класса Number, класс Integer содержит следующие методы:

public static int **parseInt(String str, int radix)**

throws NumberFormatException

Возвращает значение типа int, вычисленное по строке str в системе счисления с заданным основанием radix. Если преобразование с заданным основанием невозможно, возбуждается исключение NumberFormat Exception.

public static int **parseInt(String str)** throws NumberFormatException

Эквивалентен parse Int(str, 10).

public static Integer **valueOf(String str, int radix)** throws NumberFormatException

Аналогично parseInt(str, Radix), за исключением того, что вместо int возвращается объект Integer.

public static String **toString(int i, int radix)**

Дополнительный статический метод, который возвращает объект String, представляющий i в системе счисления с заданным основанием radix. Стандартный метод toString предполагает работу в десятичной системе счисления.

Кроме того, имеется три статических метода toHexString, toOctal String и toBinaryString, которые преобразуют аргумент типа int в шестнадцатеричную, восьмеричную или двоичную строку соответственно.

13.8. Класс Long

Класс Long расширяет Number и служит для представления типа long в виде класса. Помимо стандартных для Number методов, класс Long содержит следующие методы:

public static long **parseLong(String str, int radix)** throws NumberFormatException

Возвращает значение типа long, вычисленное по строке str в системе счисления с заданным основанием radix. Если строка не может быть преобразована в long с заданным основанием, возбуждается исключение NumberFormatException.

public static long **parseLong(String str)** throws NumberFormatException

Эквивалентен parseLong(str, 10).

public static Long **valueOf(String str, int radix)** throws NumberFormatException

Аналогично parseLong(str, radix), за исключением того, что вместо long возвращается объект Long.

public static String **toString(long l, int radix)**

Дополнительный статический метод, который возвращает объект String, представляющий l в системе счисления с заданным основанием radix. Стандартный метод toString предполагает работу в десятичной системе счисления.

По аналогии с эквивалентными методами Integer, статические методы toHexString, toOctalString и toBinaryString преобразуют аргумент типа long в шестнадцатеричную, восьмеричную или двоичную строку соответственно.

13.9. Классы Float и Double

Классы Float и Double расширяют Number и служат для представления типов float и double в виде класса. За редкими исключениями, имена методов и константы совпадают для обоих типов. Приведенный ниже список соответствует классу Float, однако float и Float всюду могут быть заменены на double и Double соответственно, что даст эквивалентные поля и методы для класса Double. Помимо стандартных методов класса Number, классы Float и Double содержат следующие методы:

public final static float **POSITIVE_INFINITY**

Значение для +.

public final static float **NEGATIVE_INFINITY**

Значение для –.

public final static float **NaN**

"Не-число" ("Not-a-Number", NaN). Константа предназначена для задания значения NaN, но не для его проверки. Чтобы узнать, является ли число NaN, следует воспользоваться методом isNaN, а не сравнением с этой константой.

public static boolean **isNaN(float val)**

Возвращает true, если val представляет собой не-число (NaN).

public static boolean **isInfinite(float val)**

Возвращает true, если val представляет собой положительную или отрицательную бесконечность.

public boolean **isNaN()**

Возвращает true, если значение текущего объекта представляет собой не-число (NaN).

public boolean **isInfinite()**

Возвращает true, если значение текущего объекта представляет собой положительную или отрицательную бесконечность.

Кроме перечисленных выше методов, Float также содержит конструктор, который получает аргумент типа double. Этот аргумент используется в качестве исходного значения после его приведения к float.

Для манипуляций с битами внутри представления числа с плавающей точкой Double содержит методы для получения битовой последовательности типа long и для ее обратного приведения к типу double. Класс Float содержит эквивалентные методы для преобразования значения типа float в битовую последовательность типа int и наоборот:

public static int **floatToIntBits(float value)**

Возвращает битовое представление значения float в виде int.

public static float **intBitsToFloat(int Bits)**

Возвращает значение float, соответствующее заданному битовому представлению.

public static int **doubleToLongBits(double value)**

Возвращает битовое представление значения double в виде long.

public static double **longBitsToDouble(long Bits)**

Возвращает значение double, соответствующее заданному битовому представлению.

Упражнение 13.3

Напишите программу, которая читает файл, состоящий из элементов вида "*тип значение*", где *тип* — название одного из классов (Boolean, Character и т. д.), а *значение* — строка, которую может воспринимать конструктор заданного типа. Для каждого элемента файла создайте объект нужного типа с указанным значением и добавьте его к вектору Vector. Выведите окончательный результат.

Содержание | Далее

Глава 14
СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Глендаур: Я духов вызывать могу из бездны.
Хотспер: И я могу, и каждый это может,
Вопрос лишь, явятся ль они на зов.
Вильям Шекспир, “Генрих IV”, перевод Е. Бируковой

В этой главе рассказано, как работать с общими функциями runtime-системы Java и операционной системы. К ним относятся: определение системных свойств, математические вычисления, запуск других программ, управление памятью, отсчет времени. Эти функции предоставляются четырьмя классами Java:

- Класс Runtime описывает состояние runtime-системы Java. Объект этого класса отвечает за доступ к функциям времени выполнения — например, управлению сборщиком мусора и прекращением работы.
- Класс Process представляет выполняемый процесс, созданный вызовом метода Runtime.exec.
- Класс System содержит статические методы для работы с состоянием системы в целом. Некоторые методы System оперируют с текущим runtime-контекстом.

Класс Math содержит статические методы для выполнения многих стандартных математических вычислений — например, для определения значения тригонометрических функций и логарифмов.

14.1. Стандартный поток ввода/вывода

Вы можете осуществлять стандартные операции ввода/вывода с помощью трех системных потоков, которые являются статическими полями класса System,— System.in, System.out и System.err:

public static InputStream **in**

Стандартный входной поток для чтения символьных данных.

public static OutputStream **out**

Стандартный выходной поток для вывода сообщений.

public static PrintStream **err**

Стандартный поток для вывода сообщений об ошибках. Пользователи часто перенаправляют стандартный вывод программы в файл, однако приложение при этом должно иметь возможность вывести сообщение об ошибке так, чтобы пользователь его увидел. Поток err предназначен для тех сообщений об ошибках, которые не перенаправляются вместе со стандартным выводом. Потоки out и err являются объектами класса PrintStream, поэтому для вывода сообщений в err используются те же методы, что и для out.

14.2. Управление памятью

Хотя Java не позволяет явно уничтожать ненужные объекты, вы можете непосредственно вызвать сборщик мусора, используя метод gc класса Runtime. Класс Runtime также содержит метод runFinalization для вызова ожидающих блоков завершения (finalizers). Класс Runtime содержит два метода для вывода информации о состоянии памяти:

public long **freeMemory()**

Возвращает примерное количество свободных байтов системной памяти.

public long **totalMemory()**

Возвращает общее количество байтов системной памяти.

Класс System содержит статические методы gc и runFinalization, которые вызывают соответствующий метод для текущего runtime-контекста.

Не исключено, что метод Runtime.gc не сможет освободить дополнительную память за счет избавления от “мусора” — его может и не быть, к тому же не все сборщики мусора могут находить ненужные объекты по требованию. Тем не менее перед созданием большого количества объектов (особенно в приложениях, критических по времени, на работе которых могут отрицательно сказаться накладные расходы по сборке мусора) все же стоит вызвать метод gc. Он приносит двойную пользу: вы начинаете работу с максимальным объемом свободной памяти и сокращаете вероятность вызова сборщика мусора во время выполнения программы. Следующий метод освобождает всю возможную память:

```
public static void fullGC() {
     Runtime rt = Runtime.getRuntime();
     long isFree = rt.freeMemory();
     long wasFree;
     do {
          wasFree = isFree;
          rt.gc();
          isFree = rt.freeMemory();
     } while (isFree >> wasFree);
     rt.runFinalization();
}
```

Данный метод в цикле вызывает gc, при этом объем свободной памяти freeMemory увеличивается до определенного предела, после достижения которого дальнейшие вызовы gc, скорее всего, ни к чему не приведут. Затем мы обращаемся к runFinalization, чтобы немедленно выполнить все завершающие действия, не позволяя сборщику мусора отложить их на потом.

Обычно вам незачем использовать runFinalization, поскольку методы finalize вызываются сборщиком мусора асинхронно. Однако при некоторых обстоятельствах (например, при нехватке ресурса, освобождаемого методом finalize) вынужденное исполнение всех возможных завершающих действий способно принести пользу. Конечно, нет никакой гарантии, что ожида-ющие завершения объекты используют данный ресурс, так что вызов run Finalization может оказаться бесполезным.

14.3. Системные свойства

Существует ряд системных свойств, которые хранятся внутри класса System в виде объекта класса Properties. Они определяют системное окружение и используются классами, которым необходима соответствующая информация. Например, приведем распечатку свойств одной системы:

```
#System properties
#Tue Feb 27 19:45:22 EST 1996
java.home=/lab/east/tools/java/java
java.version=1.0.1
file.separator=/
line.separator=\n
java.vendor=Sun Microsystem Inc.
user.name=arnold
os.arch=sparc
os.name=Solaris
java.vendor.url=http://www.sun.com/
user.dir=/vob/java_prog/src
java.class.path=.:./classes:/home/arnold/java/lib/
classes.zip:/home/arnold/java/classes
java.class.version=45.3
os.version=2.x
path.separator=:
user.home=/home/arnold
```

Все перечисленные выше свойства определены во всех системах, хотя их значения, конечно же, меняются. Некоторые из них применяются стандартными пакетами Java. Например, класс File использует свойство file.separator для построения и анализа путей к файлам. Программисты тоже могут задействовать эти свойства. Следующий метод ищет файл конфигурации в личной папке пользователя:

```
public static File personalConfig(String fileName) {
     String home = System.getPropety("user.home");
     if (home == null)
          return null;
     else
          return new File(home, fileName);
}
```

Ниже перечислены все методы класса System, которые служат для работы с системными свойствами:

public static Properties **getProperties()**

Возвращает объект класса Properties, представляющий системные свойства.

public static void **setProperties(Properties props)**

Задает системные свойства, используя для этого заданный объект класса Properties.

public static String **getProperty(String key)**

Возвращает текущее значение системного свойства, заданного в виде строки key. Эквивалентно

System.getProperties().getProperty(key);

public static String **getProperty(String key,**

String defaultValue)

Возвращает текущее значение системного свойства, заданного в виде строки key. Если оно не определено, возвращается defaultValue. Эквивалентно

System.getProperties().getProperty(key, def);

Значения свойств хранятся в виде строк, однако некоторые строки представляют другие типы — например, целые или логические. Существуют специальные методы, которые являются статическими методами соответствующих классов-оболочек, для чтения свойств и преобразования их в значения примитивных типов. Каждый из таких методов получает строковый параметр с именем свойства, интересующего программиста. Некоторые методы имеют и второй параметр (обозначенный ниже как def) со значением по умолчанию, которое возвращается в том случае, если свойство с данным именем не найдено. Методы, в которых этот параметр отсутствует, в этом случае возвращают объект, содержащий 0 для числового типа или false — для логического. Все эти методы преобразуют значения в стандартный для Java формат примитивного типа.

public static boolean **Boolean.getBoolean(String name)**

public static Integer **Integer.getInteger(String name)**

public static Integer Integer.getInteger(String name, int def)

public static Long **Long.getLong(String nm)**

public static Long **Long.getLong(String nm, long def)**

Метод getBoolean отличается от других тем, что он возвращает логическое значение (boolean) вместо объекта класса Boolean. Если свойство не найдено, getBoolean передает false.

14.4. Создание процессов

Как упоминалось выше, в программах Java могут одновременно выполняться несколько потоков. Большинство систем, на которых функционирует среда Java, также поддерживают запуск нескольких программ. Приложения Java могут вызывать новые программы, обращаясь к одной из двух форм метода System.exec. Каждый успешный вызов exec создает новый объект Process, который представляет собой работающую программу. Вы можете запросить информацию о состоянии процесса и вызвать методы, управляющие его ходом. Существуют две основные формы метода exec:

public Process **exec(String[] cmdarray)** throws IOException

Выполняет в текущей системе команду, заданную в объекте cmdarray, и возвращает объект Process (описанный ниже), который представляет запущенный процесс. В cmdarray[0] задается имя команды, а во всех последующих строках массива — аргументы.

public Process **exec(String command)** throws IOException

Эквивалентен первой форме exec, в которой элементы массива собраны в одну строку и разделены символами-разделителями. Приведем пример использования этой формы exec:

String cmd = “/bin/ls” + opts + “ ” + dir;

Process child = Runtime.getRuntime().exec(cmd);

Порожденный процесс носит название *процесса-потомка* (*child process*). По аналогии, создающий процесс называется родителем. Метод exec возвращает объект Process для каждого запущенного процесса-потомка. Этот объект обладает двумя важными аспектами по отношению к процессу-потомку: во-первых, он содержит методы для обращения ко входному и выходному потоку процесса-потомка, а также к его потоку ошибок:

public abstract OutputStream **getOutputStream()**

Возвращает поток OutputStream, соединенный со входным потоком процесса-потомка. Данные, записанные в этот поток, считываются процессом-потомком в качестве ввода. Поток является буферизованным.

public abstract InputStream **getInputStream()**

Возвращает поток InputStream, соединенный с выходным потоком процесса-потомка. Когда процесс-потомок записывает выходные данные, они могут читаться из этого потока. Поток является буферизованным.

public abstract InputStream **getErrorStream()**

Возвращает поток InputStream, соединенный с выходным потоком ошибок процесса-потомка. Когда процесс-потомок выдает сообщения об ошибках, они могут читаться из этого потока. Поток является небуферизованным, чтобы гарантировать немедленное поступление информации об ошибках .

Ниже в качестве примера приводится программа, которая соединяет стандартные потоки Java с потоками нового процесса, чтобы весь ввод пользователя поступал в указанную программу, а весь ее вывод был виден пользователю:

```
public static Process UserProg(String cmd)
     throws IOException
{
     Process proc = Runtime.getRuntime().exec(cmd);
     plugTogether(System.in,  proc.getOutputStream());
     plugTogether(System.out, proc.getInputStream());
     plugTogether(System.err, proc.getErrorStream());
     return proc;
}
```

Предполагается, что метод plugTogether соединяет два потока, читая данные из одного из них и записывая их в другой.

Упражнение 14.1

Напишите метод plugTogether. Подсказка: воспользуйтесь многопоточной моделью.

Второй аспект взаимоотношений между процессом-родителем и процессом-потомком заключается в том, что объект Process содержит методы для управления процессом и определения его статуса:

public abstract int **waitFor()** throws InterruptedException

Сколь угодно долго ожидает завершения процесса-потомка и возвращает значение, переданное им методу System.exit или его аналогу (ноль означает успешное завершение, все остальное — неудачу). Если процесс уже завершился, то просто возвращается значение.

public abstract int **exitValue()**

Возвращает завершающий код данного процесса. Если процесс еще активен, то exitValue возбуждает исключение IllegalStateException.

public abstract void **destroy()**

Уничтожает процесс. Ничего не делает, если процесс уже завершился. Если объект Process будет уничтожен сборщиком мусора, это *не* означает уничтожения самого процесса; просто в дальнейшем вы не сможете контролировать его.

Например, следующий метод возвращает строковый массив, который содержит результаты работы команды ls с заданными параметрами. В случае неудачного завершения команды возбуждается исключение LSFailedException:

```
// java.io.*  и java.util.* импортированы
public String[] ls(String dir, String opts)
     throws LSFailedException
{
     try {
          // запустить процесс
          String[] cmdArray = { "/bin/ls", opts, dir };
          Process child = Runtime.getRuntime().exec(cmdArray);
          DataInputStream
               in = new DataInputStream(child.getInputStream());

          // прочитать выходные данные
          Vector lines = new Vector();
          String line;
          while ((line = in.readLine()) != null)
               lines.addElement(line);
          if (child.waitFor() != 0)  // если выполнение ls
                                     // было неудачным
               throw new LSFailedException(child.exitValue());
          String[] retval = new String{lines.size()];
          lines.copyInto(retval);
          return retval;
     } catch (LSFailedException e) {
          throw e;
     } catch (Exception e) {
          throw new LSFailedException(e.toString());
     }
}
```

Класс Process является абстрактным. Каждая реализация Java должна содержать один или несколько классов, расширяющих Process, которые могут взаимодействовать с процессами на уровне операционной системы. Такие классы могут обладать расширенным набором функций, полезных для программирования в данной среде. Информация о расширенных функциях должна содержаться в документации.

Две другие формы exec позволяют задать набор *переменных среды* (*environment variables*) — системно-зависимых переменных, доступных для нового процесса. Переменные среды передаются методу exec в виде строкового массива, каждый элемент которого задает имя и значение переменной среды в форме *имя* = *значение. Имя* не может содержать пробелов, хотя *значение* может быть произвольным. Переменные среды передаются вторым параметром:

public Process **exec(String command, String[] env)**

throws IOException

public Process **exec(String[] command, String[] env)**

throws IOException

Способ интерпретации переменных среды процессом-потомком является системно-зависимым. Механизм переменных среды поддерживается потому, что он существует на самых разных платформах. Для передачи информации между Java-программами следует пользоваться свойствами, а не переменными среды.

Упражнение 14.2

Напишите программу, которая выполняет exec для аргументов, переданных в командной строке, и выводит результаты работы, причем каждой строке предшествует ее номер.

Упражнение 14.3

Напишите программу, которая выполняет exec для аргументов, переданных в командной строке, и выводит результаты, прекращая работу процесса, когда на выходе появляется заранее определенная строка.

14.5. Класс Runtime

Объекты класса Runtime описывают состояние runtime-системы Java и те операции, которые она может выполнить. Для получения объекта Runtime, соответствующего текущему runtime-контексту, следует вызвать статический метод Runtime.getRuntime.

Одна из операций, выполняемых текущим runtime-контекстом, — получение входного или выходного потока, переводящего символы локального набора в их Unicode-эквиваленты. Многие существующие системы работают с национальными алфавитами, использующими 8-разрядные или иные наборы символов, конфликтующие с Unicode. Runtime-контекст предоставляет средства для перевода символов потока, работающего с локальным набором, в эквивалентные им символы Unicode. Например, клавиатура может генерировать 8-разрядный символьный код Oriya. Если воспользоваться потоком System.in, который читает символы с этой клавиатуры и получает от нее локализованный входной поток, символы Oriya будут переводиться в их 16-разрядные эквиваленты Unicode в диапазоне \u0b00–\u0b7f. Локализованный выходной поток выполняет обратное преобразование.

Runtime-контекст может быть уничтожен, для этого следует вызвать его метод exit и передать ему код завершения. Метод уничтожает все потоки в текущем runtime-контексте, независимо от их состояния. При этом для уничтожения потоков не используется исключение ThreadDeath; они просто останавливаются без выполнения условий finally. Для уничтожения всех программных потоков в вашей группе лучше пользоваться методом Thread Group.stop, который позволяет потокам выполнить завершающие действия в соответствии с условиями finally.

По традиции, завершающий код exit, равный нулю, означает успешное завершение задачи, а отличный от нуля — неудачу. Существует два способа передачи информации с кодом завершения. Первый — немедленно вызвать exit и остановить все потоки. Второй — убедиться, что все потоки “чисто” завершились , и только потом вызывать exit. Приведенный ниже пример останавливает все потоки в текущей группе, дает им завершиться и затем вызывает exit:

```
public static void safeExit(int status) {
        // получить список всех потоков
        Thread myThrd = Thread.currentThread();
        ThreadGroup thisGroup = myThrd.getThreadGroup();
        int count = thisGroup.activeCount();
        Thread[] thrds = new Thread[count + 20]; // +20 на всякий
                                                 //  случай
        thisGroup.enumerate(thrds);

        // остановить все потоки
        for (int i = 0; i << thrds.length; i++) {
                if (thrds[i] != null && thrds[i] != myThrd)
                        thrds[i].stop();
        }

        // дождаться завершения всех потоков
        for (int i = 0; i << thrds.length; i++) {
                if (thrds[i] != null && thrds[i] != myThrd) {
                        try {
                                thrds[i].join();
                        } catch (InterruptedException e) {
                                // пропустить поток
                        }
                }
        }

        // теперь можно выходить
        System.exit(status);
}
```

Упражнение 14.4

Модифицируйте метод safeExit так, чтобы он обрабатывал потоки, которые могли быть созданы после вызова enumerate. Кроме того, метод должен пропускать потоки-демоны, которые будут уничтожаться автоматически.

14.6. Разное

Два метода класса System не принадлежат ни к одной из категорий:

public static long **currentTimeMillis()**

Возвращает текущее время по Гринвичу в миллисекундах, считая от начала эпохи (00:00:00 UTC, 1 января 1970 года). Время возвращается в виде значения long, поэтому переполнение наступит лишь в 292280995 году — для большинства практических целей этого вполне хватает. Для более сложных приложений может пригодиться класс Date; см. раздел “Класс Date”.

public static void **arraycopy(Object src, int srcPos, Object dst, int dstPos, int count)**

Копирует содержимое исходного массива начиная с элемента src[srcPos] в целевой массив с элемента dst[dstPos]. Копируется ровно count элементов. Все ссылки на элементы должны лежать в пределах массива, иначе возбуждается исключение IndexOutOfBoundsException. Типы данных исходного массива должны быть совместимы с типами целевого массива, иначе возбуждается исключение ArrayStoreException. “Совместимость” следует понимать следующим образом: для массивов, содержащих ссылки на объекты, каждый объект исходного массива должен присваиваться соответствующему элементу целевого массива. Для массивов со значениями встроенных типов типы должны совпадать, а не просто быть совместимыми по присваиванию; метод arraycopy не может применяться для копирования массива short в массив int.

Метод arraycopy правильно работает с перекрывающимися массивами, поэтому он может применяться для копирования одной части массива в другую. Например, вы можете сдвинуть все содержимое массива на один элемент к началу, как это было сделано в методе squeezeOut.

Два трассировочных метода класса Runtime, traceInstructions и trace MethodCalls, также не относятся ни к одной категории. Каждому из них передается логический аргумент; если он равен true, включается трассировка инструкций или вызовов методов соответственно. Чтобы отключить трассировку, следует вызвать метод с аргументом равным false. Каждая реализация может поступать с этими вызовами так, как сочтет нужным, в том числе и игнорировать их, если некуда вывести результаты трассировки. Вероятно, эти методы будут применяться в первую очередь в средах разработки.

14.7. Безопасность

Класс System содержит два метода для работы с объектом класса Security Manager. Класс SecurityManager включает в себя методы, которые разрешают или запрещают открытие сокетов (sockets), доступ к файлам, создание программных потоков и т. д. Подробности о работе менеджера безопасности приведены в спецификации “Java Language Specification”.

public static void **setSecurityManager(SecurityManager s)**

Задает системный объект, который является менеджером безопасности. Данное значение может быть присвоено только один раз, чтобы при настройке безопасности системы можно было рассчитывать на то, что оно не изменится.

public static SecurityManager **getSecurityManager()**

Выдает системный объект менеджера безопасности. Описание менеджера безопасности слишком усложнило бы эту книгу; за подробностями обращайтесь к онлайновой документации.

14.8. Класс Math

Класс Math состоит из статических констант и методов, предназначенных для математических вычислений общего назначения. Все операции выполняются в арифметике double.

Константа Math.E представляет значение числа e (2.7182818284590452354), а Math.PI — значение числа p (3.14159265358979323846). Значения углов в методах задаются в радианах, а все параметры и возвращаемые значения имеют тип double, если явно не оговорено обратное:

Функция	Значение
sin(a)	синус a
cos(a)	косинус a
tan(a)	тангенс a
asin(v)	арксинус v, где v лежит в диапазоне [–1.0, 1.0]
acos(v)	арккосинус v, где v лежит в диапазоне [–1.0, 1.0]
>atan(v)	арктангенс v, возвращается в диапазоне [–p/2, p/2]
atan2(x, y)	арктангенс x/y, возвращается в диапазоне [–p, p]
exp(x)	e^x
pow(y,x)	y^x
log(x)	натуральный логарифм x
sqrt(x)	квадратный корень из x
ceil(x)	наименьшее целое число x
floor(x)	наибольшее целое число x
rint(x)	возвращает округленное целое значение x в виде double
round(x)	возвращает (int)floor(x+0.5) в виде double или float
abs(x)	возвращает модуль x для любого числового типа
max(x,y)	возвращает наибольшее из чисел x и y, относящихся к любому числовому типу
min(x,y)	возвращает наименьшее из чисел x и y, относящихся к любому числовому типу

Статический метод Math.IEEERemainder вычисляет остаток в соответствии со стандартом IEEE-754. Оператор вычисления остатка %, описанный в разделе 5.15.1, подчиняется правилу

(x/y)*y + x%y == x

При этом сохраняется всего один вид симметрии, а именно: если x%y равно z, то изменение знака x или y изменит знак z, но не повлияет на абсолютную величину. Например, 7%2.5 дает 2.0, а -7%2.5 равняется 2.0. Стандарт IEEE определяет поведение остатка для x и y иначе, сохраняя симметрию расположения на числовой оси — результат Math.IEEERemainder(-7, 2.5) будет равен -0.5. Оператор вычисления остатка делает значения симметричными относительно нуля на числовой оси, тогда как механизм работы с остатком по стандарту IEEE разносит получившиеся величины на y единиц. Метод присутствует потому, что обе разновидности остатка могут пригодиться.

Статический метод random генерирует псевдослучайное число r в диапазоне 0,0 r 1,0. Средства для более точного управления псевдослучайными числами рассматриваются в разделе “Класс Random” на стр. .

Упражнение 14.5

Напишите программу-калькулятор, которая работает со всеми этими функциями, а также (по меньшей мере) с базовыми операторами +, -, *, / и %. Вероятно, проще всего будет реализовать калькулятор с обратной польской нотацией, поскольку приоритет операторов значения не имеет.

Содержание | Далее

Глава 5
ЛЕКСЕМЫ, ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

В этом нет ничего особенного.
Все, что от вас требуется, —
это нажимать нужные клавиши в нужный момент,
а инструмент будет играть сам.
Иоганн Себастьян Бах

В этой главе рассматриваются основные “строительные блоки” Java —типы, операторы и выражения. Мы уже видели довольно много Java-программ и познакомились с их компонентами. В этой главе приводится детальное описание базовых элементов.

5.1. Набор символов

Большинству программистов приходилось иметь дело с исходными текстами программ, в которых использовалось одно из двух представлений символов: кодировка ASCII и ее разновидности (в том числе Latin-1) и EBCDIC. Оба этих набора содержат символы, используемые в английском и некоторых других западно-европейских языках.

В отличие от них, программы на языке Java написаны в *Unicode* — 16-разрядном наборе символов. Первые 256 символов Unicode представляют собой набор Latin-1, а основная часть первых 128 символов Latin-1 соответствует 7-разрядному набору символов ASCII. В настоящее время окружение Java может читать стандартные файлы в кодировке ASCII или Latin-1, немедленно преобразуя их в Unicode. /В Java используется Unicode 1.1.5 с исправленными ошибками. Справочная информация приведена в разделе "Библиография"/

В настоящее время лишь немногие текстовые редакторы способны работать с символами Unicode, поэтому Java распознает escape-последовательности вида \udddd, которыми кодируются символы Unicode; каждому d соответствует шестнадцатеричная цифра (ASCII-символы 0–9, а также a–f или A–F для представления десятичных значений 10–15). Такие последовательности допускаются в любом месте программы — не только в символах и строковых константах, но также и в идентификаторах. В начале последовательности может стоять несколько u; следовательно, символ ••• записывается и как \u0b87, и как \uu0b87. /Использование "множественных u" может показаться странной, но на то есть веские причины. При переводе Unicode-файла в формат ASCII, приходится кодировать символы Unicode, лежащие за пределами ASCII-диапазона, в виде escape-последовательностей. Таким образом, இ представляется в виде \u0b87. При обратном переводе осуществляется обратная замена; но что произойдет, если исходный текст в кодировке Unicode вместо символа ••• содержал \u0b87? В этом случае при обратной замене исходный текст изменится (синтаксический анализатор не заметит никаких изменений - но не читатель программы!) Выход заключается в том, чтобы при прямом переводе вставлять дополнительные u в уже существующие \udddd, а при обратном - убирать их, и, если u не останется, заменять escape-последовательность эквивалентным символом Unicode./

5.2. Комментарии

Комментарии в Java бывают трех видов:

```
// комментарий  - игнорируются символы от // до конца строки
/* комментарий */       - игнорируются символы между /* и следующим */, включая
                          завершающие символы строк \r, \n и \r\n.
/** комментарий */      - игнорируются символы между /** и следующим */, включая
                          перечисленные выше завершающие символы.  Документирующие
                          комментарии должны располагаться непосредственно после
                          объявления класса, члена класса или конструктора; они
                          включаются в автоматически генерируемую документацию.
```

Когда мы говорим “символы”, то имеем в виде *любые* символы Unicode. Комментарии в Java могут включать произвольные символы Unicode: “инь-янь” (\u262f), восклицание (\u203d) или “снеговика” (\u2603).

В Java не разрешаются вложенные комментарии. Приведенный ниже текст (как бы соблазнительно он ни выглядел) компилироваться не будет:

```
/* Закомментируем  до лучших времен; пока не реализовано

     /* Сделать что-нибудь этакое */
     universe.neatStuff();

*/
```

Первая комбинация символов /* начинает комментарий; ближайшая парная */ заканчивает его, оставляя весь последующий код синтаксическому анализатору, который сообщает об оставшихся символах */ как о синтаксической ошибке. Лучший способ временно убрать фрагмент из программы — либо поместить // в начале каждой строки, либо вставить конструкцию if (false):

```
if (false) {
     // Вызвать метод, когда он будет работать
     dwim();
}
```

Разумеется, данный фрагмент предполагает, что метод dwim определен где-то в другом месте программы.

5.3. Лексемы

Лексемами (tokens) языка называются “слова”, из которых состоит программа. Синтаксический анализатор разбивает исходный текст на отдельные лексемы и пытается понять, из каких операторов, идентификаторов и т. д. состоит программа. В языке Java символы-разделители (пробелы, табуляция, перевод строки и возврат курсора) применяются исключительно для разделения лексем или содержимого символьных или строковых литералов. Вы можете взять любую работающую программу и заменить произвольное количество символов-разделителей между лексемами (то есть разделителей, не входящих в строки и символы) на другое количество разделителей (не равное нулю) — это никак не повлияет на работу программы.

Разделители *необходимы* для отделения лексем друг от друга, которые бы в противном случае представляли бы собой одно целое. Например, в операторе

return 0;

нельзя убрать пробел между return и 0, поскольку это приведет к появлению неправильного оператора

return0;

состоящего всего из одного идентификатора return0. Дополнительные разделители облегчают чтение вашей программы, несмотря на то что синтаксический анализатор их игнорирует. Обратите внимание: комментарии считаются разделителями.

Алгоритм деления программы на лексемы функционирует по принципу “чем больше, тем лучше”: он отводит для следующей лексемы как можно больше символов, не заботясь о том, что при этом может произойти ошибка. Следовательно, раз ++ оказывается длиннее, чем +, выражение

j = i+++++i; // НЕВЕРНО

неверно интерпретируется как

j = i++ ++ +i; // НЕВЕРНО

вместо правильного

j = i++ + ++i;

5.4. Идентификаторы

Идентификаторы Java, используемые для именования объявленных в программе величин (переменных и констант) и меток, должны начинаться с буквы, символа подчеркивания (_) или знака доллара ($), за которыми следуют буквы или цифры в произвольном порядке. Многим программистам это покажется знакомым, но в связи с тем, что исходные тексты Java-программ пишутся в кодировке Unicode, понятие “буква” или “цифра” оказывается значительно более широким, чем в большинстве языков программирования. “Буквы” в Java могут представлять собой символы из армянского, корейского, грузинского, индийского и практически любого алфавита, который используется в наше время.

Следовательно, наряду с идентификатором kitty можно пользоваться идентификаторами maиka, кошка, پیشی, பூனைக்குட்டி и 猫. /Эти слова означают "кошка" или "котенок" на английском, сербо-хорватском, русском, фарси, тамильском и японском языках соответственно. Если в других языках они имеют иное значение, мы искренне надеемся, что оно не является оскорбительным; в противном случае приносим свои извинения и заверяем, что оскорбление было ненамеренным./ Термины “буква” и “цифра” в Unicode трактуются довольно широко, но если какой-либо символ считается буквой или цифрой в неком языке, то, по всей вероятности, он имеет аналогичный смысл и в Java. Полные определения этих понятий приводятся в таблицах “Цифры Unicode” и “Буквы и цифры Unicode”.

Любые расхождения в символах, входящих в состав идентификаторов, делают два идентификатора различными. Регистр символов имеет значение: A, a, á, Á и т. д. являются разными идентификаторами. Символы, которые выглядят одинаково или почти одинаково, нетрудно спутать друг с другом. Например, латинская заглавная n (N) и греческая заглавная ν (N) выглядят практически одинаково, однако им соответствуют разные символы Unicode (\u004e и \u039d соответственно). Единственная возможность избежать ошибок заключается в том, чтобы каждый идентификатор был написан только на одном языке (и, следовательно, включал символы известного набора), чтобы программист мог понять, что вы имеете в виду — E или E. /Одна из этих букв входит в кириллицу, а другая - в ASCII. Отличите одну от другой, и вы получите приз./

Идентификаторы в языке Java могут иметь произвольную длину.

5.4.1. Зарезервированные слова Java

Ключевые слова Java не могут использоваться в качестве идентификаторов. Приведем список ключевых слов Java (слова, помеченные символом *, зарезервированы, но в настоящее время не применяются):

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

Хотя слова null, true и false внешне похожи на ключевые, формально они относятся к литералам (как, скажем, число 12) и потому отсутствуют в приведенной выше таблице. Тем не менее вы не можете использовать слова null, true и false (как и 12) в качестве идентификаторов, хотя они и могут входить в состав идентификатора. Формально null, true и false не являются ключевыми словами, но к ним относятся те же самые ограничения.

5.5. Примитивные типы

Некоторые зарезервированные слова представляют собой названия типов. В Java предусмотрены следующие примитивные типы:

```
boolean либо true, либо false
char    16-разрядный символ Unicode 1.1.5
byte    8-разрядное целое со знаком, дополненное по модулю 2
short   16-разрядное целое со знаком, дополненное по модулю 2
int     32-разрядное целое со знаком, дополненное по модулю 2
long    64-разрядное целое со знаком, дополненное по модулю 2
float   32-разрядное число с плавающей точкой (IEEE 7541985)
double  64-разрядное число с плавающей точкой (IEEE 7541985)
```

Каждому из примитивных типов языка Java, за исключением short и byte, соответствует одноименный класс пакета java.lang. Значения типов short и byte всегда преобразуются в int перед выполнением любых вычислений — приведенный выше формат используется только для хранения, но не для вычислений (см. “Тип выражения”). В классах языка, служащих оболочками для примитивных типов (Boolean, Character, Integer, Long, Float и Double), также определяется ряд полезных констант и методов. Например, в классах-оболочках для некоторых примитивных типов определяются константы MIN_VALUE и MAX_VALUE.

В классах Float и Double определены константы NEGATIVE_INFINITY, POSITIVE_INFINITY и NaN, а также метод isNaN, который проверяет, не является ли значение с плавающей точкой “не-числом” (Not a Number) — то есть результатом неверной операции, вроде деления на ноль. Значение NaN может использоваться для обозначения недопустимого значения, подобно тому как значение null для ссылок не указывает ни на какой конкретный объект. Классы-оболочки подробно рассматриваются в главе 13.

5.6. Литералы

Для каждого типа Java определяется понятие *литералов*, которые представляют собой постоянные значения данного типа. Несколько следующих подразделов описывают способы записи литералов (неименованных констант) в каждом из типов.

5.6.1. Ссылки на объекты

Для ссылок на объекты существует всего один литерал — null. Он может находиться всюду, где допускается использование ссылки. Чаще всего null представляет ссылку на недопустимый или несуществующий объект. null не относится ни к одному типу, даже к типу Object.

5.6.2. Логические значения

В типе boolean имеются два литерала — true и false.

5.6.3. Целые значения

Целые константы являются последовательностями восьмеричных, десятичных или шестнадцатеричных цифр. Начало константы определяет основание системы счисления: 0 (ноль) обозначает восьмеричное число (основание 8); 0x или 0X обозначает шестнадцатеричное число (основание 16); любой другой набор цифр указывает на десятичное число (основание 10). Следующие числа имеют одинаковое значение:

29 035 0x1D 0X1d

Целые константы относятся к типу long, если они заканчиваются символом L или l, как 29L; желательно пользоваться L, потому что l легко спутать с 1 (цифрой один). В противном случае считается, что целая константа относится к типу int. Если литерал типа int непосредственно присваивается переменной типа short или byte и его значение находится в пределах диапазона допустимых значений для типа переменной, то операции с литералом осуществляются так, словно он относится к типу short или byte соответственно.

5.6.4. Значения с плавающей точкой

Число с плавающей точкой представляется в виде десятичного числа с необязательной десятичной точкой, за которым (также необязательно) может следовать порядок. Число должно содержать как минимум одну цифру. В конце числа может стоять символ F или f для обозначения константы с одинарной точностью или же символ d или D для обозначения константы с двойной точностью. Следующие литералы обозначают одно и то же значение:

18. 1.8e1 .18E2

Константы с плавающей точкой относятся к типу double, если только они не завершаются символом f или F — в этом случае они имеют тип float, как константа 18.0f. Завершающий символ D или d определяет константу типа double. Ноль может быть положительным (0.0) или отрицательным (-0.0). Положительный ноль равен отрицательному, но при использовании в некоторых выражениях они могут приводить к различным результатам. Например, выражение 1d/0d равно +, а 1d/-0d равно –.

Константа типа double не может присваиваться переменной типа float, даже если ее значение лежит в пределах диапазона float. Для присваивания значений переменным и полям типа float следует использовать константы типа float или привести double к float.

5.6.5. Символы

Символьные литералы заключаются в апострофы — например, ‘Q’. Некоторые служебные символы могут представляться в виде *escape-последовательностей*. К их числу относятся:

\n переход на новую строку (\u000A)

\t табуляция (\u0009)

\b забой (\u0008)

\r ввод (\u000D)

\f подача листа (\u000C)

\\ обратная косая черта (\u005C)

\’ апостроф (\u0027)

\" кавычка (\u0022)

\ddd символ в восьмеричном представлении, где каждое d соответствует цифре от 0 до 7

Восьмеричные символьные константы могут состоять из трех или менее цифр и не могут превышать значения \377 (\u00ff). Символы, представленные в шестнадцатеричном виде, всегда должны состоять из четырех цифр.

5.6.6. Строки

Строковые литералы заключаются в двойные кавычки: “along”. В них могут входить любые escape-последовательности, допустимые в символьных константах. Строковые литералы являются объектами типа String. Более подробно о строках рассказывается в главе 8.

Символы перехода на новую строку не могут находиться в середине строковых литералов. Если вы хотите вставить такой символ в строку, воспользуйтесь escape-последовательностью \n.

В строках может применяться восьмеричная запись символов, но для предотвращения путаницы (в тех случаях, когда символы, представленные таким образом, соседствуют с другими символами) необходимо указывать все три восьмеричные цифры. Например, строка “\0116" эквивалентна строке ”\t6", тогда как строка “\116" эквивалентна ”N".

5.7. Объявления переменных

В *объявлении* указывается тип, уровень доступа и другие атрибуты идентификатора. Объявление состоит из трех частей: сначала приводится список *модификаторов*, за ним следует *тип*, и в завершение следует список *идентификаторов*.

Модификаторы могут отсутствовать в объявлении переменной. Модификатор static объявляет, что переменная сохраняет свое значение после выхода из метода; модификатор final объявляет, что значение переменной присваивается всего один раз, при ее инициализации. Модификатор final может использоваться только для полей.

Тип в объявлении указывает на то, какие значения могут принимать объявляемые величины и как они должны себя вести.

Между объявлением переменной по отдельности или одновременно с несколькими другими переменными нет никаких различий. Например, объявление:

float[] x, y;

равносильно

float[] x;

float[] y;

Объявления могут находиться в произвольном месте исходного текста программы. Вы не обязаны ставить их в начале класса, метода или блока. В общем случае, идентификатором можно пользоваться в любой момент после его объявления в некотором блоке (см. раздел “Операторы и блоки”), за одним исключением: нестатические поля недоступны в статических методах.

Поля с модификатором final должны инициализироваться при объявлении.

Объявлению члена класса может предшествовать один из нескольких модификаторов. Модификаторы могут следовать в произвольном порядке, но мы рекомендуем выработать некоторое соглашение и придерживаться его. В этой книге используется следующий порядок: сначала следуют модификаторы доступа (public, private или protected), затем static, затем synchronized, и, наконец, final. Использование единого порядка модификаторов облегчает чтение исходного текста программы.

5.7.1. Значение имени

Каждый созданный идентификатор существует в некотором *пространстве имен (namespace)*. Имена идентификаторов в пределах одного пространства имен должны различаться. Когда вы используете идентификатор для того, чтобы присвоить имя переменной, классу или методу, то для определения значения имени производится поиск в следующем порядке:

1. Локальные переменные, объявленные в блоке, цикле for или среди параметров обработчика исключений. Блок представляет собой один или несколько операторов, заключенных в фигурные скобки. Переменные также могут объявляться во время инициализации в цикле for.

2. Параметры метода или конструктора, если код входит в метод или конструктор.

3. Члены данного класса или интерфейсного типа, то есть его поля и методы, в том числе все унаследованные члены.

4. Импортированные типы с явным именованием.

5. Другие типы, объявленные в том же пакете.

6. Импортированные типы с неявным именованием.

7. Прочие пакеты, доступные в системе.

В каждом из вложенных блоков или операторов for могут объявляться новые имена. Чтобы избежать путаницы, вы не можете воспользоваться вложением для переобъявления параметра или идентификатора из внешнего блока или оператора for. Так, после появления локального идентификатора или параметра с именем über вы не можете создать во вложенном блоке новый, отличный от него идентификатор с тем же именем über.

Пространства имен разделяются в зависимости от типа идентификатора. Имя переменной может совпадать с именем пакета, типа, метода, поля или метки оператора. Вырожденный случай может выглядеть следующим образом:

```
class Reuse {
     Reuse Reuse(Reuse Reuse) {
          Reuse:
               for (;;) {
                    if (Reuse.Reuse(Reuse) == Reuse)
                         break Reuse;
               }
               return Reuse;
     }
}
```

Описанный выше порядок просмотра означает, что идентификаторы, объявленные внутри метода, могут скрывать внешние идентификаторы. Обычно скрытие идентификаторов считается проявлением плохого стиля программирования, поскольку при чтении программы приходится просматривать все уровни иерархии, чтобы выяснить, какая же переменная используется в том или ином случае.

Вложение областей видимости означает, что переменная существует лишь в том фрагменте программы, в котором планируется ее использование. Например, к переменной, объявленной при инициализации цикла for, невозможно обратиться за пределами цикла. Нередко это бывает полезно, чтобы код за пределами цикла for не смог определить, при каком значении управляющей переменной завершился цикл.

Вложение помогает усилить роль локального кода. Если бы скрытие внешних переменных не допускалось, то включение нового поля в класс или интерфейс могло бы привести к нарушению работы существующих программ, в которых используются переменные с тем же именем. Концепция областей видимости предназначена скорее для общей защиты системы, нежели для поддержки повторного использования имен.

5.8. Массивы

Массив представляет собой упорядоченный набор элементов. Элементы массива могут иметь примитивный тип или являться ссылками на объекты, в том числе и ссылками на другие массивы. Строка

int[] ia = new int[3];

объявляет массив с именем ia, в котором изначально хранится три значения типа int.

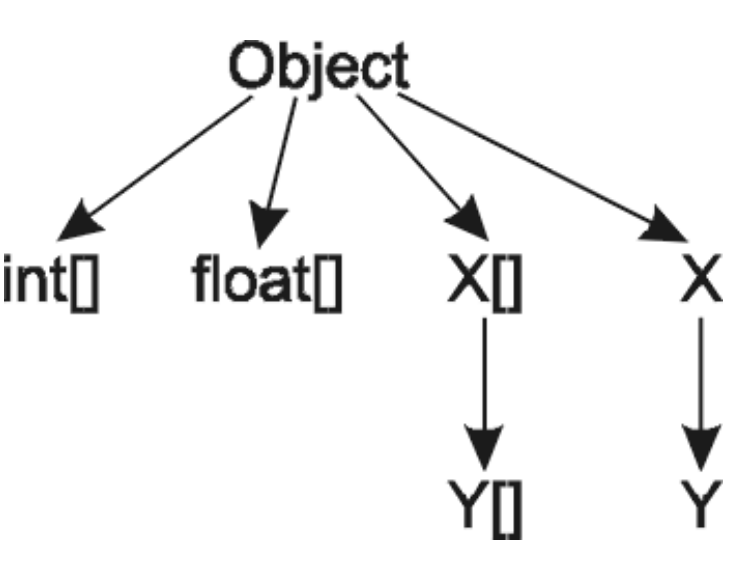
При объявлении переменной-массива размер не указывается. Количество элементов в массиве задается при его *создании* оператором new, а не при объявлении. Размер объекта-массива фиксируется в момент его создания и не может изменяться в дальнейшем. Обратите внимание: фиксируется именно размер *объекта-массива*; в приведенном выше примере ia может быть присвоена ссылка на любой массив другого размера.

Первый элемент массива имеет индекс 0 (ноль), а последний — индекс *размер*–1. В нашем примере последним элементом массива является ia[2]. При каждом использовании индекса проверяется, лежит ли он в диапазоне допустимых значений. При выходе индекса за его пределы возбуждается исключение IndexOutOfBounds.

Размер массива можно получить из поля length. Для нашего примера следующий фрагмент программы перебирает все элементы массива и выводит каждое значение:

```
for (int i =0; i << ia.length; i++)
     System.out.println(i + ": " + ia[i]);
```

Массивы всегда являются неявным расширением класса Object. Если у вас имеется класс X, расширяющий его класс Y и массивы каждого из этих классов, то иерархия будет выглядеть следующим образом:



Благодаря этому обстоятельству массивы ведут себя полиморфно. Вы можете присвоить массив переменной типа Object, после чего осуществить обратное преобразование. Массив объектов типа Y допускается использовать всюду, где разрешено присутствие массива объектов базового типа X.

Как и к любым другим созданным объектам, к массивам применяется сборка мусора.

Основное ограничение на “объектность” массивов заключается в том, что они не могут расширяться для включения в них новых методов. Так, следующая конструкция является недопустимой:

```
class ScaleVector extends double[] { //
     // ...
}
```

При объявлении массива объектов вы на самом деле объявляете массив переменных соответствующего типа. Рассмотрим следующий фрагмент:

```
Attr[] attrs = new Attr[12];

for (int i = 0; i << attrs.length; i++)
     attrs[i] = new Attr(names[i], values[i]);
```

После выполнения первого оператора new, attrs содержит ссылку на массив из 12 переменных, инициализированных значением null. Объекты Attr создаются только при выполнении цикла.

Если вы пожелаете, Java допускает присутствие квадратных скобок после переменной, а не после типа, как в следующем объявлении:

int ia[] = new int[3];

Оно эквивалентно приведенному выше. Тем не менее первый вариант все же считается более предпочтительным, поскольку тип объявляется в одном месте.

5.8.1. Многомерные массивы

Элементами массивов в Java могут быть другие массивы. Например, фрагмент программы для объявления и вывода двумерной матрицы может выглядеть следующим образом:

```
float[][] mat = new float[4][4];
setupMatrix(mat);
for (int y = 0; y << mat.length; y++) {
     for (int x = 0; x << mat[y].length; x++)
          System.out.println(mat[x][y] + " ");
     System.out.println();
}
```

Первый (левый) размер массива должен задаваться при его создании. Другие размеры могут указываться позже. Использование более чем одной размерности является сокращением для вложенного набора операторов new. Приведенный выше массив может быть создан следующим образом:

```
float[][] mat = new float[4][];
for (int y = 0; y << mat.length; y++)
     mat[y] = new float[4];
```

Одно из преимуществ многомерных массивов состоит в том, что каждый вложенный массив может иметь свои размеры. Вы можете имитировать работу с массивом 4x4, но при этом создать массив из четырех массивов типа int, каждый из которых имеет свою собственную длину, необходимую для хранения его данных.

Упражнение 5.1

Напишите программу, которая строит треугольник Паскаля до глубины 12. Каждый числовой ряд треугольника сохраняется в массиве соответствующей длины, а массивы рядов заносятся в массив, элементами которого являются 12 массивов типа int. Спроектируйте свое решение так, чтобы результаты выводились методом, который печатает содержимое двумерного массива с использованием длины каждого из вложенных массивов, а не константы 12. Теперь модифицируйте программу так, чтобы в ней использовалась константа, отличная от 12, а метод вывода при этом не изменился.

5.9. Инициализация

Переменная может инициализироваться при ее объявлении. Чтобы задать начальное значений переменной, следует после ее имени поставить = и выражение:

```
final double p = 3.14159;
float radius = 1.0f;    // начать с единичного радиуса
```

Если при объявлении поля класса не инициализируются, то Java присваивает им исходные значения по умолчанию. Значение по умолчанию зависит от типа поля:

Тип поля	Тип поля
boolean	false
char	‘\u0000’
целое (byte, short, int, long)	0
с плавающей точкой	+0.0f или +0.0d
ссылка на объект	null

Java не присваивает никаких исходных значений локальным переменным метода, конструктора или статического инициализатора. Отсутствие исходного значения у локальной переменной обычно представляет собой программную ошибку, и перед использованием локальных переменных их необходимо инициализировать.

Момент инициализации переменной зависит от ее области видимости. Локальные переменные инициализируются каждый раз, когда выполняется их объявление. Поля объектов и элементы массивов инициализируются при создании объекта или массива оператором new — см. “Порядок вызова конструкторов”. Инициализация статических переменных класса происходит перед выполнением какого-либо кода, относящегося к данному классу.

Инициализаторы полей не могут возбуждать проверяемые исключения или вызывать методы, которые могут это сделать, так как не существует способа перехватить эти исключения. Для нестатических полей можно обойти данное правило, присваивая исходное значение в конструкторе (конструктор может обрабатывать исключения). Для статических полей аналогичный выход состоит в том, чтобы присвоить исходное значение внутри статического инициализатора, обрабатывающего исключение.

5.9.1. Инициализация массивов

Чтобы инициализировать массив, следует задать значения его элементов в фигурных скобках после его объявления. Следующее объявление создает и инициализирует объект-массив:

```
String[] dangers = { "Lions", "Tigers", "Bears" };
```

Это равносильно следующему фрагменту:

```
String[] dangers = new String[3];

dangers[0] = "Lions";
dangers[1] = "Tigers";
dangers[2] = "Bears";
```

Для инициализации многомерных массивов может использоваться вложение инициализаторов отдельных массивов. Приведем объявление, в котором инициализируется матрица размеров 4x4:

```
double[][] identityMatrix = {
             { 1.0, 0.0, 0.0, 0.0 },
             { 0.0, 1.0, 0.0, 0.0 },
             { 0.0, 0.0, 1.0, 0.0 },
             { 0.0, 0.0, 0.0, 1.0 },
          };
```

5.10. Приоритет и ассоциативность операторов

Приоритетом (precedence) оператора называется порядок, в котором он выполняется по отношению к другим операторам. Различные операторы имеют различные приоритеты. Например, приоритет условных операторов выше, чем у логических, поэтому вы можете написать

```
if (i >>= min && i <<= max)
     process(i);
```

не сомневаясь в порядке выполнения операторов. Поскольку * (умножение) имеет более высокий приоритет, чем — (вычитание), значение выражения

5 * 3 — 3

равно 12, а не нулю. Приоритет операторов можно изменить с помощью скобок; например, если бы в предыдущем выражении вам было нужно получить именно ноль, то для этого достаточно поставить скобки:

5 * (3 — 3)

Когда два оператора с одинаковыми приоритетами оказываются рядом, порядок их выполнения определяется ассоциативностью операторов. Поскольку + (сложение) относится к лево-ассоциативным операторам, выражение

a + b + c

эквивалентно следующему:

(a + b) + c

Ниже все операторы перечисляются в порядке убывания приоритетов. Все они являются бинарными, за исключением унарных операторов, операторов создания и преобразования типа (также унарных) и тернарного условного оператора. Операторы с одинаковым приоритетом приведены в одной строке таблицы:

постфиксные операторы	[] . (параметры) expr++ expr—
унарные операторы	++expr —expr +expr -expr ~ !
создание и преобразование типа	new (тип)expr
операторы умножения/деления	* / %
операторы сложения/вычитания	+ -
операторы сдвига	<< >> >>>
операторы отношения	< > >= <= instanceof
операторы равенства	== !=
поразрядное И	&
поразрядное исключающее ИЛИ	^
поразрядное включающее ИЛИ	|
логическое И	&&
логическое ИЛИ	||
условный оператор	?:
операторы присваивания	= += -= *= /= %= >>= <<= >>>= &= ^= |=

Все бинарные операторы, за исключением операторов присваивания, являются *лево-ассоциативными*. Операторы присваивания являются *право-ассоциативными* — другими словами, выражение a=b=c эквивалентно a=(b=c).

Приоритет может изменяться с помощью скобок. В выражении x+y*z сначала y умножается на z, после чего к результату прибавляется x, тогда как в выражении (x+y)*z сначала вычисляется сумма x и y, а затем результат умножается на z.

Присутствие скобок часто оказывается необходимым в тех выражениях, где используется поразрядная логика или присваивание осуществляется внутри логического выражения. В качестве примера рассмотрим следующий фрагмент:

while ((v = stream.next()) != null)

processValue(v);

Приоритет операторов присваивания ниже, чем у операторов равенства; без скобок наш пример был бы равносилен следующему:

while (v = (stream.next() != null)) // НЕВЕРНО

processValue(v);

что, вероятно, отличается от ожидаемого порядка вычислений. Кроме того, конструкция без скобок, скорее всего, окажется неверной — она будет работать лишь в маловероятном случае, если v имеет тип boolean.

Приоритет поразрядных логических операторов &, ^ и | также может вызвать некоторые затруднения. Бинарные поразрядные операторы в сложных выражениях тоже следует заключать в скобки, чтобы облегчить чтение выражения и обеспечить правильность вычислений.

В этой книге скобки употребляются довольно редко — лишь в тех случаях, когда без них смысл выражения будет неочевидным. Приоритеты операторов являются важной частью языка и их нужно знать. Многие программисты склонны злоупотреблять скобками. Старайтесь не пользоваться скобками там, где без них можно обойтись — перегруженная скобками программа становится неудобочитаемой и начинает напоминать LISP, не приобретая, однако, ни одного из достоинств этого языка.

5.11. Порядок вычислений

Язык Java гарантирует, что операнды в операторах вычисляются слева направо. Например, в выражении x+y+z компилятор вычисляет значение x, потом значение y, складывает эти два значения, вычисляет значение z и прибавляет его к предыдущему результату. Компилятор не станет вычислять значение y перед x или z — перед y или x.

Такой порядок имеет значение, если вычисление x, y и z имеет некоторый побочный эффект. Скажем, если при этом будут вызываться методы, которые изменяют состояние объекта или выводят что-нибудь на печать, то изменение порядка вычислений отразится на работе программы. Язык гарантирует, что этого не произойдет.

Все операнды всех операторов, за исключением &&, || и ?: (см. ниже), вычисляются перед выполнением оператора. Это утверждение оказывается истинным даже для тех операций, в ходе которых могут возникнуть исключения. Например, целочисленное деление на ноль приводит к запуску исключения ArithmeticException, но происходит это лишь после вычисления обоих операндов.

5.12. Тип выражения

У каждого выражения имеется определенный тип. Он задается типом компонентов выражения и семантикой операторов. Если арифметический или поразрядный оператор применяется к выражению целого типа, то результат будет иметь тип int, если только в выражении не участвует значение типа long — в этом случае выражение также будет иметь тип long. Все целочисленные операции выполняются с точностью int или long, так что меньшие целые типы short и byte всегда преобразуются в int перед выполнением вычислений.

Если хотя бы один из операндов арифметического оператора относится к типу с плавающей точкой, то при выполнении оператора используется вещественная арифметика. Вычисления выполняются с точностью float, если только по крайней мере один из операндов не относится к типу double; в этом случае вычисления производятся с точностью double, и результат также имеет тип double.

Оператор + выполняет конкатенацию для типа String, если хотя бы один из его операндов относится к типу String или же переменная типа String стоит в левой части оператора +=.

При использовании в выражении значение char преобразуется в int по-средством обнуления старших 16 бит. Например, символ Unicode \uffff является эквивалентом целого значения 0x0000ffff. Несколько иначе рассматривается значение типа short, равное 0xffff, — с учетом знака оно равно –1, поэтому его эквивалент в типе int будет равен 0xffffffff.

5.13. Приведение типов

Java относится к языкам *с сильной типизацией* — это означает, что во время компиляции практически всегда осуществляется проверка на совместимость типов. Java предотвращает неверные присваивания, запрещая все сколько-нибудь сомнительные операции, и поддерживает механизм приведения типов для тех случаев, когда совместимость может быть проверена только во время выполнения программы. Мы будем рассматривать приведение типов на примере операции присваивания, но все сказанное относится и к преобразованиям внутри выражений, и к присваиванию значений параметрам методов.

5.13.1. Неявное приведение типов

Некоторые приведения типов происходят автоматически, без вмешательства с вашей стороны. Существует две категории *неявных* приведений.

Первая категория неявных приведений типа относится к примитивным значениям. Числовой переменной можно присвоить любое числовое значение, входящее в допустимый диапазон данного типа. Тип char может использоваться всюду, где допускается использование int. Значение с плавающей точкой может быть присвоено любой переменной с плавающей точкой, имеющей ту же или большую точность.

Java также поддерживает неявные приведения целых типов в типы с плавающей точкой, но не наоборот — при таком переходе не происходит потери значимости, так как диапазон значений с плавающей точкой шире, чем у любого из целых типов.

Сохранение диапазона не следует путать с сохранением точности. При некоторых неявных преобразованиях возможна потеря точности. Например, рассмотрим преобразование long в float. Значения float являются 32-разрядными, а значения long — 64-разрядными. float содержит меньше значащих цифр, чем long, даже несмотря на то, что этот тип способен хранить числа из большего диапазона. Присваивание значения long переменной типа float может привести к потере данных. Рассмотрим следующий фрагмент:

```
long orig = 0x7effffffffffffffL;
float fval = orig;
long lose = (long)fval;

System.out.println("orig = " + orig);
System.out.println("fval = " + fval);
System.out.println("losw = " + lose);
```

Первые два оператора создают значение long и присваивают его переменной float. Чтобы продемонстрировать, что при этом происходит потеря точности, мы производим явное приведение fval к long и присваиваем значение другой переменной (явное приведение типов рассматривается ниже). Результаты, выводимые программой, позволяют убедиться в том, что значение float потеряло часть своей точности, так как значение исходной переменной orig типа long отличается от того, что было получено при явном обратном приведении значения переменной fval к типу long:

```
orig = 9151314442816847871
fval = 9.15131e+18
lose = 9151314442816847872
```

Второй тип неявного приведения — приведение по ссылке. Объект, относящийся к некоторому классу, включает экземпляры каждого из супертипов. Вы можете использовать ссылку на объект типа в тех местах, где требуется ссылка на любой из его супертипов.

Значение null может быть присвоено ссылке на объект любого типа, в том числе и ссылке на массив.

5.13.2. Явное приведение и instanceof

Когда значение одного типа не может быть присвоено переменной другого типа посредством неявного приведения, довольно часто можно воспользоваться *явным приведением типов (cast)*. Явное приведение требует, чтобы новое значение нового типа как можно лучше соответствовало старому значению старого типа. Некоторые явные приведения недопустимы (вы не сможете преобразовать boolean в int), однако разрешается, например, явное приведение double к значению типа long, как показано в следующем фрагменте:

```
double d = 7.99;
long l = (long)d;
```

Когда значение с плавающей точкой преобразуется к целому типу, его дробная часть отбрасывается; например, (int)-72.3 равняется –72. В классе Math имеются методы, которые иначе осуществляют округление чисел с плавающей точкой при преобразовании в целое — см. раздел “Класс Math”.

Значение double также может явно преобразовываться к типу float, а значение целого типа — к меньшему целому типу. При приведении double к float возможна потеря точности или же появление нулевого или бесконечного значения вместо существовавшего ранее конечного.

Приведение целых типов заключается в “срезании” их старших битов. Если значение большего целого умещается в меньшем типе, к которому осуществляется преобразование, то ничего страшного не происходит. Однако если величина более широкого целого типа лежит за пределами более узкого типа, то потеря старших битов изменяет значение, а возможно — и знак. Фрагмент:

```
short s = -134;
byte b = (byte)s;

System.out.println("s = " + s + ", b = " + b);
```

выводит следующий результат (поскольку старшие биты s теряются при сохранении значения в b):

s = -134, b = 122

char можно преобразовать к любому целому типу и наоборот. При приведении целого типа в char используются только младшие 16 бит, а остальные биты отбрасываются. При преобразовании char в целый тип старшие 16 бит заполняются нулями. Тем не менее впоследствии работа с этими битами осуществляется точно так же, как и с любыми другими. В приведенном ниже фрагменте программы максимальный символ Unicode преобразуется к типу int (неявно) и к типу short (явно). Значение типа int (0x0000ffff) оказывается положительным, поскольку старшие биты символа обнулены. Однако при приведении к типу short получается отрицательная величина, так как старшим битом типа short является знаковый бит:

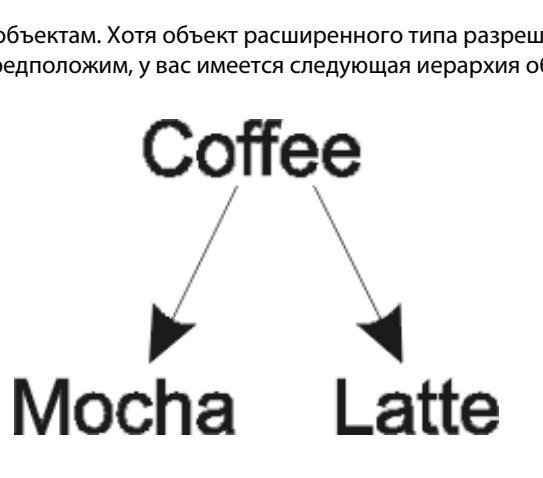
```
class CharCast {
public static void main(String[] args) {
int i = '\uffff';
short s = (short)'\uffff';

System.out.println("i = " + i);
System.out.println("s = " + s);
}
}
```

А вот как выглядит результат работы:

i = 65535

s = -1

Явное приведение типов может применяться и к объектам. Хотя объект расширенного типа разрешается использовать вместо объекта супертипа, обратное, вообще говоря, неверно. Предположим, у вас имеется следующая иерархия объектов:



Ссылка типа Coffee не обязательно относится к типу Mocha — объект также может иметь тип Latte. Следовательно, неверно, вообще говоря, ставить ссылку на объект типа Coffee там, где требуется ссылка на объект типа Mocha. Подобное приведение называется *сужением (narrowing)*, или *понижающим приведением*, в иерархии классов. Иногда его также называют *ненадежным приведением (unsafe casting)*, поскольку оно не всегда допустимо. Переход от типа, расположенного ниже в иерархии, к расположенному выше называется *повышающим приведением* типа; кроме того, употребляется термин *надежное приведение*, поскольку оно работает во всех случаях.

Но иногда вы совершенно точно знаете, что объект Coffee на самом деле является экземпляром класса Mocha. В этом случае можно осуществлять явное понижающее приведение. Это делается следующим образом:

Mocha fancy = (Mocha)joe;

Если такое приведение сработает (то есть ссылка joe действительно указывает на объект типа Mocha), то ссылка fancy будет указывать на тот же объект, что и joe, однако с ее помощью можно получить доступ к дополнительным функциям, предоставляемым классом Mocha. Если же преобразование окажется недопустимым, будет возбуждено исключение ClassCastException. В случае, если приведение даже потенциально не может быть правильным (например, если бы Mocha вообще не являлся подклассом того класса, к которому относится joe), будет выдано сообщение об ошибке во время компиляции программы. Таким образом предотвращаются возможные проблемы, связанные с неверными предположениями по поводу иерархии классов.

Иногда метод не требует, чтобы объект относился к расширенному типу, но может предоставить объектам расширенного типа дополнительные функции. Можно выполнить приведение типа и обработать исключение, однако использование исключений для подобных целей оказывается сравнительно медленным и потому считается проявлением плохого стиля. Для определения того, относится ли объект к конкретному типу, применяется метод instanceof, который возвращает true для допустимых преобразований:

```
public void quaff(Coffee joe) {
// ...
if (joe instanceof Mocha) {
Mocha fancy = (Mocha)joe;
// ... использовать функциональность Mocha
}
}
```

Ссылка null не указывает ни на какой конкретный объект, так что результат выражения

null instanceof Type

всегда равен false для любого типа Type.

5.13.3. Строковое приведение

Класс String отличается от остальных: это неявно используется в операторе конкатенации +, а строковые литералы ссылаются на объекты String. Примеры нам уже встречались в программах: при выполнении конкатенации Java пытается преобразовать в String все, что еще не относится к этому типу. Подобные приведения определены для всех примитивных типов и осуществляются вызовом метода toString объекта (см. раздел “Метод toString”).

Если преобразовать в String пустую ссылку, то результатом будет строка “null”. Если для данного класса метод toString не определен, то используется метод, унаследованный от класса Object и возвращающий строковое представление типа объекта.

5.14. Доступ к членам

Доступ к членам объекта осуществляется с помощью оператора . — например, obj.method(). Оператор . может применяться и для доступа к статическим членам либо по имени класса, либо по ссылке на объект. Если для доступа к статическим членам используется ссылка на объект, то выбор класса осуществляется на основании объявленного типа ссылки, а не фактического типа объекта. Для доступа к элементам массивов служат квадратные скобки — например, array[i].

Если использовать . или [] со ссылкой, значение которой равно null, то возбуждается исключение NullPointerException (кроме того случая, когда вы используете . для вызова статического метода). Если индекс массива выходит за его пределы, возбуждается исключение IndexOutOfBounds. Проверка осуществляется при каждом обращении к элементу массива. /По крайней мере runtime-система ведет себя именно так, но компилятор часто может избежать проверки в тех случаях, когда он действительно уверен, что все в порядке, например, если значение переменной цикла всегда лежит в допустимом диапазоне./

Для правильного вызова метода необходимо предоставить аргументы в нужном количестве и правильного типа, чтобы в классе нашелся ровно один метод с совпадающей сигнатурой. Если метод не перегружался, дело обстоит просто — с именем метода ассоциируется всего один набор параметров. Выбор оказывается несложным и в том случае, если был объявлен всего один метод с заданным именем и количеством аргументов.

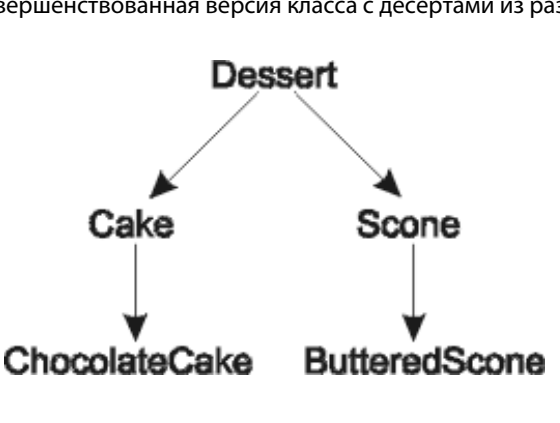
Если существует два и более перегруженных метода с одинаковым количеством параметров, выбор правильной версии становится несколько более сложным. Для этого Java пользуется следующим алгоритмом, называемым “алгоритмом самого точного совпадения”:

1. Найти все методы, к которым может относиться данный вызов, — то есть построить список всех перегруженных методов с нужным именем и типами параметров, которым могут быть присвоены значения всех аргументов. Если находится всего один метод со всеми подходящими аргументами, вызывается именно он.

2. У нас остался список методов, каждый из которых подходит для вызова. Проделаем следующую процедуру. Рассмотрим первый метод в списке. Если вместо параметров в первом методе могут быть использованы параметры еще какого-либо метода из списка — исключим первый метод из списка. Эта процедура повторяется до тех пор, пока остается возможность удаления методов.

3. Если остался ровно один метод, то совпадение для него оказывается самым точным, и он будет вызываться в программе. Если же осталось несколько методов, то вызов является неоднозначным; точного совпадения не существует, а вызов оказывается недопустимым.

Например, предположим, что у нас имеется усовершенствованная версия класса с десертами из раздела 3.2:



Допустим также, что у нас имеется несколько перегруженных методов, которые вызываются для конкретных комбинаций параметров Dessert:

```
void moorge(Dessert d, Scone s)       { /* Первая форма */ }
void moorge(Cake c, Dessert d)        { /* Вторая форма */ }
void moorge(ChocolateCake cc,Scone s) { /* Третья форма */ }
```

Теперь рассмотрим следующие вызовы moorge:

```
moorge(dessertRef, sconeRef);
moorge(chocolateCakeRef, dessertRef);
moorge(chocolateCakeRef, butteredSconeRef);
moorge(cakeRef, sconeRef); // НЕВЕРНО
```

В первом вызове используется первая форма moorge, потому что типы параметров и аргументов в точности совпадают. Во втором вызове используется вторая форма, потому что только в ней переданные аргументы могут быть присвоены в соответствии с типом параметров. В обоих случаях вызываемый метод определяется после выполнения первого шага описанного выше алгоритма.

С третьим вызовом дело обстоит сложнее. Список потенциальных кандидатов включает все три формы, потому что ссылка chocolateCakeRef может быть присвоена первому параметру, а ButteredScone — второму параметру во всех трех формах, и ни для одной из сигнатур не находится точного совпадения. Следовательно, после шага 1 у нас имеется набор из трех методов-кандидатов.

На шаге 2 из набора исключаются все методы с менее точным совпадением. В нашем случае первая форма исключается из-за того, что совпадение для третьей формы оказывается более точным. Действительно, рассмотрим третий и первый методы. Ссылка на ChocolateCake (из третьей формы) может быть присвоена параметру типа Desert (из первой формы), а ссылка на Scone (из третьей формы) непосредственно присваивается параметру типа Scone (в первой форме). Вторая форма исключается из набора по аналогичным соображениям. В итоге количество возможных методов сократилось до одного (третьей формы moorge), и именно этот метод и будет вызван.

Последний вызов является недопустимым. После шага 1 набор возможных методов состоит из первой и второй форм. Поскольку параметры любой из этих форм не могут быть присвоены параметрам другой, на шаге 2 не удается исключить из набора ни одну из этих форм. Следовательно, мы имеем дело с неоднозначным вызовом, по поводу которого компилятор не может принять никакого решения, что является недопустимым.

Эти же правила относятся и к примитивным типам. Например, значение int может быть присвоено переменной float, и при рассмотрении перегруженного вызова этот факт будет принят во внимание — точно так же, как и возможность присваивания ссылки ButteredScone ссылке Scone.

Перегруженные методы не могут отличаться одним лишь типом возвращаемого значения и/или списком возбуждаемых исключений, поскольку в противном случае при выборе запускаемого метода возникало бы слишком много неоднозначностей. Например, если бы существовало два метода doppelgänger, которые бы отличались только тем, что один из них возвращает int, а другой — short, то отдать предпочтение одному из них в следующем операторе было бы невозможно:

double d = doppelgänger();

Аналогичная проблема существует и для исключений, поскольку любое их количество (а также все или ни одно) может быть перехвачено во фрагменте программы, вызывающем перегруженный метод. Вам не удастся определить, какой из двух методов должен вызываться, если они отличаются только возбуждаемыми исключениями.

5.15. Арифметические операторы

Java поддерживает семь арифметических операторов, которые работают с любыми числовыми типами:

+ сложение

- вычитание

* умножение

/ деление

% остаток

Java также поддерживает унарный минус (-) для изменения знака числа. Знак может быть изменен оператором следующего вида:

val = -val;

Кроме того, имеется и унарный плюс — например, +3. Унарный плюс включен для симметрии, без него было бы невозможно записывать константы вида +2.0.

5.15.1. Целочисленная арифметика

Целочисленная арифметика выполняется с дополнением по модулю 2 — то есть при выходе за пределы своего диапазона допустимых значений (int или long) величина приводится по модулю, равному величине диапазона. Таким образом, в целочисленной арифметике никогда не происходит переполнения, встречаются лишь выходы значения за пределы диапазона.

При целочисленном делении происходит округление по направлению к нулю (то есть 7/2 равно 3, а –7/2 равно –3). Деление и остаток для целых типов подчиняются следующему правилу:

(x/y)*y + x%y == x

Следовательно, 7%2 равно 1, а –7%2 равно –1. Деление на ноль или нахождение остатка от деления на 0 в целочисленной арифметике не допускается и приводит к запуску исключения ArithmeticException.

Арифметические операции с символами представляют собой целочисленные операции после неявного приведения char к типу int.

5.15.2. Арифметика с плавающей точкой

Для работы с плавающей точкой (как для представления, так и для совершения операций) в Java используется стандарт IEEE 7541985. В соответствии с ним допускаются как переполнение в сторону бесконечности (значение превышает максимально допустимое для double или float), так и вырождение в ноль (значение становится слишком малым и неотличимым от нуля для double или float). Также имеется специальное представление NaN (“Not A Number”, то есть “не-число”) для результатов недопустимых операций — например, деления на ноль.

Арифметические операции с конечными операндами работают в соответствии с общепринятыми нормами. Знаки выражений с плавающей точкой также подчиняются этим правилам; перемножение двух чисел с одинаковым знаком дает положительный результат, тогда как при перемножении двух чисел с разными знаками результат будет отрицательным.

Сложение двух бесконечностей с одинаковым знаком дает бесконечность с тем же знаком. Если знаки различаются — ответ равен NaN. Вычитание бесконечностей с одинаковым знаком дает NaN; вычитание бесконечностей с разными знаками дает бесконечность, знак которой совпадает со знаком левого операнда. Например, (-(-)) равно . Результат любой арифметической операции, в которой участвует величина NaN, также равен NaN. При переполнении получается бесконечность с соответствующим знаком, а при вырождении — ноль с соответствующим знаком. В стандарте IEEE имеется отрицательный ноль, который равен +0.0, однако 1f/0f равно положительной бесконечности, а 1f/-0f равно отрицательной бесконечности.

Если -0.0 == 0.0, как же отличить отрицательный ноль, полученный в результате вырождения, от положительного? Его следует использовать в выражении, в котором участвует знак, и проверить результат. Например, если значение x равно отрицательному нулю, то выражение 1/x будет равно отрицательной бесконечности, а если положительному — то положительной бесконечности.

Операции с бесконечностями выполняются по стандартным математическим правилам. Сложение (или вычитание) конечного числа с любой бесконечностью также дает бесконечность. Например, (-+x) дает - для любого конечного x.

Бесконечность может быть получена за счет соответствующей арифметической операции или использования имени бесконечности для объектов типа float или double: POSITIVE_INFINITY или NEGATIVE_INFINITY. Например, Double.NEGATIVE_INFINITY представляет значение отрицательной бесконечности для объектов типа double.

Умножение бесконечности на ноль дает в результате NaN. Умножение бесконечности на ненулевое конечное число дает бесконечность с соответствующим знаком.

Деление, а также деление с остатком может давать бесконечность или NaN, но никогда не приводит к исключениям. В таблице перечислены результаты арифметических операций при различных значениях операндов:

x	y	x/y	x%y
Конечное	±0.0	±∞	NaN
Конечное	±∞	±0.0	x
±0.0	±0.0	NaN	NaN
±∞	Конечное	±∞	NaN
±∞	±∞	NaN	NaN

Во всех остальных отношениях нахождение остатка при делении с плавающей точкой происходит аналогично нахождению целочисленного остатка. Вычисление остатка методом Math.IEEERemainder описано в разделе 14.8.

5.15.3. Арифметика с плавающей точкой и стандарт IEEE-754

Арифметика с плавающей точкой в языке Java представляет собой подмножество стандарта IEEE-754-1985. Тем читателям, которым необходимо полное понимание всех связанных с этим аспектов, следует обратиться к спецификации языка Java “The Java Language Specification”. Вот краткая сводка ключевых отличий:

- *Непрерываемая арифметика:* в Java отсутствуют какие-либо исключения или иные события, сигнализирующие об исключительных состояниях в смысле IEEE: деление на ноль, переполнение, вырождение или потеря точности. В арифметике Java не предусмотрены какие-либо действия при появлении значения NaN.
- *Округление:* в арифметике Java происходит *округление к ближайшему* — неточный результат округляется к ближайшему представимому значению; при наличии двух одинаково близких значений предпочтение отдается тому, у которого младший бит равен 0. Это соответствует стандарту IEEE. Однако при преобразовании числа с плавающей точкой к целому в Java происходит округление по направлению к нулю. Java не допускает выбираемых пользователем режимов округления для вычислений с плавающей точкой: округления вверх, вниз или по направлению к нулю.
- *Условные операции:* в арифметике Java отсутствуют реляционные предикаты, которые бы реализовывали понятие упорядоченности, за исключением !=. Тем не менее все возможные случаи, кроме одного, могут быть смоделированы программистом с использованием существующих операций отношения и логического отрицания. Исключением является отношение “упорядочено, но не равно”, которое при необходимости может быть представлено в виде x<y || x>y.
- *Расширенные форматы:* арифметика Java не поддерживает никаких расширенных форматов, за исключением того, что double может выступать в качестве расширения формата с одинарной точностью. Наличие других расширенных форматов не является требованием стандарта.

5.15.4. Конкатенация строк

Оператор + может применяться для конкатенации строк. Приведем пример:

```
String boo = "boo";
String cry = boo + "hoo";
cry += "!";
System.out.println(cry);
```

А вот как выглядит результат работы:

boohoo!

Оператор + также используется для конкатенации объектов типа String со строковым представлением любого примитивного типа или объекта. Например, следующий фрагмент заключает строку в кавычки-“елочки” (guillemet characters), которые используются для цитирования во многих европейских языках:

```
public static String guillemete(String quote) {
return '"' + quote + '"';
}
```

Неявное преобразование примитивных типов и объектов в строки происходит лишь при использовании + или += в выражениях со строками — и нигде более. Например, методу, вызываемому с параметром типа String, нужно передавать именно String — вы не сможете передать объект или float и рассчитывать на его неявное преобразование.

5.16. Операторы приращения и уменьшения

Операторы ++ и — применяются соответственно для приращения и уменьшения значений. Выражение i++ эквивалентно i = i + 1, если не считать того, что в первом случае i вычисляется всего один раз. Например, оператор

arr[where()]++;

всего один раз вызывает метод where и использует результат в качестве индекса массива. С другой стороны, в операторе

arr[where()] = arr[where()] + 1;

метод where будет вызываться дважды: один раз при вычислении правостороннего индекса, а во второй раз — при вычислении левостороннего индекса. Если при каждом вызове where возвращается новое значение, результат будет отличаться от приведенного выше выражения, в котором использован оператор ++.

Операторы приращения и уменьшения могут быть либо *префиксными*, либо *постфиксными* — то есть стоять либо до, либо после своего операнда. Если оператор стоит перед операндом (префикс), то операция приращения/уменьшения выполняется *до* возвращения результата выражения. Если же оператор стоит *после* (постфикс), то операция выполняется *после* использования результата. Пример:

```
class IncOrder {
public static void main(String[] args) {
int i = 16;
System.out.println(++i + " " + i++ + " " + i);
}
}
```

Результат работы будет выглядеть так:

17 17 18

Первое выведенное значение равно i после выполнения префиксного приращения до 17; второе значение равно i после этого приращения, но до выполнения следующего, постфиксного приращения до 18; наконец, значение i выводится после его постфиксного приращения в середине оператора вывода.

Операции приращения и уменьшения ++ и — могут применяться к переменным типа char для получения следующего или предыдущего символа в кодировке Unicode.

5.17. Операторы отношения и условный оператор

В Java имеется стандартный набор операторов отношения и условных операторов:

> больше

>= больше или равно

< меньше

<= меньше или равно

== равно

!= не равно

Все эти операторы возвращают логические значения. Унарный оператор ! инвертирует логическую величину, и, следовательно, !true — это то же самое, что и false. Проверка логических величин обычно производится непосредственно — если x и y относятся к логическому типу, то считается более изящным написать

```
if (x || !y) {
// ...
}
```

нежели

```
if (x == true || y == false) {
// ...
}
```

Результаты логических операций могут объединяться посредством операторов && и ||, которые означают “логическое И” и “логическое ИЛИ” соответственно. По возможности эти операторы стараются обойтись без вычисления своего второго операнда. Например:

```
if (w && x) {          // внешнее "если"
if (y || z) {  //  внутреннее "если"
// ...   //  тело внутреннего "если"
}
}
```

Внутреннее “если” выполняется лишь в том случае, если оба значения (w и x) равны true. Если w равно false, то Java не станет вычислять x. Тело внутреннего “если” выполняется, если хотя бы одно из значений, y или z, равно true. Если y равно true, то Java не вычисляет z.

Это правило часто используется во многих программах на Java для обеспечения правильности или эффективности работы. Например, данный метод делает безопасным следующий фрагмент:

```
if (ix >>= 0 && ix << array.length && array[ix] != 0) {
// ...
}
```

Сначала выполняется проверка диапазона — только в том случае, если переменная x лежит в допустимых пределах, она будет использована для доступа к элементу массива array.

С логическими значениями используются только операторы равенства == и !=, поскольку вопрос о том, что больше — true или false — является бессмысленным.

Данное обстоятельство может использоваться для имитации “логического исключающего ИЛИ”. В приведенном ниже фрагменте метод sameSign будет выполняться лишь в том случае, если x и y имеют одинаковый знак (или одновременно равны нулю), в противном случае выполняется differentSign:

```
differentSign:
if (x << 0 == y << 0)
sameSign();
else
differentSign();
```

Значения с плавающей точкой подчиняются стандартному упорядочению (–1.0 меньше 0.0, которое, в свою очередь, меньше положительной бесконечности), за исключением аномального значения NaN. Все операторы отношения, которые сравнивают NaN с числом, всегда возвращают false, кроме оператора !=, который всегда возвращает true. Данное утверждение остается истинным, даже если оба операнда равны NaN. Например, значение выражения:

Double.NaN == Double.NaN

всегда равно false. Чтобы проверить, является ли некоторое значение NaN, используйте специальные средства, определяемые на уровне типа: статические методы Float.isNaN(float) и Double.isNaN(Double).

Две ссылки на объект всегда могут проверяться на равенство. Выражение ref1==ref2 равно true, если обе ссылки указывают на один и тот же объект или обе они равны null, даже если ссылки относятся к различным объявленным типам. В противном случае возвращается false.

С объектами String операторы равенства работают не так, как можно было бы ожидать. Для заданных объектов str1 и str2, относящихся к типу String, выражение str1==str2 проверяет, указывают ли эти две ссылки на один и тот же объект String. Оно *не* проверяет, совпадает ли содержимое этих строк. Равенство содержимого определяется методом String.equals, описанным в главе 8.

5.18. Поразрядные операции

Существуют следующие бинарные поразрядные операции:

& поразрядное И

| поразрядное включающее ИЛИ

^ поразрядное исключающее ИЛИ (XOR)

Кроме того, имеется унарный оператор дополнения ~, который изменяет состояние каждого бита операнда на противоположное. Дополнение целого значения 0x00003333 равняется 0xffffcccc.

Другие операторы, работающие с операндами на уровне битов, осуществляют сдвиг битов в целых значениях. Это следующие операторы:

<< сдвиг влево с заполнением правых битов нулями

>> сдвиг вправо с заполнением левых битов содержимым

старшего (знакового) бита

>>>> > сдвиг вправо с заполнением левых битов нулями

В левой части оператора сдвига указывается сдвигаемое значение, а в правой — количество разрядов. Например, var>>>2 сдвигает биты переменной var на два разряда вправо и заполняет два старших бита нулями.

Правила типов для операторов сдвига несколько отличаются от тех, что применяются к прочим бинарным операторам. Тип результата оператора сдвига совпадает с типом левого операнда — то есть сдвигаемого значения. Если в левой части оператора сдвига стоит величина типа int, то результат сдвига также будет иметь тип int, даже если количество сдвигаемых разрядов было представлено в виде значения типа long.

Если количество разрядов сдвига превышает количество бит в слове или если оно окажется отрицательным, то фактическое количество разрядов сдвига будет отличаться от заданного. Оно будет представлять собой заданное количество с применением маски, равной размеру типа за вычетом единицы. Например, для 32-разрядного значения типа int используется маска 0x1f (31), так что выражения n<<35 и n<<-29 будут эквивалентны n<<3.

Поразрядные операторы могут также использоваться с логическими значениями. & и | возвращают те же значения, что и их логические аналоги && и ||, с одним важным отличием: в поразрядных операторах всегда вычисляются оба операнда, тогда как в логических операторах оба операнда вычисляются лишь в случае необходимости.

Поразрядный оператор ^ возвращает значение true, если операнды различаются — лишь один из них равен true, но не оба сразу. ^ предоставляет еще одну возможность для моделирования “логического исключающего ИЛИ”:

```
if ((x << 0) ^ (y << 0))
differentSign();
else
sameSign();
```

Поразрядные операторы могут применяться только для значений целого и логического типа, но не для значений с плавающей точкой или ссылок. Операторы сдвига применяются только для целых типов. В тех редких случаях, когда вам понадобится манипулировать с битами в представлении величины с плавающей точкой, можно воспользоваться методами преобразования для классов Float и Double, рассмотренными в разделе “Float и Double” на стр. .

5.19. Условный оператор

Условный оператор позволяет всего в одном выражении выбрать одно из двух значений на основании логической переменной. Приводимая ниже запись:

```
value = (userSetIt ? usersValue : defaultValue);
```

равнозначна следующей:

```
if (userSetIt)
value = usersValue;
else
value = defaultValue;
```

Главное отличие между операторами if и ?: заключается в том, что последний обладает собственным значением. Условный оператор обеспечивает более компактную запись, однако не все программисты соглашаются с тем, что он лучше воспринимается читателем программы. Мы пользуемся тем или иным вариантом в зависимости от обстоятельств. Использование скобок, окружающих выражения условного оператора, является вопросом личного вкуса, и на практике встречаются самые различные варианты. Сам язык не требует присутствия скобок.

Выражения-результаты (второе и третье) должны иметь тип, совместимый с операцией присваивания. Каждое из них должно быть таким, чтобы оно присваивалось другому без явного приведения типа. Тип результата условного оператора совпадает с более общим из типов двух возможных результатов. Например, в операторе

double scale = (halveit ? 1 : 0.5);

результаты относятся к типам int (1) и float (0.5). Значение типа int может быть присвоено переменной double, поэтому условный оператор также имеет тип double. Это правило сохраняется и для ссылок — если значение одного типа может присваиваться другому, то типом операции будет наиболее общий (наименее расширяемый) из них. Если ни один из этих типов не может быть присвоен другому, то операция является недопустимой.

Условный оператор иногда называют или оператором “вопросительный знак/точка” из-за формы его записи, или “тернарным оператором”, поскольку это единственный тернарный (трех-операндный) оператор в языке Java.

5.20. Операторы присваивания

Простой знак = является основной формой оператора присваивания. Java поддерживает много других разновидностей присваивания. Любой арифметический или бинарный поразрядный оператор может быть объединен с = для образования оператора присваивания. Например:

arr[where()] += 12;

приводит к тому же результату, что и

arr[where()] = arr[where()] + 12;

за исключением того, что в первой записи выражение в левой части вычисляется всего один раз.

Для заданной переменной var типа Type, значения expr и бинарного оператора op запись

var op= expr

эквивалентна следующей:

var = (Type)((var) op (expr))

за исключением того, что значение var вычисляется всего один раз. Это означает, что запись op= допустима лишь в том случае, если оператор op может быть использован для типов, участвующих в выражении. Так, вы не сможете применить <<= с переменными типа double, потому что оператор сдвига << не работает с double.

Обратите внимание на использование скобок в приведенной выше записи. Выражение

a *= b + 1

эквивалентно

a = a * (b + 1)

но не

a = a * b + 1

Хотя a += 1 — то же самое, что и ++a, запись с использованием ++ счи- тается более наглядной, и потому ей отдается предпочтение.

5.21. Имена пакетов

Имя пакета представляет собой последовательность идентификаторов, разделяемых точками (.). В течение некоторого времени текстовые редакторы с поддержкой Unicode еще будут оставаться редкостью, так что в именах пакетов, предназначенных для широкого распространения, стоит ограничиваться ASCII-символами.

Упражнение 5.2

На основании того, что вы узнали в этой главе (но без написания программ на Java!), определите, какие из приведенных ниже выражений являются неверными, а также укажите тип и значение верных выражений:

```
3 <<<< 2L -1
(3L <<<< 2) -1
10 << 12 == 6 >> 17
10 <<<< 12 == 6 >>>> 17
13.5e-1 % Float.POSITIVE_INFINITY
Float.POSITIVE_INFINITY + Double.POSITIVE_INFINITY
Double.POSITIVE_INFINITY + Float.POSITIVE_INFINITY
0.0 / -0.0 == -0.0 / 0.0
Integer.MAX_VALUE + Integer.MIN_VALUE
Long.MAX_VALUE + 5;
(short)5 * (byte)10
(i << 15 ? 1.72e3f : 0)
i++ + i++ + --i    // исходное значение i = 3
```

Содержание | Далее

Издательский дом ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Приложение А
Родные методы

Их прозвали “чудо-рабочими”, когда один из них поинтересовался, каким гаечным ключом нужно забивать шуруп в стену

Джордж Браун, конгрессмен, Сан-Бернардино, Калифорния.

Иногда возникают ситуации, когда приложение или библиотека не могут быть написаны исключительно на языке Java, и тогда приходится создавать код на другом языке, который, вероятно, более точно учитывает специфику используемой платформы. Обычно потребность в этом возникает в следующих случаях:

- Уже имеется большой объем работающего программного кода. Проще написать “прокладку” для этого кода на Java, чем переписывать его заново.
- Приложение должно пользоваться системными средствами, отсутствующими в классах Java.
- Среда Java не обладает достаточным быстродействием для приложений, критичных по времени, и реализация их на другом языке может оказаться более эффективной.

Чтобы помочь программисту в подобных ситуациях, Java позволяет реализовывать *родные (native)* методы на каком-либо из локальных (родных) языков программирования, обычно C или C++. Родные методы объявляются следующим образом:

public native void unlock() throws IOException;

Ключевое слово *native* представляет собой еще один модификатор для объявляемого метода. Родные методы реализуются на родных языках, поэтому для них не существует программного кода на Java. Тем не менее, они вызываются из Java-программ, как и любые другие методы.

Класс, содержащий родной метод, в соответствии с требованиями безопасности, не может загружаться по сети и выполняться. Более конкретно — классы с родными методами не могут использоваться в аплетах<$Iaплет>. Даже если вопросы безопасности вас не интересуют, родной код не дает тех гарантий переносимости, которые предоставляет Java. Любой код на Java, использующий родные методы, должен отдельно переноситься на каждую целевую платформу.

И все же родные методы могут приносить пользу, особенно если вы будете полагаться на общедоступные библиотеки. Однако код Java, содержащий родные методы, ни в коем случае не может применяться в качестве аплета, запускаемого удаленным пользователем, поскольку аплет должен соответствовать требованиям переносимости и безопасности.

При использовании родных методов также приходится жертвовать защитными ограничениями Java. Так, в традиционных языках, может происходить выход за границы массива и появление неопределенных значений указателей. Все ограничения на родные методы задаются только тем языком, на котором они написаны.

В этой главе рассказано, как реализовать родной метод на языке C в системах семейства POSIX (к которым относятся, например, Windows NT и большинство реализаций UNIX). Некоторые детали в вашей системе могут отличаться от описанных, а многие среды поддерживают и другие языки кроме C, например, C++. Информацию об этом можно найти в вашей документации. Здесь мы рассматриваем многие важные аспекты согласования языка C с системой Java компании Sun, версия 1.0.2. Возможно, в вашей локальной среде присутствуют изменения и усовершенствования или используется совершенно иная схема согласования. В частности, описанные здесь способы связывания (binding) родных методов наверняка изменятся в будущих версиях. Но даже с учетом этих обстоятельств, данная глава поможет вам понять некоторые аспекты связывания родных методов, не зависящие от конкретной схемы, принятой в вашей среде.

А.1 Обзор

При стыковке программ на Java с языком C возникают следующие основные проблемы:

- Как происходит согласование имен? Полное имя метода в Java имеет вид *пакет.класс.метод*, однако в C нет ни пакетов, ни классов. Кроме того, согласование усложняется тем, что в идентификаторах Java используется кодировка Unicode, а в идентификаторах C — кодировка ASCII, поэтому необходим дополнительный перевод символов Unicode в символы, разрешенные в C.
- Как разрешается проблема различных парадигм вызова? Например, каждый нестатический метод в Java располагает ссылкой *this*, которая, в сущности, является неявным параметром метода. В C нет ни методов, ни неявных параметров.
- Как происходит согласование типов? Родная реализация метода должна обращаться к полям объекта *this*, и, возможно, методам или полям объектов других типов. Как представить классы Java в языке C?
- Как происходит согласование ошибок? Java сообщает о них при помощи исключений, но в C исключения отсутствуют.
- Как происходит согласование средств безопасности? Java следит за выходом за границы массивов и преобразованиями типов, а также осуществляет сборку мусора для борьбы с утечкой памяти и “зависшими” указателями. C и C++ не обладают этими возможностями, так как же производить такие проверки? А что должно происходить в языках типа Pascal, где такая проверка присутствует?
- Как происходит согласование работы с памятью? Как программа на языке C создает объекты Java?

Решая эти и другие проблемы, приходится идти на компромиссы. Например, C и C++ не обладают средствами безопасности Java в работе с массивами, поэтому при согласовании предполагается, что родные методы C и C++ небезопасны в этом отношении. Хотя такой выход и не идеален, он все же выглядит вполне естественно по отношению к C и C++, для которых скорость считается более важной, чем страховка. Например, чтобы реализовать подобную проверку в C или C++, пришлось бы обращаться ко всем элементам массива посредством проверочных функций Java. Такой вариант выглядит неестественно и медленно работает, а поскольку основным доводом в пользу родных методов является скорость, подобный компромисс окажется неверным. К тому же он не будет нормально работать с существующим кодом, в котором используется стандартная для C и C++ парадигма работы с массивами.

А.2 Согласование с C и C++

Согласование Java с языком C происходит довольно прямолинейно. Для стыковки родных методов с вызовами C используется сгенерированный заголовочный файл, содержащий все необходимые объявления типов и сигнатуры функций, а также программные “заглушки” на C, которые помогают runtime-системе Java вызывать эти методы.

Мы рассмотрим только основные моменты согласования. Программа, которая используется здесь в качестве примера, представляет собой текст класса *LockableClass* из пакета *local*: /К сожалению, мы не смогли воспользоваться соглашением об именах пакетов, поскольку это привело бы к удлинению идентификаторов и затруднило бы работу с книгой./

```
package local;

import java.io.*;

class LockableFile extends File {
     LockableFile(String path) {
          super(path);
     }

     // допустимые параметры lock()
     public final static int READ = 0,
                             WRITE = 1;

     public native void lock(int type) throws IOException;
     public native void unlock() throws IOException;
     private int fd = -1;

     static {
          System.load("LockableFile");
     }
}
```

После того, как эта программа будет обработана компилятором Java, следует сгенерировать заголовочный файл с помощью утилиты **javah**, передав ей имя класса, для которого создается данный файл. Утилита сгенерирует файл, содержащий все объявления и определения на языке C, необходимые для стыковки. В нашем примере команда будет выглядеть следующим образом:

javah local.LockableFile

Имя сгенерированного файла определяется по имени класса, в соответствии с описанной ниже схемой согласования имен. В нашем случае заголовочный файл будет называться **local_LockableFile.h**. Содержимое этого заголовочного файла и реализация его родных методов будут приведены ниже в данной главе.

Процесс согласования с C++ выглядит так же. Фактически, согласование с C++ сводится к включению сгенерированного заголовочного файла в объявление *extern "C"*:

```
"C":
extern "C" {
# include local_LockableFile.h
}
```

Символы, которые используются во время выполнения программы для вызова оболочек родных методов, создаются в пространстве имен C, а не так называемых “преобразованных” (mangled) имен C++, поэтому родные методы должны быть написаны на C. /На самом деле это не совсем так —приложив некоторые усилия, опытный программист сможет обойти это ограничение, но для простоты описания и реализации будет лучше согласиться с ним./ Основное следствие заключается в том, что вы не сможете использовать перегрузку методов C++ для реализации родных методов. В сущности, правильнее было бы сказать, что для родных методов прямая стыковка с C++ вообще не используется, однако возможна косвенная стыковка за счет вызовов функций C в C++. Разумеется, согласование программ на Java с C++ может быть улучшено, и несомненно это будет сделано в будущих версиях.

Реализуя родные методы на C или C++, вы должны связать откомпилированный код с приложением на Java; для этого необходимо создать библиотеку динамического связывания и соединить ее со своей программой. Чаще всего программист выделяет статический блок, наподобие приведенного выше в классе *LockableFile*, а затем вызывает один из двух статических методов класса *System*, предназначенных для загрузки библиотек:

public synchronized void ***load(String pathname)***

> Загружает динамическую библиотеку, расположенную по заданному полному имени *pathname*. В некоторых случаях полное имя модифицируется в соответствии с требованиями локальной системы. Если файл не обнаружен или не найдены символы, необходимые для работы библиотеки, возбуждается исключение *UnsatisfiedLinkError*.

public synchronized void ***loadLibrary(String libname)***

> Загружает динамическую библиотеку с указанным именем *libname*. Вызов *LoadLibrary* должен осуществляться в статическом инициализаторе первого загружаемого класса (то есть класса, содержащего вызываемый метод *main*). Попытки многократной загрузки одной и той же библиотеки игнорируются. Если библиотека не найдена, возбуждается исключение *UnsatisfiedLinkError*.

Метод *load* требует указания полного имени файла, однако во время разработки лучше пользоваться именно этой версией, поскольку она нормально сообщает о неопределенных символах. Метод *loadLibrary* переводит любую ошибку, включая неопределенные символы, в ошибку “библиотека не найдена”. Если в библиотеке присутствуют неопределенные символы, то подобный перевод скроет от вас важную информацию об истинной причине происходящего. Если вы будете пользоваться методом *load* до того момента, когда родные методы заработают, после чего замените его методом *loadLibrary*, то сможете добиться более подробной информации в ходе разработки и переносимости кода после ее завершения.

Методы загрузки библиотек класса *System* представляют собой сокращения для вызова тех же самых методов класса *Runtime*, представляющего текущий runtime-контекст.

А.2.1 Имена

Для перевода имен методов и полей на язык C используются полные имена, включающие названия пакетов, в которых все точки (.) заменяются подчеркиваниями (_). В тех языках, где не поддерживается возможность перегрузки методов (к ним относится C) вы не сможете реализовать в классе несколько методов с одинаковыми именами, поскольку им будет соответствовать одно и то же имя функции.

Аналогично переводятся и имена типов — за тем исключением, что перед именем типа ставится префикс *Class*. В программе на C тип для *LockableFile* будет называться *Classlocal_LockableFile*. Для каждого класса также необходим дескриптор (handle), поскольку он используется для внутреннего представления ссылок. Имя дескриптора совпадает с именем класса, но вместо префикса *Class* используется префикс *H*. Таким образом, тип дескриптора для *LockableFile* будет называться *Hlocal_LockableFile*.

Символы имен, относящиеся к набору ASCII (символы, меньшие или равные *\u007f*) переходят в C и в имена пакетов без изменений. Все остальные символы переводятся в вид *_0dddd*, где *d* — цифры, используемые в символьном представлении Java. Например, символ г (код *\u00e3*) будет представлен идентификатором *_000e3*. Косая черта (/) в имени пакета переходит в символ подчеркивания (_).

А.2.2 Методы

Каждый родной метод представляется в виде функции. Например, метод *lock* будет называться *local_LockableFile_lock* и иметь соответствующие параметры. Первый параметр функции — это дескриптор объекта, для которого вызывается метод (ссылка *this*). Для статических методов такой дескриптор всегда равен *null*. Ниже дескрипторы рассматриваются более подробно.

Для вызова методов нужен дополнительный слой в виде файлов-заглушек (stub files). Последние также генерируются утилитой **javah**, но с параметром **-stubs**:

javah -stubs local.LockableFile

Такая команда генерирует исходный файл на языке C, который должен быть скомпилирован и загружен в динамическую библиотеку вместе с реализациями родных методов. Имя этого файла совпадает с именем заголовочного файла, однако расширение **h** изменяется на **c** — в нашем случае это будет файл **local_LockableFile.c**.

А.2.3 Типы

В приведенной ниже таблице показано соответствие между примитивными типами Java и типами языка C, когда они используются в качестве параметров методов или полей (согласование типов для массивов рассматривается ниже).

Тип Java	Тип C
boolean	long
byte	long
short	long
int	long
long	int64_t
float	float
double	double
char	long

Классы Java представляются в языке C структурами (*struct*). Все нестатические поля класса являются членами структуры, а их имена совпадают с именами в Java (за исключением символов Unicode, отображаемых в эквиваленты *_0dddd*). Это означает, что класс, содержащий родные методы, не может иметь два нестатических поля с одинаковыми именами; в противном случае структура на языке C содержала бы члены с одинаковыми именами, что запрещено.

Проблема возникает с теми классами, которые скрывают поля суперклассов — кстати, еще одна причина, по которой не следует скрывать имена полей. Тем не менее, проблема относится даже к тем классам, которые содержат закрытые поля с тем же именем, поскольку такие поля присутствуют в структуре наравне со всеми остальными. Вы не сможете выяснить, актуальна эта проблема или нет, пока она не возникнет при написании родного метода. Дело обстоит еще хуже, если скрываемые поля находятся в суперклассе, который невозможно изменить. В таких случаях дело заходит в тупик — вы не сможете написать родной метод без модификации имен полей класса.

Каждая статическая константа с атрибутом *final* представлена константой *#define* с префиксами (именем пакета и класса). Статические поля, не являющиеся *final*, не переводятся ни во что. Например, тип и константы для класса *LockableFile* определяются в заголовочном файле следующим образом:

```
typedef struct Classlocal_LockableFile {
     struct Hjava_lang_String *path;
/* недоступное статическое поле: separator */
/* недоступное статическое поле: separatorChar */
/* недоступное статическое поле: pathSeparator */
/* недоступное статическое поле: pathSeparatorChar */
#define local_LockableFile_READ 0L
#define local_LockableFile_WRITE 1L
     long fd;
} Classlocal_LockableFile;
```

Ссылки на объекты представляются типом “дескриптор”, в составном имени которого *Class* заменяется на *H*. Ссылка на класс *LockableFile* будет называться *Hlocal_LockableFile*. Макрос *unhand* получает дескриптор и возвращает указатель на структуру, которая представляется этим дескриптором.

Ниже приведены сигнатуры функций языка C, в которых ниже мы определим родные методы класса *LockableFile*:

```
extern void local_Lockable_File_lock(
     struct Hlocal_LockableFile *, long);
extern void local_Lockable_File_unlock(
     struct Hlocal_LockableFile *);
```

В Java об ошибках сигнализируют исключения. В языке C исключений нет. Чтобы возбудить исключение из C, следует вызвать функцию *SignalError* и затем выйти. Runtime-система Java обнаруживает и возбуждает исключение, о котором сигнализировала функция *SignalError*. Ниже вы увидите несколько примеров того, как это делается.

А.2.5 Средства безопасности

Средства безопасности языка Java не имеют аналогов в C. Вы полностью отвечаете за работу программы на C (как это обычно бывает) без какой-либо автоматической помощи со стороны Java.

А.2.6 Работа с памятью

Родные методы могут создавать новые объекты Java с помощью функций, описанных ниже.

А.3 Пример

Давайте рассмотрим возможную реализацию родных методов для класса *LockableFile*. Прежде всего, мы должны сгенерировать заголовочный файл и файл-заглушку и скомпилировать последний. Затем нужно написать сами реализации методов. Например, метод *lock* может быть реализован следующим образом:

```
#include "local_LockableFile.h"
#include <<javaString.h>>
#include <<fcntl.h>>
#include <<errno.h>>

void
local_LockableFile_lock(
     struct Hlocal_LockableFile *this_h,
     long mode)
{
     Classlocal_LockableFile *this = unhand(this_h);
     struct flock lock;

     if (this->>fd == -1 && !open_fd(this))
          return;       /* произошла ошибка */

     if (!setup_lock(&lock, mode))
          return;
     if(fcntl(this->>fd, F_SETLKW, &lock) == -1)
          SignalError(EE(),
               "java/io/IOException", strerror(errno));
}
```

Сначала мы включаем нужные заголовочные файлы — сгенерированный файл для *LockableFile*, вспомогательный заголовочный файл для работы со строками *<<javastring.h>>*, системный заголовочный файл *<<fcntl.h>>*, определяющий вызовы для осуществления блокировки в POSIX, и системный заголовочный файл *<<errno.h>>* для обработки ошибок, полученных в ходе вызовов системных функций.

Первая строка в реализации метода *lock* переводит дескриптор *this_h* в указатель на структуру *Classlocal_LockableFile*, для чего используется макрос *unhand*, который возвращает объект, соответствующий данному дескриптору. Ссылкам *null* в Java ставится в соответствие указатели на дескрипторы, которые также равны *NULL*. Чтобы убедиться в том, что передача ссылки *null* не приведет к ошибкам, необходимо проверить дескриптор перед тем, как вызывать для него макрос *unhand* — это демонстрируется в последующих примерах.

Во второй строке *local_LockableFile_lock* объявляется структура *flock*. Структура *flock* используется для работы с блокировкой в POSIX. Реализации *open_fd* и *setup_lock* приводятся в разделе “Внутреннее строение *LockableFile*”.

Далее мы проверяем, имеется ли файловый дескриптор. Если он отсутствует и функция *open_fd* не может открыть файл, видимо, было возбуждено исключение, сигнализирующее об ошибке, поэтому мы просто выходим из функции. Если же дескриптор имеется, то структуру *flock* необходимо подготовить вызовом внутренней функции *setup_lock*. Вызов функции также может закончиться неудачей (например, если *mode* имеет недопустимое значение) и возбуждением исключения — и в этом случае мы выходим из функции. Функции *open_fd* и *setup_lock* являются частью кода, специфичного для POSIX.

Затем мы пытаемся заблокировать файл с помощью режима *F_SETLKW* функции POSIX с именем *fcntl*, который при необходимости ожидает возможности блокировки. Если *fcntl* возвращает -1, то попытка блокировки оказалась неудачной, и мы возбуждаем исключение, вызывая runtime-функцию Java с именем *SignalError*:

void ***SignalError(ExecEnv *exenu, char *type, char *constructor)***

> Сигнализирует о том, что после выхода из родного метода должно быть возбуждено исключение. Структура *exenu* обычно возвращается функцией *EE* и представляет текущее состояние среды. Параметр *type* является полным именем класса возбуждаемого объекта-исключения, в котором каждая точка (.) заменяется чертой (/). Последний параметр содержит строковое описание исключения или *NULL* при его отсутствии.

В нашем случае используется значение функции POSIX с именем *strerror*, которая возвращает строку с описанием номера ошибки. Практически это самое лучшее, что мы можем сделать для описания ошибок в родных методах.

Функция *SignalError* лишь подготавливает исключение; она не возбуждает его. В языке C исключения не предусмотрены, поэтому возбудить их из программы невозможно. После выхода из функции, содержащей реализацию родного метода, runtime-система проверяет флаги и по ним определяет, был ли получен сигнал о возбуждении исключения. Если такой сигнал получен, то runtime-система возбуждает исключение. Подобная схема позволяет в программе на C выполнить необходимые завершающие действия после “возбуждения” исключения. В функции *local_LockableFile_lock* такие действия не требуются, поэтому после “возбуждения” исключения мы просто выходим из нее.

Реализация *unlock* выглядит проще. Мы подготавливаем структуру *flock* и вызываем *fcntl* для режима *F_UNLCK* (снятие блокировки). И снова при неудаче возбуждается исключение, содержащее строку с описанием ошибки:

```
void
local_LockableFile_unlock(
     struct Hlocal_LockableFile *this_h)
{
     Classlocal_LockableFile *this = unhand(this_h);
     struct flock lock;

     lock.l_whence = lock.l_start = lock.l_len = 0;
     lock.l_type = F_UNLCK;
     if (fcntl(this->>fd, F_SETLKW, &lock) == -1)
          SignalError(EE(),
               "java/io/IOException", strerror(errno));
}
```

В класс *LockableFile* можно внести ряд усовершенствований. Например, создать функцию для проверки того, заблокирован ли файл и если да, то каким программным потоком. Можно сконструировать специальный вариант *lock* без ожидания, который лишь осуществляет блокировку, если она возможна, а в противном случае завершает свою работу. Или создать отдельные исключения для каждой ошибки, чтобы ошибка “файл не существует” отличалась от “доступ запрещен”.

Упражнение А.1

Если у вас имеется доступ к системе, отличной от POSIX и поддерживающей блокировку файлов, реализуйте класс *LockableFile* с использованием ее механизмов.

Упражнение А.2

Если вы работаете только с POSIX-совместимыми системами или выполнили упражнение А.1, включите в класс описанные выше возможности.

А.3.1 Внутреннее строение *LockableFile*

Для полноты картины приведем текст внутренних функций, используемых классом *LockableFile*. Статические функции *open_fd* и *setup_lock* используются при реализации родных методов *lock* и *unlock*:

```
static int
open_fd(Classlocal_LockableFile *this)
{
     char *path = allocCString(this->>path);
     if ((this->>fd = open(path, O_RDWR)) == -1)
          SignalError(EE(),
               "java/io/IOException", strerror(errno));
     free(path);      /* больше не требуется */
     return (this->>fd != -1);
}

static int
setup_lock(
     struct flock *lock,
     long mode)
{
     lock->>l_whence = lock->>l_start = lock->>l_len = 0;
     switch (mode) {
          case local_LockableFile_READ:
               lock->>l_type = F_RDLCK;
               break;
          case local_LockableFile_WRITE:
               lock->>l_type = F_WRLCK;
               break;
          default:
               SignalError(EE(),
                    "java/lang/IllegalArgumentException", NULL);
               return 0;
     }
     return 1;
}
```

А.4 Строки

Родные методы часто должны использовать строковые объекты *String*. Заголовочный файл *<<javaString.h>>* определяет несколько функций, помогающих в решении этой задачи. Все функции, преобразующие объекты *String* языка Java в строки C, переносят в них только младшие 8 бит символов Unicode. Эти функции работают с дескрипторами, которые передаются родным методам, хранятся в структурах языка C или создаются с помощью приведенных ниже функций.

char ****allocCString****(Hjava_lang_String *str)*

> Вызывает функцию *malloc* языка C для создания буфера, размер которого достаточен для хранения строки. Когда вы закончите работать со строкой, не забудьте вызвать *free* для возвращенного указателя.

char ****javaString2CString****(Hjava_lang_String *str, char buffer[], int length)*

> Копирует до *length* символов из *str* в *buffer*. Размещение и освобождение буфера лежит на совести программиста. Для удобства функция возвращает *buffer*.

char ****makeCString****(Hjava_lang_String *str)*

> Возвращает строку языка C, которая может быть уничтожена сборщиком мусора. Сборщик мусора в поисках ссылок сканирует не только объекты Java, но и данные C, поэтому строка не может быть уничтожена им в случае, если указатель на нее используется в родном методе.

int ***javaStringLength****(Hjava_lang_String *str)*

> Возвращает длину строки Java. Функции передается параметр-дескриптор.

unicode ****javaString2unicode****(Hjava_lang_String *str, unicode *buf, int len)*

> Копирует до *len* символов Unicode из *str* в буфер *buf*. Тип данных *unicode* определяется при включении файла *<<native.h>>* и представляет собой 16-разрядное символьное значение.

Для создания новых объектов используется функция *makeJavaString*:

Hjava_lang_String ****makeJavaString****(char *str, int len)*

> Возвращает дескриптор нового строкового объекта, созданного на основе строки *str* языка C, с использованием начальных *len* байтов строки. Длина не должна учитывать нуль-байт, завершающий строку в языке C.

Если какой-либо из этих методов столкнется с ошибкой (например, нехваткой памяти), он вызывает *SignalError* с соответствующим исключением и возвращает *NULL*. В этом случае необходимо выполнить нужные завершающие действия и выйти из функции.

Ниже показано, как можно написать родной метод с использованием функции *strxfrm*, соответствующей стандарту ANSI C, которая по обычной 8-разрядной строке стандарта Latin-1 создает порядковую строку, соответствующую локальному языковому контексту. Созданная строка может использоваться для сравнения исходной строки с другими строками. Локальный языковой контекст определяет порядок сортировки принятый во французском языке, норвежском, испанском и т. д. Сортировка строк с помощью функции *strxfrm* выполняется следующим образом:

1. Вызовите *strxfrm* для двух строк, чтобы создать для каждой из них порядковую строку.
1. Сравните порядковые строки функцией *strcmp*, которая возвращает отрицательное, равное нулю или положительное число, если первая строка соответственно меньше, равна или больше второй.
1. Упорядочите исходные строки в зависимости от результатов вызова *strcmp* для двух порядковых строк.

Эта процедура может пригодиться, если вашей программе часто приходится сортировать строки в соответствии с локальным языковым порядком. Другая возможность заключается в том, чтобы применить *strcoll* вместо *strcmp*. Вызов функции *strcoll* обходится дороже, чем обращение к *strcmp*, но он не требует хранения порядковых строк для их последующего использования.

Приведем пример класса, который содержит вспомогательные методы, учитывающие особенности языков при работе со строками:

```
package local;

public class LocalString {
    /** возвращает порядковую строку для str */
    public native static String xfrm(String str);

    /** сортирует массив в соответствии с локальным контекстом */
    public native static String[] sort(String[] input);

    static {
        System.loadLibrary("LocalString");
    }
}
Метод sort мы рассмотрим в следующем разделе.
Вот один из способов реализации xfrm:
HString *
local_LocalString_xfrm(
    struct Hlocal_LocalString *this_h,
    Hjava_lang_String *str_h)
{
    Hjava_lang_String *retval;
    char *str, *xfrm;
    size_t xfrm_len;

    set_locale();
    str = allocString(str_h);
    if (str == Null)
        return NULL; /* функция allocString() вызвала SignalError */
    xfrm_len = strxfrm(Null, str, 0);
    if ((xfrm = (char *)malloc(xfrm_len + 1)) == NULL) {
          SignalError(EE(),
               "java/lang/OutOfMemorytException", NULL);
          return NULL;
     }
     strxfrm(xfrm, str, xfrm_len);
     retval = makeJavaString(xfrm, xfrm_len);
     free(xfrm);
     free(str);
     return retval;
}
```

Первое, что мы должны сделать - это настроить локальный контекст функцией set_locale:

```
#include <<locale.1>>

void
set_locale()
{
     static int done_set = 0;

     if (!done_set) {
          setlocale(LC_COLLATE, "");
          done_set++;
     }
}
```

Функция *setlocale* устанавливает алгоритм сравнения строк для локального контекста, заданного переменными среды, на что указывает параметр "". После выхода из *set_locale* мы выделяем место под новую строку языка C, содержимое которой совпадает с содержимым передаваемого параметра, и проверяем возможные ошибки. Далее мы используем вариант функции *strxfrm*, который возвращает количество символов для порядковой строки, выделяем буфер, рассчитанный на данное количество символов плюс один символ для нуль-байта, и заполняем буфер функцией *strxfrm*. Затем мы вызываем *makeJavaString*, чтобы создать новый объект *String*, содержащий порядковую строку. Перед тем, как возвращать ее, необходимо освободить выделенную память. Наконец, мы возвращаем объект *String*, содержащий порядковую строку.

Ничто не может помешать программе на C модифицировать символы в структуре *Classjava_lang_String*, соответствующей объекту *String* языка Java. Тем не менее, это нарушает гарантию того, что объекты *String* доступны только для чтения, на которую часто полагаются runtime-система Java, сгенерированный программный код и классы. Например, два объекта *String* с одинаковым содержимым часто могут совместно использовать одну и ту же область памяти. Модификация одного объекта *String* не только нарушает гарантию — она может привести к одновременной модификации других объектов *String*.

Упражнение А.3

Напишите родной метод, который выводит содержимое объекта *String*.

Упражнение А.4

Напишите родной метод, который возвращает системную строку, недоступную классам Java — например, имя текущего рабочего каталога или папки.

Упражнение А.5

Напишите класс, который использует родные методы для того, чтобы предоставить программам на Java доступ к какой-нибудь библиотеке вашей системы.

А.5 Массивы

Массивы в Java являются типизированными — они могут состоять из значений примитивного типа (массив *int*) или из объектов класса. Тип массива Java учитывается при его переводе в C. Существуют специальные типы массивов для примитивных значений и универсальный тип для массивов, содержащих объекты. Каждый массив в языке C представлен структурой следующего вида:

```
typedef struct {
     CType *body;
} ArrayOfJavaType;
```

В каждой структуре имеется поле с именем *body*, указывающее на элементы массива; *CType* — тип языка C, которому соответствует тип элементов массива, а *JavaType* — имя типа в языке Java. В таблице показано, как происходит перевод различных массивов из Java в C:

Тип массива в Java	Имя структуры	Тип *body*	Тип размещаемого массива в C
boolean	ArrayOfInt	long	T_BOOLEAN
byte	ArrayOfByte	char	T_BYTE
short	ArrayOfShort	short	T_SHORT
int	ArrayOfInt	long	T_INT
long	ArrayOfLong	int64_t	T_LONG
float	ArrayOfFloat	float	T_FLOAT
double	ArrayOfDouble	double	T_DOUBLE
char	ArrayOfChar	unicode	T_CHAR
Object	ArrayOfObject	Hobject	T_CLASS

Доступ к элементам массива осуществляется стандартным образом, в виде *body[i]*; максимальный индекс на единицу меньше количества элементов в массиве. Функция *obj_Length* возвращает количество элементов в заданном массиве.

Для создания массива используется функция *ArrayAlloc*:

Handle ****ArrayAlloc****(int type, int size)*

> Создает новый массив заданного типа *type*, который должен быть одним из типов в приведенной выше таблице. Если параметр *type* равен *T_CLASS*, создается один дополнительный элемент, указывающий на объект класса, соответствующего типу элементов массива.

Приведем в качестве примера функцию, которая создает массив объектов заданного типа:

```
HArrayOfObject *
alloc_class_array(
     char *type,
     int cnt)
{
     HArrayOfObject *retval;

     retval = (HArrayOfObject *)ArrayAlloc(T_CLASS, cnt);
     if (retval == NULL) {
          SignalError(EE(),
               "java/lang/OutOfMemorytException", NULL);
          return NULL;
     }
     unhand(retval)->>body[cnt] =
          (HObject *)FindClass(EE(), type, TRUE);
     return retval;
}
```

Сначала мы пытаемся создать массив типа *T_CLASS* и проверяем, удалось ли нам это. Затем - получаем объект *Class* для заданного типа. Функция *FindClass* получает в качестве параметров среду выполнения, имя типа в виде строки языка C и логическое значение, которое определяет необходимость загрузки класса в том случае, если он не был ранее загружен. Функция *EE* возвращает текущее состояние среды выполнения.

Объект, возвращаемый функцией *FindClass*, вставляется после конца массива и используется runtime-системой для проверки того, что каждый заносимый в массив элемент относится к правильному типу. Чтобы создать массив, который может содержать любые объекты, следует воспользоваться классом *"java/lang/Object"*.

Реализация *LocalString.sort* показывает, как работает вся эта инфраструктура. Сначала давайте посмотрим, как реализована сама функция *local_LocalString_sort*:

```
#include "local_LocalString.h"
#include <<stdlib.h>>
#include ,javaString.h>>

HArrayOfString *
local_LocalString_sort(
     struct Hlocal_LocalString *this_h,
     HArrayOfString *strngs_h)
{
     ClassArrayOfString *in_strings;
     HArrayOfString *retval = NULL;
     ClassArrayOfString *retstrs;
     char **string = NULL;
     int i, str_cnt;

     if (strings_h ==NULL) { /* проверить ссылку */
          SignalError(EE(),
               "java/lang/NullPointerException", "null array");
          return NULL;
     }
     set_locale();

     in_strings = unhand(strings_h);
     str_cnt = obj_length(strings_h);
     strings = (char **)malloc(str_cnt * sizeof *strings);
     if (strings == NULL) {
          SignalError(EE(),
               "java/lang/OutOfMemorytException", NULL);
          return NULL;
     }
     for (i = 0; i << str_cnt; i++) {
          if (in_strings->>body[i] == NULL) {
               SignalError(EE(),
               "java/lang/NullPointerException",
               "Null string in array");
               goto cleanup;
          }
          strings[i] = makeCString(in_strings->>body[i]);
          if (strings[i] == NULL)
               goto cleanup; /* функция SignalError() уже вызвана */
     }
     qsort(strings, str_cnt, sizeof *strings, cmp);
     retval = (HArrayOfString *)
          alloc_class_array("java/lang/String", str_cnt);
     retstrs = unhand(retval);
     for (i = 0; i << str_cnt; i++) {
          retstrs->>body[i] =
               makeJavaString(strings[i], strlen(strings[i]));
     }

cleanup:
     free(strings);
     return retval;
}
```

Сначала мы проверяем, действительно ли был передан массив. Затем устанавливается локальный контекст, как это делалось в *LocalString.xfrm*.

Затем — создаем строковый массив, в котором будет храниться содержимое сортируемых объектов *String*. Для создания строкового массива необходимо знать, сколько строк в него входит — это число получается вызовом *obj_length* для дескриптора строкового массива. Затем мы используем функцию *malloc* для выделения памяти под указатели и проверяем возвращаемое ею значение.

Проверка ошибок чрезвычайно важна. Родные методы, не анализирующие возможные ошибки, нарушают те гарантии безопасности, которые Java предоставляет программистам. Например, если бы мы пропустили проверку равенства ссылки *null*, то при попытке использования этого указателя вместо возбуждения исключения *NullPointerException* все бы кончилось крахом программного потока, а возможно — и всего приложения.

Получив место для хранения строк языка C, мы в цикле перебираем элементы входного массива и сохраняем копии исходных строк в массиве сортировки. При этом мы не забываем проверять наличие *null*-ссылок среди этих элементов. В нашей функции была использована функция *makeCString* — главным образом для того, чтобы показать, как ей пользоваться. Кроме того, это упрощает программу, поскольку сборщик мусора будет сам уничтожать все возвращаемые строки, в том числе и при возникновении ошибок.

Теперь массив *strings* содержит эквиваленты исходных строк в языке C. Мы вызываем стандартную библиотечную функцию C с именем *qsort*, чтобы отсортировать массив, и передаем ей в качестве последнего аргумента функцию (в данном случае — *cmp*):

```
static int
cmp(const void *str1, const void *str2)
{
        return strcoll(*(char **)str1, *(char **)str2);
}
```

Функция сравнения *cmp* преобразует свои параметры-указатели к типу *char *** (*qsort* требует, чтобы параметры имели тип *void **; предполагается, что программист сам произведет все необходимые приведения типов). Затем вызывается стандартная библиотечная функция C с именем *strcoll*, которая сравнивает две строки с учетом локального контекста. Эта функция возвращает отрицательное, равное нулю или положительное число, если первая строка соответственно меньше, равна или больше второй в локальном языковом контексте упорядочения строк. То же самое функция *qsort* ожидает от своей функции сравнения, так что значение, возвращаемое *strcoll*, может возвращаться и самой функцией *cmp*.

После выполнения *qsort*, строки оказываются отсортированными в соответствии с функцией *strcoll*. Все, что остается — построить новый массив объектов *String* и заполнить его результатами. Далее, строится массив с помощью рассмотренной выше функции *alloc_class_array* и затем в цикле вызывается *makeJavaString* для задания каждого объекта *String*. Наконец, мы освобождаем массив *strings*, созданный функцией *malloc*, и возвращаем результат.

Упражнение А.6

Модифицируйте класс *LocalString*, чтобы он мог работать с объектами, каждый из которых обладает собственным локальным контекстом, вместо того, чтобы полагаться на один общий контекст. Методы перестанут быть статическими, и понадобится новое строковое поле, определяющее контекст. Помните, что функция POSIX с именем *setlocale* устанавливает локальный контекст лишь до следующего вызова *setlocale*.

А.6 Создание объектов

Вы можете создавать объекты Java внутри реализаций родных методов с помощью функции *execute_java_constructor*:

HObject ****execute_java_constructor****(ExecEnv *ee, char *className, ClassClass *classObj, char *signature, ...)*

> Создает новый объект указанного типа, задаваемого одним из двух параметров *className* или *ClassObj* (не используемый параметр должен быть равен *NULL*). Для создания объекта вызывается конструктор, описываемый строкой *signature*. За параметром *signature* следуют параметры конструктора.

Например, создание нового объекта типа *Simple* с помощью безаргументного конструктора класса происходит следующим образом:

execute_java_contructor(NULL, "Simple", NULL, "()")

В данном случае сигнатура конструктора выглядит тривиально. Чтобы воспользоваться конструктором, который получает один или несколько параметров, необходимо включить их типы в сигнатуру, и поместить их значения в нужном порядке после строки сигнатуры. Типы в строке сигнатуры аналогичны тем, которые возвращаются *Class.getName*, и используют односимвольные сокращения для примитивных типов. Применяются следующие сокращения:

Z	boolean
I	int
C	char
J	long
B	byte
F	float
S	short
D	double

Чтобы избежать конфликтов между именами классов/интерфейсов и этими буквами, типы объектов получают имена вида " *Ltype*", где *type* — полное имя класса или интерфейса, в котором разделители-точки заменяются косой чертой, а в конце ставится точка с запятой. Например, параметр, представляющий собой объект *String*, будет выглядеть как " *Ljava/lang/String;*" . Для массивов указывается тип массива с префиксом *[*; так, массив значений типа *long* будет иметь тип " *[J*" . В многомерных типах используются несколько квадратных скобок. Массив *String[][]* будет выглядеть как " *[[Ljava/labg/String;*" .

Если вам все же непонятно, как указать конкретный тип в такой строке, вы можете написать класс Java, создающий объект нужного типа, и вызвать для него метод *getClass().getName()*. Затем, если полученное имя является именем типа, замените все точки на */*, поставьте *L* спереди и *;* сзади, если эти символы отсутствуют.

Аргументы, которые представляют собой объекты, передаются в виде указателей на дескрипторы.

Ниже показано, как происходит создание двух объектов *Attr*; первый из них использует конструктор с одним аргументом, которому передается только имя, а второй — конструктор с двумя аргументами, которому сообщается исходное значение.

```
oneArgAttr = (struct HAttr *)
        execute_java_constructor(EE(), "Attr", NULL,
                "(Ljava/lang/String;)", attrStr);

twoArgAttr = (struct HAttr *)
        execute_java_constructor(EE(), "Attr", NULL,
                "(Ljava/lang/String;Ljava/lang/Object;)",
                attrStr, attrStr);
```

Точка с запятой выполняет функцию терминатора (завершающего символа) типа, а не разделителя параметров. Конструктор, получающий два параметра типа *long* и два параметра типа *double*, описывается строкой " *(JJDD)*" .

А.7 Вызов методов Java

Вызов методов Java из программ на C напоминает вызов конструкторов Java. Для этого используются следующие основные функции:

long ****execute_java_static_method****(ExecEnv *ee, ClassClass *cb, char *method_name, char *signature, ...)*

> Выполняет статический метод класса, описываемого параметром *cb*.

long ****execute_java_dynamic_method****(ExecEnv *ee, HObject *obj, char *method_name, char *signature, ...)*

> Выполняет нестатический (динамический) метод для заданного объекта.

В обоих функциях, параметр *method_name* является именем вызываемого метода, а *signature* описывает передаваемые аргументы. В отличие от конструкторов, вы также должны объявить возвращаемый методом тип после закрывающей скобки в сигнатуре. Возвращаемый тип указывается с использованием тех же сокращений, что и для типа параметров, с дополнительной буквой *V* для *void*. Примеры приведены ниже. Вы должны сами привести тип *long* к возвращаемому типу метода или проигнорировать его для методов типа *void*.

Для получения структуры класса, используемой при вызове статических методов, применяется одна из двух функций *FindClass*:

ClassClass ****FindClass****(ExecEnv *ee, char *class_name, bool_t resolve)*

> Возвращает указатель на структуру *ClassClass* для заданного класса. Как и прежде, параметр типа *ExecEnv* следует получить от функции *EE*. Логическая величина *resolve* аналогична одноименному параметру, используемому методом *ClassLoader.loadClass* в разделе 13.2.

ClassClass ****FindClassFromClass****(ExecEnv *ee, char *class_name, bool_t resolve, ClassClass *from)*

> Возвращает указатель на объект-класс для заданного класса с использованием объекта *ClassLoader* класса *from*.

Приведем пример, в котором метод *System.out.println()* вызывается для вывода сведений о работе родного метода. Статический родной метод *grindAway* класса *Crunch*, приведенный ниже, осуществляет некоторые трудоемкие вычисления и возвращает результат типа *double*:

```
double
Crunch_grindAway(struct HCrunch *this_h)
{
        ClassClass *myClass;
        HObject *out;
        long i;
        double result;
        ExecEnv *ee = EE(); /* используется в нескольких местах */

        myClass = FindClass(ee, "Crunch", TRUE);
        out = (HObject *)execute_java_static_method(ee, myClass,
                "outStream", "()Ljava/io/PrintStream;");
        if (exceptionOccurred(ee))
                return 0.0;
        for (i = 0; i << NUM_PASSES; i++) {
                execute_java_dynamic_method(ee, out,
                        "println", "(I)V", i);
                if (exceptionOccurred(ee))
                        return 0.0;  // необходимо что-то вернуть
                /* .. вычисления ... */
        }
        return result;
}
```

Во фрагменте программы, предшествующем циклу, мы получаем дескриптор объекта *java.io.PrintStream* для *System.out*. Мы не можем непосредственно использовать значение статического поля *System.out*, потому что статические поля не имеют своего представления в родных методах; из-за этого приходится прибегать к обходным маневрам. В данном случае мы создаем статический метод, возвращающий нужное значение, и вызываем его из родного метода:

```
public static java.io.PrintStream outStream() {
        return System.out;
}
```

Родной код должен получить указатель на структуру *ClassClass* для класса *Crunch*, поэтому мы вызываем *FindClass*. После этого можно вызвать функцию *execute_java_static_method* с указанием имени вызываемого метода и его сигнатуры, включающей тип возвращаемого значения. Мы преобразуем возвращаемое значение *long* к типу, необходимому для конкретного метода. В данном случае нам требуется общий дескриптор *HObject*, а не дескриптор для конкретного типа *java.io.PrintStream*.

После вызова метода проверяется, не было ли возбуждено исключение. Это стоит делать после каждого вызова метода или конструктора Java из родного кода, поскольку в противном случае вы можете пропустить исключение и иметь неприятности в будущем. Если мы выходим из функции по причине возникновения исключения, то возвращается фиктивное значение, которое игнорируется программой.

После того, как получен доступ к выходному потоку, можно начинать циклические вычисления. При проходе каждого цикла его номер выводится методом *System.out.println*. Для вызова этого метода мы используем функция *execute_java_dynamic_method*, передавая ей в качестве параметров объект, для которого вызывается метод, имя метода, его сигнатуру и аргументы. Нам нужна версия *println*, получающая аргумент типа *int*; этот метод имеет тип *void*, поэтому мы используем сигнатуру " *(I)V"* и передаем целое число, которое нужно вывести (*i*) после аргумента-сигнатуры. И снова при возврате из метода необходимо проверить, не возбуждено ли исключение.

Для многих типов, возвращаемых методами Java, приведение *long* к нужному типу происходит элементарно. Однако типы *double* и *long* в Java являются 64-разрядными, тогда как в большинстве существующих компиляторов C тип *long* 32-разрядный, и поэтому возвращаемое значение будет иметь только половинную длину. Хотя существуют различные способы получения всех 64 бит возвращаемого значения, о них не говорится в этой книге из-за их машинной зависимости.

А.8 Последнее предупреждение

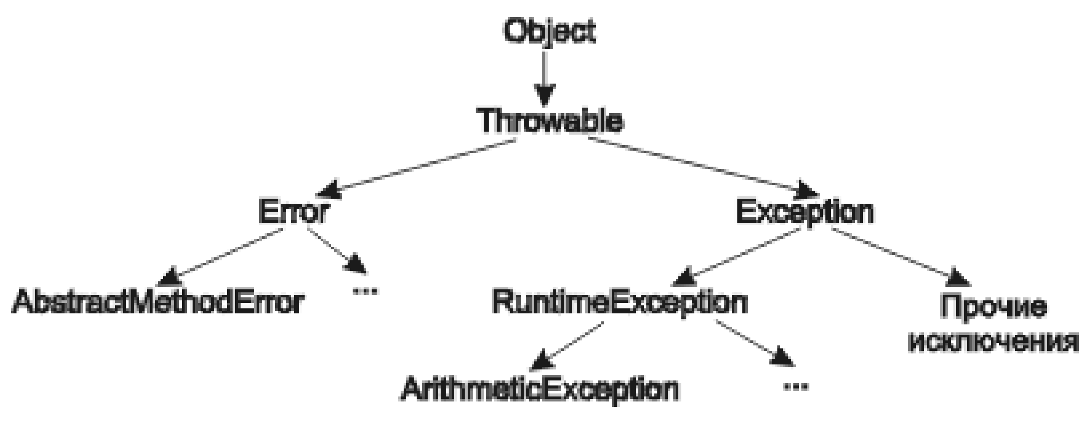
Мы должны снова предупредить вас о том, что конкретная схема стыковки, описанная здесь, в будущем обязательно изменится. Улучшения могут произойти как в плане реализации, так и на концептуальном уровне. Схема стыковки с C++ будет обладать другими характеристиками, и, возможно, повлечет за собой изменения в схеме стыковки с C, сохраняя, однако, совместимость. Кроме того, создатели будущих сред разработки могут вообще отказаться от использования всех принципов, примененных в данной схеме. Мы надеемся, что в любом случае приведенный здесь материал поможет вам понять некоторые общие аспекты, возникающие при стыковке различных языков программирования, и освоить схему связывания родных методов, которая будет использоваться в вашей системе.

Содержание | Далее

Приложение Б
Runtime-исключения в Java

Компьютер не бывает эмоциональным. Он может дать точное математическое описание, но забудет об интонации.
Фрэнк Заппа

Runtime-система Java возбуждает исключения двух основных видов: runtime-исключения, расширяющие класс *RuntimeException*, и ошибки, которые расширяют класс *Error*. Исключения обоих видов являются непроверяемыми (см. раздел 7.3). Верхняя часть иерархии исключений выглядит следующим образом:



Исключения *Error* сигнализируют об очень серьезных проблемах, после которых программа обычно завершается, и которые никогда (или почти никогда) не должны перехватываться. Исключения *Error* не являются расширениями *RuntimeException*, так что программист, пытающийся написать универсальное условие *catch* для перехвата всех исключений *Exception* и *RuntimeException* (обычно делать этого не следует) не сможет перехватить исключения *Error*. Разумеется, после возникновения любого исключения будут выполнены условия *finally* операторов *try*, так как все исключения, в том числе и *Error,* просматривают стек вызовов. Следовательно, вы всегда сможете выполнить необходимые завершающие действия.

Программист может самостоятельно расширить классы *RuntimeException* и *Error*, чтобы создать свои собственные варианты непроверяемых исключений — то есть таких исключений, которые можно возбуждать без указания их в условии *throws*. Мы сообщаем об этом по единственной причине — чтобы вы знали, что этого делать не следует. Условие *throws* предусмотрено именно для того, чтобы при вызове метода были видны все возможные аспекты его поведения. Порождая свое исключение от *RuntimeException* или *Error*, вы сообщаете о нем ложные сведения (будто оно запускается runtime-системой). Кроме того, другие разработчики, читающие вашу программу, полагают, что условие *throws* дает им информацию о возможном поведении вашего метода; вы нарушаете это предположение.

Даже если вы пишете программу для своего собственного использования, не стоит создавать непроверяемые исключения: программисты, не полностью понимающие работу вашего кода, могут упустить нечто важное. Кроме того, вероятно, что через несколько месяцев после написания программы именно *вы* окажетесь тем человеком, который будет изменять ее без полного понимания происходящего. Одно из правил создания понятных программ — считать классы *RuntimeException* и *Error* нерасширяемыми.

Все классы *Error* и *RuntimeException* содержат по меньшей мере два конструктора: один вызывается без аргументов, а второй получает объект *String* с описанием. Исключения, которые прямо или косвенно расширяют *RuntimeException* или *Error*, не объявляются в условии *throws*, поскольку они могут произойти в любой момент, что делает их объявление излишним.

Исключение *CloneNotSupportedException* непосредственно порождается от класса *Exception*, поскольку каждая программа, которая вызывает метод *clone*, возбуждающий данное исключение, должна явным образом его обработать. Оно рассмотрено в разделе “Дублирование объектов.

Настоящая глава делится на две части — одна посвящена классам *RuntimeException*, а другая — классам *Error*. Для каждого исключения приводится его значение, описание ситуации, в которой оно возбуждается, а также все дополнительные конструкторы.

Б.1 Классы *RuntimeException*

ArithmeticException extends RuntimeException

Возникла исключительная ситуация во время вычислений (например, деление целого числа на ноль).

ArrayStoreException *extends RuntimeException*

Попытка сохранения в массиве объекта неверного типа.

ClassCastException *extends RuntimeException*

Попытка недопустимого приведения типа.

IllegalArgumentException *extends RuntimeException*

Метод получил неверный аргумент (например, метод *String.equals* вызван для объекта, который не относится к типу *String*).

IllegalMonitorStateException *extends RuntimeException*

Механизм *wait/notify* использован за пределами синхронного кода.

IllegalThreadStateException *extends IllegalArgumentException*

Состояние потока не допускает выполнения требуемой операции.

IndexOutOfBoundsException *extends RuntimeException*

Runtime-система генерирует это исключение при выходе индекса массива или объекта *String* за пределы диапазона допустимых значений.

NegativeArraySizeException *extends RuntimeException*

Попытка создания массива отрицательного размера.

NullPointerException *extends RuntimeException*

Для доступа к полю или методу использована *null*-ссылка. Это же исключение сигнализирует о передаче методу параметра *null*, если для данного параметра это значение является недопустимым. Используется аналогично *IllegalArgumentException*.

NumberFormatException *extends IllegalArgumentException*

Неверное содержимое строки, в которой должно было находиться число. Исключение возбуждается такими методами, как *Integer.parseInt*.

SecurityException *extends RuntimeException*

Попытка выполнения действия, запрещенного системой безопасности — обычно объектом *SecurityManager* для текущего runtime-контекста.

Б.2 Классы Error

AbstractMethodError extends IncompatibleClassChangeError

Вызван абстрактный метод. Это может произойти лишь в очень редких случаях, так что вы никогда не столкнетесь с этим исключением.

ClassFormatError *extends LinkageError*

Загружаемый класс или интерфейс имеет неверный формат (обычно это связано с использованием “преобразованных” (mangled) имен).

IllegalAccessError *extends IncompatibleClassChangeError*

Исключение неразрешенного доступа.

IncompatibleClassChangeError *extends linkageError*

При загрузке класса или интерфейса было обнаружено изменение, несовместимое с информацией об этом классе или интерфейсе. Например, в период времени между компиляцией класса и компиляцией использующей его программы, из класса был удален незакрытый метод.

InstantiationError *extends IncompatibleClassChangeError*

Интерпретатор попытался создать объект абстрактного класса или интерфейса.

InternalError *extends VirtualMachineError*

Произошел внутренний сбой runtime-системы. В нормальных условиях такая ошибка не должна возникнуть.

LinkageError *extends Error*

Исключения класса *LinkageError* и его подклассов означают, что класс тем или иным образом зависит от другого класса и что связь между ними не может быть установлена.

NoClassDefFoundError *extends LinkageError*

Нужный класс не найден.

NoSuchFieldError *extends IncompatibleClassChangeError*

Поле отсутствует в классе или интерфейсе.

NoSuchMethodError *extends IncompatibleClassChangeError*

Метод отсутствует в классе или интерфейсе.

OutOfMemoryError *extends VirtualMachineError*

Нехватка памяти.

StackOverflowError *extends VirtualMachineError*

Переполнение стека. Может свидетельствовать о бесконечной рекурсии.

ThreadDeath *extends Error*

Исключение *ThreadDeath* возбуждается потоком-“жертвой” при его уничтожении методом *thread.stop*. Если исключение *ThreadDeath* перехватывается, его необходимо возбудить повторно, чтобы поток был уничтожен. Если *ThreadDeath* не перехватывается, то обработчик ошибок верхнего уровня не выводит никаких сообщений.

UnknownError *extends VirtualMachineError*

Произошла неизвестная, но серьезная ошибка.

UnsatisfiedLinkError *extends LinkageError*

Ошибка связывания внутри родного метода. Обычно это означает, что библиотека, реализующая родной метод, содержит неопределенные символы, которые не были найдены ни в одной библиотеке.

VerifyError *extends LinkageError*

Произошла ошибка верификации — то есть во время загрузки класс не прошел проверку, в ходе которой обычно выясняется не нарушает ли класс каких-нибудь требований безопасности Java.

VirtualMachineError *extends Error*

Нарушена работа виртуальной машины, или наблюдается нехватка ресурсов.

Содержание | Далее

Приложение B
Полезные таблицы

Таблица 1. Ключевые слова

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

Ключевые слова, помеченные символом ‹, в настоящее время не используются

Таблица 2. Специальные символы, содержащие \

Последовательность	Значение
\n	переход на новую строку (\u000A)
\t	табуляция (\u0009)
\b	забой (\u0008)
\r	ввод (\u000D)
\f	подача листа (\u000C)
\\\\	обратная косая черта (\u005C)
\\'	апостроф (\u0027)
\\"	кавычка (\u0022)
\ddd	символ в восьмеричном представлении, где каждое *d* соответствует восьмеричной цифре от 0 до 7
\uddd	символ Unicode, где каждое *d* соответствует шестнадцатеричной цифре (*0–9, a–f, A–F*)

Таблица 3. Приоритет операторов

постфиксные операторы	[] . (параметры) expr++ expr--
унарные операторы	++expr --expr +expr -expr ~ !
создание и преобразование типа	new (тип)expr
операторы умножения/деления	* / %
операторы сложения/вычитания	+ -
операторы сдвига	<<<< >>>> >>>>>>
операторы отношения	<< >> >>= <<= instanceof
операторы равенства	== !=
поразрядное И	&
поразрядное исключающее ИЛИ	^
поразрядное включающее ИЛИ	|
логическое И	&&
логическое ИЛИ	||
условный оператор	?:
операторы присвоения	= += -= *= /= %= >>>>= <<<<= >>>>>>= &= ^= | =

Таблица 4. Цифры Unicode

Unicode	Описание
\u0030–\u0039	Цифры ISO-latin-1 (и ASCII)
\u0660–\u0669	Арабско-индийские цифры
\u06f0–\u06f9	Восточные арабско-индийские цифры
\u0966–\u096f	Цифры деванагари
\u09e6–\u09ef	Цифры бенгали
\u0a66–\u0a6f	Цифры гурмукхи
\u0ae6–\u0aef	Цифры гуджарати
\u0b66–\u0b6f	Цифры ория
\u0be7–\u0bef	Тамильские цифры (только девять — без нуля)
\u0c66–\u0c6f	Цифры телугу
\u0ce6–\u0cef	Цифры каннада
\u0d66–\u0d6f	Малайские цифры
\u0e50–\u0e59	Тайские цифры
\u0ed0–\u0ed9	Цифры лао
\uff10–\uff19	Цифры полной ширины

Таблица 5. Буквы и цифры Unicode

\u0041–\u005a	Буквы верхнего регистра ISO-latin-1 и ASCII ('*A*'–'*Z*')
\u0061–\u007a	Буквы нижнего регистра ISO-latin-1 и ASCII ('*a*'–'*z*')
\u00c0–\u00d6	Дополнительные буквы ISO-latin-1
\u00d8–\u00f6	Дополнительные буквы ISO-latin-1
\u00f8–\u00ff	Дополнительные буквы ISO-latin-1
\u0100–\u1fff	Расширенная кодировка Latin-A, расширенная кодировка Latin-B, расширения IPA, буквы-модификаторы интервалов, диакритические знаки, базовый греческий алфавит, греческий и коптский алфавиты, кириллица, армянский, иврит расширенный-A, базовый иврит, иврит расширенный-B, базовый арабский, расширенный арабский, деванагари, бенгали, гурмукхи, гуджарати, ория, тамильский, телугу, каннада, малайский, тайский, лао, базовый грузинский, расширенный грузинский, хангульский, латинский расширенный дополнительный, греческий расширенный
\u3040–\u9fff	Хирагана, катакана, бопомофо, хангульский совместимый, CJK, символы и месяцы CJK, CJK совместимый, хангульский, хангульский дополнительный-A, хангульский дополнительный-B, единые идеографы CJK
\uf900–\ufdff	Совместимые идеографы CJK, алфавитные формы, арабские презентационные формы-A
\ufe70–\ufefe	Арабские презентационные формы-B
\uff10–\uff19	Цифры полной ширины
\uff21–\uff3a	Латинский полной ширины, верхний регистр
\uff41–\uff5a	Латинский полной ширины, нижний регистр
\uff66–\uffdc	Катакана и хангульский половинной ширины

Примечание: Символ Unicode является буквой или цифрой, если он принадлежит одному из диапазонов, содержащихся в таблице, и также определен как символ Unicode.

Примечание: Символ Unicode является буквой, если он присутствует в таблице “Буквы и цифры Unicode”, но отсутствует в таблице “Цифры Unicode”.

Таблица 6. Java 1.0 и Java 1.0.2: Отличия между Java 1.0 и Java 1.0.2, существенные для данной книги (с разделами, к которым они относятся)

- Константы *MIN_VALUE* и *MAX_VALUE* класса *Character* в Java 1.0 ошибочно присутствовали в классе *Boolean*. См. раздел 13.5.
- В классах *String* и *Character* в Java 1.0 некоторые свойства символов (принадлежность к верхнему/нижнему регистру, цифрам и т.д.) определялись только для подмножества символов Unicode, принадлежащего к набору ISO-Latin-1 (с *\u0000* по *\u00ff*); все символы за пределами этого диапазона считались буквами без регистра. Кроме того, отсутствовали методы класса *Character*, возвращавшие сведение о классе символа помимо принадлежности к верхнему или нижнему регистру (например, методы заглавного регистра и *isLetter*). См. раздел 8.2, раздел 8.4 и раздел 13.5.
- Java 1.0 не гарантирует, что литералам *String* с одинаковыми значениями соответствуют одинаковые ссылки, хотя иногда это было так. См. раздел 8.2.
- Список букв и цифр, используемых в идентификаторах Java 1.0, несколько отличается от списка Java 1.0.2 за пределами диапазона ISO-Latin-1. См. табл. 4 и табл. 5.
- В Java разрешалась (и даже ошибочно наделялась смыслом) комбинация ключевых слов *private protected*.
- Классы-оболочки *Integer* и *Long* в Java 1.0 не содержали методов *toHexString*, *toOctalString* и *toBinaryString*. См. раздел 13.7 и раздел 13.8.

Содержание

Глава 3
РАСШИРЕНИЕ КЛАССОВ

Вы поймете меня, если я скажу, что могу проследить свою родословную вплоть до частиц первичной протоплазмы.
Гильберт и Салливан, The Mikado

Во время экскурсии мы кратко познакомились с тем, как происходит *расширение*, или *субклассирование*, благодаря которому расширенный класс может использоваться вместо исходного. Такая возможность носит название *полиморфизма* — это означает, что объект данного класса может выступать в нескольких видах, либо самостоятельно, либо в качестве объекта расширяемого им суперкласса. Класс по отношению к расширяемому им классу называется *подклассом*, или *расширенным классом*; расширяемый класс, в свою очередь, называется *суперклассом*.

Набор членов класса (методов и полей), доступных за его пределами, в совокупности с их ожидаемым поведением часто именуется контрактом класса. Контракт — это то, что должен делать класс в соответствии с обещаниями его разработчика. Каждый раз, когда вы расширяете класс и добавляете в него новые функции, вы тем самым создаете новый класс с расширенным контрактом. Тем не менее к модификации части контракта, *унаследованной* от расширяемого класса, следует подходить с осторожностью; можно улучшить способ реализации контракта, однако изменения не должны нарушать контракт суперкласса.

3.1. Расширенный класс

Каждый класс из тех, что встречались нам в этой книге, является расширенным, независимо от того, объявлен ли он с ключевым словом extends или нет. Даже такие классы, как Body, которые вроде бы не расширяют других классов, на самом деле неявно происходят от принадлежащего Java класса Object. Другими словами, Object находится в корне всей иерархии классов. В нем объявляются методы, которые реализованы во всех объектах. Переменные типа Object могут ссылаться на любые объекты, будь это экземпляры класса или массивы.

Например, при разработке класса для списка, элементы которого могут быть объектами произвольного типа, можно предусмотреть в нем поле типа Object. В такой список не могут входить значения примитивных типов (int, boolean и т. д.), однако при необходимости можно создать объекты этих типов с помощью классов-оболочек (Integer, Boolean и т. д.), описанных в главе 13.

Для демонстрации работы с подклассами начнем с базового класса для хранения атрибутов, представленных в виде пар имя/значение. Имена атрибутов являются строками (например, “цвет” или “расположение”). Атрибуты могут иметь произвольный тип, поэтому их значения хранятся в переменных типа Object:

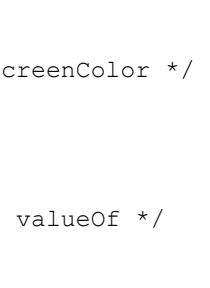
```
class Attr {
        private String name;
        private Object value = null;

        public Attr(String nameOf) {
                name = nameOf;
        }

        public Attr(String nameOf, Object valueOf) {
                name = nameOf;
                value = valueOf;
        }

        public String nameOf() {
                return name;
        }

        public Object valueOf() {
                return value;
        }

        public Object valueOf(Object newValue) {
                Object oldVal = value;
                value = newValue;
                return oldVal;
        }
}
```

Каждому атрибуту обязательно должно быть присвоено имя, поэтому при вызове конструкторов Attr им передается параметр-строка. Имя должно быть доступно только для чтения, поскольку оно может применяться, скажем, в качестве ключа хеш-таблицы или сортированного списка. Если при этом имя атрибута будет изменено за пределами класса, то объект-атрибут “потеряется”, так как в таблице или списке он будет храниться под старым именем. Значение атрибута можно менять в любой момент.

Следующий класс расширяет понятие атрибута для хранения цветовых атрибутов, которые могут быть строками, служащими для именования или описания цветов. Описания цветов могут представлять собой как названия (“красный” или “бежевый”), по которым необходимо найти нужное значение в таблице, так и числовые значения, которые могут преобразовываться в стандартное, более эффективное представление ScreenColor (которое, как предполагается, определено в другом месте программы). Преобразование описания в объект ScreenColor сопряжено с большими накладными расходами, так что для каждого атрибута эту операцию желательно выполнять только один раз. Для этого мы расширяем класс Attr и создаем на его основе класс ColorAttr, включающий специальный метод для получения преобразованного объекта ScreenColor. Данный метод реализован так, что преобразование выполняется всего один раз:

```
class ColorAttr extends Attr {
        private ScreenColor myColor; // преобразованный цвет

        public ColorAttr(String name, Object value) {
                super(name, value);
                decodeColor();
        }

        public ColorAttr(String name) {
                this(name, “transparent”);
        }

        public ColorAttr(String name, ScreenColor value) {
                super(name, value.toString());
                myColor = value;
        }

        public Object valueOf(Object newValue) {
                // сначала выполнить метод valueOf() суперкласса
                Object retval = super.valueOf(newValue);
                decodeColor();
                return retval;
        }

        /** Присвоить атрибуту ScreenColor значение,
            а не описание */
        public ScreenColor valueOf(ScreenColor newValue) {
                // сначала выполнить метод valueOf() суперкласса
                super.valueOf(newValue.toString());
                myColor = newValue;
                return newValue;
        }

        /** Вернуть преобразованный объект ScreenColor */
        public ScreenColor color() {
                return myColor;
        }

        /** Задать ScreenColor по описанию в valueOf */
        protected void decodeColor() {
                if (valueOf() == null)
                        myColor = null;
                else
                        myColor = new ScreenColor(valueOf());
        }
}
```

Сначала мы создаем новый класс ColorAttr, расширяющий класс Attr. Основные функции класса ColorAttr те же, что и у класса Attr, но к ним добавляется несколько новых. Класс Attr является суперклассом по отношению к ColorAttr, а ColorAttr — подклассом по отношению к Attr. *Иерархия классов* для нашего примера выглядит следующим образом (суперкласс на диаграмме расположен выше своего подкласса):



Расширенный класс ColorAttr выполняет три основные функции:

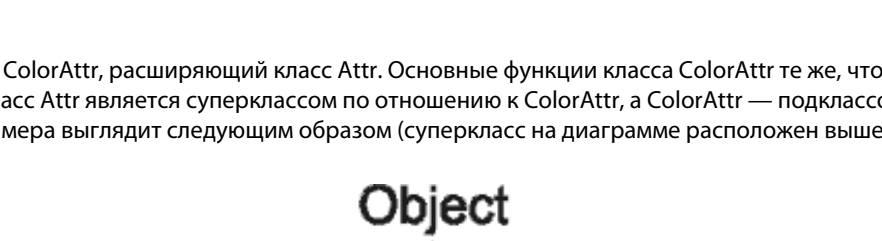
- Он обеспечивает наличие трех конструкторов: два из них копируют поведение суперкласса, а третий позволяет сразу передать объект Screen Color.
- Он перегружает и переопределяет метод valueOf базового класса, чтобы при изменении значения атрибута создался объект ScreenColor.
- Он включает новый метод color для возврата цветового описания, преобразованного в объект ScreenColor.

Упражнение 3.1

На основе класса Vehicle из главы 2 создайте расширенный класс с именем PassengerVehicle и наделите его возможностью определения числа свободных и занятых мест в машине. Включите в PassengerVehicle новый метод main, который создает несколько объектов и выводит их.

3.2. Истинное значение protected

Ранее мы кратко упомянули о том, что объявление члена класса защищенным (то есть с ключевым словом protected) означает возможность обращения к нему из классов, расширяющих данный, — однако этому замечанию не хватает формальной четкости. Выражаясь более точно, к защищенному члену класса можно обращаться через ссылку на объект, относящийся по меньшей мере к тому же типу, что и класс. Пример поможет нам разобраться с этим утверждением. Предположим, имеется следующая иерархия классов:



Поле calories в классе Dessert является защищенным. Каждый класс, расширяющий Dessert, наследует от него поле calories. Тем не менее код класса Cake может осуществлять доступ к полю calories только через ссылку на тип, являющийся Cake или его подклассом (например, тип Chocolate Cake). Код класса Cake не может обращаться к полю calories через ссылку типа Scone. Такое ограничение позволяет быть уверенным в том, что доступ к protected-полям осуществляется лишь в пределах иерархии класса. Если в коде класса Cake имеется ссылка на более общий объект Dessert, вы не можете применять ее для доступа к полю calories, однако вы можете преобразовать ее в ссылку на Cake и воспользоваться результатом — при условии, конечно, что объект, на который она указывает, действительно относится к классу Cake (по меньшей мере).

Сказанное справедливо и по отношению к защищенным методам — их можно вызывать только через ссылку на тип, относящийся по меньшей мере к тому же классу.

К защищенным статическим полям и методам можно обращаться из любого расширенного класса. Если бы поле calories было статическим, то любой метод (как статический, так и нет) в Cake, ChocolateCake и Scone мог бы обращаться к нему через ссылку на любой из типов Dessert.

Члены класса, объявленные с ключевым словом protected, также оказываются доступными для любого кода, входящего в тот же пакет. Если изображенные выше классы семейства Dessert входят в один пакет, то они могут обращаться к полям calories друг друга.

3.3. Конструкторы в расширенных классах

При расширении класса необходимо выбрать один из конструкторов суперкласса и вызывать его при конструировании объектов нового класса. Это необходимо для правильного создания части объекта, относящейся к суперклассу, помимо установки правильного исходного состояния для всех добавленных полей.

Конструктор суперкласса может вызываться в конструкторе подкласса посредством явного вызова super(). Примером может служить первый из конструкторов приведенного выше класса ColorAttr. Если вызов конструктора суперкласса не является самым первым выполняемым оператором в конструкторе нового класса, то перед выполнением последнего автоматически вызывается безаргументный конструктор суперкласса. Если же суперкласс не имеет безаргументного конструктора, вы должны явно вызвать конструктор суперкласса с параметрами. Вызов super() непременно должен быть первым оператором нового конструктора.

Вызов конструктора суперкласса демонстрируется на примере первого конструктора ColorAttr. Сначала полученные имя и значение передаются конструктору суперкласса, получающему два аргумента. Затем конструктор вызывает свой собственный метод decodeColor для того, чтобы поле myColor ссылалось на нужный цветовой объект.

Вы можете временно отложить вызов конструктора суперкласса и использовать вместо него один из конструкторов того же класса — в этом случае вместо super() используется this(). Второй конструктор ColorAttr поступает именно так. Это было сделано для того, чтобы каждому цветовому атрибуту заведомо был присвоен какой-то цвет; если он не указан, то по умолчанию присваивается цвет “transparent” (то есть “прозрачный”).

Третий конструктор ColorAttr позволяет программисту при создании объекта ColorAttr сразу же указать объект ScreenColor. Первые два конструктора преобразуют свои параметры в объекты ScreenColor при помощи метода decodeColor, вызов которого сопряжен с накладными расходами. Если программист уже располагает объектом ScreenColor, который будет использован в качестве цветового значения, было бы желательно избежать затрат на излишнее преобразование. В этом примере конструктор предназначен для повышения эффективности, а не для добавления новых возможностей.

Сигнатуры конструкторов класса ColorAttr в точности совпадают с сигнатурами конструкторов суперкласса, но это ни в коем случае не является обязательным. Иногда бывает удобно сделать так, чтобы часть или все конструкторы расширенного класса передавали нужные параметры в конструкторы суперкласса и обходились без своих параметров (или число таких параметров было бы минимальным). Нередки случаи, когда сигнатуры конструкторов расширенного класса не имеют ничего общего с сигнатурами конструкторов суперкласса.

Язык Java может создать безаргументный конструктор по умолчанию. Работа такого конструктора для расширяемого класса начинается с вызова безаргументного конструктора суперкласса. Однако, если в суперклассе отсутствует безаргументный конструктор, расширенный класс должен содержать хотя бы один конструктор. Конструктор расширенного класса по умолчанию эквивалентен следующему:

```
public class ExtendedClass extends SimpleClass {
        public ExtendedClass() {
                super();
        }
}
```

Помните, что доступность конструктора определяется доступностью класса. Так как ExtendedClass объявлен как public, конструктор по умолчанию также будет public.

3.3.1. Порядок вызова конструкторов

При создании объекта всем его полям присваиваются исходные значения по умолчанию в зависимости от их типа (ноль для всех числовых типов, ‘\u0000’ для char, false для boolean и null для ссылок на объекты). Затем происходит вызов конструктора. Каждый конструктор выполняется за три фазы:

1. Вызов конструктора суперкласса.

2. Присвоение значений полям при помощи инициализаторов.

3. Выполнение тела конструктора.

Приведем пример, который позволит нам проследить за этой процедурой:

```
class X {
        protected int xMask = 0x00ff;
        protected int fullMask;

        public X() {
                fullMask = xMask;
        }

        public int mask(int orig) {
                return (orig & fullMask);
        }
}
class Y extends X {
        protected int yMask = 0xff00;

        public Y() {
                fullMask |= yMask;
        }
}
```

Если создать объект типа Y и проследить за его конструированием шаг за шагом, то значения полей будут меняться следующим образом:

Шаг	Что происходит	xMask	yMask	fullMask
0	Присвоение полям значений по умолчанию			
0				
0				
0				
1	Вызов конструктора Y	0	0	0
2	Вызов конструктора X	0	0	0
3	Инициализация полей X	0x00ff	0	0
4	Выполнение конструктора X	0x00ff	0	0x00ff
5	Инициализация полей Y	0x00ff	0xff00	0x00ff
6	Выполнение конструктора Y	0x00ff	0xff00	0xffff

Этот порядок имеет ряд важных следствий для вызова методов во время конструирования. Обращаясь к методу, вы всегда имеете дело с его реализацией для конкретного объекта; поля, используемые в нем, могут быть еще не инициализированы. Если на приведенном выше шаге 4 конструктор X вызовет метод mask, то маска будет иметь значение 0x00ff, а не 0xffff, несмотря на то что в более позднем вызове mask (после завершения конструирования объекта) будет использовано значение 0xffff.

Кроме того, представьте себе ситуацию, в которой класс Y переопределяет реализацию mask так, чтобы в вычислениях явно использовалось поле yMask. Если конструктор X использует метод mask, на самом деле будет вызван mask класса Y, а в этот момент значение yMask равно нулю вместо ожидаемого 0xff00.

Все эти факторы следует учитывать при разработке методов, вызываемых во время фазы конструирования объекта. Кроме того, вы должны тщательно документировать любые методы, вызываемые в вашем конструкторе, чтобы предупредить каждого, кто захочет переопределить такой конструктор, о возможных ограничениях.

Упражнение 3.2

Наберите код приведенных выше классов X и Y и включите в него операторы вывода для наблюдения за значениями масок. Напишите метод main и запустите его, чтобы ознакомиться с результатами. Переопределите mask в классе Y и снова выполните тестирование.

Упражнение 3.3

Если правильные значения масок оказываются абсолютно необходимыми для конструирования, какой бы выход вы предложили?

3.4. Переопределение методов и скрытие полей

В своем новом классе ColorAttr мы *переопределили* и *перегрузили* метод valueOf, устанавливающий значение атрибута:

- *Перегрузка (overloading)* метода рассматривалась нами раньше; под этим термином понимается создание нескольких методов с одинаковыми именами, но с различными сигнатурами, по которым эти методы отличаются друг от друга.
- *Переопределение (overriding)* метода означает, что реализация метода, взятая из суперкласса, заменяется вашей собственной. Сигнатуры методов при этом должны быть идентичными. Обратите внимание: переопределению подлежат только нестатические методы.

В классе ColorAttr мы переопределили метод Attr.valueOf(Object), создав новый метод ColorAttr.valueOf(Object). Этот метод сначала обращается к реализации суперкласса с помощью ключевого слова super и затем вызывает метод decodeColor. Ссылка super может использоваться для вызова методов суперкласса, переопределяемых в данном классе. Позднее мы подробно рассмотрим ссылку super.

В переопределяемом методе должны сохраняться сигнатура и тип возвращаемого значения. Связка throws переопределяющего метода может отличаться от связки throws метода суперкласса, если только в первой не объявляются какие-либо типы исключений, не входящие в исходное определение метода. В связке throws переопределяющего метода может присутствовать меньше исключений, чем в методе суперкласса. Переопределяющий метод может вообще не иметь связки throws; в таком случае исключения в нем не проверяются.

Переопределенные методы могут иметь собственные значения атрибутов доступа. Расширенный класс может изменить права доступа к методам, унаследованным из суперкласса, но лишь в том случае, если он расширяет их. Метод, объявленный в суперклассе как protected, может быть повторно заявлен как protected (вполне обычная ситуация) или public, но не как private. Ограничивать доступ к методам по сравнению с суперклассом на самом деле было бы бессмысленно, поскольку такое ограничение очень легко обойти: достаточно преобразовать ссылку в супертип с большими правами доступа и использовать ее для вызова метода.

Поля не могут переопределяться; вы можете лишь *скрыть* их. Если объявить в своем классе поле с тем же именем, что и в суперклассе, то поле суперкласса никуда не исчезнет, однако к нему уже нельзя будет обратиться напрямую, используя одно имя. Для доступа к такому полю нужно будет воспользоваться super или другой ссылкой на тип суперкласса.

При вызове метода для некоторого объекта его реализация выбирается в зависимости от *фактического типа объекта*. При доступе к полю используется *объявленный тип ссылки*. Разобраться в этом поможет следующий пример:

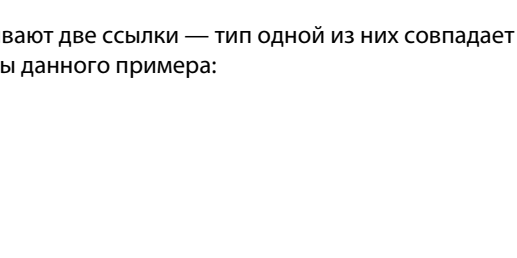
```
class SuperShow {
        public String str = “SuperStr”;

        public void show() {
                System.out.println(“Super.show: ” + str);
        }
}

class ExtendShow extends SuperShow {
        public String str = “ExtendStr”;
public void show() {
                System.out.println(“Extend.show: ” + str);
        }
        public static void main(String[] args) {
                ExtendShow ext = new ExtendShow();
                SuperShow sup = ext;
                sup.show();
                ext.show();
                System.out.println(“sup.str = ” + sup.str);
                System.out.println(“ext.str = ” + ext.str);
        }
}
```

У нас имеется всего один объект, но на него указывают две ссылки — тип одной из них совпадает с типом объекта, а другая объявлена как ссылка на суперкласс. Вот как выглядят результаты работы данного примера:

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

Метод show ведет себя именно так, как следовало ожидать: вызываемый метод зависит от фактического типа объекта, а не от типа ссылки. Когда мы имеем дело с объектом ExtendShow, вызывается метод именно этого класса, даже если доступ к нему осуществляется через ссылку на объект типа SuperShow.

Что касается поля str, то выбор класса, которому принадлежит это поле, осуществляется на основании объявленного типа *ссылки*, а не фактического типа *объекта*. В сущности, каждый объект класса ExtendShow содержит два поля типа String, каждое из которых называется str; одно из них наследуется от суперкласса и скрывается другим, собственным полем класса Extend Show:



Мы уже убедились, что переопределение методов позволяет увеличить возможности существующего кода за счет использования его с объектами, расширенные свойства которых не были предусмотрены разработчиком. Но в том, что касается скрытия полей, — довольно нелегко придумать ситуацию, при которой оно было бы полезным.

Если существующий метод получает параметр типа SuperShow и обращается к str через ссылку на объект-параметр, он всегда будет получать Super Show.str, даже если методу на самом деле был передан объект типа Extend Show. Если бы классы были спроектированы так, чтобы для доступа к строке применялся специальный метод, то в этом случае был бы вызван переопределенный метод, возвращающий ExtendShow.str. Это еще одна из причин, по которой определение классов с закрытыми данными, доступ к которым осуществляется с помощью методов, оказывается предпочтительным.

Скрытие полей разрешено в Java для того, чтобы при новой реализации существующих суперклассов можно было включить в них новые поля с атрибутами public или protected, не нарушая при этом работы подкласса. Если бы использование одинаковых имен в суперклассе и подклассе было запрещено, то включение нового поля в существующий суперкласс потенциально могло бы привести к конфликтам с подклассами, в которых это имя уже используется. В таком случае запрет на добавление новых полей к существующим суперклассам связывал бы руки программистам, которые не могли бы дополнить суперклассы новыми полями с атрибутами public или protected. Формально можно было бы возразить, что классы должны содержать только данные private, но Java поддерживает обе возможности.

3.4.1. Ключевое слово super

Ключевое слово super может использоваться во всех нестатических методах класса. При доступе к полям или вызове методов ключевое слово super представляет собой ссылку на текущий объект как экземпляр суперкласса. Использование super оказывается единственным случаем, при котором выбор реализации метода зависит от типа *ссылки*. В вызове вида super.method всегда используется реализация method из суперкласса, а не его переопределенная реализация, которая находится где-то ниже в иерархии классов.

Вызов методов с помощью ключевого слова super отличается от любых других ссылок, в которых метод выбирается в зависимости от фактического типа объекта, а не типа ссылки. При вызове метода через ссылку super вы обращаетесь к реализации метода, основанной на типе суперкласса. Приведем пример практического использования super:

```
class That {
        /** вернуть имя класса */
        protected String nm() {
                return “That”;
        }
}

class More extends That {
        protected String nm() {
                return “More”;
        }

        protected void printNM() {
                That sref = super;

                System.out.println(“this.nm() = ” + this.nm());
                System.out.println(“sref.nm() = ” + sref.nm());
                System.out.println(“super.nm() = ” + super.nm());
        }
}
А вот как выглядит результат работы printNM:
this.nm() = More
sref.nm() = More
super.nm() = That
```

Ключевое слово super может также применяться для доступа к защищенным членам суперкласса.

3.5. Объявление методов и классов с ключевым словом final

Если метод объявлен с атрибутом final, это означает, что ни один расширенный класс не сможет переопределить данный метод с целью изменить его поведение. Другими словами, данная версия метода является *окончательной*.

Подобным образом могут объявляться целые классы:

```
final class NoExtending {
        // ...
}
```

Класс, помеченный с атрибутом final, не может быть субклассирован, а все его методы также неявно являются final.

Имеется два основных довода в пользу объявления методов с атрибутом final. Первый из них — безопасность; каждый, кто пользуется классом, может быть уверен, что его поведение останется неизменным, вне зависимости от объектов, с которыми ему приходится работать.

Окончательные классы и методы повышают безопасность. Если класс является окончательным, то вы не сможете объявить расширяющий его класс и, следовательно, не сможете нарушить его контракт. Если же окончательным является метод, вы можете положиться на его реализацию (разумеется, лишь в том случае, если в нем не вызываются неокончательные методы). Например, final может использоваться для метода проверки введенного пароля validatePassword, чтобы этот метод всегда выполнял свои функции и не был переопределен с тем, чтобы при всех обстоятельствах возвращать true. Кроме того, можно пометить с атрибутом final целый класс, содержащий этот метод, чтобы запретить его расширение и избежать возможных проблем, связанных с реализацией validatePassword.

Во многих случаях уровень безопасности класса, объявленного final, может быть достигнут за счет того, что класс остается расширяемым, а каждый из его методов объявляется final. В этом случае можно быть уверенным в работе данных методов и при этом оставить возможность расширения посредством добавления новых функций без переопределения существующих методов. Разумеется, поля, на которые опираются методы final, должны быть объявлены private, иначе расширенный класс сможет все испортить за счет модификации этих полей.

Класс или метод, помеченный final, серьезно ограничивает использование данного класса. Если метод объявляется final, то вы должны быть действительно уверены в том, что его поведение ни при каких обстоятельствах не должно измениться. Вы ограничиваете гибкость класса, усложняя жизнь другим разработчикам, которые захотят воспользоваться вашим классом для повышения функциональности своих программ. Если класс помечен final, то никто не сможет расширить его, следовательно, его полезность ограничивается. Объявляя что-либо с ключевым словом final, убедитесь в том, что вытекающие из этого ограничения действительно необходимы.

Однако использование final упрощает оптимизацию программы. При вызове метода, не являющегося final, runtime-система Java определяет фактический тип объекта, связывает вызов с нужной реализацией метода для данного типа и затем вызывает данную реализацию. Но если бы, скажем, метод nameOf в классе Attr был объявлен как final и у вас имелась ссылка на объект типа Attr или любого производного от него типа, то при вызове метода можно обойтись и без всех этих действий. В простейшем случае (таком, как nameOf) вызов метода в программе можно заменить телом метода. Такой механизм известен под названием “встроенных методов” (inlining). Встроенный метод приводит к тому, что следующие два оператора становятся эквивалентными:

```
System.out.println(“id = ” + rose.name);
System.out.println(“id = ” + rose.nameOf());
```

Хотя эти два оператора приводят к одному результату, применение метода nameOf позволяет сделать поле name доступным только для чтения и предоставляет в ваше распоряжение все преимущества абстрагирования, позволяя в любой момент изменить реализацию метода.

По отношению к рассматриваемой оптимизации методы private и static эквивалентны методам final, поскольку они также не могут быть переопределены.

Некоторые проверки для классов final осуществляются быстрее. В сущности, многие из них производятся на стадии компиляции; кроме того, ошибки выявляются быстрее. Если компилятор Java имеет дело со ссылкой на класс вида final, он точно знает тип объекта, на который она указывает. Для таких классов известна вся иерархия, так что компилятор может проверить, все ли с ними в порядке. Для ссылок на объекты, не являющиеся final, многие проверки осуществляются лишь во время выполнения программы.

Упражнение 3.4

Какие из методов классов Vehicle и PassengerVehicle имеет смысл сделать final (если таковые имеются)?

3.6. Класс Object

Все классы являются явными или неявными расширениями класса Object и, таким образом, наследуют его методы. Последние делятся на две категории: общие служебные и методы, поддерживающие потоки. Работа с потоками рассматривается в главе 9. В этом разделе описываются служебные методы Object и их назначение. К категории служебных относятся следующие методы:

public boolean **equals(Object obj)**

Сравнивает объект-получатель с объектом, на который указывает ссылка obj; возвращает true, если объекты равны между собой, и false в противном случае. Если вам нужно выяснить, указывают ли две ссылки на один и тот же объект, можете сравнить их с помощью операторов == и != , а метод equals предназначен для сравнения значений. Реализация метода equals, принятая в Object по умолчанию, предполагает, что объект равен лишь самому себе.

public int **hashCode()**

Возвращает хеш-код для данного объекта. Каждому объекту может быть присвоен некоторый хеш-код, используемый при работе с хеш-таблицами. По умолчанию возвращается значение, которое является уникальным для каждого объекта. Оно используется при сохранении объектов в таблицах Hashtable, которые описаны в разделе “Класс Hashtable”.

protected Object **clone()** throws CloneNotSupportedException

Возвращает дубликат объекта. Дубликатом называется новый объект, являющийся копией объекта, для которого вызывался метод clone. Процесс дублирования объектов подробнее рассматривается ниже в этой главе, в разделе 3.8.

public final Class **getClass()**

Возвращает объект типа Class, который соответствует классу данного объекта. Во время выполнения программы на Java можно получить информацию о классе в виде объекта Class, возвращаемого методом getClass.

protected void **finalize()** throws Throwable

Завершающие операции с объектом, осуществляемые во время сборки мусора. Этот метод был подробно описан в разделе “Метод finalize”.

Методы hashCode и equals должны переопределяться, если вы хотите ввести новую концепцию равенства объектов, отличающуюся от принятой в классе Object. По умолчанию считается, что два различных объекта не равны между собой, а их хеш-коды не должны совпадать.

Если ваш класс вводит концепцию равенства, при которой два различных объекта могут считаться равными, метод hashCode должен возвращать для них одинаковые значения хеш-кода. Это происходит оттого, что механизм Hashtable полагается в своей работе на возврат методом equals значения true при нахождении в хеш-таблице элемента с тем же значением. Например, класс String переопределяет метод equals так, чтобы он возвращал значение true при совпадении содержимого двух строк. Кроме того, в этом классе переопределяется и метод hashCode — его новая версия возвращает хеш-код, вычисляемый на основании содержимого String, и две одинаковые строки имеют совпадающие значения хеш-кодов.

Упражнение 3.5

Переопределите методы equals и hashCode в классе Vehicle.

3.7. Абстрактные классы и методы

Абстрактные классы представляют собой исключительно полезную концепцию объектно-ориентированного программирования. С их помощью можно объявлять классы, реализованные лишь частично, поручив полную реализацию расширенным классам.

Абстракция оказывается полезной, когда некоторое поведение характерно для большинства или всех объектов данного класса, но некоторые аспекты имеют смысл лишь для ограниченного круга объектов, не составляющих суперкласса. В Java такие классы объявляются с ключевым словом abstract, и каждый метод, не реализованный в классе, также объявляется abstract. (Если все, что вам требуется, — это определить набор методов, которые будут где-то поддерживаться, но не предоставлять для них реализации, то вместо абстрактных классов, видимо, лучше воспользоваться интерфейсами, описанными в главе 4.)

Допустим, вы хотите создать инфраструктуру для написания кода, который будет во время выполнения программы измерять определенные показатели. Реализация такого класса может учитывать, как происходит измерение, но вы не знаете заранее, какой именно показатель будет измеряться. Большинство абстрактных классов следует именно этому принципу: для конкретной работы, выполняемой классом, необходимо, чтобы кто-то предоставил недостающую часть. В нашем примере таким недостающим звеном становится фрагмент программы, выполняющий измерения. Вот как может выглядеть соответствующий класс:

```
abstract class Benchmark {
        abstract void benchmark();

        public long repeat(int count) {
                long start = System.currentTimeMillis();
                for (int i = 0; i < count; i++)
                        benchmark();
                return (System.currentTimeMillis() — start);
        }
}
```

Класс объявлен с ключевым словом abstract, потому что так объявляется любой класс, содержащий абстрактные методы. Подобная избыточность позволяет во время чтения программы быстро понять, является ли данный класс абстрактным, и обойтись без просмотра всех методов и поиска среди них абстрактных.

Метод repeat предоставляет общую схему для проведения измерений. Он знает, как ему вызвать процедуру измерения count раз через равные интервалы времени. Если хронометраж должен быть более сложным (например, необходимо измерить время каждого запуска и вычислить статистику отклонения от среднего значения), метод можно усовершенствовать, что никак не отразится на реализации процедуры измерений в расширенном классе.

Абстрактный метод benchmark должен быть реализован в каждом подклассе, который не является абстрактным. Именно по этой причине в классе отсутствует реализация, есть лишь объявление. Приведем пример простейшего расширения класса Benchmark:

```
class MethodBenchmark extends Benchark {
        void benchmark() {
        }

        public static void main(String[] args) {
                int count = Integer.parseInt(args[0]);
                long time = new MethodBenchmark().repeat(count);
                System.out.println(count + “ methods in ” +
                                           time + “ milliseconds”);
        }
}
```

Данный класс использует benchmark для определения времени, затраченного на вызов метода. Теперь вы можете провести хронометраж, запустив приложение MethodBenchmark, сообщив ему количество повторений теста. Значение передается в программу в виде аргумента-строки, из которого оно извлекается методом parseInt класса Integer, как описано в разделе “Преобразование строк”.

Вы не можете создать объект абстрактного класса, поскольку для некоторых вызываемых методов может отсутствовать реализация.

Любой класс может переопределить методы своего суперкласса и объявить их абстрактными — конкретный метод в иерархии типов становится абстрактным. Это бывает полезно, например, когда принятая по умолчанию реализация класса не подходит на некоторых уровнях в его иерархии.

Упражнение 3.6

Напишите новый класс, который измеряет что-нибудь другое — например, время, затраченное на выполнение цикла от 0 до значения, переданного в виде параметра.

Упражнение 3.7

Измените класс Vehicle так, чтобы он содержал ссылку на объект EnergySource (источник энергии), ассоциируемый с Vehicle внутри конструктора. Класс EnergySource должен быть абстрактным, поскольку состояние заполнения для объекта GasTank (бензобак) должно отмечаться иначе, нежели для объекта Battery (аккумулятор). Включите в EnergySource абстрактный метод empty и реализуйте его в классах GasTank и Battery. Включите в Vehicle метод start, который бы проверял состояние источника энергии в начале поездки.

3.8. Дублирование объектов

Метод Object.clone помогает производить в ваших классах дублирование объектов. При дублировании возвращается новый объект, исходное состояние которого копирует состояние объекта, для которого был вызван метод clone. Все последующие изменения, вносимые в объект-дубль, не изменяют состояния исходного объекта.

При написании метода clone следует учитывать три основных момента:

- Для нормальной работы метода clone необходимо реализовать интерфейс Cloneable. /В будущих реализациях название интерфейса может быть исправлено на Clonable/
- Метод Object.clone выполняет простое дублирование, заключающееся в копировании всех полей исходного объекта в новый объект. Для многих классов такой вариант работает, но, возможно, в вашем классе его придется дополнить за счет переопределения метода (см. ниже).
- Исключение CloneNotSupportedException сигнализирует о том, что метод clone данного класса не должен вызываться.

Существует четыре варианта отношения класса к методу clone:

- Класс поддерживает clone. Такие классы реализуют Cloneable, а в объявлении метода clone обычно не указывается никаких запускаемых исключений.
- Класс условно поддерживает clone. Такой класс может представлять собой коллекцию (набор объектов), которая в принципе может дублироваться, но лишь при условии, что дублируется все ее содержимое. Такие классы реализуют Cloneable, но при этом допускают возникновение в методе clone исключения CloneNotSupportedException, которое может быть получено от других объектов при попытке их дублирования во время дублирования коллекции. Кроме того, бывает желательно разрешить возможность дублирования класса, но при этом не требовать, чтобы дублирование также поддерживалось и для всех подклассов.
- Класс разрешает поддержку clone в подклассах, но не объявляет об этом открыто. Такие классы не реализуют Cloneable, но обеспечивают реализацию clone для правильного дублирования полей, если реализация по умолчанию оказывается неправильной.
- Класс запрещает clone. Такие классы не реализуют Cloneable, а метод clone в них всегда запускает исключение CloneNotSupportedException.

Object.clone сначала проверяет, поддерживается ли интерфейс Cloneable объектом, для которого вызван метод clone; если нет — запускается исключение CloneNotSupportedException. В противном случае создается новый объект, тип которого совпадает с типом исходного, и его поля инициализируются значениями полей исходного объекта. При завершении работы Object.clone возвращает ссылку на новый объект.

Самая простая возможность создать дублируемый класс — объявить о реализации в нем интерфейса Cloneable:

```
public class MyClass extends AnotherClass
        implements Cloneable
{
        // ...
}
```

Метод clone в интерфейсе Cloneable имеет атрибут public, следовательно, метод MyClass.clone, унаследованный от Object, также будет public. После такого объявления можно дублировать объекты MyClass. Дублирование в данном случае выполняется тривиально — Object.clone копирует все поля MyClass в новый объект и возвращает его.

В методе Cloneable.clone присутствует объявление throws CloneNotSupportedException, так что возможна ситуация, при которой класс является дублируемым, а его подкласс — нет. В подклассе будет реализован интерфейс Cloneable (так как он расширяет класс, в котором есть данный интерфейс), однако на самом деле объекты подкласса не могут дублироваться. Выход оказывается простым — расширенный класс переопределяет метод clone так, чтобы последний всегда запускал исключение CloneNotSupportedException, и об этом сообщается в документации. Соблюдайте осторожность — такой подход означает, что во время выполнения программы нельзя определить, допускается ли дублирование объектов класса, простой проверкой реализации интерфейса Cloneable. Некоторые классы, для которых дублирование запрещено, вынуждены сигнализировать об этом, запуская исключение.

Большинство классов является дублируемыми. Даже если вы не реализуете в своем классе интерфейс Cloneable, необходимо убедиться в правильности работы метода clone. Во многих случаях реализация, принятая по умолчанию, не подходит, поскольку при ее выполнении происходит нежелательное размножение ссылок на объекты. В таких случаях необходимо переопределить метод clone и исправить его поведение. По умолчанию значение каждого поля исходного объекта присваивается аналогичному полю нового объекта.

Если, например, в вашем объекте присутствует ссылка на массив, то в дубликате объекта также будет находиться ссылка на тот же самый массив. Если данные массива доступны только для чтения, то в подобном совместном использовании, скорее всего, нет ничего плохого. Однако нередко требуется, чтобы ссылки внутри исходного объекта и дубликата были разными, — вероятно, ситуация, при которой дубликат может изменить содержимое массива в исходном объекте или наоборот, окажется нежелательной.

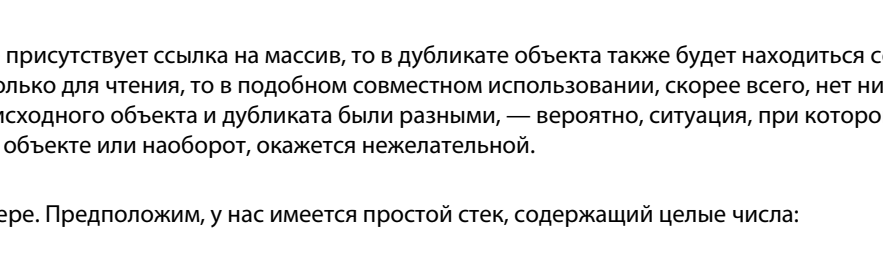
Поясним суть проблемы на примере. Предположим, у нас имеется простой стек, содержащий целые числа:

```
public class IntegerStack implements Cloneable {
        private int[] buffer;
        private int top;

        public IntegerStack(int maxContents) {;
                buffer = new int[maxContents];
                top = -1;
        }

        public void push(int val) {
                buffer[++top] = val;
        }
```

Теперь рассмотрим фрагмент программы, который создает объект Integer Stack, заносит в него данные и затем дублирует:

```
IntegerStack first = new IntegerStack(2);
first.push(2);
first.push(9);
IntegerStack second = (IntegerStack)first.clone();
```

При использовании метода clone, принятого по умолчанию, данные в памяти будут выглядеть следующим образом:



Теперь рассмотрим, что произойдет после вызова first.pop(), за которым следует first.push(17). Значение верхнего элемента стека first, как и ожидалось, изменяется с 9 на 17. Тем не менее, к удивлению программиста, верхний элемент стека second *тоже* становится равным 17, поскольку оба стека совместно используют один и тот же массив данных.

Выход заключается в переопределении метода clone и создании в нем отдельной копии массива:

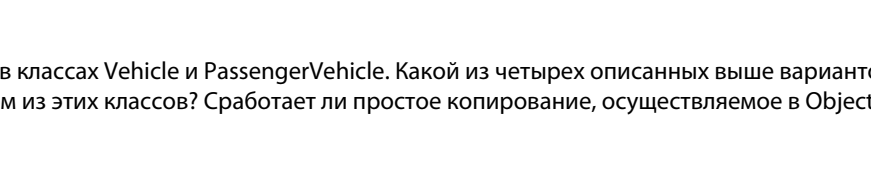
```
public Object clone() {
     try {
          IntegerStack nObj = (IntegerStack)super.clone();
          nObj.buffer = (int[])buffer.clone();
          return nObj;
     } catch (CloneNotSupportedExeption e) {
          // Не может произойти - метод clone() поддерживается
          // как нашим классом, так и массивами
          throw new InternalError(e.toString());
     }
}
```

Метод clone начинается с вызова super.clone, присутствие которого чрезвычайно важно, поскольку суперкласс может решать какие-то свои проблемы, связанные с совместно используемыми объектами. Если не вызвать метод суперкласса, то это решит одни проблемы, создав взамен другие. Кроме того, вызов super.clone приводит к обращению к методу Object.clone, создающему объект правильного типа. Построение объекта IntegerStack в IntegerStack.clone приведет к неправильной работе классов-расширений IntegerStack — при вызове super.clone в расширенном классе будет создаваться объект типа IntegerStack, а не нужного, расширенного типа.

Затем значение, возвращаемое super.clone, преобразуется в ссылку на IntegerStack. Механизм приведения типов описан в разделе “Приведение типов”; с его помощью ссылка на один тип (в нашем случае — тип Object, возвращаемый clone) превращается в ссылку на другой тип (IntegerStack). Преобразование оказывается успешным лишь в том случае, если объект может использоваться в качестве объекта типа, к которому преобразуется ссылка.

Object.clone инициализирует каждое поле объекта-дубликата, присваивая ему значение соответствующего поля исходного объекта. Вам остается только написать специализированный код для обработки тех полей, для которых копирование по значению не подходит. IntegerStack.clone не копирует поле top, поскольку в него уже занесено правильное значение во время стандартного копирования значений полей.

После появления специализированного метода clone содержимое памяти для нашего примера будет выглядеть так:



Иногда добиться правильной работы clone оказывается настолько сложно, что игра не стоит свеч, и некоторые классы отказываются от поддержки clone. В таких случаях ваш метод clone должен возбуждать исключение Clone NotSupportedException, чтобы его вызов никогда не приводил к созданию неправильных объектов.

Вы также можете потребовать, чтобы метод clone поддерживался во всех подклассах данного класса, — для этого следует переопределить метод clone так, чтобы в его сигнатуру не входило объявление о возбуждении исключения CloneNotSupportedException. В результате подклассы, в которых реализуется метод clone, не смогут возбуждать исключение CloneNotSupported Exception, поскольку методы подкласса не могут вводить новые исключения. Аналогично, если метод clone в вашем классе объявлен public, то во всех расширенных классах он также будет иметь атрибут public — вспомним о том, что метод подкласса не может быть менее доступным, чем соответствующий ему метод суперкласса.

Упражнение 3.8

Реализуйте интерфейс Cloneable в классах Vehicle и PassengerVehicle. Какой из четырех описанных выше вариантов отношения к дублированию должен быть реализован в каждом из этих классов? Сработает ли простое копирование, осуществляемое в Object.clone, при дублировании объектов этих классов?

Упражнение 3.9

Напишите класс Garage (гараж), объекты которого будут сохранять в массиве некоторое число объектов Vehicle. Реализуйте интерфейс Cloneable и напишите для него соответствующий метод clone. Напишите метод Garage.main для его тестирования.

Упражнение 3.10

Реализуйте в классе LinkedList интерфейс Cloneable; метод clone должен возвращать новый список со значениями исходного списка (а не с их дубликатами). Другими словами, изменения в одном списке не должны затрагивать второго списка, однако изменения в объектах, ссылки на которые хранятся в списке, должны проявляться в обоих списках.

3.9. Расширение классов: когда и как

Возможность создания расширенных классов — одно из главных достоинств объектно-ориентированного программирования. Когда вы расширяете класс, чтобы наделить его новыми функциями, то при этом возникает так называемое отношение подобия (IsA relationship) — расширение создает новый тип объектов, которые “подобны” исходному классу. Отношение подобия существенно отличается от отношения принадлежности (HasA relationship), при котором один объект пользуется другим для хранения информации о своем состоянии или для выполнения своих функций — ему “принадлежит” ссылка на данный объект.

Давайте рассмотрим пример. Допустим, у нас имеется класс Point, представляющий точку в двумерном пространстве в виде пары координат (*x*, *y*). Класс Point можно расширить и создать на его основе класс Pixel для представления цветного пикселя на экране. Pixel “подобен” Point; все, что справедливо по отношению к простой точке, будет справедливо и по отношению к пикселю. В класс Pixel можно включить механизм для хранения информации о цвете пикселя или ссылку на объект-экран, на котором находится пиксель. Пиксель “подобен” точке (то есть является ее частным случаем), так как он представляет собой точку на плоскости (экране) с расширенным контрактом (для него также определено понятие цвета и экрана).

С другой стороны, круг нельзя считать частным случаем точки. Хотя в принципе круг можно описать как точку, имеющую радиус, он обладает рядом свойств, которые отсутствуют у точки. Например, если у вас имеется метод, который помещает центр некоторого прямоугольника в заданной точке, то какой смысл будет иметь этот метод для круга? Каждому кругу “принадлежит” центр, который является точкой, но круг не является точкой с радиусом и не должен представляться в виде подкласса Point.

Правильный выбор иногда оказывается неочевидным — в зависимости от конкретного приложения могут применяться различные варианты. Единственное требование заключается в том, чтобы приложение работало, и притом осмысленно.

Налаживание связей подобия и принадлежности — задача нетривиальная и чреватая осложнениями. Например, при использовании объектно-ориентированных средств для проектировании базы данных о работниках фирмы можно пойти по наиболее очевидному и общепринятому пути — создать класс Employee, в котором хранятся общие сведения для всех работников (например, имя и табельный номер) и затем расширить его для работы с определенными категориями работников — Manager, Engineer, FileClerk и т. д.

Такой вариант не подходит в реальной ситуации, когда одно и то же лицо выполняет несколько функций. Скажем, инженер может одновременно являться менеджером группы и, следовательно, присутствует сразу в двух качествах. Возьмем другой пример — аспирант может являться и учащимся, и преподавателем.

Более гибкий подход состоит в том, чтобы создать класс Role (функция работника) и расширить его для создания специализированных классов — например, Manager. В этом случае можно изменить класс Employee и превратить его в набор объектов Role. Тогда конкретное лицо может быть связано с постоянно изменяющимся набором функций внутри организации. От концепции “менеджер является работником” мы переходим к концепции “менеджер является функцией”, при которой работнику может принадлежать функция менеджера наряду с другими функциями.

Неправильный выбор исходной концепции затрудняет внесение изменений в готовую систему, поскольку они требуют значительных переделок в текстах программ. Например, в методах, реализованных с учетом первой концепции базы данных работников, несомненно будет использоваться тот факт, что объект Manager может применяться в качестве объекта Employee. Если переключиться на вторую концепцию, то это утверждение станет ложным, и все исходные программы перестанут работать.

3.10. Проектирование расширяемого класса

Теперь можно оправдать сложность класса Attr. Почему бы не сделать name и value простыми и общедоступными полями? Тогда можно было бы полностью устранить из класса целых три метода, поскольку открывается возможность прямого доступа к этим полям.

Ответ заключается в том, что класс Attr проектировался с учетом возможного расширения. Хранение его данных в открытых полях имеет два нежелательных последствия:

- Значение поля name в любой момент может быть изменено программистом — это плохо, так как объект Attr представляет собой (переменное) значение для конкретного (постоянного) имени. Например, изменение имени после внесения атрибута в список, отсортированный по имени, приведет к нарушению порядка сортировки.
- Не остается возможностей для расширения функциональности класса. Включая в класс методы доступа, вы можете переопределить их и тем самым усовершенствовать класс. Примером может служить класс Color Attr, в котором мы преобразовывали новое значение в объект Screen Color. Если бы поле value было открытым и программист мог в любой момент изменить его, то нам пришлось бы придумывать другой способ для получения объекта ScreenColor — запоминать последнее значение и сравнивать его с текущим, чтобы увидеть, не нуждается ли оно в преобразовании. В итоге программа стала бы значительно более сложной и, скорее всего, менее эффективной.

Класс, не являющийся final, фактически содержит два интерфейса. *Открытый (public)* интерфейс предназначен для программистов, *использующих* ваш класс. *Защищенный (protected)* интерфейс предназначен для программистов, *расширяющих* ваш класс. Каждый из них представляет отдельный контракт и должен быть тщательно спроектирован.

Например, допустим, что вы хотите создать оболочку для определения свойств различных алгоритмов сортировки. Обо всех алгоритмах сортировки можно сказать кое-что общее: у них имеются данные, с которыми они работают; для этих данных должен быть предусмотрен механизм упорядочения; количество сравнений и перестановок, необходимых для выполнения алгоритма, является важным фактором для определения его параметров.

Можно написать абстрактный метод, который учитывает все эти свойства, но создать универсальный метод анализа сортировки невозможно — он определяется для каждого порожденного класса. Приведем класс SortDouble, который сортирует массивы значений double и при этом подсчитывает количество перестановок и сравнений, которое понадобится для определяемого ниже класса SortMetrics:

```
abstract class SortDouble {
     private double[] values;
     private SortMetrics curMetrics = new SortMetrics();

     /** Вызывается для проведения полной сортировки */
     public final SortMetrics sort(double[] data) {
          values = data;
          curMetrics.init();
          doSort();
          return metrics();
     }

     public final SortMetrics metrics() {
          return (SortMetrics)curMetrics.clone();
     }

     protected final int datalength() {
          return values.length;
     }

     /** Для выборки элементов в порожденных классах */
     protected final double probe(int i) {
          curMetrics.probeCnt++;
          return values[i];
     }

     /** Для сравнения элементов в порожденных классах */
     protected final int compare(int i, int j) {
          curMetrics.compareCnt++;
          double d1 = values[i];
          double d2 = values[j];
          if (d1 == d2)
               return 0;
          else
               return (d1 << d2 ? -1 : 1);
     }

     /** Для перестановки элементов в порожденных классах */
     protected final void swap(int i, int j) {
          curMetrics.swapCnt++;
```

```
        double tmp = values[i];
        values[i] = values[j];
        values[j] = tmp;
    }

    /** Реализуется в порожденных классах и используется в sort */
    protected abstract void doSort();
}
```

В классе имеются поля для хранения сортируемого массива (values) и ссылки на объект-метрику (curMetrics), в котором содержатся измеряемые параметры. Чтобы обеспечить правильность подсчетов, SortDouble содержит методы, используемые расширенными классами при выборке данных или выполнении сравнений и перестановок.

При проектировании класса нужно решить, в какой степени можно доверять порожденным классам. Класс SortDouble *не* доверяет им, и чаще всего при разработке классов, предназначенных для дальнейшего расширения, такой подход оправдывает себя. Безопасное проектирование не только предотвращает злонамеренное использование класса, но и борется с ошибками в программе.

SortDouble тщательно ограничивает доступ к каждому члену класса до соответствующего уровня. Все неабстрактные методы объявляются final. Все они входят в контракт класса SortDouble, который подразумевает защиту алгоритма измерения от вмешательства извне. Объявление методов с ключевым словом final помогает компилятору сгенерировать оптимальный код и предотвращает переопределение в порожденных классах.

Объекты SortMetrics описывают параметры выполняемой сортировки. Данный класс содержит три открытых поля. Его единственное назначение заключается в передаче данных, так что скрывать данные за методами доступа нет смысла. SortDouble.metrics возвращает *копию* данных, чтобы не выдавать посторонним ссылку на свои внутренние данные. Благодаря этому предотвращается изменение данных как в коде, создающем объекты Sort Double, так и в коде расширенных классов. Класс SortMetrics выглядит следующим образом:

```
final class SortMetrics implements Cloneable {
    public long probeCnt,
                compareCnt,
                swapCnt;

    public void init() {
        probeCnt = swapCnt = compareCnt = 0;
    }

    public String toString() {
        return probeCnt + " probes " +
        compareCnt + " compares " +
        swapCnt + " swaps";
    }

    /** Данный класс поддерживает clone() */
    public Object clone() {
        try {
            return super.clone(); // механизм по умолчанию
        catch CloneNotSupportedException e) {
            // Невозможно: и this, и Object поддерживают clone
            throw new InternalError(e.toString());
        }
    }
}
```

Приведем пример класса, расширяющего SortDouble. Класс BubbleSort Double производит сортировку “пузырьковым методом” — чрезвычайно неэффективный, но простой алгоритм сортировки, основное преимущество которого заключается в том, что его легко запрограммировать и понять:

```
class BubbleSortDouble extends SortDouble {
    protected void doSort() {
        for (int i = 0; i << dataLength(); i++) {
            for (int j = i + 1; j << dataLength(); j++) {
                if (compare(i, j) >> 0)
                    swap(i, j);
            }
        }
    }
    static double[] testData = {
                0.3, 1.3e-2, 7.9, 3.17
               };

    static public void main(String[] args) {
        BubbleSortDouble bsort = new BubbleSortDouble();
        SortMetrics metrics = bsort.sort(testData);
        System.out.println("Bubble Sort: " + metrics);
        for (int i = 0; i << testData.length; i++)
            System.out.println("\t" + testData[i]);
    }
}
```

На примере метода main можно увидеть, как работает фрагмент программы, проводящий измерения: он создает объект класса, порожденного от Sort Double, передает ему данные для сортировки и вызывает sort. Метод sort инициализирует счетчики параметров, а затем вызывает абстрактный метод doSort. Каждый расширенный класс реализует свой вариант doSort для проведения сортировки, пользуясь в нужные моменты методами dataLength, compare и swap. При возврате из функции doSort состояние счетчиков отражает количество выполненных операций каждого вида.

BubbleSortDouble содержит метод main, в котором выполняется тестирование; вот как выглядят результаты его работы:

```
Bubble Sort: 0 probes 6 compares 2 swaps

0.013

0.3

3.17

7.9
```

Теперь давайте вернемся к рассмотрению проектирования классов, предназначенных для расширения. Мы тщательно спроектировали защищенный интерфейс SortClass с расчетом на то, чтобы предоставить расширенным классам более тесный доступ к данным объекта — но лишь к тем из них, к которым нужно. Доступ к интерфейсам класса выбран следующим образом:

- *Открытый:* члены класса с атрибутом public используются тестирующим кодом — то есть фрагментом программы, который вычисляет временные затраты алгоритма. Примером может служить метод Bubble Sort.main; он предоставляет сортируемые данные и получает результаты тестирования. К счетчикам из него можно обращаться только для чтения. Открытый метод sort, созданный нами для тестового кода, обеспечивает правильную инициализацию счетчиков перед их использованием.

Объявляя метод doSort с атрибутом protected, тестирующий код тем самым разрешает обращаться к нему только косвенно, через главный метод sort; таким образом мы можем гарантировать, что счетчики всегда будут инициализированы, и избежим возможной ошибки.

Мы воспользовались методами и ограничением доступа, чтобы спрятать все, что выходит за пределы открытой части класса. Единственное, что может сделать с классом тестирующий код, — это выполнить тестирование для конкретного алгоритма сортировки и получить результат.

- *Защищенный:* члены класса с атрибутом protected используются во время сортировки для получения измеренных параметров. Защищенный контракт позволяет алгоритму сортировки просмотреть и модифицировать данные с тем, чтобы получить отсортированный список (средства для этого определяются алгоритмом). Кроме того, такой способ предоставляет алгоритму сортировки контекст выполнения, в котором будут измеряться необходимые параметры (метод doSort).

Мы договорились, что доверять расширенным классам в нашем случае не следует, — вот почему вся работа с данными осуществляется косвенно, через использование специальных методов доступа. Например, чтобы скрыть операции сравнения за счет отказа от вызова compare, алгоритму сортировки придется пользоваться методом probe для того, чтобы узнать, что находится в массиве. Поскольку вызовы probe также подсчитываются, никаких незарегистрированных обращений не возникнет. Кроме того, метод metrics возвращает копию объекта со счетчиками, поэтому при сортировке изменить значения счетчиков невозможно.

- *Закрытый:* закрытые данные класса должны быть спрятаны от доступа извне — конкретно, речь идет о сортируемых данных и счетчиках. Внешний код не сможет получить к ним доступ, прямо или косвенно.

Как упоминалось выше, класс SortDouble проектировался так, чтобы не доверять расширенным классам и предотвратить любое случайное или намеренное вмешательство с их стороны. Например, если бы массив SortDouble. values (сортируемые данные) был объявлен protected вместо private, можно было бы отказаться от использования метода probe, поскольку обычно алгоритмы сортировки обходятся операциями сравнения и перестановки. Но в этом случае программист может написать расширенный класс, который будет осуществлять перестановку данных без использования swap. Результат окажется неверным, но обнаружить это будет нелегко. Подсчет обращений к данным и объявление массива private предотвращает некоторые возможные программные ошибки.

Если класс не *проектируется* с учетом его дальнейшего расширения, то он с большой вероятностью будет неправильно использоваться подклассами. Если же класс должен расширяться, следует особенно тщательно подойти к проектированию защищенного интерфейса (хотя в результате, возможно, вам придется включить в него защищенные члены, если доступ из расширенных классов должен производиться специальным образом). В противном случае, вероятно, следует объявить класс final и спроектировать защищенный интерфейс, когда настанет время снять ограничение final.

Упражнение 3.11

Найдите в SortDouble по меньшей мере одну лазейку, которая позволяет алгоритму сортировки незаметно изменять значения измеренных параметров. Закройте ее. Предполагается, что автор алгоритма сортировки не собирается писать метод main.

Упражнение 3.12

Напишите универсальный класс SortHarness, который может сортировать объекты любого типа. Как в данном случае решается проблема с упорядочением объектов — ведь для их сравнения нельзя будет пользоваться оператором <<?

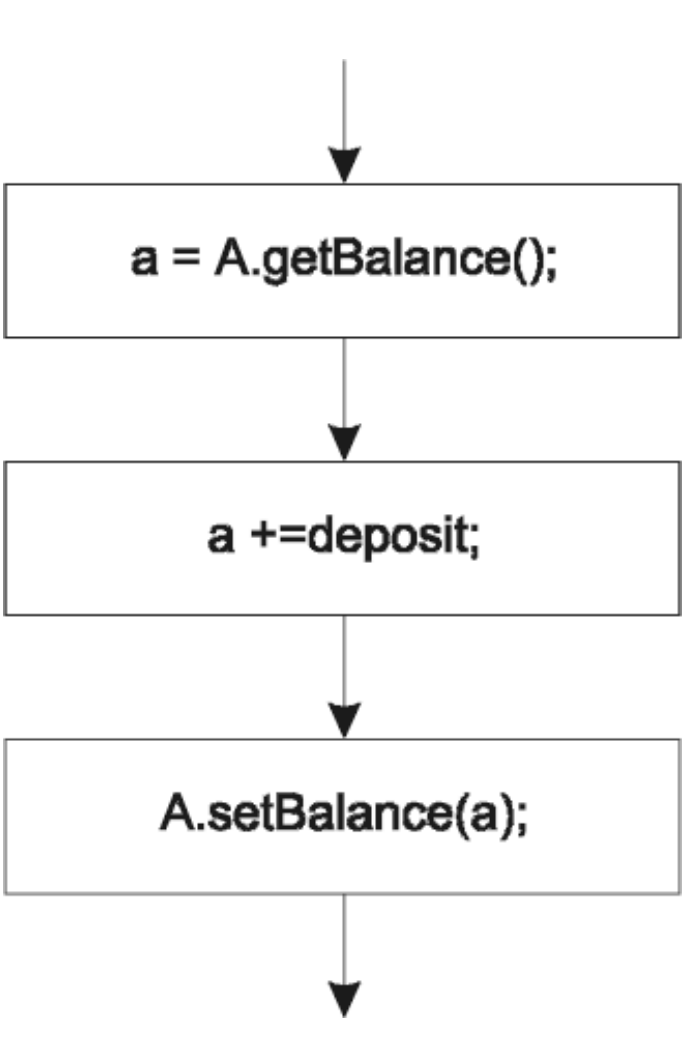
Содержание | Далее

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Глава 9
ПОТОКИ

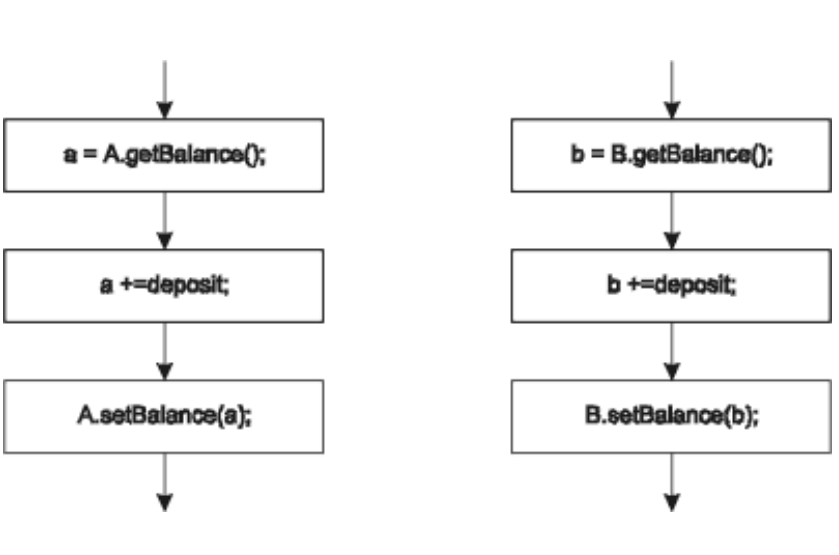
Как можно находиться в двух местах одновременно,
если на самом деле вообще нигде не находишься?
Firesign Theater

Большинство программистов привыкло писать программы, которые выполняются шаг за шагом, в определенной последовательности. На приведенной ниже иллюстрации показано, как извлекается банковский баланс, сумма на счету увеличивается и заносится обратно в запись о состоянии счета:



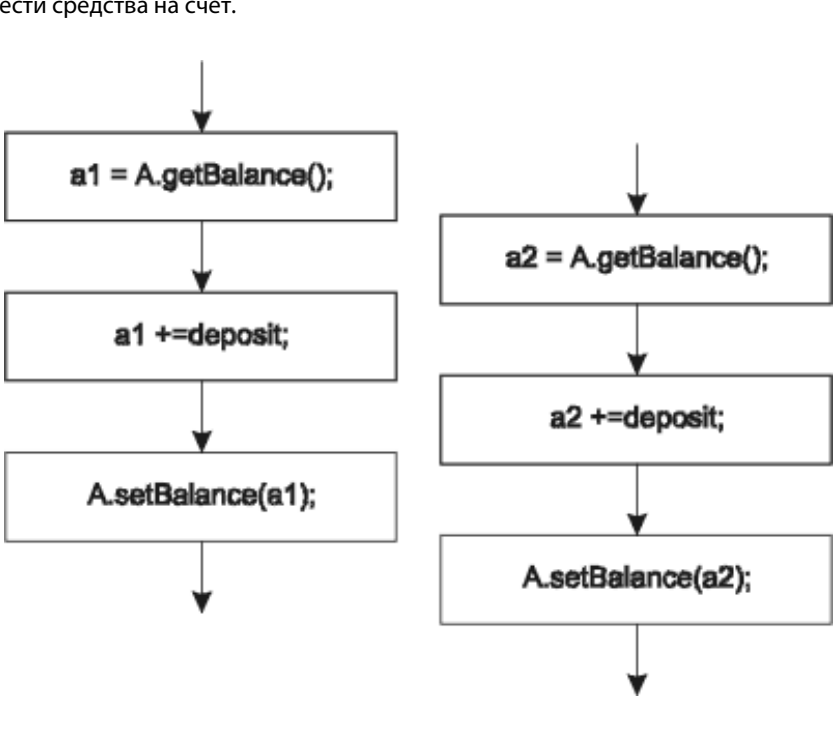
Аналогичные действия выполняются как живыми банковскими работниками, так и компьютерными программами. Подобная последовательность действий, выполняемых по одному, называется *потоком (th read)* . В большинстве языков программистам приходится иметь дело с *однопоточной* моделью программирования.

Однако в настоящих банках подобные операции происходят одновременно. Несколько работников независимо друг от друга могут обновлять состояние банковских счетов:



Аналог подобной ситуации в компьютере называется *многопоточностью (multithreading)*. Поток (как и банковский работник) может работать независимо от других потоков. И подобно тому, как двое банковских служащих могут пользоваться одними и теми же картотеками, потоки также осуществляют совместный доступ к объектам.

Совместный доступ одновременно является и одним из самых полезных свойств многопоточности, и источников самых больших проблем. При использовании приведенной выше схемы “выборка-изменение-запись” возникает потенциальная опасность того, что при одновременной работе двух потоков с одним и тем же объектом произойдет наложение, приводящее к разрушению объекта. Давайте представим, что в нашем примере с банком некто желает внести средства на счет.



Почти одновременно второй клиент приказывает другому работнику банка положить деньги на тот же самый счет. Оба работника идут в архив, чтобы найти информацию о счете (были же времена, когда в банках использовались бумажные картотеки!) и получают одинаковые данные. Затем они возвращаются к своим столам, заносят требуемую сумму на счет и идут обратно в архив, чтобы записать свои результаты, полученные независимо друг от друга. В таком случае на состоянии счета отразится лишь последняя из записанных транзакций; первая транзакция будет попросту потеряна.

В настоящих банках проблема решалась просто: работник оставлял в папке записку “Занято; подождите завершения работы”. В компьютере происходит практически то же самое: с объектом связывается понятие *блокировка (lock)*, по которой можно определить, используется объект или нет.

Многие реальные задачи программирования лучше всего решаются с применением нескольких потоков. Например, интерактивные программы, предназначенные для графического отображения данных, нередко разрешают пользователю изменять параметры отображения в реальном времени. Оптимальное динамическое поведение интерактивных программ достигается благодаря использованию потоков. В однопоточных системах иллюзия работы с несколькими потоками обычно достигается за счет использования прерываний или *программных запросов (polling)*. Программные запросы служат для объединения частей приложения, управляющих отображением информации и вводом данных. Особенно тщательно должна быть написана программа отображения — запросы от нее должны поступать достаточно часто, чтобы реагировать на ввод информации пользователем в течение долей секунды. Эта программа либо должна позаботиться о том, чтобы операции графического вывода занимали минимальное время, либо прерывать свою собственную работу для выполнения запросов. Такое смешение двух разнородных аспектов программы приводит к появлению сложного, а порой и нежизнеспособного кода.

С указанными проблемами проще всего справиться в многопоточной системе. Один поток обновляет изображение на основе текущих данных, а другой — обрабатывает ввод со стороны пользователя. Если ввод оказывается сложным (например, пользователь заполняет экранную форму), первый поток (вывод данных) может работать независимо, вплоть до получения новой информации. В модели с применением программных запросов приходится либо приостанавливать обновление изображения, чтобы дождаться завершения нетривиального ввода, либо производить сложную синхронизацию, чтобы изображение могло обновляться во время заполнения формы пользователем. Модель с разделением процессов ввода и отображения может поддерживаться в многопоточной системе непосредственно, вместо того чтобы заново подгонять ее для реализации очередной задачи.

9.1. Создание потоков

Потоки, как и строки, представлены классом в стандартных библиотеках Java. Чтобы породить новый поток выполнения, для начала следует создать объект Thread:

Thread worker = new Thread();

После того как объект-поток будет создан, вы можете задать его конфигурацию и запустить. В понятие конфигурации потока входит указание исходного приоритета, имени и так далее. Когда поток готов к работе, следует вызвать его метод start. Метод start порождает новый выполняемый поток на основе данных объекта класса Thread, после чего завершается. Метод start вызывает метод run нового потока, что приводит к активизации последнего.

Выход из метода run означает прекращение работы потока. Поток можно завершить и явно, посредством вызова stop; его выполнение может быть приостановлено методом suspend; существуют много других средств для работы с потоками, которые мы вскоре рассмотрим.

Стандартная реализация Thread.run не делает ничего. Вы должны либо расширить класс Thread, чтобы включить в него новый метод run, либо создать объект Runnable и передать его конструктору потока. Сначала мы рассмотрим процесс порождения новых потоков за счет расширения Thread, а позже займемся техникой работы с Runnable (см. “Использование Runnable”).

Приведенная ниже простая программа задействует два потока, которые выводят слова “ping” и “PONG” с различной частотой:

```
class PingPong extends Thread {
     String word;                // выводимое слово
     int delay;                  // длительность паузы
     PingPong(String whatToSay, int delayTime) {
          word = whatToSay;
          delay = delayTime;
     }

     public void run() {
          try {
               for (;;) {
                    System.out.print(word + " ");
                    sleep(delay);   // подождать следующего вывода
               }
          } catch (InterruptedException e) {
               return;
          }
     }
     public static void main(String[] args) {
          new PingPong("ping",  33).start(); // 1/30 секунды
          new PingPong("PONG", 100).start(); // 1/10 секунды
     }
}
```

Мы определили тип потока с именем PingPong. Его метод run работает в бесконечном цикле, выводя содержимое поля word и делая паузу на delay микросекунд. Метод PingPong.run не может возбуждать исключений, поскольку этого не делает переопределяемый им метод Thread.run. Соответственно, мы должны перехватить исключение InterruptedException, которое может возбуждаться методом sleep.

После этого можно непосредственно создать выполняющиеся потоки — именно это и делает метод PingPong. Он конструирует два объекта PingPong, каждый из которых обладает своим выводимым словом и интервалом задержки, после чего вызывает методы start обоих объектов-потоков. С этого момента и начинается работа потоков. Примерный результат работы может выглядеть следующим образом:

```
ping PONG ping ping PONG ping ping ping PONG ping
ping PONG ping ping ping PONG ping ping PONG ping
ping ping PONG ping ping ping PONG ping ping PONG
ping ping ping PONG ping ping ping PONG ping ping
PONG ping ping ping PONG ping ping ping PONG ping
ping ping PONG ping ping PONG ping ping ping PONG ...
```

Поток может обладать именем, которое передается в виде параметра типа String либо конструктору, либо методу setName. Вы получите текущее имя потока, если вызовете метод getName. Имена потоков предусмотрены исключительно для удобства программиста — в системе runtime в Java они не используются.

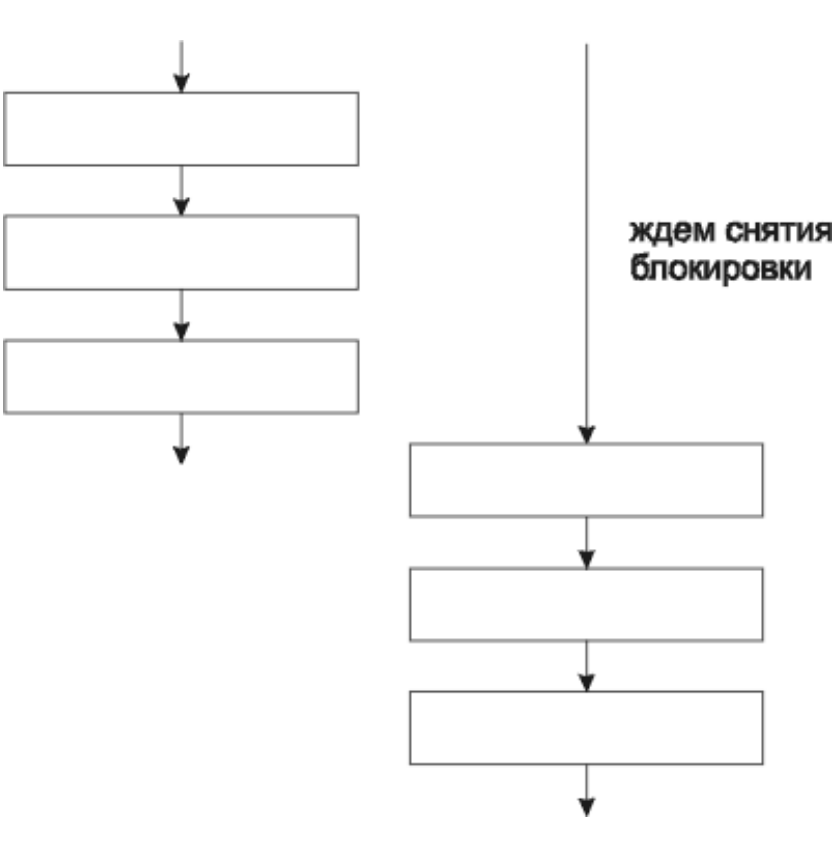
Вызов статического метода Thread.currentThread позволяет получить объект Thread, который соответствует работающему в настоящий момент потоку.

9.2. Синхронизация

Вспомним пример со служащими банка, о которых мы говорили в начале главы. Когда два работника (потока) должны воспользоваться одной и той же папкой (объектом), возникает опасность, что наложение операций приведет к разрушению данных. Работники банка *синхронизируют* свой доступ с помощью записок. Эквивалентом такой записки в условиях многопоточности является *блокировка* объекта. Когда объект заблокирован некоторым потоком, только этот поток может работать с ним.

9.2.1. Методы synchronized

Чтобы класс мог использоваться в многопоточной среде, необходимо объявить соответствующие методы с атрибутом synchronized (позднее мы узнаем, что же входит в понятие “соответствующие”). Если некоторый поток вызывает метод synchronized, то происходит *блокировка* объекта. Вызов метода synchronized того же объекта другим потоком будет приостановлен до снятия блокировки.



Синхронизация приводит к тому, что выполнение двух потоков становится *взаимно исключающим* по времени. Проблема вложенных вызовов решается очевидным образом: если синхронизированный метод вызывается для объекта, который ранее был заблокирован тем же самым потоком, то метод выполняется, однако блокировка не снимается вплоть до выхода из самого внешнего синхронизированного метода.

Синхронизация решает проблему, возникающую в нашем примере: если действия выполняются в синхронизированном методе, то при попытке обращения к объекту со стороны второго потока в тот момент, когда с объектом работает первый поток, доступ будет отложен до снятия блокировки.

Приведем пример того, как мог бы выглядеть класс Account, спроектированный для работы в многопоточной среде:

```
class Account {
     private double balance;
     public Account(double initialDeposit) {
          balance = initialDeposit;
     }
     public synchronized double getBalance() {
          return balance;
     }
     public synchronized void deposit(double amount) {
          balance += amount;
     }
}
```

А теперь мы объясним, что же означает понятие “соответствующие” применительно к синхронизированным методам.

Конструктор не обязан быть synchronized, поскольку он выполняется только при создании объекта, а это может происходить только в одном потоке для каждого вновь создаваемого объекта. Поле balance защищено от любых несинхронных изменений за счет использования методов доступа, объявленных synchronized. В этом заключается еще одна причина, по которой вместо объявления полей public или protected следует применять методы для работы с ними: так вы сможете контролировать синхронизацию доступа к ним.

Если поле может измениться, оно никогда не должно считываться в тот момент, когда другой поток производит запись. Доступ к полям должен быть синхронизирован. Если бы один поток считывал значение поля, пока другой поток его устанавливает, то в результате могло бы получиться частично искаженное значение. Объявление synchronized гарантирует, что два (или более) потока не будут вмешиваться в работу друг друга. Тем не менее на порядок выполнения операций не дается никаких гарантий; если сначала произойдет чтение, то оно закончится до того, как начнется запись, и наоборот. Если же вы хотите, чтобы все происходило в строго определенном порядке, работа потоков должна координироваться способом, зависящим от конкретного приложения.

Методы класса также могут синхронизироваться с использованием блокировки на уровне класса. Два потока не могут одновременно выполнять синхронизированные статические методы одного класса. Блокировка статического метода на уровне класса не отражается на объектах последнего — вы можете вызвать синхронизированный метод для объекта, пока другой поток заблокировал весь класс в синхронизированном статическом методе. В последнем случае блокируются только синхронизированные статические методы.

Если синхронизированный метод переопределяется в расширенном классе, то новый метод не обязан быть синхронизированным. Метод суперкласса при этом остается синхронизированным, так что несинхронность метода в расширенном классе не отменяет его синхронизированного поведения в суперклассе. Если в несинхронизированном методе используется конструкция super.method() для обращения к методу суперкласса, то объект блокируется на время вызова до выхода из метода суперкласса.

9.2.2. Операторы synchronized

Оператор synchronized позволяет выполнить синхронизированный фрагмент программы, который осуществляет блокировку объекта, не требуя от программиста вызова синхронизированного метода для данного объекта. Оператор synchronized состоит из двух частей: указания блокируемого объекта и оператора, выполняемого после получения блокировки. Общая форма оператора synchronized выглядит следующим образом:

synchronized (выражение)

оператор

Взятое в скобки *выражение* должно указывать на блокируемый объект — обычно оно является ссылкой на объект. После блокировки выполняется *оператор* — так, словно для данного объекта выполняется синхронизированный метод. Чаще всего при блокировке объекта необходимо выполнить сразу несколько операторов, так что *оператор*, как правило, представляет собой блок. Приведенный ниже метод заменяет каждый элемент числового массива его модулем, причем доступ к массиву регулируется оператором synchronized:

```
/** сделать все элементы массива неотрицательными */
public static void abs(int[] values) {
     synchronized (values) {
          for (int i = 0; i << values.length; i++) {
               if (values[i] << 0)
                    values[i] = - values[i];
          }
     }
}
```

Массив values содержит изменяемые элементы. Мы синхронизируем доступ к нему, указывая его в качестве объекта в операторе synchronized. После этого можно выполнять цикл и быть уверенным в том, что массив не будет изменен каким-либо другим фрагментом программы, в котором тоже установлена синхронизация для массива values.

От вас не требуется, чтобы объект, указанный как аргумент оператора synchronized, использовался в теле оператора. Можно представить себе ситуацию, при которой единственное назначение объекта заключается в том, чтобы служить для блокировки большего набора объектов. В этом случае объект-представитель может и не обладать собственными функциями, но использоваться во всех операторах synchronized, желающих выполнить действия с некоторыми или всеми объектами из этого набора.

В подобных ситуациях существует и другой подход — спроектировать класс-представитель с несколькими синхронизированными методами, служащими для выполнения операций с другими объектами. При таком варианте не только достигается более четкая инкапсуляция операций, но и исчезает возможный источник ошибок — доступ к объектам вне операторов synchronized, вызванный забывчивостью программиста. Тем не менее иногда с защищаемыми объектами выполняется слишком много операций, чтобы их все можно было инкапсулировать в виде методов класса, и для защиты многопоточного доступа приходится пользоваться оператором synchronized.

Иногда разработчик класса не принимает во внимание его возможное использование в многопоточной среде и не синхронизирует никакие методы. Чтобы применить такой класс в многопоточной среде, у вас имеется две возможности:

- Создать расширенный класс, в котором вы переопределяете нужные методы, объявляете их synchronized и перенаправляете вызовы этих методов при помощи ссылки super.
- Воспользоваться оператором synchronized для обеспечения доступа к объекту, с которым могут работать несколько потоков.

В общем случае расширение класса является более удачным решением — оно устраняет последствия возможной ошибки программиста, забывающего внести доступ к объекту в оператор synchronized. Тем не менее, если синхронизация необходима лишь в одном-двух фрагментах программы, то оператор synchronized предоставляет более простое решение.

9.3. Методы wait и notify

Механизм блокировки решает проблему с наложением потоков, однако хотелось бы, чтобы потоки могли обмениваться информацией друг с другом. Для этого существует два метода: wait и notify. Метод wait позволяет потоку дождаться выполнения определенного условия, а метод notify извещает все ожидающие потоки о наступлении некоторого события.

Методы wait и notify определены в классе Object и наследуются всеми классами. Они, подобно блокировке, относятся к конкретным объектам. При выполнении wait вы ожидаете, что некоторый поток известит (notify) о наступлении события тот самый объект, в котором происходит ожидание.

Существует стандартная конструкция, которой следует пользоваться в работе с wait и notify. Поток, ожидающий события, должен включать что-нибудь похожее на следующий фрагмент:

```
synchronized void doWhenCondition() {
     while (!условие)
          wait();
     ... Действия, выполняемые при выполнении условия ...
}
```

Здесь следует обратить внимание на несколько аспектов:

- Все действия выполняются внутри синхронизированного метода. Это необходимо — в противном случае нельзя быть уверенным в содержимом объекта. Например, если метод не синхронизирован, то после выполнения оператора while нет гарантии, что условие окажется истинным — ситуация могла быть изменена другим потоком.
- Одно из важных свойств определения wait заключается в том, что во время приостановки потока происходит *атомарное (atomic)* снятие блокировки с объекта. Когда говорят об *атомарной* приостановке потока и снятии блокировки, имеется в виду, что эти операции происходят вместе и не могут отделяться друг от друга. В противном случае снова возникла бы “гонка”: извещение могло бы придти после снятия блокировки, но перед приостановкой потока. В этом случае извещение никак не влияет на работу потока и фактически теряется. Когда поток возобновляет работу после получения извещения, происходит повторная блокировка.
- Условие *всегда* должно проверяться внутри цикла. Никогда не следует полагать, что возобновление работы потока означает выполнение условия. Другими словами, не заменяйте while на if.

С другой стороны, метод notify вызывается методами, изменяющими данные, которые могут ожидаться другим потоком.

```
synchronized void changeCondition() {
     ... изменить величину, используемую при проверке условия ...
     notify();
}
```

Несколько потоков могут ждать один и тот же объект. Извещение notify возобновляет тот поток, который ждет дольше всех. Если необходимо возобновить все ожидающие потоки, используйте метод notifyAll.

Приводимый ниже класс реализует концепцию очереди. Он содержит методы, которые используются для помещения элементов в очередь и их удаления:

```
class Queue {
        // первый и последний элементы очереди
     Element head, tail;

     public synchronized void append(Element p) {
          if (tail == null)
               head = p;
          else
               tail.next = p;
          p.next = null;
          tail = p;
          notify(); // сообщить ожидающим потокам о новом элементе
     }

     public synchronized Element get() {
          try {
               while(head == null)
                    wait();  // ожидать появления элемента
          } catch (InterruptedException e) {
               return;
          }

          Element p = head;    // запомнить первый элемент
          head = head.next;    // удалить его из очереди
          if (head == null)    // проверить, не пуста ли очередь
               tail = null;
          return p;
     }
}
```

Такая реализация очереди во многом напоминает ее воплощение в однопоточной системе. Отличий не так уж много: методы синхронизированы; при занесении нового элемента в очередь происходит извещение ожидающих потоков; вместо того чтобы возвращать null для пустой очереди, метод get ждет, пока какой-нибудь другой поток занесет элемент в очередь. Как занесение, так и извлечение элементов очереди может осуществляться несколькими потоками (а не обязательно одним).

9.4. Подробности, касающиеся wait и notify

Существует три формы wait и две формы notify. Все они входят в класс Object и выполняются для текущего потока:

public final void wait(long timeout) throws InterruptedException

Выполнение текущего потока приостанавливается до получения извещения или до истечения заданного интервала времени timeout. Значение timeout задается в миллисекундах. Если оно равно нулю, то ожидание не прерывается по тайм-ауту, а продолжается до получения извещения.

public final void **wait(long timeout, int nanos)** ••throws InterruptedException

Аналог предыдущего метода с возможностью более точного контроля времени; интервал тайм-аута представляет собой сумму двух параметров: timeout (в миллисекундах) и nanos (в наносекундах, значение в диапазоне 0–999999).

public final void **wait()** throws InterruptedException

Эквивалентно wait(0).

public final void **notify()**

Посылает извещение ровно *одному* потоку, ожидающему выполнения некоторого условия. Потоки, которые возобновляются лишь после выполнения данного условия, могут вызвать одну из разновидностей wait. При этом выбрать извещаемый поток невозможно, поэтому данная форма notify используется лишь в тех случаях, когда вы точно знаете, какие потоки ожидают событий, какие это события и сколько длится ожидание. Если вы не уверены в каком-либо из этих факторов, вероятно, следует воспользоваться методом notifyAll.

public final void **notifyAll()**

Посылает извещения *всем* потокам, ожидающим выполнения некоторого условия. Обычно потоки стоят, пока какой-то другой поток не изменит некоторое условие. Используя этот метод, управляющий условием поток извещает все ожидающие потоки об изменении условия. Потоки, которые возобновляются лишь после выполнения данного условия, могут вызывать одну из разновидностей wait.

Все эти методы реализованы в классе Object. Тем не менее они могут вызываться только из синхронизированных фрагментов, с использованием блокировки объекта, в котором они применяются. Вызов может осуществляться или непосредственно из такого фрагмента, или косвенно — из метода, вызываемого в фрагменте. Любая попытка обращения к этим методам для объектов за пределами синхронизированных фрагментов, для которых действует блокировка, приведет к возбуждению исключения IllegalMonitorState Exception.

9.5. Планирование потоков

Java может работать как на однопроцессорных, так и на многопроцессорных компьютерах, в однопоточных и многопоточных системах, так что в отношении потоков даются лишь общие гарантии. Вы можете быть уверены в том, что исполнимый (runnable) поток с наивысшим приоритетом будет работать и что все потоки с тем же приоритетом получат некоторую долю процессорного времени. Функционирование потоков с низшим приоритетом гарантируется лишь в случае блокировки всех потоков с высшим приоритетом. /Читателю следует отличать блокировку объекта (lock), о которой говорилось выше, от блокировки потока (block). Терминология, сложившаяся в отечественной литературе, может стать источником недоразумений. - Примеч. перев./ На самом деле не исключено, что потоки с низшим приоритетом будут работать и без таких решительных мер, но полагаться на это нельзя.

Поток называется *заблокированным*, если он приостановлен или выполняет заблокированную функцию (системную или функцию потока). В случае блокировки потока Java выбирает исполнимый поток с наивысшим приоритетом (или один из таких потоков, если их несколько) и начинает его выполнение.

Runtime-система Java может приостановить поток с наивысшим приоритетом, чтобы дать поработать потоку с тем же приоритетом, — это означает, что все потоки, обладающие наивысшим приоритетом, со временем выполняются. Тем не менее это вряд ли можно считать серьезной гарантией, поскольку “со временем” — понятие растяжимое. Приоритетами следует пользоваться лишь для того, чтобы повлиять на политику планирования для повышения эффективности. Не стоит полагаться на приоритет потоков, если от этого зависит правильность работы алгоритма.

Начальный приоритет потока совпадает с приоритетом того потока, который создал его. Для установки приоритета используется метод setPriority с аргументом, значение которого лежит между константами MIN_PRIORITY и MAX_PRIORITY класса Thread. Стандартный приоритет для потока по умолчанию равен NORM_PRIORITY. Приоритет выполняемого потока может быть изменен в любой момент. Если потоку будет присвоен приоритет ниже текущего, то система может запустить другой поток, так как исходный поток может уже не обладать наивысшим приоритетом. Метод getPriority возвращает приоритет потока.

В общем случае постоянно работающая часть вашего приложения должна обладать более низким приоритетом, чем поток, занятый обработкой более редких событий — например, ввода информации пользователем. Скажем, когда пользователь нажимает кнопку с надписью STOP, он ждет, что приложение немедленно остановится. Если обновление изображения и ввод информации осуществляются с одинаковым приоритетом и во время нажатия кнопки происходит вывод, на то, чтобы поток ввода смог среагировать на нажатие кнопки, может потребоваться некоторое время. Даже несмотря на то, что поток вывода обладает более низким приоритетом, он все равно будет выполняться большую часть времени, поскольку поток пользовательского интерфейса будет заблокирован в ожидании ввода. С появлением введенной информации поток пользовательского интерфейса заставит поток вывода среагировать на запрос пользователя. По этой причине приоритет потока, который должен выполняться постоянно, устанавливается равным MIN_PRIORITY, чтобы он не поглощал все доступное процессорное время.

Несколько методов класса Thread управляют планировкой потоков в системе:

public static void sleep(long millis) throws InterruptedException

Приостанавливает работу текущего потока как минимум на указанное число миллисекунд. “Как минимум” означает, что не существует гарантий возобновления работы потока точно в указанное время. На время возобновления может повлиять планировка потоков в системе, гранулярность и точность системных часов, а также ряд других факторов.

public static void **sleep(long millis, int nanos)** •• throws InterruptedException

Приостанавливает работу текущего потока как минимум на указанное число миллисекунд и дополнительное число наносекунд. Значение интервала в наносекундах лежит в диапазоне 0–999999.

public static void **yield()**

Текущий поток передает управление, чтобы дать возможность работать и другим исполняемым потокам. Планировщик потоков выбирает новый поток среди исполняемых потоков в системе. При этом может быть вызван поток, только что уступивший управление, если его приоритет окажется самым высоким.

Приведенный ниже пример демонстрирует работу yield. Приложение получает список слов и создает потоки, предназначенные для вывода отдельного слова в списке. Первый параметр приложения определяет, должен ли каждый поток передавать управление после каждого вызова println; значение второго параметра равно количеству повторений слова при выводе. Остальные параметры представляют собой слова, входящие в список:

```
class Babble extends Thread {
     static boolean doYield; // передавать управление другим потокам?
     Static int howOften;    // количеств повторов при выводе
     String word;            // слово
     Babble(String whatToSay) {
          word = whatToSay;
     }

     public void run() {
          for (int i = 0; i << howOften; i++) {
               System.out.println(word);
               if (doYield)
                    yield(); // передать управление другому потоку
          }
     }

     public static void main(String[] args) {
          howOften = Integer.parseInt(args[1]);
          doYield = new Boolean(args[0]).booleanValue();

          // создать поток для каждого слова и присвоить ему
          // максимальный приоритет
          Thread cur = currentThread();
          cur.setPriority(Thread.MAX_PRIORITY);
          for (int i = 2; i << args.length; i++)
               new babble(args[i]).start();
     }
}
```

Когда потоки работают, не передавая управления друг другу, им отводятся большие кванты времени — обычно этого бывает достаточно, чтобы закончить вывод в монопольном режиме. Например, при запуске программы с присвоением doYield значения false:

Babble false 2 Did DidNot

результат будет выглядеть следующим образом:

Did

Did

DidNot

DidNot

Если же каждый поток передает управление после очередного println, то другие потоки также получат возможность работать. Если присвоить doYield значение true:

Babble true 2 Did DidNot

то остальные потоки также смогут выполняться между очередными выводами и, в свою очередь, будут уступать управление, что приведет к следующему:

Did

DidNot

Did

DidNot

Приведенные выше результаты являются приблизительными. При другой реализации потоков они могут быть другими, хотя даже при одинаковой реализации разные запуски программы могут дать разные результаты. Однако при любой реализации вызов yield повышает шансы других потоков в споре за процессорное время.

9.6. Взаимная блокировка

Если вы имеете дело с двумя потоками и с двумя блокируемыми объектами, может возникнуть ситуация *взаимной блокировки (deadlock)*, при которой каждый объект дожидается снятия блокировки с другого объекта. Представим себе, что объект X содержит синхронизированный метод, внутри которого вызывается синхронизированный метод объекта Y, который, в свою очередь, также содержит синхронизированный метод для вызова синхронизированного метода объекта X. Каждый объект ждет, пока с другого объекта не будет снята блокировка, и в результате ни один из них не работает. Подобная ситуация иногда называется “смертельными объятиями” (deadly embrace). Рассмотрим сценарий, в соответствии с которым объекты jareth и cory относятся к некоторому классу Friend ly:

1. Поток 1 вызывает синхронизированный метод jareth.hug. С этого момента поток 1 осуществляет блокировку объекта jareth.

2. Поток 2 вызывает синхронизированный метод cory.hug. С этого момента поток 2 осуществляет блокировку объекта cory.

3. Теперь cory.hug вызывает синхронизированный метод jareth.hugBack. Поток 1 блокируется, поскольку он ожидает снятия блокировки с cory (в настоящее время осуществляемой потоком 2).

4. Наконец, jareth.hug вызывает синхронизированный метод cory.hugBack. Поток 2 также блокируется, поскольку он ожидает снятия блокировки с jareth (в настоящее время осуществляемой потоком 1).

Возникает взаимная блокировка — cory не работает, пока не снята блокировка с jareth, и наоборот, и два потока навечно застряли в тупике.

Конечно, вам может повезти, и один из потоков завершит весь метод hug без участия второго. Если бы этапы 3 и 4 следовали бы в другом порядке, то объект jareth выполнил бы hug и hugBack еще до того, как cory понадобилось бы заблокировать jareth. Однако в будущих запусках того же приложения планировщик потоков мог бы сработать иначе, приводя к взаимной блокировке. Самое простое решение заключается в том, чтобы объявить методы hug и hugBack несинхронизированными и синхронизировать их работу по одному объекту, совместно используемому всеми объектами Friendly. Это означает, что в любой момент времени во всех потоках может выполняться ровно один метод hug — опасность взаимной блокировки при этом исчезает. Благодаря другим, более хитроумным приемам удается одновременно выполнять несколько hug без опасности взаимной блокировки.

Вся ответственность в вопросе взаимной блокировки возлагается на вас. Java не умеет ни обнаруживать такую ситуацию, ни предотвращать ее. Подобные проблемы с трудом поддаются отладке, так что предотвращать их надо на стадии проектирования. В разделе “Библиография” приведен ряд полезных ссылок на книги, посвященные вопросу проектирования потоков и решению проблем блокировки.

9.7. Приостановка потоков

Поток может быть *приостановлен (suspended)*, если необходимо быть уверенным в том, что он возобновится лишь с вашего разрешения. Для примера допустим, что пользователь нажал кнопку C ANCEL во время выполнения длительной операции. Работу следует приостановить до того момента, когда пользователь подтвердит (или нет) свое решение. Фрагмент программы может выглядеть следующим образом:

```
Thread spinner;   //поток, выполняющий обработку

public void userHitCancel() {
     spinner.suspend();              // приостановка
     if (askYesNo("Really Cancel?"))
          spinner.stop();            // прекращение операции
     else
          spinner.resume();          // передумал!
}
```

Метод userHitCancel сначала вызывает suspend для потока, выполняющего операцию, чтобы остановить его вплоть до вашего распоряжения. Затем пользователь должен ответить, действительно ли он хочет отменить операцию. Если да, то метод stop снимает поток; в противном случае метод resume возобновляет работу потока.

Приостановка ранее остановленного потока, а также возобновление работы потока, который не был приостановлен, не приводит ни к каким нежелательным последствиям.

9.8. Прерывание потока

В некоторых методах класса Thread упоминается *прерывание (interrupting)* потока. Соответствующие методы зарезервированы для возможности, которая вскоре будет включена в Java. На момент написания этой книги они еще не полностью реализованы; попытка их вызова приводит к возбуждению исключения NoSuchMethodError и уничтожению вызывающего потока. Вполне возможно, что к тому моменту, когда вы будете читать эту книгу, эти методы уже будут реализованы. В данном разделе приводится их краткий обзор.

Концепция “прерывания” оказывается полезной, если выполняемому потоку необходимо предоставить некоторую степень контроля над моментом обработки события. Например, в цикле вывода может понадобиться информация из базы данных, извлекаемая посредством транзакции; если при этом поступает запрос на прекращение работы, желательно дождаться нормального завершения транзакции. Поток пользовательского интерфейса может реализовать такой запрос, прерывая поток вывода и давая ему возможность дождаться конца транзакции. Подобная схема будет хорошо работать лишь в том случае, если поток вывода “хорошо себя ведет” и в конце каждой транзакции проверяет, не поступил ли запрос на прерывание (и прекращает работу в этом случае).

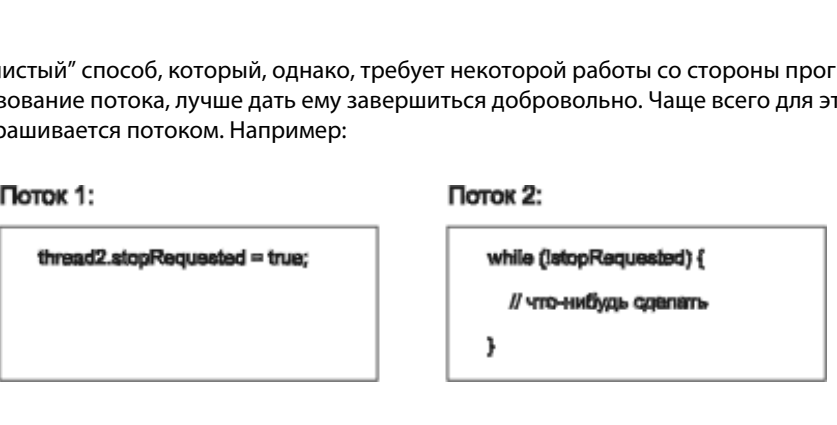
Прерывание потока в общем случае не должно влиять на его работу, однако некоторые методы (такие, как sleep или wait) возбуждают исключение InterruptedException. Если в вашем потоке во время прерывания выполнялся один из таких методов, то будет возбуждено прерывание Interrupted Exception.

Для работы с прерываниями используются несколько методов. Метод interrupt посылает прерывание в поток; метод isInterrupted проверяет факт прерывания потока; статический метод interrupted проверяет, прерывался ли текущий поток.

9.9. Завершение работы потока

Работа потока прекращается, когда происходит выход из его метода run. Так происходит нормальное завершение потока, но вы можете остановить поток и по-другому.

Желательно использовать самый “чистый” способ, который, однако, требует некоторой работы со стороны программиста: вместо того чтобы насильственно прекращать существование потока, лучше дать ему завершиться добровольно. Чаще всего для этого используют логическую переменную, значение которой опрашивается потоком. Например:



```
thread2.stopRequested = true;
```

```
while (!stopRequested) {
    // что-нибудь сделать
}
```

Самый прямолинейный способ завершить поток — вызвать его метод stop, который запустит объект ThreadDeath, указав ему в качестве цели нужный поток. ThreadDeath является подклассом класса Error, а не Exception (объяснение того, почему так было сделано, приводится в приложении Б). Программистам не следует перехватывать ThreadDeath, если только они не должны выполнить какие-нибудь *чрезвычайно неординарные* завершающие действия, с которыми не справится finally. Если уж вы перехватываете ThreadDeath, обязательно возбудите объект-исключение заново, чтобы поток мог “умереть”. Если же ThreadDeath не перехватывается, то обработчик ошибок верхнего уровня просто уничтожает поток, не выводя никаких сообщений.

Поток также может возбудить ThreadDeath для самого себя, чтобы завершить свою собственную работу. Это может пригодиться, если поток углубился на несколько уровней ниже метода run и вам не удается легко сообщить run о том, что пора заканчивать.

Другой форме метода stop можно вместо ThreadDeath передать какое-то другое исключение. Хотя обычно возбуждение исключений оказывается не самым лучшим способом для обмена информацией между потоками, вы можете использовать эту форму общения для того, чтобы послать потоку какое-то сообщение. Например, если некоторый поток выполняет длительные вычисления для определенных входных значений, то интерфейсный поток может разрешить пользователю изменить эти значения прямо во время вычислений. Конечно, вы можете просто завершить поток и начать новый. Тем не менее, если промежуточные результаты вычислений могут использоваться повторно, то вместо завершения потока можно создать новый тип исключения Restart Calculation и воспользоваться методом stop, чтобы запустить новое исключение в поток. При этом поток должен перехватить исключение, рассмотреть новые входные значения, по возможности сохранить результаты и возобновить вычисления.

Один поток может ожидать завершения другого потока. Для этого применяется один из методов join. Простейшая форма этого метода ждет завершения определенного потока:

```
class CalcThread extends Thread {
     private double Result;

     public void run() {
          Result = calculate();
     }

     public double result() {
          return Result;
     }

     public double calculate() {
          // ...
     }
}

class join {
     public static void main(String[] args) {
          CalcThread calc = new CalcThread();
          calc.start();
          doSomethingElse();
          try {
               calc.join();
               System.out.println("result is "
                    + calc.result());
          } catch (InterruptedException e) {
               System.out.println("No answer: interrupted");
          }
     }

}
```

Сначала создается новый тип потока, CalcThread, выполняющий некоторые вычисления. Мы запускаем поток, некоторое время занимаемся другими делами, после чего пытаемся присоединиться (join) к потоку. На выходе из join можно быть уверенным, что метод CalcThread.run завершился, а значение Result получено. Это сработает независимо от того, окончился ли поток CalcThread до doSomethingElse или нет. Когда поток завершается, его объект никуда не исчезает, так что вы можете к нему обращаться.

При вызове других форм join им передаются интервалы тайм-аута, подобные тем, какие используются для метода sleep. Имеются три формы join:

public final void **join()** throws InterruptedException

Ожидает безусловного завершения потока, для которого вызывается метод.

public final synchronized void **join(long millis)** •• throws InterruptedException

Ожидает завершения потока или истечения заданного числа миллисекунд (в зависимости от того, что произойдет раньше). Аргумент, равный нулю, означает ожидание без тайм-аута.

public final synchronized void **join(long millis, int nanos)** •• throws InterruptedException

Ожидает завершения потока или тайм-аута с более точным контролем времени. Суммарное время тайм-аута, равное 0 наносекунд, снова означает ожидание без тайм-аута. Количество наносекунд находится в диапазоне 0–999999.

Вызов метода destroy для потока — самая решительная мера. Этот метод уничтожает поток без выполнения нормальных завершающих действий, к которым относится и снятие блокировки со всех объектов потока, так что применение destroy может навечно заблокировать другие потоки. По возможности старайтесь избегать вызова destroy.

9.10. Завершение приложения

Работа каждого приложения начинается с запуска одного потока — того, в котором выполняется метод main. Если ваше приложение не создает других потоков, то после выхода из main оно завершается. Но давайте предположим, что в приложении возникают другие потоки; что произойдет с ними после выхода из main?

Существует две разновидности потоков: потоки-пользователи (users) и потоки-демоны (daemons). Наличие оставшихся потоков-пользователей приводит к продолжению работы приложения, тогда как потоки-демоны могут уничтожаться. После снятия последнего потока-пользователя происходит закрытие всех потоков-демонов, и работа приложения на этом заканчивается. Для пометки потока-демона применяется метод setDaemon(true), а метод get Daemon проверяет значение соответствующего флага. По умолчанию “демонический” статус потока совпадает со статусом его потока-создателя. После того как поток начнет выполняться, изменить данное свойство невозможно; при попытке сделать это возбуждается исключение IllegalThreadState Exception.

Если новый поток порождается в методе main, то он наследует от своего создателя статус потока-пользователя. После завершения main приложение будет выполняться до того, как завершится и этот порожденный поток. В исходном потоке нет ничего особенного — просто он оказывается первым при конкретном запуске приложения. После этого такой поток ничем не отличается от всех остальных потоков. Приложение работает до тех пор, пока не будут завершены *все* его потоки-пользователи. С точки зрения runtime-системы, исходный поток создается лишь для того, чтобы породить другой поток и умереть, предоставляя порожденному потоку выполнять всю работу. Если вы хотите, чтобы ваше приложение завершалось вместе с завершением исходного потока, необходимо помечать все создаваемые потоки как потоки-демоны.

9.11. Использование Runnable

В интерфейсе Runnable абстрагируется концепция некой сущности, выполняющей программу во время своей активности. Интерфейс Runnable объявляет всего один метод:

public void run();

Класс Thread реализует интерфейс Runnable, поскольку поток как раз и является такой сущностью — во время его активности выполняется программа. Мы уже видели, что для осуществления каких-то особых вычислений можно расширить класс Thread, однако во многих случаях это не слишком просто. Прежде всего, расширение классов производится на основе одиночного наследования — если некоторый класс расширяется для того, чтобы он мог выполняться в потоке, то одновременно расширить и его, и Thread не удастся. Кроме того, если вам нужна только возможность выполнения, то вряд ли вы захотите наследовать и все накладные расходы, связанные с Thread.

Во многих случаях проще реализовать Runnable. Объект Runnable может выполняться в отдельном потоке — для этого следует передать его конструктору Thread. Если объект Thread конструируется с объектом Runnable, то реализация Thread.run вызывает метод run переданного объекта.

Приведем версию класса PingPong, в которой используется интерфейс Runnable. Сравнение этих двух версий показывает, что они выглядят почти одинаково. Наиболее существенные отличия заключаются в супертипе (Runnable вместо Thread) и методе main:

```
class RunPingPong inplements Runnable {
     String word;                // выводимое слово
     int delay;                  // длительность паузы
     PingPong(String whatToSay, int delayTime) {
          word = whatToSay;
          delay = delayTime;
     }

     public void run() {
          try {
               for (;;) {
                    System.out.print(word + " ");
                    Thread.sleep(delay); // подождать следующего
                                         // вывода
               }
          } catch (InterruptedException e) {
               return;                     // завершить поток
          }
     }

     public static void main(String[] args) {
          Runnable ping = new RunPingPong("ping",  33);
          Runnable pong = new RunPingPong("PONG", 100);
     }
}
```

Сначала определяется новый класс, реализующий интерфейс Runnable. Код метода run в этом классе совпадает с его реализацией в классе PingPong. В методе main создаются два объекта RunPingPong с разными временными интервалами; затем для каждого из них создается и немедленно запускается новый объект Thread.

Существует четыре конструктора Thread, которым передаются объекты Runnable:

public **Thread(Runnable target)**

Конструирует новый объект Thread, использующий метод run указанного класса target.

public **Thread(Runnable target, String name)**

Конструирует новый объект Thread с заданным именем name, использующий метод run указанного класса target.

public **Thread(ThreadGroup group, Runnable target)**

Конструирует новый объект Thread, входящий в заданную группу ThreadGroup и использующий метод run указанного класса target.

public Thread(ThreadGroup group, Runnable target, String name)

Конструирует новый объект Thread с заданным именем name, входящий в заданную группу ThreadGroup и использующий метод run указанного класса target.

9.12. Ключевое слово volatile

Механизм синхронизации помогает в решении многих проблем, однако, если вы откажетесь от его использования, сразу несколько потоков смогут одновременно изменять значение некоторого поля. Если это делается намеренно (может быть, для синхронизации доступа используются другие средства), следует объявить поле с ключевым словом volatile. Например, если у вас имеется переменная, значение которой постоянно отображается потоком графического вывода и может изменяться несинхронизированными методами, то фрагмент вывода может выглядеть следующим образом:

```
currentValue = 5;
for (;;) {
        display.showValue(currentValue);
        Thread.sleep(1000); // подождать 1 секунду
}
```

Если бы значение currentValue не могло изменяться внутри метода ShowValue, то компилятор мог бы предположить, что величина current Value остается в цикле постоянной, и просто использовать константу 5 вместо вызова showValue.

Однако, если во время выполнения цикла значение currentValue может быть изменено другим потоком, то предположение компилятора будет неверным. Объявление поля currentValue с ключевым словом volatile не позволяет компилятору делать подобные предположения.

9.13. Безопасность потоков и ThreadGroup

При программировании работы нескольких потоков (часть которых создается библиотечными вызовами) бывает полезно отчасти ограничить их возможности, чтобы потоки не мешали друг другу.

Потоки делятся на *группы потоков* в целях безопасности. Группа потоков может входить в состав другой группы, что позволяет создавать определенную иерархию. Потоки внутри группы могут изменять другие потоки, входящие в ту же группу, а также во все группы, расположенные ниже в иерархии. Поток не может модифицировать потоки за пределами своей собственной и всех подчиненных групп.

Эти ограничения используются для защиты потоков от произвола со стороны других потоков. Если новые потоки помещаются в отдельную группу внутри уже существующей группы, то приоритеты порожденных потоков могут изменяться, однако новые потоки не смогут повлиять на приоритеты существующих потоков или любых потоков за пределами своей группы.

Каждый поток принадлежит к некоторой группе. Ограничения, накладываемые на потоки, входящие в группу, описываются объектом ThreadGroup. Группа может задаваться в конструкторе потока; по умолчанию каждый новый поток помещается в ту же группу, в которую входит его поток-создатель. После завершения потока соответствующий объект удаляется из группы.

public **Thread(ThreadGroup group, String name)**

Конструирует новый поток с заданным именем name (может быть равно null), принадлежащий конкретной группе.

После того как объект будет создан, вы уже не сможете изменить связанный с ним объект ThreadGroup. Чтобы узнать, какой группе принадлежит некоторый поток, следует вызвать его метод getThreadGroup. Кроме того, можно проверить, допустима ли модификация потока, — для этого вызовите его метод checkAccess. Этот метод возбуждает исключение SecurityException, если вы не можете модифицировать поток, и просто завершается в противном случае (метод имеет тип void).

Группы потоков могут быть *группами-демонами*. Такие группы автоматически уничтожаются, когда в них не остается ни одного потока. То, что группа является группой-демоном, никак не влияет на “демонизм” принадлежащих ей потоков или групп. Статус группы-демона определяет лишь то, что происходит с ней, когда группа становится пустой.

Группы потоков также могут использоваться для задания максимального приоритета потоков, входящих в нее. После вызова метода setMaxPriority, задающего максимальный приоритет группы, при любой попытке поднять приоритет потока выше указанного значения происходит его незаметное понижение до объявленного максимума. Вызов этого метода не влияет на существующие потоки. Чтобы быть уверенным в том, что приоритет некоторого потока всегда будет превышать приоритет всех остальных потоков группы, следует установить для него приоритет MAX_PRIORITY, после чего установить максимальный приоритет группы равным MAX_PRIORITY-1. Ограничение относится и к самой группе потоков — при попытке установить для нее максимальный приоритет, превышающий текущее значение, произойдет незаметное понижение затребованного приоритета:

```
static public void maxThread(Thread thr) {
        ThreadGroup grp = thr.getThreadGroup();
        thr.setPriority(Thread.MAX_PRIORITY);
        grp.setMaxPriority(thr.getPriority() - 1);
}
```

Данный метод сначала устанавливает для потока наивысший приоритет, после чего опускает максимально допустимый приоритет группы ниже приоритета этого потока. Новый максимальный приоритет группы устанавливается на единицу меньшим приоритета группы, а не Thread. MAX_PRIORITY-1, поскольку действующий максимум группы может ограничить ваше право задавать для потока приоритет MAX_PRIORITY. Вам нужно, чтобы приоритет потока был наивысшим из возможных в данной группе, а максимальный приоритет группы был ниже него, и при этом неважно, какими будут конкретные значения.

ThreadGroup содержит следующие конструкторы и методы:

public **ThreadGroup(String name)**

Создает новую группу ThreadGroup, принадлежащую группе ThreadGroup текущего потока. Имена групп, как и имена потоков, не используются runtime-системой. Если имя равно null, возбуждается исключение Null PointerException. Этим объекты ThreadGroup отличаются от объектов Thread, у которых наличие имени необязательно.

public **ThreadGroup(ThreadGroup parent, String name)**

Создает новую группу ThreadGroup с заданным именем, принадлежащую указанной группе ThreadGroup. Как и в других конструкторах, наличие имени является обязательным.

public final String **getName()**

Возвращает имя группы ThreadGroup.

public final ThreadGroup **getParent()**

Возвращает родительскую группу ThreadGroup или null, если ее не существует.

public final void **setParent(boolean daemon)**

Устанавливает “демонический” статус группы.

public final boolean **isDaemon()**

Возвращает “демонический” статус группы.

public final synchronized void **setMaxPriority(int maxPri)**

Устанавливает максимальный приоритет группы.

public final int **getMaxPriority()**

Возвращает текущий максимальный приоритет группы.

public final boolean **parentOf(ThreadGroup g)**

Проверяет, является ли текущая группа родителем группы g или же совпадает с ней. Лучше представлять себе этот метод в терминах “является частью”, так как группа является частью самой себя.

public final void **checkAccess()**

Возбуждает исключение SecurityException, если текущий поток не имеет права на модификацию группы. В противном случае метод просто завершается.

public final synchronized void **destroy()**

Уничтожает текущую группу типа ThreadGroup. Группа, в которой содержатся потоки, не может быть уничтожена; при попытке сделать это возбуждается исключение IllegalThreadStateException. Это означает, что метод destroy не может применяться для уничтожения всех потоков группы — это необходимо сделать вручную, воспользовавшись описанными ниже методами перечисления. Если в группу входят другие группы, то они также должны быть пустыми.

Для просмотра содержимого группы используются два параллельных набора методов: один из них служит для получения информации о потоках, а другой — о группах потоков, принадлежащих данной группе. Пример использования этих методов можно найти в методе safeExit.

public synchronized int **activeCount()**

Возвращает примерное количество активных потоков в группе, включая потоки, содержащиеся в подгруппах. Значение будет лишь примерным, поскольку к моменту его получения количество активных потоков может измениться; во время вызова activeCount одни потоки могут завершиться, а другие — начать работу.

public int enumerate(Thread[] threadsInGroup, boolean recurse)

Заполняет массив threadsInGroup ссылками на все активные потоки в группе до заполнения массива. Если значение recurse равно false, то перечисляются лишь потоки, непосредственно входящие в группу; если же оно равно true, то перечисляются все потоки в иерархии. Thread Group.enumerate, в отличие от ThreadGroup.activeCount, позволяет определить, включаете ли вы потоки в подгруппах или нет. Это значит, что вы можете получить разумную оценку для размера массива, необходимого для хранения результатов рекурсивного перечисления, однако для перечисления, не учитывающего подгрупп, такая оценка окажется завышенной.

public int **enumerate(Thread[] threadsInGroup)**

Эквивалентно enumerate(threadsInGroup, true).

public synchronized int **activeGroupCount()**

Аналогичен методу activeCount, однако подсчитывает не потоки, а группы, в том числе и во всех подгруппах. “Активный” (active) в данном случае означает “существующий”. Неактивных групп не бывает; термин используется лишь для соблюдения единого стиля с activeCount.

public int **enumerate(ThreadGroup[] groupsInGroup, ••** boolean recurse)

Аналогичен методу enumerate для потоков, однако заполняет массив ссылками на объекты-группы типа ThreadGroup вместо объектов-потоков Thread.

public int **enumerate(ThreadGroup[] groupsInGroup)**

Эквивалентно enumerate(groupsInGroup, true).

Объекты ThreadGroup могут также использоваться для управления потоками, входящими в группу. Перечисленные ниже методы воздействуют на все потоки, входящие в группу и во все ее подгруппы:

public final synchronized void **stop()**

Завершает все потоки в группе и во всех ее подгруппах.

public final synchronized void **suspend()**

Приостанавливает все потоки в группе и во всех ее подгруппах.

public final synchronized void **resume()**

Возобновляет все потоки в группе и во всех ее подгруппах.

Эти методы предоставляют единственную возможность прямого использования объекта ThreadGroup для задания параметров потоков.

В классе Thread также имеется два статических метода для работы с группой, в которую входит текущий поток. Они представляют собой сокращенную запись для последовательного вызова getCurrentThread, getThread Group и вызова метода для найденной группы:

public static int **activeCount()**

Возвращает количество активных потоков в группе, в которую входит текущий поток.

public static int **enumerate(Thread[] tarray)**

Возвращает количество потоков в группе, в которую входит текущий поток.

Класс ThreadGroup также содержит метод, вызываемый при завершении потока, из-за неперехваченного прерывания:

public void **uncaughtException(Thread[] thr, Throwable exc)**

Вызывается при завершении потока, вызванном неперехваченным прерыванием.

Данный метод является открытым, так что вы можете переопределить его для обработки неперехваченных прерываний по своему желанию. Реализация, принятая по умолчанию, вызывает метод uncaughtException группы-родителя, если таковая имеется, или метод Throwable. printStackTrace в противном случае. Например, при разработке графической оболочки было бы желательно отобразить содержимое стека в окне, вместо того чтобы просто вывести его в System.out, как это делает метод printSt a ckTrace. Вы можете переопределить uncaughtException в своей группе, чтобы создать нужное окно и перенаправить в него содержимое стека.

9.14. Отладка потоков

Несколько методов класса Thread предназначены для облегчения отладки многопоточных приложений. Эти средства используются для вывода информации о состоянии программы. Ниже приведен список методов класса Thread, помогающих в процессе отладки:

public String **toString()**

Возвращает строковое описание потока, включающее его имя, приоритет и имя группы.

public String **countStackFrames()**

Возвращает количество кадров стека в потоке.

public static void **dumpStack()**

Выводит в System.out содержание стека для текущего потока.

Также существует ряд отладочных средств для отслеживания состояния группы потоков. Следующие методы вызываются для объектов ThreadGroup и выдают информацию об их состоянии:

public String **toString()**

Возвращает строковое описание группы, включающее ее имя и приоритет.

public synchronized void **list()**

Выводит в System.out список содержимого группы и ее подгрупп.

Содержание | Далее

Глава 10
ПАКЕТЫ

Библиотека — это арсенал свободы.
Источник неизвестен

Под понятием “пакет” подразумевается объединение взаимосвязанных классов, интерфейсов и подпакетов. Концепция пакета оказывается полезной по нескольким причинам:

- Пакеты позволяют группировать родственные интерфейсы и классы.
- В интерфейсах и классах, входящих в пакет, могут использоваться популярные имена (вроде get или put), которые имеют смысл в данном контексте, но конфликтуют с теми же именами в других пакетах.
- Пакет может включать типы и члены, с которыми можно работать только в пределах данного пакета. Соответствующие идентификаторы доступны для программ пакета, но закрыты для внешних методов.

Давайте рассмотрим пример пакета для нашего класса атрибутов, использованного в предыдущих главах. Назовем пакет attr. Каждый исходный файл, классы и интерфейсы которого принадлежат пакету attr, должен указывать на свою принадлежность к пакету объявлением package:

```
package attr;
```

Тем самым объявляется, что все классы и интерфейсы, определенные в этом исходном файле, являются частью пакета attr. Объявление package должно находиться в самом начале файла, до любых объявлений классов или интерфейсов; в файле может присутствовать всего одно объявление package. Имя пакета является неявным префиксом для всех имен типов, включенных в пакет.

Если фрагменту программы вне пакета понадобится обратиться к типам, входящим в пакет, у него имеется две возможности. Первая — указывать перед каждым именем типа префикс (имя пакета). Такой вариант будет вполне разумен, если вам приходится иметь дело всего с несколькими членами пакета.

Другая возможность доступа к типам пакета заключается в частичном или полном *импортировании* пакета. Программист, который захочет воспользоваться пакетом attr, может вставить следующую строку в начало своего исходного файла (после своего объявления package, но перед всем прочим):

```
import attr.*;
```

После этого он может обращаться к типам пакета просто по имени — например, Attributed. Пакет неявно импортирует сам себя, так что все, что определено в нем, становится доступным для всех остальных типов пакета.

Механизмы package и import помогают программисту предотвращать потенциальные конфликты имен. Если в пакете, предназначенном для других целей (скажем, лингвистических), тоже встретится класс с именем Attributed, предназначенный для хранения языковых атрибутов, то у программиста, пожелавшего использовать оба пакета в одном файле, имеется несколько вариантов:

- Обращаться к типам по их полным именам — например, attr.Attributed и lingua.Attributed.
- Импортировать attr.Attributed или attr.*, после чего использовать простое имя Attributed вместо attr.Attributed и полное имя lingua.Attributed.
- Сделать обратное — импортировать lingua.Attributed или lingua.*, после чего использовать простое имя Attributed вместо lingua.Attributed и полное имя attr.Attributed.

10.1. Имена пакетов

Имя пакета должно использоваться только один раз, так что выбор содержательного и уникального имени составляет важный аспект проектирования пакета. Однако сейчас, когда программисты всей планеты разрабатывают пакеты на языке Java, невозможно выяснить, какие имена пакетов ими используются. Следовательно, выбор уникального имени представляет некоторую проблему. Если вы уверены, что пакет будет использоваться только внутри вашей организации, то можно привлечь к делу выбора имени внутреннего арбитра — это позволит быть уверенным, что все пакеты будут иметь отличающиеся имена.

Тем не менее в нашем огромном мире такой подход не сработает. Идентификаторы пакетов Java представляют собой обычные имена, так что неплохой способ обеспечить их уникальность — включить в них имя домена организации в Internet. Если вы работаете в компании Magic, Inc., то пакет с атрибутами может быть объявлен следующим образом:

```
package COM.magic.attr;
```

Обратите внимание: компоненты имени следуют в обратном порядке по сравнению с обычным именем домена, а имя домена верхнего уровня (в нашем случае COM) написано прописными буквами. Это сделано для того, чтобы избежать конфликтов с именами пакетов, которые не следуют этому соглашению — их названия могут случайно совпасть с именем домена верхнего уровня, но вряд ли при этом они будут написаны прописными буквами.

Если вы следуете данному соглашению, конфликты возникать не будут (разве что внутри вашей организации). Если проблемы все же возникли (скажем, в очень большой организации), можно пойти дальше и уточнить имя домена. Во многих крупных компаниях имеются внутренние домены с именами east, europe и т. д. Вы можете уточнить имя пакета с использованием имени внутреннего домена:

```
package COM.magic.japan.attr;
```

При этом имена пакетов могут стать довольно длинными, однако такая схема относительно безопасна — никто из тех, кто ее использует, не выберет имя, совпадающее с названием вашего пакета.

Во многих средах разработки имена пакетов отражаются на уровне файловой системы — часто требуется, чтобы все программы из одного пакета находились в определенной папке или каталоге, а имя этого каталога соответствовало имени пакета. Подробности можно узнать в документации, относящейся к вашей среде разработки.

10.2. Пакетный доступ

Классы и интерфейсы, входящие в пакет, обладают одним из двух уровней доступа: пакетным (package) и открытым (public). Открытый класс или интерфейс доступен для программ, не входящих в пакет. Типы, не являющиеся открытыми, обладают областью видимости на уровне пакета: они доступны для всех программ того же пакета, но скрыты за его пределами, в том числе даже от программ во вложенных пакетах.

Члены классов тоже обладают уровнем доступа. Член, не объявленный с ключевым словом public, protected или private, может использоваться любой программой, входящей в пакет, но остается невидим за пределами пакета. Другими словами, по умолчанию идентификаторы обладают “пакетным” уровнем доступа, за исключением членов интерфейсов, которые являются открытыми.

Поля или методы, не объявленные в пакете с ключевым словом private, доступны для всех программ этого пакета. Следовательно, классы, входящие в тот же пакет, считаются “дружественными”, или “заслуживающими доверия”. Однако подпакеты не пользуются доверием в своих внешних пакетах. Например, защищенные и пакетные идентификаторы в пакете dit недоступны для программ в пакете dit.dat и наоборот.

10.3. Содержимое пакета

Если в исходном файле отсутствует объявление package, то входящие в него типы считаются принадлежащими к “безымянному” пакету.

К проектированию пакета следует подходить внимательно и включать в него только функционально связанные классы и интерфейсы. Классы пакета могут свободно обращаться к незакрытым членам друг друга. Защита членов класса предназначена для предотвращения некорректных действий со стороны классов, обладающих доступом к деталям внутренней реализации других классов. В пределах пакета не существует никаких ограничений доступа, кроме priv a te, и в итоге может получиться так, что посторонний класс получил к другим классам более близкий доступ, чем вам хотелось бы.

Кроме того, пакет должен составляться исходя из логического разделения задач, чтобы помочь программистам, которые ищут полезные для себя интерфейсы и классы. Если пакет состоит из несвязанных классов, будет сложно понять, что же из этого можно применить в работе. Логическое разделение способствует повторному использованию вашего кода, поскольку программистам будет легче ориентироваться в нем.

Если пакет состоит из взаимосвязанных типов, вы сможете давать им очевидные имена, избегая при этом конфликтов с посторонними типами из этого же пакета.

Пакет может быть составной частью другого пакета. Например, пакет java.lang является вложенным, то есть пакет lang входит в более обширный пакет java. Пакет java не содержит ничего, кроме других пакетов. Вложение позволяет построить иерархическую систему имен для взаимосвязанных пакетов.

Например, чтобы создать набор пакетов для адаптивных систем (скажем, нейросетей или генетических алгоритмов), можно воспользоваться вложенными пакетами, разделяя их имена точками:

```
package adaptive.neuralNet;
```

Исходный файл с таким объявлением входит в пакет adaptive.neuralNet, который, в свою очередь, является подпакетом пакета adaptive. Пакет adaptive может содержать классы, относящиеся к общим адаптивным алгоритмам, — например, методы с общей постановкой генетических проблем или способы измерения каких-то показателей. Каждый пакет, который находится ниже в иерархии (например, adaptive.neuralNet или adaptive. genetic), содержит классы, предназначенные для конкретного типа адаптивных алгоритмов.

Вложение пакетов является средством организации взаимосвязанных пакетов и не имеет отношения к правам доступа. Классы пакета adaptive. genetic не смогут работать с закрытыми на пакетном уровне идентификаторами из adaptive или adaptive.neuralNet. Область видимости пакета не выходит за его пределы. Вложение позволяет группировать взаимосвязанные пакеты и помогает программистам искать классы в иерархической структуре, но не дает никаких других преимуществ.

Содержание | Далее

Глава 11
ПАКЕТ ВВОДА/ВЫВОДА

С точки зрения программиста,
пользователь — это периферийное устройство,
вводящее символы в ответ
на команду read.
Питер Уильямс

Ввод/вывод в Java описывается в терминах потоков. Потоком /К сожалению, в отечественной литературе одним словом "поток" переводятся совершенно разнородные термины thread (см. выше) и stream, что создает определенную двусмысленность. В тех случаях, когда контекст не дает однозначного толкования, для перевода thread используется уточняющий термин "программный поток" - Примеч. перев./ называется упорядоченная последовательность данных, которая имеет источник (входной поток) или приемник (выходной поток). Потоки ввода/вывода избавляют программиста от необходимости вникать в конкретные детали операционной системы и позволяют осуществлять доступ к файлам. В основе работы всех потоков лежит ограниченный набор базовых интерфейсов и абстрактных классов; большинство типов потоков (например, потоки для работы с файлами) поддерживают базовые методы, иногда — с минимальными модификациями. Самый лучший способ освоения ввода/вывода в Java заключается в изучении базовых интерфейсов и абстрактных классов. В качестве примера мы рассмотрим файловые потоки.

Основным исключением из этой модели являются потоки данных, которые читают и записывают значения базовых типов Java, таких как int или string. Эти потоки поддерживают более широкий набор методов, спроектированный с учетом их специфики. Они рассматриваются во второй части этой главы, начиная с раздела 11.16.

Пакет ввода/вывода в Java называется java.io. Он импортируется во всех листингах этой главы, даже если строка import и не входит в пример.

На момент написания книги не существовало стандартной библиотеки для манипуляций с выходными числовыми или строковыми форматами — скажем, задания минимальной или максимальной ширины или желательной точности для числа с плавающей точкой.

11.1. Потоки

В пакете java.io определяется несколько абстрактных классов для базовых входных и выходных потоков. Затем эти абстрактные классы расширяются, и на их основе создаются некоторые полезные типы потоков. Потоки почти всегда являются парными: если существует FileInputStream, то есть и FileOutputStream.

Кроме того, существуют классы для работы с именами файлов, класс потока с возможностью чтения/записи с именем RandomAccessFile и анализатор для деления входного потока на отдельные лексемы.

Класс IOException используется многими методами java.io для сигнализации об исключительных состояниях. Некоторые классы, являющиеся расширениями IOException, сообщают о конкретных проблемах, однако в большинстве случаев все же применяются объекты IOException со строкой-описанием. Подробности приведены в разделе 11.20.

Перед тем как рассматривать конкретные виды входных и выходных потоков, следует ознакомиться с базовыми абстрактными классами InputStream и OutputStream. Иерархия типов пакета java.io изображена на рис. 11.1.

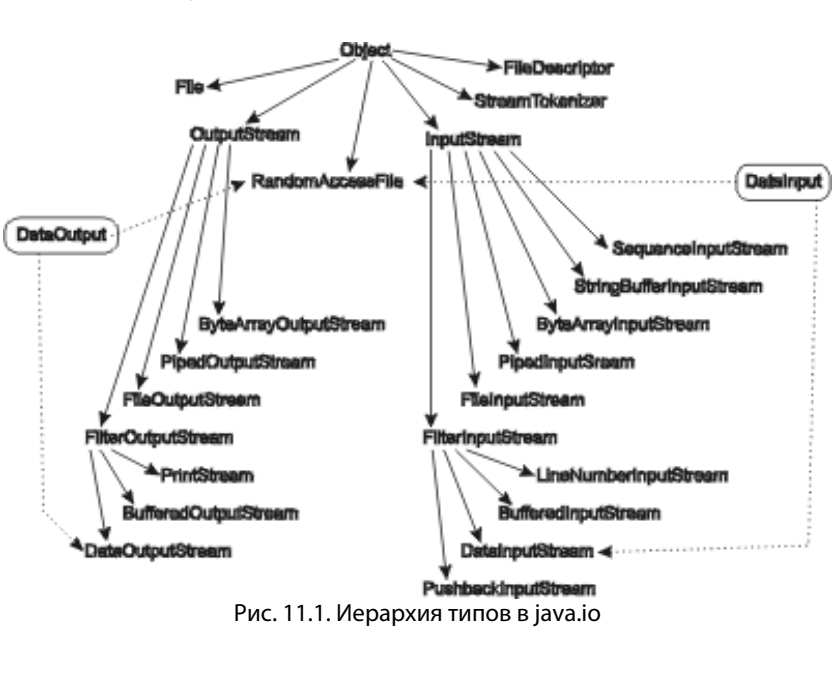


Рис. 11.1. Иерархия типов в java.io

11.2. Класс InputStream

В абстрактном классе InputStream объявляются методы для чтения из заданного источника. InputStream является базовым классом для большинства входных потоков в java.io и содержит следующие методы:

public **InputStream()**

Класс InputStream содержит только безаргументный конструктор.

public abstract int **read()** throws IOException

Читает из потока один байт данных и возвращает прочитанное значение, лежащее в диапазоне от 0 до 255 (не от –128 до 127). При достижении конца потока возвращается флаг –1. Метод блокирует работу программы до появления значения на входе.

public int **read(byte[] buf)** throws IOException

Читает данные в массив байтов. Метод блокирует работу программы до появления вводимого значения, после чего заполняет buf всеми прочитанными байтами, в количестве не более buf.length. Метод возвращает фактическое количество прочитанных байтов или –1 при достижении конца потока.

public int read(byte[] buf, int off, int len) throws IOException

Читает данные в байтовый подмассив. Метод блокирует работу программы до начала ввода, после чего заполняет часть массива buf, начиная со смещения off, в количестве до len байтов, если не встретится конец массива buf.

public long **skip(long count)** throws IOException

Пропускает до count байтов во входном потоке. Количество пропущенных байтов может быть меньше count из-за достижения конца потока. Возвращает фактическое количество пропущенных байтов.

public int **available()** throws IOException

Возвращает количество байтов, которые могут быть прочитаны без блокировки работы программы.

public void **close()** throws IOException

Закрывает входной поток. Метод должен вызываться для освобождения любых ресурсов (например, файловых дескрипторов), связанных с потоком. Если не сделать это, то ресурсы будут считаться занятыми, пока сборщик мусора не вызовет метод finalize данного потока.

Приведенная ниже программа подсчитывает общее количество символов и разделителей (white-space characters) в файле:

```
import java.io.*;

class CountSpace {
     public static void main(String[] args)
          throws IOException
     {
          InputStream in;
          if (args.length == 0)
               in = System.in;
          else
               in = new FileInputStream(args[0]);
          int ch;
          int total;
          int spaces = 0;
          for (total = 0; (ch = in.read()) != -1; total++) {
               if (Character.isSpace((char)ch)
                    spaces++;
          }

          System.out.println(total + " chars, "
               + spaces + " spaces");
     }
}
```

Программа либо берет имя файла из командной строки, либо читает данные из стандартного входного потока, System.in. Входной поток представлен переменной in. Если имя файла не задано, используется стандартный входной поток; если же оно указано, то создается объект FileInputStream, являющийся расширением InputStream.

Цикл for подсчитывает как общее количество символов в файле, так и количество символов-разделителей; для идентификации последних применяется метод isSpace класса Character. В конце происходит вывод результатов. Вот как они выглядят, если программа используется с файлом, содержащим ее собственный исходный текст:

434 chars, 109 spaces

Возможно, вам захочется присвоить значение total с помощью метода available, однако для потока System.in такой вариант не сработает. Метод available возвращает количество байтов, которые могут быть прочитаны *без блокировки*. Для файла оно обычно представляет собой его длину. Если же поток System.in будет связан с клавиатурой, то возвращаемое методом значение может быть равно нулю; если необработанные символы ввода отсутствуют, то следующий вызов read приведет к блокировке.

11.3. Класс OutputStream

Абстрактный класс OutputStream во многих отношениях напоминает InputStream; он абстрагирует поток байтов, направляемых в приемник. Класс содержит следующие методы:

public **OutputStream()**

Класс OutputStream содержит только безаргументный конструктор.

public abstract void **write(int b)** throws IOException

Записывает в поток байт b. Байт передается в виде значения int, поскольку он часто является результатом арифметической операции над байтами. Выражения, в которых входят данные типа byte, имеют тип int, так что параметр типа int позволяет использовать результат без преобразования в byte. Тем не менее обратите внимание на то, что передаются только младшие 8 бит значения int — старшие 24 бита при этом теряются. Метод блокирует работу программы до завершения записи байта.

public void **write(byte[] buf)** throws IOException

Записывает в поток содержимое массива байтов. Метод блокирует работу программы до завершения записи.

public void **write(byte[] buf, int offset, int len)** •• throws IOException

Записывает в поток часть массива байтов, которая начинается с buf [offset] и насчитывает до count байтов, если ранее не будет встречен конец массива.

public void **flush()** throws IOException

Очищает поток, то есть направляет в него все байты, находящиеся в буфере.

public void **close()** throws IOException

Закрывает поток. Метод должен вызываться для освобождения любых ресурсов, связанных с потоком.

Если явно не указывается противное, то при обнаружении ошибки в выходном потоке все эти методы возбуждают исключение IOException.

Ниже приводится приложение, копирующее свой входной поток в выходной и попутно заменяющее некоторые символы. Приложение Translate получает два параметра: строку from и строку to. Если во входном потоке встречается символ, входящий в строку from, он заменяется символом строки to, находящимся в той же позиции:

```
import java.io.*;

class Translate {
     public static void main(String[] args) {
          InputStream in = System.in;
          OutputStream out = System.out;

          if (args.length != 2)
               error ("must provide from/to arguments");

          String from = args[0], to = args[1];
          int ch, i;

          if (from.length() != to.length())
               error ("from and to must be same length");

          try {
               while ((ch = in.read()) != 1) {
                    if ((i = from.indexOf(ch)) != -1)
                         out.write(to.charAt(i));
                    else
                         out.write(ch);
               }
          } catch (IOException e) {
               error ("I/O Exception: " + e);
          }
     }

     public static void error(String err) {
          System.err.print("Translate: " + err);
          System.exit(1);  // ненулевое значение означает
                           // неблагополучное завершение
     }
}
```

Упражнение 11.1

Перепишите приведенную выше программу Translate в виде метода, который пересылает символы из InputStream в OutputStream, а метод трансляции (правило замены символов) и потоки являются параметрами. Для каждого типа InputStream и OutputStream, о которых говорилось в этой главе, напишите новый метод main, в котором бы учитывалась возможность трансляции символов при вводе или выводе. Если потоки ввода и вывода оказываются симметричными, то для них может применяться общий метод main.

11.4. Стандартные типы потоков

Как видно из рис. 11.1, в пакете java.io определяются несколько типов потоков. Обычно они составляют пары ввода/вывода:

- Конвейерные потоки Piped спроектированы для парного использования, при котором байты, записываемые в PipedOutputStream, могут читаться из PipedInputStream.
- Байтовые потоки ByteArray осуществляют ввод/вывод в массив байтов.
- Фильтрующие потоки Filtered представляют собой абстрактные классы байтовых потоков, в которых с читаемыми байтами выполняются некоторые операции-фильтры. Объект FilterInputStream получает ввод от другого объекта InputStream, некоторым образом обрабатывает (фильтрует) байты и возвращает результат. Фильтрующие потоки могут объединяться в последовательности, при этом несколько фильтров превращаются в один сквозной фильтр. Аналогичным образом осуществляется и фильтрация вывода — для этого применяются различные классы Filter OutputStream.
- Буферизованные потоки Buffered расширяют понятие фильтрующих потоков, добавляя буферизацию, чтобы при каждом вызове read и write не приходилось обращаться к файловой системе.
- Потоки данных Data разделяются на две категории. Интерфейсы Data Input и DataOutput определяют методы для чтения и записи данных встроенных типов, причем вывод одного из них воспринимается в качестве ввода другого. Эти интерфейсы реализуются классами DataInputStream и Data OutputStream.
- Файловые потоки File расширяют понятие фильтрующих потоков — байтовый поток в них связывается с определенным файлом. В них встроены некоторые методы, относящиеся к работе с файлами.

В пакет также входит ряд потоков ввода (вывода), для которых отсутствуют парные им потоки вывода (ввода):

- Поток SequenceInputStream преобразует последовательность объектов InputStream в один общий InputStream, благодаря чему несколько объединенных входных потоков могут рассматриваться в виде единого входного потока.
- StringBufferInputStream использует объект StringBuffer в качестве входного потока.
- LineNumberInputStream расширяет FilterInputStream и следит за нумерацией строк входного потока.
- PushbackInputStream расширяет FilterInputStream, добавляя возможность отката на один байт, что оказывается полезным при сканировании и синтаксическом анализе входного потока.
- PrintStream расширяет OutputStream и включает методы print и println для форматирования данных на выводе. К этому типу относятся потоки System.out и Sy stem.err.

Кроме указанных выше типов, имеются еще несколько полезных классов ввода/вывода:

- Класс File (не путать с потоковым классом File!) предназначен для работы с именами и путями файлов в локальной файловой системе. Он включает разделители для компонентов пути, локальный разделитель- суффикс и ряд полезных методов для работы с именами файлов.
- RandomAccessFile позволяет работать с файлами на уровне потоков с произвольным доступом. Он реализует интерфейсы DataInput и Data Output, а также большинство методов ввода/вывода классов Input Stream и OutputStream.
- Класс StreamTokenizer разбивает InputStream на отдельные лексемы. Он представляет входной поток в виде понятных “слов”, что часто бывает необходимо при синтаксическом анализе введенных пользователем выражений.

Все эти классы могут расширяться и порождать новые разновидности потоковых классов, предназначенные для конкретных приложений.

11.5. Фильтрующие потоки

Фильтрующие потоки добавляют несколько новых конструкторов к базовым конструкторам классов InputStream и OutputStream. Им передается поток соответствующего типа (входной или выходной), с которым необходимо соединить объект. Фильтрующие потоки позволяют объединять потоки в “цепочки” и тем самым создавать составной поток с большими возможностями. Приведенная программа печатает номер строки файла, в которой будет обнаружено первое вхождение заданного символа:

```
import java.io.*;

class FindChar {
     public static void main (String[] args)
          throws Exception
     {
          if (args.length != 2)
               throw new Exception("need char and file");
          int match = args[0].charAt(0);
          FileInputStream
               fileIn = new FileInputStream(filein);
          int ch;
          while ((ch == in.read()) != -1) {
               if (ch == match) {
                    System.out.println("'" + (char)ch +
                         "' at line " + in.getLineNumber());
                    System.exit(0);
               }
          }
          System.out.println(ch + " not found");
          System.exit(1);
     }
}
```

Программа создает поток класса FileInputStream с именем fileIn для чтения из указанного файла и затем вставляет перед ним объект класса LineNumberInputStream с именем in. Объекты LineNumberInputStream получают ввод из входных потоков, за которыми они закреплены, и следят за нумерацией строк. При чтении байтов из in на самом деле происходит чтение из потока fileIn, который получает эти байты из входного файла. Если запустить программу с файлом ее собственного исходного текста и буквой ‘I’ в качестве аргументов, то результат работы будет выглядеть следующим образом:

‘I’ at line 10

Вы можете “сцепить” произвольное количество объектов FilterInput Stream. В качестве исходного источника байтов допускается произвольный объект InputStream, не обязательно относящийся к классу FilterInput Stream. Возможность сцепления является одним из основных достоинств фильтрующих потоков, причем самый первый поток в цепочке не должен относиться к классу FilterInputStream.

Объекты FilterOutputStream могут сцепляться аналогичным образом. При этом байты, записанные в один выходной поток, будут подвергаться фильтрации и записываться в другой выходной поток. Все потоки, от первого до предпоследнего, должны относиться к классу FilterOutputStream, но последний поток может представлять собой любую из разновидностей Output Stream.

Применение фильтрующих потоков позволяет усовершенствовать поведение стандартных потоков. Например, чтобы всегда знать номер текущей строки в System.in, можно вставить в начало программы следующий фрагмент:

```
LineNumberInputStream
     lnum = new LinenumberInputStream(System.in);
System.in = lnum;
```

Во всем остальном тексте программы производятся обычные операции с System.in, однако теперь появляется возможность следить за нумерацией строк. Для этого используется следующий вызов:

lnum.getLineNumber();

Поток LineNumberInputStream, закрепленный за другим потоком Input Stream, следует контракту последнего, если InputStream — единственный тип, к которому мог бы относиться данный поток. System.in может быть отнесен только к типу InputStream, так что весь код программы, в котором он используется, вправе рассчитывать только на выполнение контракта этого типа. LineNu m berInputStream поддерживает более широкий спектр функций, так что замена исходного объекта на тот же самый объект с добавленными функциями нумерации строк оказывается вполне допустимой.

Упражнение 11.2

Расширьте FilterInputStream для создания класса, который осуществляет построчное чтение и возврат данных, с использованием метода, блокирующего работу программы до появления полной строки ввода.

Упражнение 11.3

Расширьте FilterOutputStream для создания класса, который преобразует каждое слово входного потока в заглавный регистр (title case). /См. раздел 13.5 - Примеч. перев/

Упражнение 11.4

Создайте пару фильтрующих потоковых классов для работы со сжатыми в произвольном формате данными; при этом поток CompressInputStream должен уметь расшифровывать данные, созданные потоком Compress OutputStream.

11.6. Класс PrintStream

Класс PrintStream используется каждый раз, когда в вашей программе встречается вызов метода print или println. PrintStream является расширением FilterOutputStream, так что передаваемые байты могут подвергаться фильтрации. Класс содержит методы print и println для следующих типов:

```
char       int      float      Object     boolean
char[]     long     double     String
```

Кроме того, простой вызов println без параметров осуществляет переход на другую строку без вывода информации.

PrintStream содержит два конструктора. Один из них — конструктор FilterOutputStream, получающий в качестве параметра объект-поток. У другого конструктора имеется второй параметр логического типа, который управляет автоматической очисткой (autoflushing) потока. Если значение этого аргумента равно true, то запись в поток символа перехода на новую строку ‘\n’ приводит к вызову метода flush. В противном случае такой символ ничем не отличается от всех остальных, и flush не вызывается. После конструирования потока его поведение в отношении автоматической очистки уже не может быть изменено.

При включении автоматической очистки вызов какого-либо из методов write, записывающего массив байтов, приводит к обращению к flush. Символы ‘\n’, которые встречаются внутри массивов, не вызывают flush, независимо от состояния флага автоматической очистки.

Методы print(String) и print(char[]) являются синхронизированными. Все остальные методы print и println реализуются с помощью этих двух методов, так что печать в объект PrintStream является безопасной при многопоточной работе.

11.7. Буферизованные потоки

Объекты классов BufferedInputStream и BufferedOutputStream обладают свойством буферизации, благодаря чему удается избежать вызова операций чтения/записи при каждом новом обращении к потоку. Эти классы часто используются в сочетании с файловыми потоками — работа с файлом на диске происходит сравнительно медленно, и буферизация позволяет сократить количество обращений к физическому носителю.

При создании буферизованного потока можно явно задать размер буфера или положиться на значение, принятое по умолчанию. Буферизованный поток использует массив типа byte для промежуточного хранения байтов, проходящих через поток.

Если метод read вызывается для пустого потока BufferedInputStream, он выполняет следующие действия: обращается к методу read потока-источника, заполняет буфер максимально возможным количеством байтов и возвращает запрошенные данные из буфера.

Аналогично ведет себя и BufferedOutputStream. Когда очередной вызов write приводит к заполнению буфера, вызывается метод write потока-приемника, направляющий содержимое буфера в поток.

Буферизованный выходной поток, используемый для записи данных в файл, создается следующим образом:

```
OutputStream bufferedFile(String path)
     throws IOExceptioon
{
     OutputStream out = new FileOutputStream(path);
     return new BufferedOutputStream(out);
}
```

Сначала для указанного пути создается FileOutputStream, затем порождается BufferedOutputStream и возвращается полученный буферизованный объект-поток. Подобная схема позволяет буферизовать вывод, предназначенный для занесения в файл.

Чтобы пользоваться методами объекта FileOutputStream, необходимо сохранить ссылку на него, поскольку для фильтрующих потоков не существует способа получить объект, следующий за данным объектом-потоком в цепочке. Перед тем как работать со следующим потоком, необходимо очистить буфер, иначе данные в буфере не достигнут следующего потока.

11.8. Байтовые потоки

Байтовые массивы, используемые в качестве источников входных или приемников выходных потоков, могут применяться для построение строк с данными для печати, декодирования данных и т. д. Эти возможности предоставляются потоками ByteArray. Методы потоков ByteArray являются синхронизированными, а следовательно — безопасными в условиях многопоточной среды.

Класс ByteArrayInput использует в качестве источника данных массив типа byte. Он содержит два конструктора:

public **ByteArrayInputStream(byte[] buf)**

Создает объект ByteArrayInputStream по заданному байтовому массиву. Массив используется непосредственно, а не копируется. Достижение конца массива buf означает завершение ввода данных из потока.

public ByteArrayInputStream(byte[] buf, int offset, int length)

Создает объект ByteArrayInputStream по заданному байтовому массиву, однако используется лишь часть массива buf от buf[offset] до buf [offset +length-1] или до конца массива (в зависимости от того, какая величина окажется меньше).

Класс ByteArrayOutput осуществляет вывод в динамически увеличиваемый байтовый массив. Он содержит следующие конструкторы и методы:

public **ByteArrayOutputStream()**

Создает объект ByteArrayOutputStream, размер которого выбирается по умолчанию.

public **ByteArrayOutputStream(int size)**

Создает объект ByteArrayOutputStream с заданным исходным размером.

public synchronized byte[] **toByteArray()**

Метод возвращает копию данных. Это позволяет программисту работать с массивом, не изменяя выходных данных.

public int **size()**

Возвращает текущий размер буфера.

public String **toString(int hiByte)**

Создает новый объект String на основе содержимого байтового массива. Старшие 8 бит каждого 16-разрядного символа в строке устанавливаются равными 8 младшим битам hiByte. Также имеется переопределенная безаргументная форма toString, эквивалентная toString(0).

11.9. Класс StringBufferInputStream

StringBufferInputStream читает данные из строки String, а не из байтового массива. Класс содержит единственный конструктор, параметром которого является строка — источник ввода. Работа с символами строки осуществляется так, как если бы это были байты. Например, приведенная ниже программа читает символы из командной строки или из System.in:

```
class Factor {
     public static void main(String[] args) {
          if (args.kength == 0) {
               factorNumbers(System.in);
          } else {
               InputStream in;
               for (int i = 0; i << args.lengthl i++) {
                    in = new StringBufferInputStream(args[i]);
                    factorNumbers(in);
               }
          }
     }
     // ...
}
```

Если команда вызывается без параметров, то factorNumbers берет числа из стандартного входного потока. Если же в командной строке присутствуют параметры, то для каждого из них создается объект StringBufferInput Stream и вызывается метод factorNumbers. Входные данные этого метода рассматриваются как единая последовательность байтов, независимо от того, взяты ли они из командной строки или из стандартного входного потока .

Обратите внимание на то, что в конструктор StringBufferInputStream передается объект класса String, а не StringBuffer.

Парного потока для StringBufferOutputStream не существует. При необходимости его можно имитировать, применяя метод toString к потоку Byt eArrayOutputStream.

11.10. Файловые потоки и FileDescriptor

Ввод и вывод в приложениях часто связан с чтением/записью файлов. Файловый ввод/вывод в Java представлен двумя потоками — FileInput Stream и FileOutputStream. Объекты каждого из этих типов создаются одним из трех конструкторов:

- Конструктор с параметром типа String, содержащим имя файла.
- Конструктор с параметром File (см. раздел “Класс File”).
- Конструктор с параметром класса FileDescriptor.

Файловый дескриптор FileDescriptor представляет собой системно-зависимый объект, служащий для описания открытого файла. Он может быть получен вызовом метода getFD для любого объекта класса File или Random AccessFile. Объекты FileDescriptor позволяют создавать новые потоки File или RandomAccessFile для тех же файлов, что и другие потоки, но при этом не требуется знание имени файлов. Необходимо соблюдать осторожность и следить за тем, чтобы различные потоки не пытались одновременно совершать с файлом различные операции. Например, невозможно предсказать, что случится, когда два потока попытаются одновременно записать информацию в один и тот же файл с использованием двух разных объектов File Descriptor.

Метод flush класса FileOutputStream гарантирует лишь сброс содержимого буфера в файл. Он *не* гарантирует, что данные будут записаны на диск — файловая система может осуществлять свою собственную буферизацию.

11.11. Конвейерные потоки

Конвейерные (piped) потоки используются парами, предназначенными для ввода/вывода; байты, записанные во входной поток пары, считываются на выходе. Конвейерные потоки безопасны в многопоточной среде; на самом деле, один из вполне надежных способов работы с конвейерными потоками заключается в использовании двух программных потоков — одного для чтения, а другого для записи. В случае заполнения конвейера происходит блокировка программного потока, осуществляющего запись. Если же чтение и запись производятся в одном программном потоке, то он блокируется навсегда.

В приведенном ниже примере создается новый программный поток, получающий входные данные от некоторого объекта-генератора, а его вывод направляется в объект OutputStream:

```
class Pipe {
     public static void main(String[] args) {
          try {
               PipedOutputStream out = new PipeOutputStream();
               PipedInputStream in = new PipedInputStream(out);

               // генератор данных выводит данные
               // в предоставленный ему выходной поток
               DataGenerator data = new DataGenerator(out);
               data.setPriority(Thread.MIN_PRIORITY);
               data.start();

               int ch;
               while ((ch = in.read()) != -1)
                    System.out.print((char)ch);
               System.out.println();
          } catch (IOException e) {
               System.out.println("Exception: " + e);
          }
     }
}
```

Мы создаем конвейерные потоки, задавая PipedInputStream в качестве параметра конструктора PipedOutputStream. Порядок значения не имеет: с тем же успехом можно было передавать выходной поток конструктору входного. Важно, чтобы парные потоки ввода/вывода были соединены друг с другом. Далее конструируется объект DataGenerator и выходным потоком сгенерированных данных назначается PipedOutputStream. Затем в цикле происходит чтение данных от генератора и запись их в системный выходной поток. В конце необходимо убедиться, что последняя выводимая строка будет должным образом завершена.

11.12. Класс Seq uenceInputStream

Класс SequenceInputStream создает единый входной поток, читая данные из одного или нескольких входных потоков: сначала первый поток читается до самого конца, затем — следующий за ним, и так далее, до последнего потока. Этот класс содержит два конструктора: один — для простейшего случая двух входных потоков, которые передаются в качестве параметров конструктора; другой конструктор предназначен для произвольного количества входных потоков, в нем используется абстрактное представление Enumeration, описанное в главе 12. Реализация интерфейса Enumeration позволяет получить упорядоченный список объектов любого типа. Для потока Sequence InputStream перечисление может содержать только объекты типа Input Stream. Если в нем окажется что-либо еще, то при попытке получения объекта из списка возбуждается исключение SequenceInputStream.

Например, приложение Factor вызывает метод factor Numbers для каждого аргумента, входящего в командную строку. Все числа обрабатываются отдельно, так что подобное разобщение параметров не имеет особого значения. Тем не менее, если бы ваше приложение суммировало числа из входного потока, то было бы необходимо собрать все значения воедино. В приведенном ниже приложении SequenceInputStream используется для создания единого потока из объектов StringBufferInputStream для каждого из параметров:

```
import java.io.*;
import java.util.Vector;

class Sum {
     public static void main(String[] args) {
          InputStream in;  // поток, из которого читаются числа
          if (args.length == 0) {
               in = System.in;
          } else {
               InputStream stringIn;
               Vector inputs = new Vector(args.length);
               for (int i = 0; i << args.length; i++) {
                    String arg = args[i] + " ";
                    stringIn = new StringBufferInputStream(arg);
                    inputs.addElement(stringIn);
               }
               in = new SequenceInputStream(inputs.elements());
          }

          try {
               double total = sumStream(in);
               System.out.println("The sum is " + total);
          } catch (IOException e) {
               System.out.println(e);
               System.exit(-1); //
          }
     }
     // ...
}
```

Если параметры отсутствуют, то для ввода данных используется System.in. В противном случае создается объект Vector, размер которого позволяет хранить столько объектов StringBufferInputStream, сколько аргументов в командной строке. Затем мы создаем поток для каждого из аргументов и добавляем в концы строк пробелы, чтобы разделить их. Затем потоки заносятся в вектор streams. После завершения цикла мы вызываем метод elements вектора, чтобы получить объект Enumeration с элементами. Enumeration используется в конструкторе SequenceInputStream, который сцепляет все потоки параметров в единый поток InputStream. Затем все числа в этом потоке суммируются методом sumStream и выводится результат. Реализация sumStream приведена в примере из раздела “Класс StreamTokenizer”. /Конечно, проблему можно было решить и иначе - получить единую строку, в которую входят все параметры, и создать один поток StringBufferInputStream./

Кроме того, можно было создать и новую реализацию Enumeration, которая бы обращалась за каждым аргументом к потоку StringInputStream. Подробности приведены в разделе “Интерфейс Enumeration”.

11.13. Класс LineNumberInputStream

Объекты класса LineNumberInputStream позволяют следить за нумерацией строк во время чтения данных из входного потока. Метод getLine Number, возвращает текущий номер строки. Нумерация строк начинается с единицы.

Текущий номер строки может быть задан методом setLineNumber. Это может оказаться полезным, когда вы работаете с несколькими входными потоками как с одним целым, однако нумерация строк должна осуществляться относительно начала каждого из потоков. Например, если SequenceInput Stream используется для чтения из нескольких файлов как из одного потока, то может возникнуть надобность в отдельной нумерации строк для каждого из файлов, из которого поступили данные.

Упражнение 11.5

Напишите программу, которая читает заданный файл и ищет в нем некоторое слово. Программа должна выводить каждую строку, в которой встретилось это слово, и ее номер.

11.14. Класс PushbackInputStream

Класс PushbackInputStream обеспечивает возможность отката на один символ потока назад. Это особенно полезно при разделении входного потока на отдельные лексемы. Например, чтобы определить, где кончается лексема, часто приходится читать символ, следующий за ее концом. После просмотра символа, завершающего текущую лексему, необходимо вернуть его во входной поток, чтобы он послужил началом следующей лексемы. В приведенном ниже примере класс PushbackInputStream используется для поиска самой длинной последовательности повторений любого байта в потоке:

```
import java.io.*;

class SequenceCount {
     public static void main(String[] args) {
          try {
               PushbackInputStream
                    in = new PushbackInputStream(System.in);
               int max = 0;     // длина найденной последовательности
               int maxB = -1;   // байт, из которого она состоит
               int b;           // текущий байт входного потока
               do {
                    int cnt;
                    int b1 = in.read();  // первый байт
                                         // в последовательности
                    for (cnt = 1; (b = in.read()) == b1; cnt++)
                         continue;
                    if (cnt >> max) {
                         max = cnt; // запомнить длину
                         maxB = b1; // запомнить байт
                    }
                    in.unread(b);   // откат к началу
                                    // следующей последовательности
               } while (b != -1);   // продолжать до конца потока
               System.out.println(max + " bytes of " + maxB);
          } catch (IOException e) {
               System.out.println(e);
               System.exit(1);
          }
     }
}
```

При достижении конца одной последовательности происходит чтение байта, с которого начинается следующая последовательность. Метод unread позволяет вернуться на одну позицию назад, чтобы снова прочитать байт при выполнении цикла do для следующей последовательности.

Буфер отката представляет собой защищенное поле типа int с именем pushBack. Подклассы могут модифицировать это поле. Значение –1 показывает, что буфер отката пуст. Любое другое значение возвращается в качестве первого байта входного потока методом Pushback.read.

11.15. Класс StreamTokenizer

Разделение входного потока на отдельные лексемы встречается довольно часто, поэтому пакет java.io содержит специальный класс StreamTokenizer для выполнения простейшего лексического анализа. В настоящее время этот класс в полной мере работает лишь с младшими 8 битами Unicode, составляющими подмножество символов Latin-1, поскольку внутренний массив класса, хранящий информацию о категориях символов, состоит только из 256 элементов. Символы, превышающие \u00ff, считаются алфавитными. Хотя в подавляющем большинстве случаев это действительно так (собственно, большая часть символов относится к алфавитным), вы, например, не сможете назначить в качестве ограничителя символ ‘?’ (\u270D). Даже с учетом этого условия выделение лексем во многих случаях происходит нормально.

Чтобы выделить лексемы в потоке, следует создать объект StreamTokenizer на основе объекта InputStream и затем установить параметры анализа. Цикл сканирования вызывает метод nextToken, который возвращает тип следующей лексемы в потоке. С некоторыми типами лексем связываются значения, содержащиеся в полях объекта StreamTokenizer.

Данный класс спроектирован в первую очередь для анализа потоков, содержащих текст в стиле Java; он не универсален. Тем не менее многие файлы конфигурации достаточно похожи на Java и могут успешно анализироваться. При разработке новых файлов конфигурации или других данных можно придать им сходство с текстами на Java, чтобы анализировать их с помощью StreamTokenizer и за счет этого сэкономить усилия.

Когда метод nextToken распознает следующую лексему, он возвращает ее тип и присваивает это же значение полю ttype. Имеются четыре типа лексем:

- TT_WORD: обнаружено слово. Найденное слово помещается в поле sval типа String.
- TT_NUMBER: обнаружено число. Найденное число помещается в поле nval типа double. Распознаются только десятичные числа с плавающей точкой (с десятичной точкой или без нее). Анализатор не распознает 3.4e79 как число с плавающей точкой, или 0xffff как шестнадцатеричное число.
- TT_EOL: обнаружен конец строки.
- TT_EOF: обнаружен конец файла.

Символы входного потока делятся на специальные и ординарные. Специальными считаются символы, которые особым образом обрабатываются в процессе анализа, — пробелы, символы, образующие числа и слова, и так далее. Все остальные символы относятся к ординарным. Если следующий символ потока является ординарным, то тип лексемы совпадает с символом. Например, если в потоке встречается символ ‘B’ и он не является специальным, то тип лексемы (и поле ttype) равен эквиваленту символа ‘B’ в типе int.

В качестве примера давайте рассмотрим реализацию метода Sum.sum Stream из класса Sum:

```
static double sumStream(InputStream in) throws IOException {
     StreamTokenizer nums = new StreamTokenizer(in);
     double result = 0.0;
     while (nums.nextToken() == StreamTokenizer.TT_NUMBER)
          result +=nums.nval;
     return result;
}
```

Объект StreamTokenizer создается для исходного потока, после чего в цикле происходит чтение лексем из потока. Если обнаруженная лексема является числом, то оно прибавляется к накапливаемому результату. Когда числа во входном потоке кончаются, возвращается окончательное значение суммы.

Приведем еще один пример. Данная программа читает содержимое файла, ищет в нем атрибуты в виде пар *имя=значение* и сохраняет их в объектах AttributedImpl, описанных в разделе “Реализация интерфейсов”:

```
public static Attributed readAttrs(String file)
     throws IOException
{
     FileInputStream fileIn = new FileInputStream(file);
     StreamTokenizer in = new StreamTokenizer(fileIn);
     AttributedImpl attrs = new AttributedImpl();
     Attr attr = null;
     in.commentChar('#');    // '#' - комментарий до конца строки
     in.ordinaryChar('/');   // ранее являлся символом комментария
     while (in.nextToken() != StreamTokenizer.TT_EOF) {
          if (in.ttype() == StreamTokenizer.TT_WORD) {
               if (attr != null) {
                    attr.valueOf(in.sval);
                    attr = null;        // использован
               } else {
                    attr = new Attr(in.sval);
                    attrs.add(attr);
               }
          } else if (in.ttype == '=') {
               if (attr == null)
                    throw new IOException("misplaced '='");
          } else {
               if (attr == null)        // ожидалось слово
                    throw new IOException("bad Attr name");
               attr.valueOf(new Double(in.nval));
               attr = null;
          }
     }
     return attrs;
}
```

В файле атрибутов символ # используется для обозначения начала комментариев, игнорируемых во время поиска. Программа ищет в потоке строковую лексему, за которой может (хотя и не обязан) следовать знак =, сопровождаемый строкой или числом. Каждый такой атрибут заносится в объект Attr, добавляемый к набору атрибутов объекта AttributedImpl. После завершения анализа файла возвращается набор атрибутов.

Задавая символ # в качестве символа комментария, мы тем самым устанавливаем его категорию. Анализатор распознает несколько категорий символов, которые определяются следующими методами:

public void **wordChars(int low, int hi)**

Символы в этом диапазоне образуют слова; они могут входить в лексему типа TT_WORD. Допускается многократный вызов этого метода с разными диапазонами. Слово состоит из одного или нескольких символов, входящих в любой их допустимых диапазонов.

public void **whitespaceChars(int low, int hi)**

Символы в этом диапазоне являются разделителями. При анализе они игнорируются; их единственное назначение заключается в разделении лексем — например, двух последовательных слов. Как и в случае wordChars, можно вызывать этот метод несколько раз, при этом объединение всех диапазонов определяет набор символов-разделителей.

public void **ordinaryChar (int ch)**

Символ ch является ординарным. Ординарный символ при анализе потока возвращается сам по себе, а не в виде лексемы. В качестве иллюстрации см. приведенный выше пример.

public void **ordinaryChars (int low, int hi)**

Символы в диапазоне являются ординарными.

public void **commentChar (int ch)**

Символ ch начинает однострочный комментарий — символы от ch до ближайшего конца строки считаются одним длинным разделителем.

public void **quoteChar (int ch)**

Пары символов ch являются ограничителями для строковых констант. Когда в потоке распознается строковая константа, символ ch возвращается в качестве лексемы, а поле sval содержит тело строки (без символов-ограничителей). При чтении строковых констант обрабатываются некоторые стандартные символы Java в записи с \ (например, \t), но не все. Строки, воспринимаемые StreamTokenizer, представляют собой подмножество строк Java. Особенно жесткий запрет накладывается на использование \xxxx, \', \" или (к сожалению) \Q, где символ Q совпадает с символом-ограничителем ch. В потоке могут присутствовать несколько разных символов-ограничителей, но строки должны начинаться и заканчиваться одним и тем же ограничителем. Другими словами, строка, которая начинается одним символом-ограничителем, продолжается до следующего вхождения того же символа; если в середине строки встречается другой символ-ограничитель, то он просто считается частью строки.

public void **parseNumbers()**

Указывает на необходимость выделения чисел из потока. StreamTokenizer выдает числа с плавающей точкой двойной точности и возвращает тип лексемы TT_NUMBER, а значение лексемы помещается в поле nval. Просто отказаться от поиска чисел невозможно — для этого придется либо вызвать ordinaryChars для всех символов, входящих в состав числа (не забудьте о десятичной точке и знаке “минус”), либо вызвать resetSyntax.

public void **resetSyntax()**

Сбрасывает синтаксическую таблицу, в результате чего все символы становятся ординарными. Если вы вызовете resetSyntax и затем начнете читать поток, то nextToken всегда будет выдавать следующий символ потока, как будто вы используете метод InputStream.read.

Не существует методов для определения категории заданного символа или для добавления новых категорий. Ниже приведены значения параметров по умолчанию для только что созданного объекта StreamTokenizer:

wordChars(‘a’, ‘z’);

wordChars(‘A’, ‘Z’);

wordChars(128 + 32, 255);

whitespaceChars(0, ‘ ‘);

commentChar(‘/’);

quoteChar(‘”’);

quoteChar(‘\’’);

parseNumbers();

Остальные методы управляют поведением анализатора:

public void **eolIsSignificant(boolean flag)**

Если значение flag равно true, то конец строки является существенным, и nextToken может возвращать TT_EOL. В противном случае концы строк считаются символами-разделителями и TT_EOL никогда не возвращается. Значение по умолчанию равно false.

public void **slashStarComments(boolean flag)**

Если значение flag равно true, анализатор распознает комментарии вида /*...*/. Значение по умолчанию равно false.

public void **slashSlashComments(boolean flag)**

Если значение flag равно true, анализатор распознает комментарии от // до конца строки. Значение по умолчанию равно false.

public void **lowerCaseMode(boolean flag)**

Если значение flag равно true, все символы в лексемах типа TT_WORD преобразуются в нижний регистр, если имеется соответствующий эквивалент (то есть к слову применяется метод String.toLowerCase). Значение по умолчанию равно false.

Имеется также несколько методов общего назначения:

public void **pushBack()**

Заносит предыдущую лексему обратно в поток. Следующий вызов nextToken снова вернет ту же самую лексему. Глубина отката ограничивается одной лексемой; несколько последовательных вызовов pushBack эквивалентны одному вызову.

public int **lineno()**

Возвращает текущий номер строки. Обычно это бывает полезно для вывода сообщений о найденных ошибках.

public String **toString()**

Возвращает строковое представление последней возвращенной лексемы, включающее номер строки.

Упражнение 11.6

Напишите программу, которая получает входные данные в форме “*имя оператор значение*”, где *имя* — одно из трех имен по вашему выбору, *оператор* равен +, – или =, а *значение* является числом. Примените все операторы к именованным величинам, а в конце работы программы выведите все три значения. Усложним задание — воспользуйтесь классом Hashtable, который применялся при разработке AttributedImpl, чтобы можно было работать с произвольным количеством именованных величин, не обязательно тремя.

11.16. Потоки данных

Хотя возможность чтения и записи байтовых потоков достаточно полезна, часто бывает необходимо пересылать в потоке данные определенного типа. Интерфейсы DataInput и DataOutput определяют методы для пересылки примитивных типов Java в потоке. Реализация этих интерфейсов по умолчанию представлена классами D ataInputStream и DataOutputStream. Сначала мы рассмотрим интерфейсы, а затем их реализации.

Интерфейсы для входных и выходных потоков данных являются практически зеркальными отражениями друг друга. В приведенной ниже таблице содержатся параллельные методы чтения и записи для каждого из типов:

Read	Write	Тип
readBoolean	writeBoolean	boolean
readChar	writeChar	char
readByte	writeByte	byte
readShort	writeShort	short
readInt	writeInt	int
readLong	writeLong	long
readFloat	writeFloat	float
readDouble	writeDouble	double
readUTF	writeUTF	String в формате UTF

UTF является сокращением от “Unicode Transmission Format” — символы Unicode переводятся в Unicode-1-1-UTF-8, компактную двоичную форму, спроектированную для кодировки 16-разрядных символов Unicode в 8-разрядных байтах.

Кроме этих парных методов, DataInput содержит несколько своих собственных:

public abstract void **readFully(byte[] buf)** throws IOException

Читает последовательность байтов в buf, блокируя работу программы до завершения чтения.

public abstract void **readFully(byte[] b, int off, int len)** •• throws IOException

Читает последовательность байтов в buf, начиная с позиции offset, в количестве len байтов, если не встретится конец массива buf. Работа программы блокируется до завершения чтения всех байтов.

public abstract int **skipBytes(int n)** throws IOException

Пропускает байты в потоке. Работа программы блокируется до пропуска всех n байтов.

public abstract String **readLine()** throws IOException

Читает строку вплоть до нахождения символов \n, \r или пары \r\n. Символы, завершающие строку, не включаются в нее. При достижении конца входного потока возвращается null.

public abstract int **readUnsignedByte()** throws IOException

Читает 8-разрядный байт без знака и возвращает его в виде значения типа in t.

public abstract int **readUnsignedShort()** throws IOException

Читает 16-разрядное целое типа short без знака и возвращает его в виде значения типа int, что дает число в диапазоне 0–65535 (2 $^{16–1).}$

Интерфейс DataInput обрабатывает встреченный конец файла, возбуждая исключение EOFException. Класс EOFException является расширением IOException.

Интерфейс DataOutput содержит методы, сигнатуры которых совпадают с сигнатурами трех разновидностей метода write класса OutputStream, и в дополнение к ним поддерживает еще три метода:

public abstract void **writeBytes(String s)** throws IOException

Записывает строку в виде последовательности байтов. Старший байт каждого символа при этом теряется, так что данный метод следует применять только для строк, содержащих символы в диапазоне от \u0000 до \u00ff, если только вы не готовы смириться с потерей данных.

public abstract void **writeChars(String s)** throws IOException

Записывает строку в виде последовательности значений типа char.

Чтение строк, записанных этими методами, должно осуществляться циклическим вызовом метода readChar, поскольку не существует метода readBytes или readChars, возвращающего то же количество символов, которое было записано методом writeBytes или writeChars. Вам придется сначала записать длину строки или воспользоваться специальным символом-маркером, отмечающим ее конец. В первом случае можно воспользоваться методом readFully, чтобы считать полный массив байтов, однако с writeChars это уже не сработает, так как вам нужны значения типа char, а не byte.

11.16.1. Классы потоков данных

Для каждого интерфейса Data имеется соответствующий поток. Кроме того, класс RandomAccessFile реализует оба интерфейса для входных и выходных потоков данных (см. раздел "Класс RandomAccessFile"). Каждый из классов Data представляет собой расширение класса Filter, так что потоки данных могут использоваться для фильтрации других потоков. Следовательно, каждый из них должен иметь конструкторы, которые получают в качестве параметра другой входной или выходной поток. Например, фильтрация может применяться при записи данных в файл — для этого следует создать объект DataOutputStream до объекта FileOutputStream, а затем, при считывании данных из файла, поместить DataInputStream перед объектом FileInput Stream.

Упражнение 11.7

Включите в класс Body из главы 2 метод, который записывает содержимое объекта в DataOutputStream, и конструктор, который считывает состояние объекта из DataInputStream.

11.17. Класс RandomAccessFile

Класс RandomAccessFile предоставляет более совершенный механизм для работы с файлами, чем файловые потоки. Он не является расширением Input Stream или OutputStream, поскольку может осуществлять любую из операций чтения/записи или оба действия сразу. Режим работы с файлом указывается в качестве параметра для различных конструкторов. Класс Random AccessFile реализует оба интерфейса Data InputStream и DataOutput Stream, поэтому он может применяться для чтения/записи встроенных типов Java.

Хотя класс RandomAccessFile не является расширением входных и выходных потоковых классов, имена и сигнатуры содержащихся в нем методов совпадают с вызовами read и write. Хотя это означает, что вам не придется учить новый набор имен и семантик для выполнения той же самой задачи, объекты класса RandomAccessFile не могут использоваться там, где требуется присутствие объектов InputStream или OutputStream. Тем не менее вы можете использовать объекты RandomAccessFile вместо объектов-потоков DataInput или DataOutput.

Класс RandomAccessFile содержит три конструктора:

public **RandomAccessFile(String name, String mode)** throws •• IOException

Создает объект RandomAccessFile для заданных имени файла и режима. Режим указывается в виде "r" или "rw" для доступа по чтению или чтению/записи соответственно. Любое другое значение режима приводит к возбуждению I OException.

public **RandomAccessFile(File file, String mode)** throws •• IOException

Создает объект RandomAccessFile для заданного объекта класса File и режима.

public **RandomAccessFile(FileDescriptor fd)** throws IOException

Создает объект RandomAccessFile для заданного объекта fd типа File Descriptor (см. раздел "Файловые потоки и FileDescriptor").

Термин "произвольный доступ" (random access), вынесенный в название типа, обозначает возможность установки файлового указателя чтения/записи в любую позицию внутри файла с последующим выполнением нужной операции. Эта возможность обеспечивается следующими методами:

public long **getFilePointer()** throws IOException

Возвращает текущее смещение (в байтах) от начала файла.

public void **seek(long pos)** throws IOException

Устанавливает файловый указатель в заданную позицию (в байтах). Следующий считанный или записанный байт будет иметь смещение pos.

public long **length()** throws IOException

Возвращает длину файла.

Упражнение 11.8

Напишите программу для чтения файла, который состоит из отдельных элементов, разделяемых строками, начинающимися с символов %%. Программа должна создавать сводный файл, содержащий начальную позицию для каждого такого элемента. Затем напишите программу, которая печатает случайный элемент на основании сводного файла (см. описание метода Math.random в разделе "Класс Math").

11.18. Класс File

Класс File содержит стандартные средства для работы с именами файлов. Его методы позволяют разделять полные имена, включающие путь, на составные части и запрашивать у файловой системы информацию о файлах.

Объект File обычно связан с полным именем файла, причем необязательно существующего. Например, чтобы выяснить, представляет ли некоторое имя существующий в системе файл, следует сначала создать объект File для данного имени, после чего вызвать для этого объекта метод exists.

Полное имя /В этом разделе следует отличать полное имя файла от имени объекта класса файл. - Примеч. перев./ делится на две части: одна определяет каталог (или папку), а другая — собственно файл. Для разделения компонентов пути служит символ, хранящийся как значение типа char в статическом поле separatorChar или значение типа String в статическом поле separator. Последнее вхождение этого символа в путь отделяет каталог от имени файла.

Объекты File создаются одним из трех конструкторов:

public **File(String path)**

Создает объект File для работы с заданным файлом path. Если параметр равен null, возбуждается исключение NullPointerException.

public **File(String dirName, String name)**

Создает объект File для работы с файлом name, находящимся в каталоге dirName. Если параметр dirName равен null, используется только компонент name. В противном случае вызов конструктора эквивалентен следующему:

File(dirname + File.separator + name)

public **File(File fileDir, String name)**

Создает объект File для заданного объекта-каталога fileDir типа File и файла с именем name. Вызов конструктора эквивалентен следующему:

File(fileDir.getPath(), name)

Четыре метода get предназначены для получения информации о компонентах полного имени объекта File. В приведенном ниже фрагменте программы вызывается каждый из этих методов:

```
File src = new File("ok", "FileMethods");
System.out.println("getName() = " + src.getName());
System.out.println("getPath() = " + src.getPath());
System.out.println("getAbsolutePath() = "
        + src.getAbsolutePath());
System.out.println("getParent() = " + src.getParent());
```

Вот как выглядит результат работы программы:

getName() = FileMethods

getPath() = ok/FileMethods

getAbsolutePath() = /vob/java_prog/src/ok/FileMethods

getParent() = ok

Несколько методов возвращают логические значения, которые характеризуют файл, представленный объектом File:

- exists: возвращает true, если файл существует в файловой системе.
- canRead: возвращает true, если файл существует и доступен для чтения.
- canWrite: возвращает true, если файл существует и доступен для записи.
- isFile: возвращает true, если файл существует и является обычным (то есть не каталогом или файлом особого типа).
- isDirectory: возвращает true, если файл является каталогом.
- isAbsolute: возвращает true, если путь представляет собой полное имя файла.

Класс File содержит и другие полезные методы:

public long **lastModified()**

Возвращает дату последней модификации файла. Возвращаемое значение должно использоваться только для сравнения дат последних модификаций различных файлов и для выяснения того, редактировался ли некоторый файл после другого, а также для проверки того, модифицировался ли файл вообще. Временем последней модификации файла не следует пользоваться для других целей.

public long **length()**

Возвращает длину файла в байтах.

public boolean **mkdir()**

Создает каталог и возвращает true в случае успеха.

public boolean **mkdirs()**

Создает все каталоги, входящие в заданный путь, и возвращает true в случае успеха. Метод обеспечивает возникновение конкретного каталога, даже если для этого потребуется создать другие, не существующие в данный момент каталоги, которые находятся выше в иерархии.

public boolean **renameTo(File new_name)**

Переименовывает файл и возвращает true в случае успеха.

public boolean **delete()**

Удаляет файл или каталог, представленный объектом File, и возвращает true в случае успеха.

public String[] **list()**

Возвращает список файлов в каталоге. Если объект File представляет собой не каталог, а нечто иное, передается null; в противном случае возвращается массив с именами файлов. Список содержит все файлы каталога, за исключением эквивалентов "." и ".." (текущий и родительский каталоги соответственно).

public String[] **list(FilenameFilter filter)**

Использует заданный фильтр для составления списка файлов в каталоге (см. ниже раздел "Интерфес Filename Filter").

Переопределенный метод File.equals заслуживает особого упоминания. Два объекта File считаются равными в том случае, если совпадают их полные имена, а не в том, если они представляют один и тот же файл в системе. Метод File.equals не может применяться для выяснения того, соответствуют ли два объекта File одному и тому же файлу.

Для создания файлов используются объекты классов FileOutputStream или RandomAccessFile, а не объекты класса File.

Наконец, остается упомянуть, что символ File.pathSeparatorChar и строка File.pathSeparator представляют символ, разделяющий каталоги или файлы в полном имени. Например, в системе UNIX для разделения компонентов полного имени используется двоеточие: ".:/bin:/usr/bin". Следовательно, в системе UNIX символ pathSepar a torChar представляет собой двоеточие.

Полное имя файла хранится в защищенном строковом поле с именем String. Подклассы File могут при необходимости непосредственно обращаться к этому полю или модифицировать его.

Упражнение 11.9

Напишите метод, который получает в качестве параметра полное имя файла и выводит всю информацию о соответствующем файле (если он существует).

11.19. Интерфейс FilenameFilter

Интерфейс FilenameFilter позволяет создавать объекты, которые фильтруют списки файлов и удаляют из них ненужные. Он содержит всего один метод:

boolean **accept(File dir, String name)**

Возвращает true, если файл с именем name в каталоге dir должен входить в отфильтрованный список.

В следующем примере объект FilenameFilter используется для того, чтобы в список включались только каталоги:

```
import java.io.*;

class DirFilter implements FilenameFilter {
        public boolean accept(File dir, String name) {
                return new File(dir, name).isDirectory();
        }

        public static void main(String[] args) {
                File dir = new File(args[0]);
                String[] files = dir.list(new DirFilter());
                System.out.println(files.length + "dir(s):");
                for (int i = 0; i << files.length; i++)
                        System.out.println("\t" + files[i]);
        }
}
```

Сначала мы создаем объект File, который представляет собой каталог, указанный в командной строке. Затем мы конструируем объект DirFilter и передаем его в качестве параметра методу list. Для каждого имени, входящего в каталог, list вызывает метод accept объекта-фильтра и включает имя в список лишь в том случае, если объект-фильтр возвращает true. Для нашего метода accept значение true показывает, что имя соответствует каталогу.

Упражнение 11.10

С помощью интерфейса FilenameFilter напишите программу, которая получает в качестве параметров имя каталога и расширение файла и выводит список всех файлов каталога с заданным расширением.

11.20. Классы IOException

Для сообщений обо всех ошибках ввода/вывода, обнаруженных классами пакета java.io, должны использоваться исключения, являющиеся подклассом IOException. Большинство классов проектировалось для целей общего назначения, так что основная часть исключений также носит универсальный характер. Например, методы класса InputStream, возбуждающие IOException, не могут заранее предсказать, какие именно возникнут исключения, так как каждый конкретный потоковый класс может возбудить некоторый подкласс IOException, сигнализируя тем самым об ошибке, относящейся лишь к этому потоку. Например, фильтрующие входные и выходные потоки лишь передают без обработки исключения от объектов, на основе которых они создавались и которые могут представлять собой потоки любого типа.

В пакете java.io используются четыре подкласса IOException:

EOFException extends IOException

Возбуждается интерфейсами потоков данных при достижении конца ввода, как запланированном, так и неожиданном.

FileNotFoundException extends IOException

Возбуждается конструкторами файловых потоков, если файл, имя которого передается в качестве параметра, не найден.

InterruptedIOException extends IOException

Возбуждается любым потоком, когда в операцию ввода/вывода вмешивается прерывание программного потока (см. раздел "Прерывание потока"). Фактически операции ввода/вывода переводят исключение InterruptedException в InterruptedIO Exception.

UTFDataFormatException extends IOException

Возбуждается методом DataInputStream.readUTF, если считываемая строка имеет неверный синтаксис UTF.

Если не считать этих конкретных исключений, то для сообщений обо всех особых состояниях в java.io используется исключение IOException, содержащее строку с описанием конкретной ошибки — например, использование несоединенного конвейерного потока или попытка отката на несколько символов назад в потоке PushbackInputStream.

Содержание | Далее

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Глава 12
СТАНДАРТНЫЕ ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА

Компьютеры бесполезны —
они могут только давать ответы.
Пабло Пикассо

Пакет java.util содержит ряд стандартных вспомогательных интерфейсов и классов. Некоторые из них уже использовались в предыдущих главах — например, классы Date и Hashtable. Имеются и другие полезные интерфейсы и классы:

Коллекции:

- BitSet: битовый вектор с динамическим изменением размера.
- Enumeration: интерфейс, который возвращает объект, используемый для перечисления набора объектов (например, элементов, содержащихся в конкретной хеш-таблице).
- Vector: вектор, состоящий из элементов типа Object, с динамическим изменением размера.
- Stack: расширение класса Vector, в котором добавлены методы для работы с простейшим стеком LIFO (“последним пришел, первым вышел”).
- Dictionary: абстрактный класс, содержащий алгоритмы для работы с парами ключ/значение.
- Hashtable: реализация Dictionary, в которой для сопоставления ключа со значением используется хеш-код.
- Properties: расширение Hashtable, в котором строковые ключи сопоставляются со строковыми значениями.

Концепции проектирования:

- Observer/Observable: с помощью этой пары интерфейс/класс вы можете сделать свой объект “наблюдаемым” (Observable) — закрепить за ним один или более объектов-наблюдателей (Observer), которые будут извещаться в том случае, если с наблюдаемым объектом происходит что-то интересное.

Прочее:

- Date: работа с датами с точностью до одной секунды.
- Random: объекты, генерирующие последовательности псевдослучайных чисел.
- StringTok enizer: деление строки на лексемы с учетом символов-ограничителей. По умолчанию ими считаются разделители (whitespace).

12.1. Класс BitSet

Класс BitSet позволяет создать битовый вектор, размер которого изменяется динамически. Фактически BitSet представляет собой набор битов со значениями true или false размером до 232–1, причем изначально все биты равны false. Для хранения набора выделяется объем памяти, необходимый для хранения вектора вплоть до старшего бита, который устанавливался или сбрасывался в программе — все превышающие его биты считаются равными false.

При создании объекта BitSet можно явно задать исходный размер набора или воспользоваться безаргументным конструктором для установки размера по умолчанию.

public void **set(int bit)**

Устанавливает бит в позиции bit, присваивая ему значение true.

public void **clear(int bit)**

Сбрасывает бит в позиции bit, присваивая ему значение false.

public boolean **get(int bit)**

Возвращает значение бита в позиции bit.

public void **and(BitSet other)**

Выполняет операцию логического И над данным набором и other и присваивает результат данному набору.

public void **or(BitSet other)**

Выполняет операцию логического ИЛИ над данным набором и other и присваивает результат данному набору.

public void **xor(BitSet other)**

Выполняет операцию исключающего логического ИЛИ над данным набором и other и присваивает результат данному набору.

public int **size()**

Возвращает позицию старшего бита в наборе, который может быть установлен или сброшен без необходимости увеличения набора.

public int **hashCode()**

Возвращает хеш-код для набора, определяемый значениями битов. Соблюдайте осторожность и не изменяйте биты набора, пока объект BitSet находится в хеш-таблице, иначе он будет потерян.

public boolean **equals(BitSet other)**

Возвращает true, если все биты other совпадают с битами в данном наборе.

В приведенном ниже классе с помощью объекта BitSet происходит пометка символов, встречающихся в строке. Объект можно распечатать и посмотреть, какие же символы входят в строку:

```
public class WhichChars {
     private BitSet used = new BitSet();

     public WhichChars(String str) {
          for (int i = 0; i << str.length(); i++)
               used.set(str.charAt(i));    // установить бит,
                                           // соответствующий символу
     }

     public String toString() {
          String desc = "[";
          int size = used.size();
          for (int i = 0; i << size; i++) {
               if (used.get(i))
                    desc += (char)i;
          }
          return desc + "]";
     }
}
```

12.2. Интерфейс Enumeration

Большинство классов-коллекций использует интерфейс Enumeration в качестве средства для перебора объектов, входящих в коллекцию. Кроме того, этот интерфейс используется и другими классами, входящими в библиотеки Java, а также в пользовательские программы. Обычно при этом создается собственный класс, который реализует интерфейс Enumeration и содержит один или несколько методов, возвращающих объект Enumeration. Интерфейс Enumeration объявляет два метода:

public abstract boolean **hasMoreElements()**

Возвращает true, если перебор элементов перечисления еще не закончен. Метод может многократно вызываться между последовательными вызовами nextElement.

public abstract Object **nextElement()**

Возвращает следующий элемент перечисления. Вызовы этого метода осуществляют последовательный перебор всех элементов. Если следующего элемента не существует, возбуждается исключение NoSuchElemen tException.

Приведем типичный цикл, в котором Enumeration используется для перебора элементов класса-коллекции, в данном случае — элементов хеш-таблицы:

```
Enumeration e = table.elements();
while (e.hasMoreElements())
     doSomethingWith(e.nextElement());
```

Контракт Enumeration не гарантирует *фиксации исходного состояния (snapshot guarantee)*. Другими словами, если содержимое коллекции изменяется во время итерации, это может отразиться на значениях, возвращаемых методами. Например, если в реализации nextElement используется содержимое исходной коллекции, то удаление объектов из списка во время перебора может иметь разрушительные последствия. Если бы исходное состояние было зафиксировано, то работа с объектами осуществлялась бы в том виде, в каком они находились на момент создания Enumeration. Вы можете рассчитывать на фиксацию исходного состояния лишь в том случае, если метод, возвращающий объект Enumeration, явно это гарантирует.

12.3. Реализация интерфейса Enumeration

При разработке новых классов-коллекций может возникнуть необходимость в собственной реализации Enumeration. Приведенный выше класс Which Chars фактически представляет собой коллекцию для работы с набором символов исходной строки. Следующий класс реализует интерфейс Enumeration для того, чтобы возвращать символы, представленные объектом BitSet в WhichChars:

```
class EnumerateWhichChars implements Enumeration {
     private BitSet bits;
     private int pos;      // следующая проверяемая позиция
     private int setSize;  // количество бит (для оптимизации)

     EnumerateWhichChars(BitSet whichBits) {
          bits = whichBits;
          setSize = whichBits.size();
          pos = 0;
     }

     public boolean hasMoreElements() {
          while (pos << setSize && !bits.get(pos))
               pos++;
          return (pos << setSize);
     }

     public Object nextElement() {
          if (hasMoreElements())
               return new Character((char)pos++);
          else
               return null;
     }
}
```

Класс перебирает биты, входящие в BitSet, и возвращает объекты Character со значениями символов, которым соответствуют установленные биты в объекте BitSet. Метод hasMoreElements перемещает текущую позицию к следующему возвращаемому элементу. Он написан так, чтобы его можно было многократно использовать для каждого вызова nextElement.

Теперь в класс WhichChars необходимо включить метод, который возвращает объект-перечисление:

```
public Enumeration characters() {
     return new EnumeratrWhichChars(used);
}
```

Обратите внимание: метод characters возвращает объект класса Enumeration, а не EnumerateWhichChars. Класс EnumerateWhichChars не предназначен для открытого использования, поэтому реализацию перечисления можно скрыть. Если только вы не захотите возвращать объект-перечисление с новыми открытыми возможностями, следует скрывать тип объекта, чтобы оставить для себя возможность изменить его реализацию по своему усмотрению.

12.4. Класс Vector

Класс Vector предназначен для работы с массивом переменного размера, состоящим из элементов Object. Новые элементы могут добавляться в начало, середину или конец вектора, и к любому элементу можно обратиться посредством индекса. Массивы в языке Java имеют фиксированный размер, так что объекты Vector оказываются полезными в тех случаях, когда в момент создания массива неизвестно максимальное количество сохраняемых элементов или это количество велико, а достигается оно редко.

Методы класса Vector делятся на три категории:

- Методы для модификации вектора.
- Методы для получения объектов, хранящихся в векторе.
- Методы, управляющие процессом расширения вектора, когда его емкости оказывается недостаточно.

Безаргументный конструктор создает объект Vector, размер которого регулируется в соответствии с принятыми по умолчанию правилами. Другие конструкторы рассматриваются ниже, вместе с методами управления размером вектора.

Многие методы класса изменяют содержимое вектора. Все они, кроме setElements, при необходимости осуществляют динамическое изменение размера вектора в соответствии со своими потребностями.

public final synchronized void **setElementAt(Object obj, int index)**

Присваивает obj элементу вектора с индексом index. Старое значение этого элемента пропадает. При задании индекса, превышающего текущий размер вектора, возбуждается исключение IndexOutOfBoundsException. Чтобы убедиться в корректности индекса перед его применением, используйте метод setSize (см. ниже).

public final synchronized void **removeElementAt(Object obj, int index)**

Удаляет элемент вектора с индексом index. Элементы, находящиеся после удаленного, сдвигаются к началу, а размер вектора уменьшается на 1.

public final synchronized void **insertElementAt(Object obj, int index)**

Вставляет элемент obj в позицию index. Элементы, следующие после удаленного, сдвигаются, чтобы освободить место для вставки.

public final synchronized void **addElement(Object obj)**

Добавляет элемент obj к концу вектора.

public final synchronized boolean **removeElement(Object obj)**

Эквивалентен методу indexOf(obj) и — в случае удачного поиска — вызову removeElementAt для найденного индекса. Если объект не является элементом вектора, removeElement возвращает false (метод indexOf описывается ниже).

public final synchronized void **removeAllElements()**

Удаляет все элементы вектора. Вектор становится пустым.

Класс Polygon предназначен для хранения списка объектов Point, которые представляют собой вершины многоугольника:

```
import java.util.Vector;

public class Polygon {
     private Vector vertices = new Vector();

     public void add(Point p) {
          vertices.addElements(p);
     }

     public void remove(Point p) {
          vertices. RemoveElement(p);
     }

     public int numVertices() {
          return vertices.size();
     }

     // ... другие методы ...
}
```

Существует ряд методов, предназначенных для просмотра содержимого вектора. При задании недопустимого индекса возбуждается исключение Index OutOfBoundsException. Все методы, которые ищут элемент в векторе, используют метод Object.equals для сравнения искомого объекта с элементами Vector.

public final synchronized Object **elementAt(int index)**

Возвращает элемент с индексом index.

public final boolean **contains(Object obj)**

Возвращает true, если obj является элементом вектора.

public final synchronized int **indexOf(Object obj, int index)**

Ищет первое вхождение заданного объекта obj начиная с позиции index, и возвращает его индекс или –1, если объект не найден.

public final int **indexOf(Object obj)**

Эквивалентен indexOf(obj,0).

public final synchronized int **lastIndexOf(Object obj,**

int index)

Осуществляет поиск obj в обратном направлении от позиции index и возвращает его индекс или –1, если объект не найден.

public final int **lastIndexOf(Object obj)**

Эквивалентен lastIndexOf(obj,size()-1).

public final synchronized void **copyInto(Object[] anArray)**

Копирует элементы вектора в заданный массив. Метод может применяться для “фотографирования” содержимого вектора.

public final synchronized Enumeration **elements()**

Возвращает Enumeration для текущего состава элементов. Для последовательной выборки элементов возвращаемого объекта применяются методы Enumeration. Исходное состояние при этом не фиксируется, поэтому для получения “фотографии” содержимого вектора пользуйтесь методом copy Into.

public final synchronized Object **firstElement()**

Возвращает первый элемент вектора. Если вектор пуст, возбуждается исключение NoSuchElementException.

public final synchronized Object **lastElement()**

Возвращает последний элемент вектора. Если вектор пуст, возбуждается исключение NoSuchElementException. Пара методов firstElement/ lastElement может использоваться для перебора элементов вектора, но существует риск изменения вектора во время выполнения цикла. Для получения “фотографии” содержимого вектора пользуйтесь методом copyInto.

Размер вектора равен количеству элементов, содержащихся в нем. Чтобы изменить размер вектора, можно добавлять или удалять элементы либо вызвать метод setSize или trimSize:

public final int **size()**

Возвращает количество элементов, содержащихся в векторе. Обратите внимание, что эта величина отличается от емкости вектора.

public final boolean **isEmpty()**

Возвращает true, если вектор не содержит ни одного элемента.

public final synchronized void **trimToSize()**

Сокращает емкость вектора до текущего размера. Этот метод используется для минимизации объема памяти, когда вектор находится в устойчивом состоянии. Последующие добавления элементов к вектору приведут к его увеличению.

public final synchronized void **setSize(int newSize)**

Устанавливает размер вектора равным newSize. Если при этом вектор сокращается, то элементы за его концом теряются; если вектор увеличивается, то новые элементы равны null.

Правильное управление емкостью вектора существенно влияет на эффективность работы с ним. Если приращение емкости оказывается небольшим, а количество добавляемых элементов велико, то слишком много времени будет тратиться на то, чтобы повторно создавать новый буфер увеличенного размера и копировать в него содержимое вектора. Лучше сразу создавать вектор, емкость которого равна максимальному возможному размеру или близка к нему. Если вы знаете, сколько элементов будет добавлено в вектор, используйте метод ensureCapacity для однократного увеличения емкости вектора. Параметры управления емкостью задаются при конструировании вектора. Для создания объектов Vector применяются следующие конструкторы:

public **Vector(int initialCapacity, int capacityIncrement)**

Создает пустой вектор с заданной исходной емкостью и запоминает ее приращение. Приращение, равное 0, означает удвоение емкости буфера при каждом его увеличении; в противном случае буфер увеличивается на capacityIncrement элементов.

public **Vector(int initialCapacity)**

Эквивалентен Vector(initialCapacity, 0).

public **Vector()**

Конструирует пустой вектор со значениями исходной емкости и приращения, заданными по умолчанию.

public final int **capacity()**

Возвращает текущую емкость вектора — количество элементов, которые могут храниться в векторе без увеличения его размера.

public final synchronized void **ensureCapacity ••** (int minCapacity)

Метод гарантирует, что емкость вектора будет не ниже заданной, и при необходимости увеличивает текущую емкость.

Приведем метод класса Polygon, который присоединяет к объекту вершины другого многоугольника:

```
public void merge(Polygon other) {
     int otherSize = other.vertices.size();

     vertices.enusreCapacity(vertices.size() + otherSize);
     for (int i = 0; i << otherSize; i++)
          vertices.addElement(other.vertices.elementAt(i));
}
```

В этом примере метод ensureCapacity используется для того, чтобы с добавлением новых вершин емкость вектора увеличивалась не более одного раза.

Реализация vector.toString выдает строку с полным описанием вектора, включающую результат вызова toString для каждого из содержащихся в нем элементов.

Помимо этих открытых методов, подклассы Vector могут использовать защищенные поля класса. Соблюдайте осторожность в работе с ними — например, методы Vector предполагают, что размер буфера превышает количество элементов вектора.

public Object **elementData[]**

Буфер, в котором хранятся элементы вектора.

public int **elementCount**

Текущее количество элементов в буфере.

public int **capacityIncrement**

Количество элементов, которое добавляется к емкости вектора при заполнении буфера elementData. Если значение этого поля равно 0, то размер буфера удваивается при каждой необходимости его увеличения.

Упражнение 12.1

Напишите программу, которая открывает файл и читает из него строки (по одной), сохраняя каждую строку в объекте Vector, сортируемом методом String.compareTo. В этом вам может пригодиться класс для чтения строк из файла, созданный в упражнении 11.2.

12.5. Класс Stack

Класс Stack расширяет Vector, добавляя методы для реализации простейшего стека объектов Object, построенного по принципу LIFO (“последним пришел, первым вышел”). Метод push заносит объект в стек, а pop выталкивает верхний элемент стека. Метод peek возвращает значение верхнего элемента стека, при этом сам элемент остается в стеке. Метод empty возвращает true, если стек пуст. Попытка вызова pop или peek для пустого объекта-стека приводит к возбуждению исключения EmptyStackException.

Чтобы выяснить, насколько далеко расположен тот или иной элемент от вершины стека, применяется метод search; 1 соответствует вершине стека. Если объект не найден, возвращается –1. Для проверки совпадения искомого объекта с объектами в стеке применяется метод Object.equals.

В приведенном ниже примере класс Stack используется для слежения за тем, у кого в данный момент находится некоторый предмет — скажем, игрушка. Имя исходного владельца попадает в стек первым. Когда кто-нибудь одалживает у него игрушку, имя должника также заносится в стек. При возврате игрушки имя должника выталкивается из стека. Последнее имя должно всегда оставаться в стеке, так как в противном случае будет утрачена информация о владельце.

```
import java.util.Stack;

public class Borrow {
     private String itemName;
     private Stack hasIt = new Stack();

     public Borrow(String name, String owner) {
          itemName = name;
          hasIt.push(owner);      // первым следует имя владельца
     }

     public void borrow(String borrower) {
          hasIt.push(borrower);
     }

     public String currentHolder() {
          return (String)hasIt.peek();
     }

     public String returnIt() {
          String ret = (String)hasIt.pop();
          if (hasIt.empty())      // случайно вытолкнутый владелец
               hasIt.push(ret);   // вернуть его обратно
          return ret;
     }
}
```

Упражнение 12.2

Добавьте метод, который использует метод search, чтобы определить количество должников.

12.6. Класс Dictionary

Абстрактный класс Dictionary фактически представляет собой интерфейс. В нем определен ряд абстрактных методов, предназначенных для хранения *элемента* с некоторым *ключом* и последующей выборки элемента по ключу. Этот интерфейс является базовым для класса Hashtable, однако класс Dictionary определен отдельно, чтобы другие реализации могли использовать разные алгоритмы для сопоставления ключа с элементом. Класс Dictionary возвращает null, сигнализируя о таких событиях, как отсутствие элемента с заданным ключом; следовательно, ни ключ, ни элемент не могут быть равны null. Если задать значение null для аргумента-ключа или элемента, возбуждается исключение NullPointerException. В случае, если вам понадобится ввести специальный элемент-маркер, следует использовать для него значение, отличное от null.

Класс Dictionary содержит следующие методы:

public abstract Object **put(Object key, Object element)**

Заносит element в словарь с ключом key. Возвращает старый элемент, хранившийся с ключом key, или null, если такого элемента нет.

public abstract Object **get(Object key)**

Возвращает объект, занесенный в словарь с ключом key, или null, если ключ не определен.

public abstract Object **remove(Object key)**

Удаляет из словаря элемент с ключом key и возвращает значение удаленного элемента или null, если ключ не определен.

public abstract int **size()**

Возвращает количество элементов в словаре.

public abstract boolean **isEmpty()**

Возвращает true, если словарь не содержит ни одного элемента.

public abstract Enumeration **keys()**

Возвращает объект-перечисление для всех ключей, входящих в словарь.

public abstract Enumeration **elements()**

Возвращает объект-перечисление для всех элементов, входящих в словарь.

Объекты-перечисления, возвращаемые методами keys и elements, не гарантируют фиксации исходного состояния, однако при создании класса, в котором используется Dictionary, можно осуществить такую гарантию в вашей собственной реализации этих методов.

12.7. Класс Hashtable

Хеш-таблицы представляют собой распространенный механизм для хранения пар ключ/элемент. Они обладают такими достоинствами, как универсальность и простота, а также высокая эффективность при хорошо продуманной генерации хеш-кода. Класс Hashtable реализует интерфейс Dictionary. Он обладает определенной емкостью и средствами, определяющими момент увеличения таблицы. Расширение хеш-таблицы требует повторного хеширования всех ее элементов в соответствии с их новым положением в увеличенной таблице, так что важно обеспечить однократное изменение таблицы.

Другой фактор, влияющий на эффективность хеш-таблицы, — процесс генерации хеш-кода по ключу. Конфликты хеш-кодов должны происходить как можно реже. Хеш-коды обязаны равномерно распределяться по диапазону возможных значений, который для класса Hashtable совпадает с полным диапазоном типа int. Если различные ключи часто приводят к одним и тем же хеш-кодам, то некоторая часть хеш-таблицы быстро переполнится, в результате чего пострадает эффективность.

Значение хеш-кода возвращается методом hashCode для объекта, являющегося ключом. По умолчанию каждый объект имеет уникальный хеш-код. Использование в качестве ключей случайно выбранных объектов приводит к порождению различных хеш-кодов. Классы String, BitSet и большинство других, переопределяющих метод equal, обычно переопределяют и hashCode. Это важно, поскольку класс Hashtable использует хеш-код для нахождения набора ключей, которые могут совпадать с заданным, и вызывает equal для каждого из таких объектов, пока не будет найден совпадающий. Если для некоторых объектов equal и hashCode окажутся несовместимыми, то при использовании объектов этого типа в качестве ключей Hastable их поведение окажется непредсказуемым.

Пример использования класса Hastable приведен в классе Attributed Impl (см. раздел “Реализация интерфейсов”), в котором объект Hashtable использован для хранения атрибутов объекта. В этом примере ключами являются строковые объекты, представляющие собой имена атрибутов, а объекты Attr были значениями атрибутов.

Кроме методов, входящих в класс Dictionary (get, put, remove, size, isEmpty, keys и elements), Hastable содержит следующие методы:

public synchronized boolean **containsKey(Object key)**

Возвращает true, если хеш-таблица содержит элемент с заданным ключом.

public synchronized boolean **contains(Object element)**

Возвращает true, если заданный element является элементом хеш-таблицы. Данная операция является более сложной, чем метод containsKey, поскольку хеш-таблица спроектирована с расчетом на эффективный поиск ключей, а не элементов.

public synchronized void **clear()**

Делает хеш-таблицу пустой.

public synchronized Object **clone()**

Создает дубликат хеш-таблицы. Ключи и элементы при этом *не* дублируются.

Объекты Hashtable автоматически увеличиваются, когда они становятся слишком заполненными. Под выражением “слишком заполненными” понимается превышение *показателя загрузки* таблицы, который представляет собой отношение количества элементов к текущей емкости таблицы. Когда таблица увеличивается, ее новая емкость примерно вдвое превышает текущую. Для повышения эффективности следует выбирать емкость, представленную простым числом, чтобы при увеличении объекта Hastable также было выбрано ближайшее простое число. Исходная емкость хеш-таблицы и показатель загрузки могут задаваться в конструкторах Hashtable:

public **Hashtable()**

Конструирует новую, пустую хеш-таблицу с принятой по умолчанию исходной емкостью и показателем загрузки, равным 0, 75.

public **Hashtable(int initialCapacity)**

Конструирует новую, пустую хеш-таблицу с заданной емкостью initial Capacity и принятым по умолчанию показателем загрузки, равным 0,75.

public **Hashtable(int initialCapacity, float loadFactor)**

Конструирует новую, пустую хеш-таблицу с заданной емкостью и показателем загрузки loadFactor, который представляет собой число, лежащее в диапазоне 0,0–1,0 и определяющее момент увеличения хеш-таблицы. Если количество элементов хеш-таблицы превышает текущую емкость, умноженную на показатель загрузки, то хеш-таблица автоматически увеличивается.

Емкость по умолчанию выбирается “разумной”, причем критерий разумности зависит от реализации. После конструирования объекта Hashtable невозможно изменить показатель загрузки или явно задать новую емкость.

При увеличении объекта Hashtable повторное хеширование осуществляется методом rehash. Метод rehash является защищенным, так что расширенные классы могут вызывать его по своему усмотрению, когда они решат, что наступило время увеличить емкость таблицы. Задать новый размер при этом невозможно — он всегда вычисляется методом rehash.

При реализации метод Hashtable.toString возвращает строку, которая полностью описывает содержимое таблицы, включая результаты вызова to String для всех ключей и элементов, входящих в нее.

Упражнение 12.3

В классе WhichChars, имеется проблема с пометкой символов в верхней части диапазона Unicode, поскольку высокие значения символов оставляют много неиспользованных битов в нижней части диапазона. Решите эту проблему с помощью класса Hashtable, сохраняя объект Character для каждого обнаруженного символа. Не забудьте написать свой класс-перечисление.

Упражнение 12.4

Теперь воспользуйтесь классом Hashtable, чтобы сохранять объект BitSet для каждого нового старшего байта (старшие 8 бит), встречающегося во входной строке, причем каждый BitSet должен содержать младшие байты вместе с данным старшим байтом. Не забудьте написать свой класс-перечисление.

Упражнение 12.5

Напишите программу, которая пользуется объектом StreamTokenizer для разбиения входного файла на слова и подсчета количества слов в файле, с выводом результата.

12.8. Класс Properties

Еще один распространенный вариант пары ключ/элемент — *список свойств*, состоящий из строковых имен и связанных с ними строковых элементов. Эта разновидность словаря часто обладает вспомогательным набором элементов по умолчанию для свойств, отсутствующих в таблице. Класс Properties является расширением Hashtable. Практически для всех манипуляций со списками свойств используются методы Hashtable, однако для получения свойств применяется один из двух методов getProperty:

public String **getProperty(String key)**

Возвращает элемент для заданного ключа key. Если ключ отсутствует в списке свойств, просматривается список свойств по умолчанию (если он существует). Метод возвращает null, если свойство не найдено.

public String **getProperty(String key, String defaultElement)**

Возвращает элемент для заданного ключа key. Если ключ отсутствует в списке свойств, просматривается список свойств по умолчанию (если он существует). Если элемент отсутствует в обоих списках, возвращается строка defaultElement.

Класс Properties содержит два конструктора: один вызывается без аргументов, а второму передается объект Properties, который представляет вспомогательный список свойств по умолчанию. Если поиск в основном списке свойств оказывается неудачным, то просматривается вспомогательный объект Properties, который, в свою очередь, может иметь собственный вспомогательный объект со свойствами по умолчанию, и так далее. Цепочка основных и вспомогательных списков свойств может иметь произвольную длину.

public **Properties()**

Создает пустой список свойств.

public **Properties(Properties defaults)**

Создает пустой список свойств с заданным вспомогательным объектом Properties для поиска свойств, отсутствующих в основном списке.

Если список свойств состоит только из строковых ключей и элементов, можно записывать или считывать его из файла или иного потока ввода/ вывода с помощью следующих методов:

public void **save(OutputStream out, String header)**

Сохраняет содержимое списка свойств в OutputStream. Строка header записывается в выходной поток в виде комментария, состоящего из одной строки. Не пользуйтесь многострочными заголовками-комментариями, иначе сохраненный список свойств не удастся загрузить. В файле сохраняются только свойства, входящие в основной список, но не во вспомогательный.

public synchronized void **load(InputStream in)** throws

IOException

Загружает список свойств из InputStream. Предполагается, что список свойств был ранее сохранен методом save. Метод загружает свойства только в основной список, но не во вспомогательный.

Для получения объекта Enumeration, представляющего собой “фотографию” ключей в списке свойств, применяется метод propertyNames:

public Enumeration **propertyNames()**

Создает объект-перечисление с перечнем всех ключей. Метод гарантирует фиксацию исходного состояния.

public void **list(PrintStream out)**

Выводит свойства из списка в заданный поток PrintStream. Метод полезен во время отладки.

После создания объекта невозможно изменить его вспомогательный перечень свойств. Если это все же необходимо сделать, можно создать подкласс класса Properties и изменить значение защищенного поля defaults, содержащее список свойств по умолчанию.

12.9. Классы Observer/Observable

Типы Observer/Observable предоставляют протокол, в соответствии с которым произвольное количество объектов-наблюдателей Observer получают уведомления о каких-либо изменениях или событиях, относящихся к произвольному количеству объектов Observable. Объект Observable производится от подкласса Observable, благодаря чему можно вести список объектов Observer, уведомляемых об изменениях в объекте Observable. Все объекты- “наблюдатели”, входящие в список, должны реализовывать интерфейс Observer. Когда с наблюдаемым объектом происходят изменения, заслужи вающие внимания, или случаются некоторые события, которые представляют интерес для Observer, вызывается метод notifyObservers объекта Observable, который обращается к методу update для каждого из объектов Observer. Метод update интерфейса Observable выглядит следующим образом:

public abstract void **update(Observable obj, Object arg)**

Метод вызывается, когда объект Observable должен сообщить наблюдателям об изменении или некотором событии. Параметр arg дает возможность передачи произвольного объекта, содержащего описание изменения или события в объекте Observer.

Механизм Observer/Observable проектировался с расчетом на универсальность. Каждый класс Observable сам определяет, когда и при каких обстоятельствах должен вызываться метод update объекта Observer.

Класс Observable реализует методы для ведения списка объектов Observer, для установки флага, сообщающего об изменении объекта, а также для вызова метода update любого из объектов Observer. Для ведения списка объектов Observer используются следующие методы:

public synchronized void **addObserver(Observer o)**

Добавляет аргумент o типа Observer к списку объектов-наблюдателей.

public synchronized void **deleteObserver(Observer o)**

Удаляет аргумент o типа Observer из списка объектов-наблюдателей.

public synchronized void **deleteObservers()**

Удаляет все объекты Observer из списка наблюдателей.

public synchronized int **countObservers()**

Возвращает количество объектов-наблюдателей.

Следующие методы извещают объекты Observer о произошедших изменениях:

public synchronized void **notifyObservers(Object arg)**

Уведомляет все объекты Observer о том, что с наблюдаемым объектом что-то произошло, после чего сбрасывает флаг изменения объекта. Для каждого объекта-наблюдателя, входящего в список, вызывается его метод update, первым параметром которого является объект Observable, а вторым — arg.

public void **notifyObservers()**

Эквивалентен notifyObservers(null).

Приведенный ниже пример показывает, как протокол Observer/Observable может применяться для наблюдения за пользователями, зарегистрированными в системе. Сначала определяется класс Users, расширяющий Observable:

```
import java.util.*;

public class Users extends Observable {
     private Hashtable loggedIn = new Hashtable();
     public void login(String name, String password)
          throws BadUserException
     {
          // метод возбуждает исключение BadUserException
          if (!passwordValid(name, password)
               throw new BadUserException(name);

          UserState state = new UserState(name);
          loggedIn.put(name, state);
          setChanged();
          notifyObservers(state);

     }

     public void logout(UserState state) {
          loggedIn.remove(state.name());
          setChanged();
          notifyObservers(state);
     }

     // ...
}
```

Объект Users содержит список активных пользователей и для каждого из них заводит объект UserState. Когда кто-либо из пользователей входит в систему или прекращает работу, то всем объектам Observer передается его объект UserState. Метод notifyObservers рассылает сообщения наблюдателям лишь в случае изменения состояния наблюдаемого объекта, так что мы должны также вызвать метод setChanged для Users, иначе notifyObservers ничего не сделает. Кроме метода setChanged, существует еще два метода для работы с флагом изменения состояния: clearChanged помечает объект Observable как неизменявшийся, а hasChanged возвращает логическое значение флага.

Ниже показано, как может выглядеть реализация update для объекта Observer, постоянно следящего за составом зарегистрированных пользователей:

```
import java.util.*;

public class Eye implements Observer {
     Users watching;

     public Eye(Users users) {
          watching = users;
          watching.addObserver(this);
     }

     public void update(Observable users, Object whichState)
     {
          if (users != watching)
               throw new IllegalArgumentException();

          UserState state = (UserState)whichState;
          if (watching.loggedIn(state))    // вход в систему
               addUser(state);             // внести в список
          else
               removeUser(state);          // удалить из списка
     }
}
```

Каждый объект Eye наблюдает за конкретным объектом Users. Когда пользователь входит в систему или прекращает работу, объект Eye извещается об этом, поскольку в его конструкторе вызывается метод addObserver для объекта User, в котором объект Eye указывается в качестве объекта-наблюдателя. При вызове метода update происходит проверка на правильность параметров и изменение выводимой информации в зависимости от того, вошел ли данный пользователь в систему или вышел.

Проверка того, что происходит с объектом UserState, в данном случае выполняется просто. Впрочем, ее можно избежать — для этого следует вместо самого объекта UserState передавать объект-оболочку, который описывает, что и с кем происходит. Такой вариант выглядит нагляднее и облегчает добавление новых возможностей без нарушения существующего программного кода.

Механизм Observer/Observable отчасти напоминает механизм wait/ notify для потоков, описанный на стр. , однако он отличается большей гибкостью и меньшим количеством ограничений. Механизм потоков гарантирует, что синхронный доступ защитит программу от нежелательных эффектов многозадачности. Механизм наблюдения позволяет организовать между участниками любую связь, не зависящую от используемых потоков. В обоих механизмах предусмотрен поставщик информации (Observable и объект, вызывающий notify) и ее потребитель (Obs e rver и объект, вызывающий wait), однако они удовлетворяют различные потребности. Используйте wait/ notify, когда механизм должен учитывать специфику потоков, и Observer/ Observable для более общих случаев.

Упражнение 12.6

Создайте реализацию интерфейса Attributed, в которой механизм Observer/Observable используется для уведомления наблюдателей об изменениях, происходящих с объектами.

12.10. Класс Date

Класс Date предоставляет в распоряжение программиста механизм для вычислений, связанных с датами и временем, а также для вывода их результатов (по умолчанию вывод осуществляется в формате, используемом в Соединенных Штатах). Вы можете установить дату и определить ее, при необходимости учитывая локальный часовой пояс.

Предполагается, что класс Date работает в соответствии со стандартом UTC (Coordinated Universal Time — координированное универсальное время), однако это не всегда возможно. Неточности возникают из-за механизмов обращения со временем, используемых в операционной системе. /Почти все современные системы временного исчисления предполагают, что одни сутки состоят из 24*60*60 секунд. В системе UTC примерно раз в год к суткам прибавляется дополнительная секунда, называемая "переходной". Большинство компьютерных часов не обладает необходимой точностью, чтобы отражать этот факт, поэтому класс Date также не учитывает его. Некоторые компьютерные стандарты определены в GMT - это название является общеупотребительным, тогда как UT представляет собой "научное" название того же самого стандарта. Различие между UTC и UT состоит в том, что стандарт UT основан на атомных часах, а UTC - на астрономических наблюдениях. На практике отличие оказывается пренебрежимо малым. Ссылки на дополнительную информацию приведены в разделе "Библиография"./ Компоненты дат задаются в единицах, принятых в стандарте UTC, и принадлежат соответствующим диапазонам. Значение, выходящее за пределы диапазона, интерпретируется правильно — например, 32 января эквивалентно 1 февраля. Диапазоны определяются следующим образом:

год год после 1900, со всеми цифрами

месяц 0–11

дата день месяца, 1–31

час 0–23

минуты 0–59

секунды 0–61 (с учетом переходной секунды).

Класс Date прост в использовании, но содержит много методов:

public **Date()**

Создает объект Date, соответствующий текущей дате/времени.

public **Date(int year, int month, int date, int hrs, int min, •• int sec)**

Создает объект Date, соответствующий заданной дате/времени.

public **Date(int year, int month, int date, int hrs, int min)**

Эквивалентно Date(year, month, date, hrs, min, 0), то есть началу текущей минуты.

public **Date(int year, int month, int date)**

Эквивалентно Date(year, month, date, 0, 0, 0), то есть полуночи заданной даты.

public **Date(String s)**

Создает дату из строки в соответствии с синтаксисом, принятым в методе parse (см. ниже).

public static long **UTC(int year, int month, int date, •• int hrs, int min, int sec)**

Вычисляет значение в стандарте UTC для указанной даты.

public static long **parse(String s)**

Анализирует строку, представляющую время, и возвращает полученное значение. Метод может работать со многими форматами, но важнее всего, что он воспринимает даты в стандарте IETF: “Sat, 12 Aug 1995 13:30:00 GMT”. Он также понимает сокращения для часовых поясов, используемые в США, но в общем случае должно использоваться смещение для часового пояса: “Sat, 12 Aug 1995 13:30:00 GMT+0430" (4 часа 30 минут к западу от Гринвичского меридиана). Если часовой пояс не указан, предпо-лагается локальный часовой пояс. При указании часового пояса стандарты GMT и UTC считаются эквивалентными.

public **Date(long date)**

Создает объект-дату. Перед созданием объекта Date происходит нормализация полей. Метод воспринимает в качестве параметра значение, возвращаемое методами parse и UTC.

public int **getYear()**

Возвращает год, всегда следующий после 1900.

public int **getMonth()**

Возвращает значение месяца в диапазоне 0–11 (с января по декабрь соответственно).

public int **getDate()**

Возвращает число месяца.

public int **getDay()**

Возвращает день недели в диапазоне 0–6 (с воскресенья до субботы соответственно).

public int **getHours()**

Возвращает час в диапазоне 0–23 (значение 0 соответствует полуночи).

public int **getMinutes()**

Возвращает минуты в диапазоне 0–59.

public int **getSeconds()**

Возвращает секунды в диапазоне 0–61.

public long **getTime()**

Возвращает время в формате UTC.

public int **getTimezoneOffset()**

Возвращает смещение часового пояса в минутах. Результат учитывает время суток, и на него может влиять летнее время — если оно учитывается, то в зависимости от времени года может присутствовать дополнительное смещение часового пояса.

public void **setYear(int year)**

Устанавливает значение года. Год должен быть после 1900.

public void **setMonth(int month)**

Устанавливает месяц.

public void **setDate(int date)**

Устанавливает число месяца.

public void **setDay(int day)**

Устанавливает день недели.

public void **setHours(int hours)**

Устанавливает час.

public void **setMinutes(int minutes)**

Устанавливает минуты.

public void **setSeconds(int seconds)**

Устанавливает секунды.

public boolean **before(Date other)**

Возвращает true, если дата объекта наступает раньше даты other.

public boolean **after(Date other)**

Возвращает true, если дата объекта наступает после даты other.

public boolean **equals(Object other)**

Возвращает true, если дата объекта представляет в стандарте UTC ту же дату, что и other.

public int **hashCode()**

Вычисляет хеш-код, чтобы объекты Date могли использоваться в качестве ключей в хеш-таблицах.

public String **toString()**

Преобразует дату в String, например: “Fri Oct 13 14:33:57 EDT 1995". /Формат строки совпадает с форматом, используемым в функции ctime в соответствии со стандартом ANSI C./

public String **toLocaleString()**

Преобразует дату в String с использованием национального формата. Другими словами, дата будет представлена в виде, принятом в локализованной операционной системе. Например, жители США привыкли видеть месяц перед числом (“June 13"), тогда как в Европе обычно используется обратный порядок (”13 June ").

public String **toGMTString()**

Преобразует дату в String с использованием конвенции Internet GMT, в форме

d mon yyyy hh:mm:ss GMT

где d — число месяца (одна или две цифры), mon — первые три буквы месяца, yyyy — год из четырех цифр, hh — часы (0–23), mm — минуты, а ss — секунды. Информация о местном часовом поясе при этом игнорируется.

12.11. Класс Random

Объекты класса Random предназначены для работы с независимыми последовательностями псевдослучайных чисел. Если вам нужна последовательность типа double и вас не интересует порядок следования чисел, можно воспользоваться методом java.lang.Math.random — он создает объект Random при первом вызове и в дальнейшем возвращает псевдослучайные числа из этого объекта. Чтобы иметь больше средств для контроля за последовательностью (например, чтобы иметь возможность задать стартовое значение), создайте объект Random и получайте числа от него.

public **Random()**

Создает новый генератор случайных чисел. Стартовое значение определяется на основании текущего времени.

public **Random(long seed)**

Создает новый генератор случайных чисел с заданным стартовым значением. Два объекта Random, созданные с одинаковым seed, будут порождать совпадающие последовательности псевдослучайных чисел.

public synchronized void **setSeed(long seed)**

Устанавливает стартовое значение генератора случайных чисел равным seed. Метод может быть вызван в любой момент — в результате произойдет сброс последовательности и последующее ее порождение на основе стартового значения.

public int **nextInt()**

Возвращает псевдослучайное значение типа int, равномерно распределенное между величинами Integer.MIN_VALUE и Integer.MAX_VALUE включительно.

public long **nextLong()**

Возвращает псевдослучайное значение типа long, равномерно распределенное между величинами Long.MIN_VALUE и Long.MAX_VALUE включительно.

public float **nextFloat()**

Возвращает псевдослучайное значение типа float, равномерно распределенное между величинами Float.MIN_VALUE и Float.MAX_VALUE включительно.

public double **nextDouble()**

Возвращает псевдослучайное значение типа double, равномерно распределенное между величинами Double.MIN_VALUE и Double.MAX_VALUE включительно.

public synchronized double **nextGaussian()**

Возвращает псевдослучайное значение типа double, подчиняющееся распределению Гаусса, с математическим ожиданием 0,0 и стандартным отклонением 1,0.

Упражнение 12.7

Для известного количества шестигранных кубиков можно вычислить теоретическую вероятность выпадения каждой из возможных сумм. Например, для двух шестигранных кубиков вероятность выпадения семи очков составляет 1/6. Напишите программу, которая сравнивает теоретическое распределение очков для известного числа кубиков с экспериментальными данными, полученными в результате многочисленных “бросков”, использующих Random для генерации чисел между 1 и 6. Имеет ли значение выбор метода, генерирующего числа?

Упражнение 12.8

Напишите программу, которая тестирует метод nextGaussian, отображая распределение для большого количества чисел в виде графика — его роль может играть гистограмма из символов *.

12.12. Класс String Tokenizer

Класс StringTokenizer делит строку на части, используя для этого символы-разделители. Последовательность лексем, выделенных из строки, фактически представляет собой упорядоченный объект-перечисление, поэтому класс StringTokenizer реализует интерфейс Enumeration. Вы можете передавать объекты StringTokenizer методам, которые обрабатывают объекты-перечисления, или воспользоваться методами Enumeration для проведения итераций. StringTokenizer также предоставляет ряд методов с более конкретной типизацией. Перечисление StringTokenizer не гарантирует фиксации исходного состояния, но это не имеет значения, поскольку объекты String доступны только для чтения. Например, для деления строки на лексемы, отделяемые запятыми и пробелами, может использоваться следующий цикл:

```
String str = "Gone, and forgotten";
StringTokenizer tokens = new StringTokenizer(str, " ,");
while (tokens.hasMoreTokens())
        System.out.println(tokens.nextToken());
```

Запятая включена в список разделителей в конструкторе StringTokenizer для того, чтобы анализатор “поглощал” запятые вместе с пробелами, оставляя только слова, которые возвращаются по одному. Результат работы примера выглядит следующим образом:

Gone

and

forgotten

Класс StringTokenizer содержит несколько методов, которые определяют, что считать словом, следует ли отдельно обрабатывать строки и числа, и так далее:

public **StringTokenizer(String str, String delim, boolean returnTokens)**

Конструирует объект StringTokenizer для строки str с использованием символов из строки delim в качестве разделителей. Логическое значение returnTokens определяет, следует ли возвращать разделители как лексемы или же пропускать их. В первом случае каждый символ-разделитель возвращается отдельно.

public **StringTokenizer(String str, String delim)**

Эквивалентен StringTokenizer(str, delim, false), то есть разделители пропускаются.

public **StringTokenizer(String str)**

Эквивалентен StringTokenizer(str, “ \t\n\r”), то есть используются стандартные символы-разделители.

public boolean **HasMoreTokens()**

Возвращает true, если в строке еще остаются лексемы.

public String **nextToken()()**

Возвращает следующую лексему в строке. Если лексем больше нет, возбуждается исключение NoSuchElementException.

public String **nextToken(String delim)**

Заменяет набор символов-разделителей на символы из строки delim и возвращает следующую лексему. Невозможно изменить набор символов-разделителей, не получая следующей лексемы.

public int **countTokens()**

Возвращает количество лексем, остающихся в строке при использовании текущего набора разделителей. Оно равно числу возможных вызовов next Token перед тем, как будет возбуждено исключение. Если вам понадобилось узнать количество лексем, то этот метод работает быстрее циклического вызова nextToken, поскольку строки-лексемы только подсчитываются, без расходов на конструирование и возврат значения.

Два метода класса StringTokenizer, унаследованные от интерфейса Enumeration (hasMoreElements и nextElement), эквивалентны методам hasMoreTokens и nextToken соответственно.

Если вам понадобится более мощный механизм для деления строки или другого входного значения на лексемы, обратитесь к разделу “Класс Stream Tokenizer”, в котором описывается класс с большими возможностями по части распознавания ввода. Чтобы воспользоваться классом Stream Tokenizer для строки, создайте для нее объект StringBufferInputStream. Тем не менее во многих случаях бывает достаточно и простого класса String Tokenizer.

Упражнение 12.9

Напишите метод, который получает строку, делит ее на лексемы с использованием стандартных символов-разделителей и возвращает новую строку, в которой первая буква каждого слова преобразована в заглавный регистр.

Содержание | Далее

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР® WWW.PITER.COM
© 1997-2002 Издательский дом "Питер". Авторские права охраняются.
Предназначено только для частного использования!
Воспроизведение материалов или частей данной книги в любом виде без письменного разрешения Издательского дома "Питер" запрещено!

Глава 4
ИНТЕРФЕЙСЫ

“Дирижирование” — это когда вы рисуете
свои “проекты” прямо в воздухе, палочкой или руками,
и нарисованное становится “инструкциями” для парней в галстуках, которые в данный момент предпочли бы
оказаться где-нибудь на рыбалке.

Фрэнк Заппа

Основной единицей *проектирования* в Java являются открытые (public) методы, которые могут вызываться для объектов. Интерфейсы предназначены для объявления типов, состоящих только из абстрактных методов и констант; они позволяют задать для этих методов произвольную реализацию. Интерфейс является выражением чистой концепции проектирования, тогда как класс представляет собой смесь проектирования и конкретной реализации.

Методы, входящие в интерфейс, могут быть реализованы в классе так, как сочтет нужным проектировщик класса. Следовательно, интерфейсы имеют значительно больше возможностей реализации, нежели классы.

4.1. Пример интерфейса

В предыдущей главе мы представили читателю класс Attr и показали, как расширить его для создания специализированных типов объектов с атрибутами. Теперь все, что нам нужно, — научиться связывать атрибуты с объектами. Для этого служат два подхода: *композиция* и *наследование*. Вы можете создать в объекте набор определенных атрибутов и предоставить программисту доступ к этому набору. Второй метод состоит в том, что вы рассматриваете атрибуты объекта как составную часть его типа и включаете их в иерархию класса. Оба подхода вполне допустимы; мы полагаем, что хранение атрибутов в иерархии класса приносит больше пользы. Мы создадим тип Attributed, который может использоваться для наделения объектов атрибутами посредством закрепления за ними объектов Attr.

Однако в Java поддерживается только *одиночное наследование (single inheritance)* при реализации — это означает, что новый класс может являться непосредственным расширением всего одного класса. Если вы создаете класс Attributed, от которого порождаются другие классы, то вам либо придется закладывать Attributed в основу всей иерархии, либо программисты окажутся перед выбором: расширять ли им класс Attributed или какой-нибудь другой полезный класс.

Каждый раз, когда вы создаете полезное средство вроде Attributed, возникает желание включить его в корневой класс Object. Если бы это разрешалось, то класс Object вскоре разросся бы настолько, что работать с ним стало бы невозможно.

В Java допускается множественное наследование интерфейсов, так что вместо того, чтобы включать возможности класса Attributed в Object, мы оформим его в виде интерфейса. Например, чтобы наделить атрибутами наш класс небесных тел, можно объявить его следующим образом:

```
class AttributedBody extends Body
implements Attributed
Разумеется, для этого нам понадобится интерфейс Attributed:
interface Attributed {
     void add(Attr newAttr);
     Attr find(String attrName);
     Attr remove(String attrName);
     java.util.Enumeration attrs();
}
```

В данном интерфейсе объявляются четыре метода. Первый из них добавляет новый атрибут в объект Attributed; второй проверяет, включался ли ранее в объект атрибут с указанным именем; третий удаляет атрибут из объекта; четвертый возвращает список атрибутов, закрепленных за объектом. В последнем из них используется интерфейс Enumeration, определенный для классов-коллекций Java. java.util.enumeration подробно рассматривается в главе 12.

Все методы, входящие в интерфейс, неявно объявляются абстрактными; так как интерфейс не может содержать собственной реализации объявленных в нем методов. Поэтому нет необходимости объявлять их с ключевым словом abstract. Каждый класс, реализующий интерфейс, должен реализовать все его методы; если же в классе реализуется только некоторая часть методов интерфейса, такой класс (в обязательном порядке) объявляется abstract.

Методы интерфейса всегда являются открытыми. Они не могут быть статическими, поскольку статические методы всегда относятся к конкретному классу и никогда не бывают абстрактными, а интерфейс может включать только абстрактные методъ.

С другой стороны, поля интерфейса всегда объявляются static и final. Они представляют собой константы, используемые при вызове методов. Например, интерфейс, в контракте которого предусмотрено несколько уровней точности, может выглядеть следующим образом:

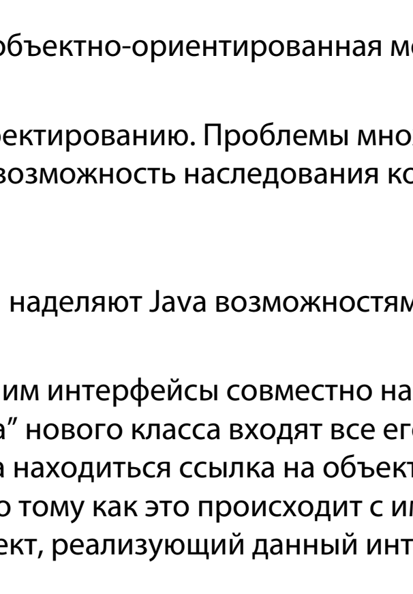
```
interface Verbose {
     int SILENT  = 0;
     int TERSE   = 1;
     int NORMAL  = 2;
     int VERBOSE = 3;

     void setVerbosity(int level);
     int getVerbosity();
}
```

Константы SILENT, TERSE, NORMAL и VERBOSE передаются методу set Verbosity; таким образом можно присвоить имена постоянным величинам, имеющим конкретное значение. Они должны быть константами, а все поля интерфейса неявно объявляются static и final.

4.2. Одиночное и множественное наследование

В языке Java новый класс может расширять всего один суперкласс — такая модель носит название *одиночного наследования*. Расширение класса означает, что новый класс наследует от своего суперкласса не только контракт, но и реализацию. В некоторых объектно-ориентированных языках используется множественное наследование, при котором новый класс может иметь два и более суперклассов.

Множественное наследование оказывается полезным в тех случаях, когда требуется наделить класс новыми возможностями и при этом сохранить большую часть (или все) старых свойств. Однако при наличии нескольких суперклассов возникают проблемы, связанные с двойственным наследованием. Например, рассмотрим следующую иерархию типов:



Обычно такая ситуация называется “ромбовидным наследованием”, и в ней нет ничего плохого — подобная структура встречается довольно часто. Проблема заключается в наследовании реализации. Если класс W содержит открытое поле goggin и у вас имеется ссылка на объект типа Z с именем zref, то чему будет соответствовать ссылка zref.goggin? Будет ли она представлять собой копию goggin из класса X, или из класса Y, или же X и Y будут использовать одну копию goggin, поскольку в действительности W входит в Z всего один раз, хотя Z одновременно является и X, и Y?

Чтобы избежать подобных проблем, в Java используется объектно-ориентированная модель с одиночным наследованием.

Одиночное наследование способствует правильному проектированию. Проблемы множественного наследования возникают из расширения классов при их реализации. Поэтому Java предоставляет возможность наследования контракта без связанной с ним реализации. Для этого вместо типа class используется тип interface.

Таким образом, интерфейсы входят в иерархию классов и наделяют Java возможностями множественного наследования.

Классы, расширяемые данным классом, и реализованные им интерфейсы совместно называются его *супертипами*; с точки зрения супертипов, новый класс является *подтипом*. В понятие “полного типа” нового класса входят все его супертипы, поэтому ссылка на объект класса может использоваться полиморфно — то есть всюду, где должна находиться ссылка на объект любого из супертипов (класса или интерфейса). Определения интерфейсов создают имена типов, подобно тому как это происходит с именами классов; вы можете использовать имя интерфейса в качестве имени переменной и присвоить ей любой объект, реализующий данный интерфейс.

4.3. Расширение интерфейсов

Интерфейсы также могут расширяться с помощью ключевого слова extended. В отличие от классов, допускается расширение интерфейсом сразу нескольких других интерфейсов:

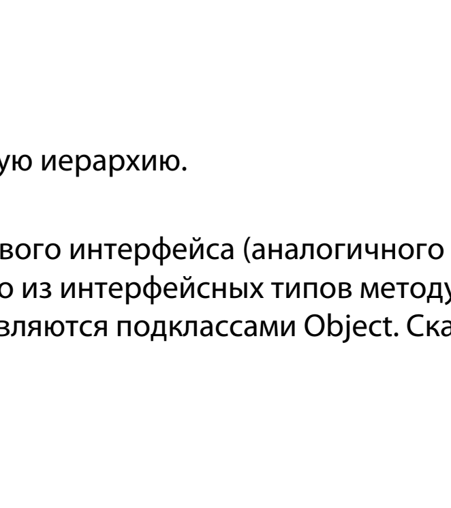
```
interface Shimmer extends FloorWax, DessertTopping {
     double amazingPrice();
}
```

Тип Shimmer расширяет FloorWax и DessertTopping; это значит, что все методы и константы, определенные в FloorWax и DessertTopping, являются составной частью его контракта, и к ним еще добавляется метод amazingPrice.

Если вы хотите, чтобы ваш класс реализовывал интерфейс и при этом расширял другой класс, то вам необходимо применить множественное наследование. Другими словами, у вас появляется класс, объекты которого могут использоваться там, где допускаются типы суперкласса и суперинтерфейса. Давайте рассмотрим следующее объявление:

```
interface W { }
interface X extends W { }
class Y implements W { }
class Z extends Y implements X { }
```

Ситуация отчасти напоминает знакомое нам “ромбовидное наследование”, но на этот раз нет никаких сомнений по поводу того, какие поля, X или Y, должны использоваться; у X нет никаких полей, поскольку это интерфейс, так что остается только Y. Диаграмма наследования будет выглядеть следующим образом:



W, X и Y могли бы быть интерфейсами, а Z — классом. Вот как бы это выглядело:

```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```

Z оказывается единственным классом, входящим в данную иерархию.

У интерфейсов, в отличие от классов, нет единого корневого интерфейса (аналогичного классу Object), лежащего в основе всей иерархии. Несмотря на это, вы можете передать выражение любого из интерфейсных типов методу, получающему параметр типа Object, потому что объект *должен* принадлежать к какому-то классу, а все классы являются подклассами Object. Скажем, для приведенного выше примера допускается следующее присваивание переменной obj:

```
protected void twiddle(W wRef) {
     Object obj = wRef;
     // ...
}
```

4.3.1. Конфликты имен

Два последних примера наглядно демонстрируют, что класс или интерфейс может быть подтипом более чем одного интерфейса. Возникает вопрос: что произойдет, если в родительских интерфейсах присутствуют методы с одинаковыми именами? Если интерфейсы X и Y содержат одноименные методы с разным количеством или типом параметров, то ответ прост: интерфейс Z будет содержать два перегруженных метода с одинаковыми именами, но разными сигнатурами. Если же сигнатуры в точности совпадают, то ответ также прост: Z может содержать лишь один метод с данной сигнатурой. Если методы отличаются лишь типом возвращаемого значения, вы не можете реализовать оба интерфейса.

Если два метода отличаются только типом возбуждаемых исключений, метод вашего класса обязан соответствовать обоим объявлениям с одинаковыми сигнатурами (количеством и типом параметров), но может возбуждать свои исключения. Однако методы в пределах класса не должны отличаться только составом возбуждаемых исключений; следовательно, должна присутствовать всего одна реализация, удовлетворяющая обоим связкам throws. Поясним сказанное на примере:

```
interface X {
     void setup() throws SomeExeption;
}

interface Y {
     void setup();
}

class Z implements X, Y {
     public void setup() {
          // ...
     }
}
```

В этом случае класс Z может содержать единую реализацию, которая соответствует X.setup и Y.setup. Метод может возбуждать меньше исключений, чем объявлено в его суперклассе, поэтому при объявлении Z.setup необязательно указывать, что в методе возбуждается исключение типа Some Exception. X.setup только разрешает использовать данное исключение. Разумеется, все это имеет смысл лишь в том случае, если одна реализация может удовлетворить контрактам обоих методов, — если два метода *подразумевают* нечто разное, то вам, по всей видимости, не удастся написать единую реализацию для них.

Если списки исключений расходятся настолько, что вам не удается объявить методы так, чтобы они удовлетворяли сигнатурам обоих интерфейсов, то вы не сможете ни расширить эти интерфейсы, ни построить реализующий их класс.

С *константами* интерфейсов дело обстоит проще. Если в двух интерфейсах имеются константы с одинаковыми именами, то вы всегда сможете объединить их в дереве наследования, если воспользуетесь уточненными (qualified) именами констант. Пусть интерфейсы PokerDeck и TarotDeck включают константы DECK_SIZE с различными значениями, а интерфейс или класс MultiDeck может реализовать оба этих интерфейса. Однако внутри Multi Deck и его подтипов вы должны пользоваться уточненными именами Poker Deck.DECK_SIZE и TarotDeck.DECK_SIZE, поскольку простое DECK_SIZE было бы двусмысленным.

4.4. Реализация интерфейсов

Интерфейс описывает контракт в абстрактной форме, однако он представляет интерес лишь после того, как будет реализован в некотором классе.

Некоторые интерфейсы являются чисто абстрактными — у них нет никакого полезного универсального воплощения, и они должны заново реализовываться для каждого нового класса. Тем не менее большая часть интерфейсов может иметь несколько полезных реализаций. В случае нашего интерфейса Attributed можно придумать несколько возможных реализаций, в которых используются различные стратегии для хранения набора атрибутов.

Одна стратегия может быть простой и быстродействующей (если набор содержит малое количество атрибутов); другую можно оптимизировать для работы с наборами редко изменяемых атрибутов; наконец, третья может предназначаться для часто меняющихся атрибутов. Если бы существовал пакет с возможными реализациями интерфейса Attributed, то класс, реализующий этот интерфейс, мог бы воспользоваться одной из них или же предоставить свой собственный вариант.

В качестве примера рассмотрим простую реализацию Attributed, в которой используется вспомогательный класс java.util.Hashtable. Позднее это будет использовано, чтобы реализовать интерфейс Attributed для конкретного набора объектов, наделяемых атрибутами. Прежде всего, класс AttributedImpl выглядит следующим образом:

```
import java.util.*;

class AttributedImpl implements Attributed
{
     protected Hashtable attrTable = new Hashtable();

     public void add(Attr newAttr) {
          attrTable.put(newAttr.nemeOf(), newAttr);
     }

     public Attr find(String name) {
          return (Attr)attrTable.get(name);
     }

     public Attr remove(String name) {
          return (Attr)attrTable.remove(name);
     }

     public Enumeration attrs() {
          return attrTable.elements();
     }
}
```

В реализации методов AttributedImpl используется класс Hashtable.

При инициализации attrTable создается объект Hashtable, в котором хранятся атрибуты. Большая часть работы выполняется именно классом Hashtable. Класс HashTable использует метод hashCode данного объекта для хеширования. Нам не приходится писать свой метод хеширования, поскольку String уже содержит подходящую реализацию hashCode.

При добавлении нового атрибута объект Attr сохраняется в хеш-таблице, причем имя атрибута используется в качестве ключа хеширования; затем по имени атрибута можно осуществлять поиск и удаление атрибутов из хеш-таблицы.

Метод attrs возвращает значение Enumeration, в котором приведены все атрибуты, входящие в набор. Enumeration является абстрактным классом, определенным в java.util и используемым классами-коллекциями типа Hash table для возвращения списков (см. раздел “Интерфейс Enumeration”). Мы также воспользуемся этим типом, поскольку он предоставляет стандартное средство для возвращения списков в Java. Фактически интерфейс Attributed определяет тип-коллекцию, поэтому применим обычный в таких случаях механизм возврата содержимого коллекции, а именно класс Enumeration. Использование Enumeration имеет ряд преимуществ: стандартные классы-коллекции вроде Hashtable, в которых применяется Enumeration, позволяют упростить реализацию Attributed.

4.5. Использование реализации интерфейса

Чтобы использовать класс (скажем, AttributedImpl), реализующий некоторый интерфейс, вы можете просто расширить класс. В тех случаях, когда такой подход возможен, он оказывается самым простым, поскольку при нем наследуются все методы вместе с их реализацией. Однако, если вам приходится поддерживать сразу несколько интерфейсов или расширять какой-то другой класс, не исключено, что придется поступить иначе. Чаще всего программист создает объект реализующего класса и *перенаправляет* в него все вызовы методов интерфейса, возвращая нужные значения.

Вот как выглядит реализация интерфейса Attributed, в которой объект AttributedImpl используется для наделения атрибутами нашего класса небесных тел:

```
import java.util.Enumeration;

class AttributedBody extends Body
     implements Attributed
{
     AttributedImpl attrImpl = new AtrributedImpl();

     AttributedBody() {
          super();
     }

     AttributedBody(String name, Body orbits) {
          super(name, orbits);
     }

     // Перенаправить все вызовы Attributed в объект attrImpl

     public void add(Attr newAttr)
          { attrImpl.add(newAttr); }
     public Attr find(String name)
          { return attrImpl.find(name); }
     public Attr remove(String name)
          { return attrImpl.remove(name); }
     public Enumeration attrs()
          { return attrImpl.attrs(); }
}
```

Объявление, в соответствии с которым AttributedBody расширяет класс Body и реализует интерфейс Attributed, определяет контракт Attributed Body. Реализация всех методов Body наследуется от самого класса Body. Реализация каждого из методов Attributed заключается в перенаправлении вызова в эквивалентный метод объекта AttributedImpl и возврате полученного от него значения (если оно имеется). Отсюда следует, что вам придется включить в класс поле типа AttributedImpl, используемое при перенаправлении вызовов, и инициализировать его ссылкой на объект AttributedImpl.

Перенаправление работает без особых хитростей и требует существенно меньших усилий, чем реализация Attributed “с нуля”. Кроме того, если в будущем появится более удачная реализация Attributed, вы сможете быстро перейти на нее.

4.6. Для чего применяются интерфейсы

Между интерфейсами и абстрактными классами существует два важных отличия:

- Интерфейсы предоставляют некоторую разновидность множественного наследования, поскольку класс может реализовать несколько интерфейсов. С другой стороны, класс расширяет всего один класс, а не два или более, даже если все они состоят только из абстрактных методов.
- Абстрактный класс может содержать частичную реализацию, защищенные компоненты, статические методы и т. д., тогда как интерфейс ограничивается открытыми методами, для которых не задается реализация, и константами.

Эти отличия обычно определяют выбор средства, которое должно применяться для конкретного случая. Если вам необходимо воспользоваться множественным наследованием, применяются интерфейсы. Однако при работе с абстрактным классом вместо перенаправления методов можно частично или полностью задать их реализацию, чтобы облегчить наследование. Перенаправление — довольно скучная процедура, к тому же чреватая ошибками, так что вам стоит лишний раз подумать, прежде чем отказываться от абстрактных методов.

Тем не менее любой сколько-нибудь важный класс (абстрактный или нет), предназначенный для расширения, должен представлять собой реализацию интерфейса. Хотя для этого потребуется немного дополнительных усилий, перед вами открывается целый спектр новых возможностей, недоступных в других случаях. Например, если бы мы просто создали класс Attributed вместо интерфейса Attributed, реализованного в специальном классе AttributedImpl, то тогда на его основе стало бы невозможно построить другие полезные классы типа AttributedBody — ведь расширять можно всего один класс. Поскольку Attributed является интерфейсом, у программистов появляется выбор: они могут либо непосредственно расширить AttributedImpl и избежать перенаправления, либо, если расширение невозможно, по крайней мере воспользоваться перенаправлением для реализации интерфейса. Если общая реализация окажется неверной, они напишут свою собственную. Вы даже можете предоставить несколько реализаций интерфейса для разных категорий пользователей. Независимо от того, какую стратегию реализации выберет программист, создаваемые объекты будут Attributed.

Упражнение 4.1

Перепишите свое решение упражнения 3.7 с использованием интерфейса, если вы не сделали этого ранее.

Упражнение 4.2

Перепишите свое решение упражнения 3.12 с использованием интерфейса, если вы не сделали этого ранее.

Упражнение 4.3

Должен ли класс LinkedList из предыдущих упражнений представлять собой интерфейс? Прежде чем отвечать на этот вопрос, перепишите его с использованием реализующего класса.

Упражнение 4.4

Спроектируйте иерархию классов-коллекций с применением одних интерфейсов.

Упражнение 4.5

Подумайте над тем, как лучше представить следующие типы (в виде интерфейсов, абстрактных или обычных классов): 1) TreeNode — для представления узлов N-арного дерева; 2) TreeWalker — для перебора узлов дерева в порядке, определяемом пользователем (например, перебор в глубину или в ширину); 3) Drawable — для представления объектов, которые могут быть нарисованы в графической системе; 4) Application — для программ, которые могут запускаться с графической рабочей поверхности (desktop).

Упражнение 4.6

Как надо изменить условия в упражнении 4.5, чтобы ваши ответы тоже изменились?

Содержание | Далее

Глава 2
КЛАССЫ И ОБЪЕКТЫ

Начнем с самого начала, хотя порядок может быть и другим.
Доктор Who, Meglos

Фундаментальной единицей программирования в Java является *класс*. В состав классов входят методы — фрагменты выполняемого кода, в которых происходят все вычисления. Классы также определяют структуру объектов и обеспечивают механизмы для их создания на основе определения класса. Вы *можете* ограничиваться в своих программах одними лишь примитивными типами (целыми, вещественными и т. д.), но практически любая нетривиальная программа на Java создает объекты и работает с ними.

В объектно-ориентированном программировании между тем, *что* делается, и тем, *как* это делается, существуют четкие различия. “Что” описывается в виде набора методов (а иногда и общедоступных данных) и связанной с ними семантики. Такое сочетание (методы, данные и семантика) часто называется *контрактом* между разработчиком класса и программистом, использующим этот класс, так как оно определяет, что происходит при вызове конкретных методов объекта.

Обычно считается, что объявленные в классе методы составляют все содержание его контракта. Кроме того, в понятие контракта входит и *семантика* этих операций, даже несмотря на то, что она может быть описана лишь в документации. Два метода могут иметь одинаковые имена и сигнатуры, но они не будут эквивалентными, если обладают различной семантикой. Например, нельзя предположить, что каждый метод с именем print предназначен для вывода копии объекта. Кто-нибудь может создать метод print и вложить в его название иную семантику — скажем, термин может быть сокращением от выражений “process interval” или “prioritize nonterminals”. Контракт (то есть совокупность сигнатуры и семантики) определяет сущность метода.

На вопрос “как” отвечает класс, на основе которого создавался данный объект. Класс определяет реализацию методов, поддерживаемых объектом. Каждый объект представляет собой *экземпляр* класса. При вызове метода объекта выполняемый код ищется в классе. Объект может использовать другие объекты, однако мы начнем изучение с простых классов, в которых все методы реализуются непосредственно.

2.1. Простой класс

Основными компонентами класса являются поля (данные) и методы (код для работы с ними). Приведем простой класс Body, предназначенный для хранения сведений о небесных телах:

```
class Body {
        public long idNum;
        public String nameFor;
        public Body orbits;

        public static long nextID = 0;
}
```

Прежде всего объявляется имя класса. Объявление класса в языке Java создает *имя типа*, так что в дальнейшем ссылки на объекты этого типа могут объявляться следующим образом:

Body mercury;

Подобное объявление указывает на то, что mercury является ссылкой на объект класса Body. Оно *не* создает сам объект, а лишь объявляет *ссылку* на него. Первоначально ссылка имеет значение null, и объект, на который ссылается mercury, не существует до тех пор, пока вы не создадите его явным образом; в этом отношении Java отличается от других языков программирования, в которых объекты создаются при объявлении переменных.

Первая версия класса Body спроектирована неудачно. Это было сделано намеренно: по мере усовершенствования класса в данной главе мы сможем сосредоточить свое внимание на некоторых возможностях языка.

Упражнение 2.1

Напишите простой класс Vehicle (транспортное средство) для хранения информации об автомашине, в котором предусмотрены (как минимум) поля для задания текущей скорости, текущего направления в градусах и имени владельца.

Упражнение 2.2

Напишите класс LinkedList (связный список), в котором имеется поле типа Object и ссылка на следующий по списку элемент LinkedList.

2.2. Поля

Переменные класса называются *полями*; примерами могут служить поля nameFor и orbits, входящие в класс Body. Каждый объект Body обладает отдельным экземпляром своих полей: значение типа long отличает данное небесное тело от остальных, переменная типа String содержит его имя, а ссылка на другой объект Body определяет небесное тело, вокруг которого оно обращается.

Наличие у каждого объекта отдельного экземпляра полей означает, что его состояние уникально. Изменение поля orbits в одном объекте класса Body никак не отражается на значениях соответствующего поля в других объектах этого класса.

Тем не менее иногда бывает необходимо, чтобы один экземпляр поля совместно использовался всеми объектами класса. Такие поля объявляются с ключевым словом static и называются *статическими полями*, или *переменными класса*. При объявлении в классе поля static все объекты этого класса разделяют одну и ту же копию статического поля.

В нашем случае класс Body содержит одно статическое поле — nextID, в котором хранится следующий используемый идентификатор небесного тела. При инициализации класса, которая происходит после его загрузки и компоновки, в поле nextID заносится нулевое значение. Ниже мы убедимся, что текущее значение nextID присваивается идентификатору каждого вновь создаваемого объекта класса Body.

Когда в этой книге используются термины “поле” или “метод”, обычно имеются в виду нестатические поля и методы. В тех случаях, когда контекст не дает однозначного толкования, мы будем пользоваться термином “нестатическое поле” или “нестатический метод”.

Упражнение 2.3

Включите в класс Vehicle (транспортное средство) статическое поле для хранения идентификатора машины, а в класс Car (автомобиль) — нестатическое поле, содержащее номер машины.

2.3. Управление доступом и наследование

Код класса всегда может обращаться ко всем полям и методам данного класса. Для управления доступом к ним из других классов, а также для управления наследованием их в подклассах члены классов могут объявляться с одним из четырех атрибутов доступа:

- Открытый (Public): к членам класса всегда можно обращаться из любого места, в котором доступен сам класс; такие члены наследуются в подклассах.
- Закрытый (Private): доступ к членам класса осуществляется только из самого класса.
- Защищенный (Protected): к данным членам разрешается доступ из подклассов и из функций, входящих в тот же пакет. Такие члены наследуются подклассами. Расширение объектов (наследование) подробно рассмотрено в главе 3.
- Пакетный: доступ к членам, объявленным без указания атрибута доступа, осуществляется только из того же пакета. Такие члены наследуются подклассами пакета. Пакеты рассматриваются в главе 10.

Поля класса Body были объявлены с атрибутом public, потому что для выполнения поставленной задачи программистам необходим доступ к этим полям. На примере более поздних версий класса Body мы убедимся, что подобное решение обычно является неудачным.

2.4. Создание объектов

Для первой версии класса Body создание и инициализация объектов, представляющих небесные тела, происходит следующим образом:

```
Body sun = new Body();
sun.idNum = Body.nextID++;
sun.nameFor = “Sol”;
sun.orbits = null; // Солнце является центром Солнечной
                                  // системы

Body earth = new Body();
earth.idNum = Body.nextID++;
earth.nameFor = “Earth”;
earth.orbits = sun;
```

Сначала мы объявили две ссылки (sun и earth) на объекты типа Body. Как упоминалось выше, объекты при этом *не* создаются — лишь объявляются переменные, которые ссылаются на объекты. Первоначальное значение ссылок равно null, а соответствующие им объекты должны явным образом создаваться в программе.

Объект sun создается посредством оператора new. Конструкция new считается самым распространенным способом построения объектов (позднее мы рассмотрим и другие возможности). При создании объекта оператором new следует указать тип конструируемого объекта и необходимые параметры. Runtime-система выделяет область памяти, достаточную для хранения полей объекта, и инициализирует ее в соответствии с рассматриваемыми ниже правилами. После завершения инициализации runtime-система возвращает ссылку на созданный объект.

Если системе не удается выделить память, достаточную для создания объекта, она может запустить сборщик мусора, который освобождает неиспользуемую память. Если же и после этого памяти все равно не хватает, оператор new возбуждает исключение OutOfMemoryError.

Создав новый объект Body, мы инициализируем его переменные. Каждый объект класса Body должен иметь уникальный идентификатор, который он получает из статического поля nextID. Программа наращивает значение nextID, чтобы идентификатор следующего объекта Body также был уникальным.

В нашем примере строится модель Солнечной системы. В ее центре находится Солнце, поэтому полю orbits объекта sun присваивается значение null — у Солнца нет объекта, вокруг которого оно бы вращалось. При создании и инициализации объекта earth (Земля) мы присвоили полю orbits значение sun. Для Луны, вращающейся вокруг Земли, поле orbits получило бы значение earth. Если бы мы строили модель Галактики, то Солнце бы также вращалось вокруг “черной дыры”, находящейся где-то в середине Млечного Пути.

Упражнение 2.4

Напишите для класса Vehicle метод main, который создает несколько объектов-автомашин и выводит значения их полей.

Упражнение 2.5

Напишите для класса LinkedList метод main, который создает несколько объектов типа Vehicle и заносит их в список.

2.5. Конструкторы

Каждый вновь созданный объект обладает некоторым исходным состоянием. Значения полей могут инициализироваться при их объявлении — иногда этого бывает достаточно. /Инициализация данных подробно рассматривается в разделе "Инициализация", однако в сущности за этим термином скрывается обычное присвоение начального значения. Если в программе полю не присваивается никакого значения, оно получит значение ноль, \u0000, false или null, в зависимости от типа./ Однако довольно часто для определения исходного состояния простой инициализации данных оказывается недостаточно; например, могут понадобиться какие-либо исходные данные, или же выполняемые операции не могут быть представлены в виде простого присваивания.

Для тех случаев, когда простой инициализации недостаточно, используются *конструкторы*. Имя конструктора совпадает с именем класса, который он инициализирует. Конструкторы, подобно методам, могут получать один или несколько параметров, однако они не являются методами и не могут возвращать никакого значения. Параметры конструктора (если они есть) указываются в скобках за именем типа при создании объекта оператором new. Конструктор вызывается после того, как переменным в экземпляре вновь создаваемого объекта будут присвоены начальные значения по умолчанию и будет выполнена непосредственная инициализация.

В усовершенствованной версии класса Body исходное состояние объекта частично устанавливается посредством инициализации, а частично — в конструкторе:

```
class Body {
        public long idNum;
        public String name = “”;
        public Body orbits = null;

        private static long nextID = 0;


        Body() {
                idNum = nextID++;
        }
}
```

Конструктор класса Body вызывается без аргументов, однако он выполняет важную функцию, а именно устанавливает во вновь создаваемом объекте правильное значение поля idNum. Простейшая ошибка, возможная при работе со старой версией класса, — скажем, вы забыли присвоить значение полю idNum или не наращивали nextID после его использования — приводила к тому, что в программе возникали разные объекты класса Body с одинаковыми значениями поля idNum. В результате возникали проблемы в той части кода, которая была основана на положении контракта, гласящем: “Все значения idNum должны быть разными”.

Возлагая ответственность за генерацию значений idNum на сам класс Body, мы тем самым предотвращаем ошибки подобного рода. Конструктор Body становится единственным местом в программе, где idNum присваивается значение. Следующим шагом является объявление поля nextID с ключевым словом private, чтобы доступ к нему осуществлялся только из класса. Тем самым мы устраняем еще один возможный источник ошибок для программистов, работающих с классом Body.

Кроме того, такое решение позволит нам свободно изменять способ присвоения значений полю idNums в объектах Body. Например, будущая реализация нашего класса может просматривать базу данных известных астрономических объектов и присваивать новое значение idNum лишь в том случае, если ранее данному небесному телу не был присвоен другой идентификатор. Такое изменение никак не скажется на существующем коде программы, поскольку он никак не участвует в присвоении значения idNum.

При инициализации полям name и orbits присваиваются некоторые разумные начальные значения. Следовательно, после приведенного ниже вызова конструктора все поля нового объекта Body будут инициализированы. После этого вы можете изменить состояние объекта, присвоив его полям нужные значения:

```
Body sun = new Body();  // значение idNum равно 0
sun.name = “Sol”;

Body earth = new Body();  // значение idNum равно 1
earth.name = “Earth”;
earth.orbits = sun;
```

Конструктор Body вызывается при создании нового объекта оператором new, но после того, как полям name и orbits будут присвоены начальные значения. Инициализация поля orbits значением null означает, что sun.orbits в нашей программе не присваивается никакого значения.

Если создание объекта для небесного тела с двумя параметрами — названием и центром обращения — будет происходить довольно часто, то можно предусмотреть отдельный конструктор, в который оба этих значения передаются в качестве параметров:

```
Body(String bodyName, Body orbitsAround) {
        this();
        name = bodyName;
        orbits = orbitsAround;
}
```

Как видите, из одного конструктора можно вызывать другой конструктор этого же класса — для этого первым выполняемым оператором должен быть вызов this(). Это называется “явным вызовом конструктора”. Если для вызова конструктора необходимы параметры, они могут передаваться. В нашем случае для присвоения значения полю idNum используется конструктор, вызываемый без аргументов. Теперь создание объектов происходит значительно проще:

```
Body sun = new Body(“Sol”, null);
Body earth = new Body(“Earth”, sun);
```

При желании можно задать отдельный конструктор с одним аргументом для тех случаев, когда для создаваемого объекта Body не существует центра вращения. Вызов такого конструктора равносилен применению конструктора с двумя аргументами, при котором второй из них равен null.

Для некоторых классов необходимо, чтобы создатель объекта предоставлял некоторые данные. Например, приложение может требовать, чтобы для всех объектов Body было указано их название. Чтобы убедиться, что всем создаваемым объектам Body передается название, необходимо включить соответствующий параметр во все конструкторы класса Body.

Приведем несколько общепринятых соображений в пользу создания специализированных конструкторов:

- Некоторые классы не обладают разумным начальным состоянием, если не передать им параметры.
- При конструировании объектов некоторых видов передача исходного состояния оказывается самым удобным и разумным выходом (примером может служить конструктор Body с двумя аргументами).
- Конструирование объектов потенциально сопряжено с большими накладными расходами, так что желательно при создании объекта сразу устанавливать правильное исходное состояние. Например, если каждый объект класса содержит таблицу, то конструктор, получающий исходный размер таблицы в качестве параметра, позволит с самого начала создать объект с таблицей нужного размера.
- Конструктор, атрибут доступа которого отличается от public, ограничивает возможности создания объектов данного класса. Например, вы можете запретить программистам, работающим с вашим пакетом, расширять тот или иной класс, если сделаете все конструкторы доступными лишь из пакета. Кроме того, можно пометить ключевым словом protected те конструкторы, которые предназначены для использования исключительно в подклассах.

Конструкторы, не получающие при вызове никаких аргументов, встречаются настолько часто, что для них даже появился специальный термин: “безаргументные” (no-arg) конструкторы.

Если вы не предоставите для класса никаких конструкторов, язык создает безаргументный конструктор по умолчанию, который не делает ничего. Этот конструктор создается автоматически лишь в тех случаях, когда нет никаких других конструкторов, — существуют классы, для которых безаргументный конструктор будет работать неверно (например, класс Attr, с которым мы познакомимся в следующей главе).

Если безаргументный конструктор должен существовать наряду с одним или несколькими конструкторами, использующими аргументы, можно явно определить его. Автоматически создаваемый безаргументный конструктор класса, не имеющего суперкласса, эквивалентен следующему (как мы увидим на примере расширенного класса в главе 3):

```
class SimpleClass {
        /** Эквивалент конструктора по умолчанию */
        public SimpleClass() {
        }
}
```

Конструктор по умолчанию имеет атрибут public, если такой же атрибут имеет класс, и не имеет его в противном случае.

Упражнение 2.6

Включите в класс Vehicle два конструктора. Первый из них — безаргументный, а другой должен получать в качестве аргумента имя владельца машины. Модифицируйте метод main так, чтобы он выдавал те же результаты, что и раньше.

Упражнение 2.7

Какие конструкторы вы бы сочли нужным добавить в класс LinkedList?

2.6. Методы

Методы класса обычно содержат код, который анализирует состояние объекта и изменяет его. Некоторые классы имеют поля public, к которым программисты могут обращаться напрямую, но в большинстве случаев такой подход оказывается не слишком удачным (см. “Проектирование расширяемого класса”). Многие классы обладают функциями, которые невозможно свести к чтению или изменению некоторой величины — для них необходимы вычисления.

Вызов метода представляет собой операцию, выполняемую с объектом посредством ссылки на него с использованием оператора:

reference.method(parameters)

Каждый метод вызывается с определенным количеством параметров. Java не поддерживает методов, у которых допускается переменное число параметров. Каждый параметр имеет строго определенный тип — примитивный или ссылочный. Кроме того, методы обладают типом возвращаемого значения, который указывается перед их именем. Например, приведем метод класса Body, который создает строку типа String с описанием конкретного объекта Body:

```
public String toString() {
        String desc = idNum + “ (” + name + “)”;
        if (orbits != null)
                desc += “ orbits ” + orbits.toString();
        return desc;
}
```

В этом методе производится конкатенация объектов String с помощью операторов + и +=. Сначала образуется строка, содержащая идентификатор и название объекта. Если данное небесное тело обращается вокруг другого, то к ней присоединяется строка с описанием центра вращения, для чего вызывается метод toString соответствующего объекта. Последовательность рекурсивных вызовов продолжает строить цепочку тел, обращающихся вокруг друг друга, пока не будет найдено тело, не имеющее центра вращения.

Метод toString не совсем обычен. Если у объекта имеется метод с именем toString, который вызывается без параметров и возвращает значение типа String, то он используется для приведения объекта к типу String, если он участвует в конкатенации строк, выполняемой оператором +. В следующих выражениях:

```
System.out.println(“Body ” + sun);
System.out.println(“Body ” + earth);
```

происходит косвенный вызов методов toString для объектов sun и earth, приводящий к следующим результатам:

```
Body 0 (Sol)
Body 1 (Earth) orbits 0 (Sol)
```

Существует несколько способов, которыми удается добиться возвращения методом нескольких значений: можно возвращать ссылку на объект, в котором эти значения хранятся в виде полей; принимать в качестве параметров ссылки на объекты, в которых должны сохраняться результаты; наконец, можно возвращать массив с результатами.

К примеру, предположим, что вам нужно написать метод для возвращения перечня финансовых операций, которые определенное лицо может выполнять с банковским счетом. Таких операций может быть несколько (зачисление и снятие средств со счета и т. д.); следовательно, метод должен возвращать несколько значений. Вот как выглядит объект Permissions, в котором сохраняются логические значения, определяющие допустимость той или иной операции:

```
class Permissions {
        public boolean canDeposit,
                canWithdraw,
                canClose;
}
А вот как выглядит метод, заполняющий эти поля:
class Account {
        public Permissions permissionsFor(Person who) {
                Permissions perm = new Permissions();
                perm.canDeposit = canDeposit(who);
                perm.canWithdraw = canWithdraw(who);
                perm.canClose = canClose(who);
                return perm;
        }

        // ... определение метода canDeposit()
}
```

Если метод не возвращает никакого значения, то на месте возвращаемого типа ставится ключевое слово void. В противном случае каждый возможный путь выполнения его операторов должен возвращать значение, которое может быть присвоено переменной объявленного типа. К примеру, метод permissions For не может возвращать значение типа String, поскольку невозможно присвоить объект типа String переменной типа Permissions. Однако вы можете объявить, что метод permissionsFor возвращает значение String, не изменяя при этом оператор return, поскольку ссылка на объект Permissions может быть присвоена переменной типа Object.

2.6.1. Значения параметров

Все параметры в Java передаются “по значению”. Другими словами, значения переменных-параметров метода являются копиями значений, указанных при его вызове. Если передать методу переменную некоторого типа, то параметр будет представлять собой копию этой переменной; ее изменение внутри метода никак не повлияет на значение переменной в коде за его пределами. Например:

```
class PassByValue {
        public static void main(String[] args) {
                double one = 1.0;

                System.out.println(“before: one = ” + one);
                halveIt(one);
                System.out.println(“after: one = ” + one);
        }

        public static void halveIt(double arg) {
                arg /= 2.0; //
                System.out.println(“halved: arg = ” + arg);
        }
}
```

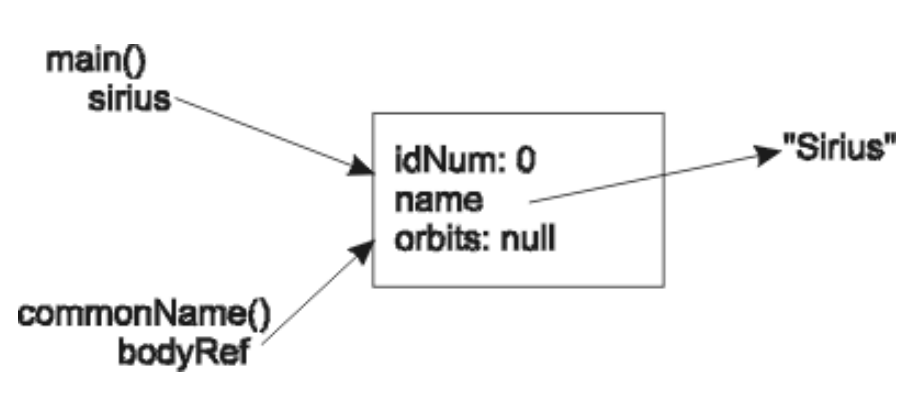
Приведенные ниже результаты показывают, что деление на два переменной arg в методе halveIt не меняет значения переменной one в методе main:

```
before: one = 1
halved: arg = 0.5
after:  one = 1
```

Однако, если параметр метода представляет собой ссылку на объект, то “по значению” передается *ссылка*, а не сам объект! Следовательно, вы можете изменять в методе тот объект, на который она ссылается, — значение ссылки остается прежним. Можно изменять любые поля объекта или вызывать методы, влияющие на его состояние, — произойдет изменение объекта во всех фрагментах программы, где имеется ссылка на него. Приведенный ниже пример наглядно показывает, чем данный случай отличается от предыдущего:

```
class PassRef {
        public static void main(String[] args) {
                Body sirius = new Body(“Sirius”, null);

                System.out.println(“before: ” + sirius);
                commonName(sirius);
                System.out.println(“after: ” + sirius);
        }

        public static void commonName(Body bodyRef) {
                bodyRef.name = “Dog Star”;
                bodyRef = null;
        }
}
Результат будет следующим:
before: 0 (Sirius)
after: 0 (Dog Star)
```

Обратите внимание на то, что название объекта изменилось, тогда как ссылка bodyRef все равно указывает на объект Body (хотя метод commonName присваивал параметру bodyRef значение null).



Приведенная выше диаграмма показывает состояние ссылок непосредственно после вызова commonName в main. Обе ссылки — sirius (в main) и bodyRef (в commonName) — указывают на один и тот же объект. Когда commonName изменяет значение поля bodyRef.name, то название изменяется в объекте, совместно используемом обоими ссылками. Однако при присвоении null ссылке bodyRef изменяется только ее значение, тогда как значение ссылки на объект sirius остается тем же самым; вспомним, что параметр bodyRef является передаваемой по значению копией sirius. Внутри метода commonName изменяется лишь значение переменной-параметра bodyRef, подобно тому как в методе halveIt изменялось лишь значение параметра-переменной arg. Если бы изменение bodyRef относилось и к значению sirius в main, то в строке, начинающейся с “after: ”, стояло бы “null”. Тем не менее переменные bodyRef в commonName и sirius в main указывают на один и тот же объект, поэтому изменения, вносимые в commonName, отражаются и в том объекте, на который ссылается sirius.

2.6.2. Применение методов для ограничения доступа

Пользоваться классом Body с несколькими конструкторами стало значительно удобнее, чем его старым вариантом, состоявшим из одних данных; кроме того, мы обеспечили правильное автоматическое присвоение значений idNum. И все же программист может все испортить, изменяя значение поля idNum после конструирования объекта, — ведь данное поле объявлено как public и открыто для любых действий. Необходимо, чтобы поле idNum содержало данные, доступные только для чтения. Подобные данные в объектах встречаются довольно часто, но в языке Java не существует ключевого слова, которое сделало бы поле за пределами класса доступным только для чтения.

Чтобы сделать поле доступным только для чтения, мы должны скрыть его. Для этого поле idNum объявляется с ключевым словом private, а в класс добавляется новый метод, с помощью которого код за пределами класса может получить значение этого поля:

```
class Body {
        private long idNum; // поле стало private

        public String name = “”;
        public Body orbits = null;
        private static long nextID = 0;

  Body() {
                idNum = nextID++;
        }
        public long id() {
                return idNum;
        }

        // ...
}
```

Начиная с этого момента программист, которому понадобилось узнать идентификатор небесного тела, должен вызвать метод id, возвращающий требуемое значение. У программиста не остается никакой возможности изменить идентификатор — в сущности, за пределами класса его можно рассматривать как величину, доступную только для чтения. Данное поле может быть изменено только внутренними методами класса Body.

Методы, регулирующие доступ к внутренним данным класса, иногда называются *методами доступа (accessor methods)*. С их помощью также можно (и, наверное, нужно) защитить поля name и orbits.

Даже если интересы приложения и не требуют полей, доступных только для чтения, объявление полей класса с ключевым словом private и создание методов для присвоения/получения их значений позволяет вам определить необходимые действия над объектом в будущем. Если программист может непосредственно обращаться к полям класса, вы неизбежно теряете контроль над тем, какие значения будут принимать эти поля и что происходит в программе при их изменении. По этим причинам в дальнейших примерах этой книги поля public встречаются очень редко.

Упражнение 2.8

Объявите поля класса Vehicle с ключевым словом private и опишите соответствующие методы доступа. Для каких полей следует предусмотреть методы, изменяющие их значения, а для каких — нет?

Упражнение 2.9

Объявите поля класса LinkedList с ключевым словом private и включите в класс соответствующие методы доступа. Для каких полей следует предусмотреть методы, изменяющие их значения, а для каких — нет?

Упражнение 2.10

Включите в класс Vehicle метод changeSpeed для задания текущей скорости машины в соответствии с передаваемым значением и метод stop для обнуления скорости.

Упражнение 2.11

Включите в класс LinkedList метод для подсчета количества элементов в списке.

2.7. Ссылка this

Мы уже видели (на стр.), как в начале работы конструктора происходит явный вызов другого конструктора класса. Специальная ссылка на объект this может также применяться внутри нестатических методов; она указывает на текущий объект, для которого был вызван данный метод.
this чаше всего используется для передачи ссылки на текущий объект в качестве параметра для других методов. Предположим, в методе необходимо пополнить список объектов, ожидающих обслуживания. Такой вызов может выглядеть следующим образом:

Service.add(this);

this неявно добавляется в начало каждой ссылки на поле или метод, если только программист не указал ссылку на другой объект. Например, присвоение значения полю str в следующем классе:

```
class Name {
        public String str;
        Name() {
                str = “”;
        }
}
равносильно следующему:
this.str = “”;
```

Обычно this используется только в случае необходимости, то есть когда имя поля, к которому вы обращаетесь, скрывается объявлением переменной или параметра. Например:

```
class Moose {
        String hairdresser;

        Moose(String hairdresser) {
                this.hairdresser = hairdresser;
        }
}
```

Поле hairdresser внутри конструктора скрывается присутствием одноименного параметра. Чтобы обратиться к полю hairdresser, а не к параметру, мы ставим перед именем ссылку this, указывая тем самым, что поле принадлежит к текущему объекту. Намеренное скрытие идентификаторов, осуществляемое подобным образом, может быть отнесено к хорошему стилю программирования лишь при идиоматическом использовании в конструкторах и методах доступа.

Помимо ссылки this, может также применяться ссылка super, с помощью которой осуществляется доступ к скрытым полям и вызов переопределенных методов суперкласса. Ключевое слово super подробно рассматривается в разделе “Переопределение методов и скрытие полей”.

2.8. Перегрузка методов

В языке Java каждый метод обладает определенной *сигнатурой*, которая представляет собой совокупность имени с количеством и типом параметров. Два метода могут иметь одинаковые имена, если их сигнатуры отличаются по количеству или типам параметров. Это называется *перегрузкой (overloading)*, поскольку простое имя метода “перегружается” несколькими значениями. Когда программист вызывает метод, компилятор по количеству и типу параметров ищет тот из существующих методов, сигнатура которого подходит лучше всех остальных. Приведем в качестве примера различные методы orbits Around нашего класса Body:

```
public Body orbitsAround() {
        return orbits;
}
public void orbitsAround(Body around) {
        orbits = around;
}
```

При подобном стиле программирования перегрузка служит для того, чтобы отличать выборку значения (параметры не передаются) от его задания (указывается аргумент, представляющий собой новое значение). Количество параметров в двух методах отличается, поэтому выбрать нужный метод будет несложно. Если orbitsAround вызывается без параметров, то используется метод, возвращающий текущее значение. Если orbitsAround вызывается с одним аргументом типа Body, то используется метод, задающий значение. Если же вызов не подходит ни под одну из этих сигнатур, то он является неверным, а программа не будет компилироваться.

Вопрос о том, как язык выбирает среди нескольких перегруженных методов тот, что нужен для данного вызова, подробно рассматривается в разделе “Доступ к членам” на стр. .

Упражнение 2.12

Включите в класс Vehicle два новых метода: один в качестве параметра получает количество градусов, на которое поворачивает машина, а другой — одну из констант Vehicle.TURN_LEFT или Vehicle.TURN_RIGHT.

2.9. Статические члены

Класс содержит члены двух видов: поля и методы. Для каждого из них задается атрибут, определяющий возможности наследования и доступа (private, protected, public или package). Кроме того, каждый из членов при желании можно объявить как static.

Для *статического* члена создается всего один экземпляр, общий для всего класса, вместо построения его копий в каждом объекте класса. В случае статических переменных (переменных класса), это ровно одна переменная, независимо от того, сколько объектов было создано на основе класса (даже если ни одного). Образцом может служить поле nextID класса Body в приведенном выше примере.

Инициализация статических полей класса происходит до того, как они используются или запускается любой из его методов. В следующем примере метод unset может быть уверен в том, что перед использованием переменной UNSET ей было присвоено значение Double.NaN:

```
class Value {
        public static double UNSET = double.NaN;

        private double V;
        public void unset() {
                V = UNSET;
        }
        // ...
}
```

2.9.1. Блоки статической инициализации

Класс также может содержать *блоки статической инициализации*, которые присваивают значения статическим полям или выполняют иную необходимую работу. Статический инициализатор оказывается наиболее полезным в тех случаях, когда простой инициализации в объявлении поля недостаточно. Например, создание статического массива часто должно выполняться одновременно с его инициализацией в операторах программы. Приведем пример инициализации небольшого массива с простыми числами:

```
class Primes {
        protected static int[] knownPrimes = new int[4];

        static {
                knownPrimes[0] = 2;
                for(int i = 1; i < knownPrimes.length; i++)
                        knownPrimes[i] = nextPrime();
        }

}
```

Статическая инициализация внутри класса выполняется в порядке слева направо и сверху вниз. Инициализатор, или статический блок, каждой из статических переменных выполняется перед следующим, начиная от первой строки исходного текста к последней. При этом можно гарантировать, что массив knownPrimes будет создан до выполнения статического блока в нашем примере.

Что произойдет, если статический инициализатор класса X вызывает метод класса Y, а статический инициализатор Y, в свою очередь, вызывает метод из класса X для задания *своих* статических величин? Подобные циклические инициализации не могут быть надежно выявлены в процессе компиляции, поскольку в момент компиляции X класс Y может еще не существовать. Если возникает подобная ситуация, то статические инициализаторы X выполняются лишь до вызова метода Y. Когда Y, в свою очередь, обратится к методу X, то последний будет выполняться без завершения статической инициализации. Все статические поля X, для которых инициализация не была выполнена, будут иметь значения по умолчанию (false, ‘\u0000’, ноль или null в зависимости от типа).

В инициализаторах статических полей не должны вызываться методы, которые, согласно их объявлениям, могут привести к возникновению проверяемых исключений. Дело в том, что инициализаторы выполняются при загрузке класса, и программа может быть еще не готова к обработке исключения.

Блок статической инициализации *может* вызывать методы, возбуждающие исключения, но лишь в том случае, если он готов сам перехватить их все.

2.9.2. Статические методы

Статические методы вызываются для целого класса, а не для каждого конкретного объекта, созданного на его основе. Подобные методы также называются *методами класса*. Статический метод может выполнять задачи, общие для всех объектов класса, — например, возвращать следующий серийный номер (в нашем примере с плеерами) или что-нибудь в этом роде.

Статический метод работает лишь со статическими переменными и статическими методами класса. Ссылка this в этом случае не может использоваться, поскольку не определен конкретный объект, для которого вызывается данный метод.

За пределами класса статические методы обычно вызываются через имя класса, а не через ссылку на конкретный объект:

```
prime = Primes.nextPrime();
knownCnt = Primes.knownPrimes.length;
```

Упражнение 2.13

Включите в класс Vehicle статический метод, который возвращает максимальное значение идентификатора, использованное на данный момент.

2.10. Сборка мусора и метод finalize

Java выполняет всю сборку программного мусора автоматически и избавляет вас от необходимости явного освобождения объектов.

Проще говоря, это означает, что память, занимаемая неиспользуемым объектом, может быть возвращена в систему. При этом никаких действий с вашей стороны не требуется — в сущности, вы ничего и не *сможете* сделать. Объект является “неиспользуемым”, когда на него отсутствуют ссылки в статических данных и в любой из переменных выполняемого в настоящий момент метода, когда не удается найти ссылку на него посредством отслеживания полей и элементов массивов статических данных и переменных методов, и так далее. Объекты создаются оператором new, но соответствующего ему оператора delete не существует. После завершения работы с объектом вы просто перестаете ссылаться на него (изменяете его ссылку так, чтобы она указывала на другой объект или null) или возвращаетесь из метода, чтобы его локальные переменные перестали существовать и не указывали на объект. Когда ссылок на объект не остается нигде, за исключением других неиспользуемых объектов, данный объект может быть уничтожен сборщиком мусора. Мы пользуемся выражением “может быть”, потому что память освобождается лишь в том случае, если ее недостаточно или если сборщик мусора захочет предотвратить ее нехватку.

Автоматическая сборка мусора означает, что вам никогда не придется беспокоиться о проблеме “зависших ссылок” (dangling references). В тех системах, где предусмотрен прямой контроль за удалением, допускается освобождение объектов, на которые ссылаются другие объекты. В таком случае ссылка становится “зависшей”, то есть она указывает на область памяти, которая в системе считается свободной. Эта “свободная” память может быть использована для создания нового объекта, и тогда “зависшая ссылка” будет указывать на нечто совершенно отличное от того, что предполагалось в объекте. В результате содержимое этой памяти может быть использовано совершенно непредсказуемым образом, и возникает полный хаос. Java решает проблему “зависших ссылок” за вас, поскольку объект, на который имеется ссылка, никогда не будет уничтожен сборщиком мусора.

Сборка мусора происходит без вашего вмешательства, однако она все же требует определенного внимания. Создание и уничтожение многочисленных объектов может помешать работе тех приложений, для которых время выполнения оказывается критичным. Программы следует проектировать так, чтобы количество создаваемых в них объектов было разумным.

Сборка мусора еще не является гарантией того, что для любого вновь создаваемого объекта всегда найдется память. Вы можете бесконечно создавать объекты, помещать их в список и делать это до тех пор, пока не кончится память, и при этом не будет ни единого неиспользуемого объекта, который можно было бы уничтожить. Например, подобная “утечка памяти” может быть обусловлена тем, что вы продолжаете поддерживать ссылки на ненужные объекты. Сборка мусора решает многие, но не все проблемы с выделением памяти.

2.10.1. Метод finalize

Обычно вы и не замечаете, как происходит уничтожение “осиротевших” объектов. Тем не менее класс может реализовать метод с именем finalize, который выполняется перед уничтожением объекта или при завершении работы виртуальной машины. Метод finalize дает возможность использовать удаление объекта для освобождения других, не связанных с Java ресурсов. Он объявляется следующим образом:

```
protected void finalize() throws Throwable {
        super.finalize();
        // ...
}
```

Роль метода finalize становится особенно существенной при работе с внешними по отношению к Java ресурсами, которые слишком важны, чтобы можно было дожидаться этапа сборки мусора. Например, открытые файлы (число которых обычно ограничено) не могут дожидаться завершающей фазы finalize — нет никакой гарантии, что объект, содержащий открытый файл, будет уничтожен сборщиком мусора до того, как израсходуются все ресурсы по открытию файлов.

Поэтому в объекты, распоряжающиеся внешними ресурсами, следует включать метод finalize, освобождающий ресурсы и предотвращающий их утечку. Кроме того, вам придется предоставить программистам возможность явного освобождения этих ресурсов. Например, в класс, который открывает файл для своих целей, следует включить метод close для закрытия файла, чтобы пользующиеся этим классом программисты могли явным образом распоряжаться количеством открытых файлов в системе.

Иногда может случиться так, что метод close не будет вызван, несмотря на то, что работа с объектом закончена. Возможно, вам уже приходилось сталкиваться с подобными случаями. Вы можете частично предотвратить “утечку файлов”, включив в класс метод finalize, внутри которого вызывается close, — таким образом можно быть уверенным, что вне зависимости от качества работы других программистов ваш класс никогда не будет поглощать открытые файлы. Вот как это может выглядеть на практике:

```
public class ProcessFile {
        private Stream File;

        public ProcessFile(String path) {
                File = new Stream(path);
        }
// ...

        public void close() {
                if (File != null) {
                        File.close();
                        File = null;
                }
        }

        protected void finalize() throws Throwable {
                super.finalize();
                close();
        }
}
```

Обратите внимание: метод close написан так, чтобы его можно было вызвать несколько раз. В противном случае, если бы метод close уже использовался ранее, то при вызове finalize происходила бы попытка повторного закрытия файла, что может привести к ошибкам.

Кроме того, в приведенном выше примере следует обратить внимание на то, что метод finalize вызывает super.finalize. Пусть это войдет у вас в привычку для всех методов finalize, которые вам придется писать. Если не использовать super.finalize, то работа вашего класса завершится нормально, но суперклассы, для которых не были выполнены необходимые завершающие действия, могут вызвать сбои. Помните, вызов super. finalize — это полезное правило, применяемое даже в том случае, если ваш класс не является расширением другого класса. Помимо того, что это послужит вам дополнительным напоминанием на будущее, вызов super.finalize в подобных ситуациях позволит в будущем породить класс, аналогичный Process File, от другого суперкласса и при этом не заботиться о переделке метода finalize.

В теле метода finalize может применяться конструкция try/catch для обработки исключений в вызываемых методах, но любые неперехваченные исключения, возникшие при выполнении метода finalize, игнорируются. Исключения подробно рассматриваются в главе 7.

Сборщик мусора может уничтожать объекты в любом порядке, а может и не уничтожать их вовсе. Память будет освобождаться в тот момент, который сборщик мусора сочтет подходящим. Отсутствие привязки к какому-то конкретному порядку позволяет сборщику мусора действовать оптимальным образом, что позволяет снизить накладные расходы на его работу. Вы можете напрямую вызвать сборщик мусора, чтобы память была освобождена раньше (см. раздел “Управление памятью”).

При завершении работы приложения вызываются методы finalize для всех существующих объектов. Это происходит независимо от того, что послужило причиной завершения; тем не менее некоторые системные ошибки могут привести к тому, что часть методов finalize не будет запущена. Например, если программа завершается по причине нехватки памяти, то сборщику мусора может не хватить памяти для поиска всех объектов и вызова их методов finalize. И все же в общем случае можно рассчитывать на то, что метод finalize будет вызван для каждого объекта.

2.10.2. Восстановление объектов в методе

Метод finalize может “воскресить” объект, снова делая его используемым — скажем, включая его в статический список объектов. Подобные действия не рекомендуются, но Java не сможет ничего сделать, чтобы предотвратить их.

Тем не менее Java вызывает finalize ровно один раз для каждого объекта, даже если данный объект уничтожается сборщиком мусора повторно из-за того, что он был восстановлен в предыдущем вызове finalize. Если подобное восстановление объектов оказывается важным для концепции вашей программы, то происходить оно будет всего один раз — маловероятно, чтобы вы добивались от программы именно такого поведения.

Если вы действительно считаете, что вам необходимо восстанавливать объекты, попробуйте тщательно пересмотреть свою программу — возможно, вы обнаружите в ней какие-то недочеты. Если же и после этого вы уверены, что без восстановления объектов никак не обойтись, то правильным решением будет дублирование или создание нового объекта, а не восстановление. Метод finalize может создать ссылку на новый объект, состояние которого совпадает с состоянием уничтожаемого объекта. Так как дублированный объект создается заново, при необходимости в будущем может быть вызван его метод finalize, который поместит еще одну копию объекта в какой-нибудь другой список — это обеспечит выживание если не самого объекта, то его точной копии.

2.11. Метод main

Детали запуска Java-приложений могут отличаться для разных систем, но всегда необходимо указать имя класса, который управляет работой приложения. При запуске программы на Java система находит и запускает метод main этого класса. Метод main должен быть объявлен как public, static и void (то есть не возвращающий никакого значения), и ему должен передаваться один аргумент типа String[]. Приведем пример метода main, выводящего значения своих параметров:

```
class Echo {
        public static void main(String[] args) {
                for (int i = 0; i < args.length; i++)
                        System.out.print(args[i] = “ ”);
                System.out.println();
        }
}
```

В массиве строк содержатся “аргументы программы”. Чаще всего они вводятся пользователем при запуске приложения. Например, в системе с использованием командной строки — такой, как UNIX или DOS Shell, — приложение Echo может быть вызвано следующим образом:

java Echo in here

В этой команде java является интерпретатором байт-кода Java, Echo — имя класса, а остальные параметры представляют собой аргументы программы. Команда java находит откомпилированный байт-код класса Echo, загружает его в виртуальную машину и вызывает метод Echo.main с аргументами, содержащимися в элементах массива String. Результат работы программы будет следующим:

in here

Имя класса не включается в число строк, передаваемых main. Оно известно заранее, поскольку это имя класса приложения.

Приложение может содержать несколько методов main, поскольку каждый из его классов может иметь такой метод. Тем не менее в каждой программе используется всего один метод main, указываемый при запуске, — в приведенном выше примере это был метод класса Echo. Присутствие нескольких методов main имеет положительный эффект — каждый класс может иметь метод main, предназначенный для проверки его собственного кода, что дает превосходную возможность для модульного тестирования класса. Мы рекомендуем пользоваться подобной техникой в ваших программах. /Метод main присутствует во многих примерах, приведенных в этой книге. Ограниченный объем не позволяет нам приводить метод main для каждого примера, но обычно мы используем этот метод при разработке собственных классов для нетривиальных приложений и библиотек./

Упражнение 2.14

Измените метод Vehicle.main так, чтобы он создавал объекты-машины для владельцев, чьи имена указаны в командной строке, и выводил информацию о новых объектах.

2.12. Метод toString

Если объект включает общедоступный (public) метод toString, который не получает параметров и возвращает объект String, то данный метод вызывается каждый раз, когда объект этого типа встречается вместо строки в операторе + или +=. Например, следующий фрагмент выводит содержимое массива небесных тел:

```
static void displayBodies(Body[] bodies) {
        for (int i = 0; i < bodies.length; i++)
                System.out.println(i + “: ” + bodies[i]);
}
```

Если повнимательнее присмотреться к вызову println, можно обнаружить два неявных приведения к строковым значениям: первое — для индекса i и второе — для объекта Body. Значения всех примитивных типов неявно преобразуются в объекты String при использовании в подобных выражениях.

В Java не существует универсального механизма для преобразования значения типа String в объект. Разумеется, вы можете включить в класс свою собственную функцию для подобного приведения. Обычно для этого используется некий аналог метода fromString, изменяющий текущее состояние объекта, либо конструктор, принимающий параметр типа String, который определяет исходное состояние объекта.

Упражнение 2.15

Включите в класс Vehicle метод toString.

Упражнение 2.16

Включите в класс LinkedList метод toString.

2.13. Родные методы

Если вам потребовалось написать на Java программу, в которой должен использоваться код, написанный на другом языке программирования, или если вам приходится напрямую работать с какой-либо аппаратурой, можно прибегнуть к помощи родных (native) методов. Родной метод может вызываться из программы на Java, но пишется он на “родном” языке — обычно на C или C++.

При использовании родных методов о переносимости и надежности программы говорить не приходится. Например, родные методы нельзя применять в коде, который должен пересылаться по сети и выполняться на другом компьютере (скажем, в апплетах) — его архитектура может отличаться от вашей, но даже если они и совпадают, удаленный компьютер может попросту не доверять вашей системе настолько, чтобы разрешить запускать у себя откомпилированную программу на C.

Сведения, касающиеся написания родных методов, приведены в Приложении A.

Содержание | Далее

Глава 7
ИСКЛЮЧЕНИЯ

Плохо подогнанное снаряжение может заставить ваш гранатомет М203 выстрелить в самый неожиданный момент.
Подобное происшествие плохо скажется на вашей репутации среди тех, кто останется в живых.
Журнал PS армии США,
август 1993 года

Во время своей работы приложение иногда сталкивается с разного рода нештатными ситуациями. При вызове метода некоторого объекта он может обнаружить у себя внутренние проблемы (неверные значения переменных), найти ошибки в других объектах или данных (например, в файле или сетевом адресе), определить факт нарушения своего базового контракта (чтение данных из закрытого потока) и так далее.

Многие программисты не проверяют все возможные источники ошибок, и на то есть веская причина: если при каждом вызове метода анализировать все мыслимые ошибки, текст программы становится совершенно невразумительным. Таким образом достигается компромисс между правильностью (проверка всех ошибок) и ясностью (отказ от загромождения основной логики программы множеством проверок).

Исключения предоставляют удобную возможность проверки ошибок без загромождения текста программы. Кроме того, исключения непосредственно сигнализируют об ошибках, а не меняют значения флагов или каких-либо полей, которые потом нужно проверять. Исключения превращают ошибки, о которых может сигнализировать метод, в явную часть контракта этого метода. Список исключений виден программисту, проверяется компилятором и сохраняется в расширенных классах, переопределяющих данный метод.

Исключение во*збуждается*, когда возникает неожиданное ошибочное состояние. Затем исключение *перехватывается* соответствующим условием в стеке вызова методов. Если исключение не перехвачено, срабатывает обработчик исключения по умолчанию, который обычно выводит полезную информацию об исключении (скажем, содержимое стека вызовов).

7.1. Создание новых типов исключений

Исключения в Java представляют собой объекты. Все типы исключений (то есть все классы, объекты которых возбуждаются в качестве исключений) должны расширять класс языка Java, который называется Throwable, или один из его подклассов. Класс Throwable содержит строку, которая может использоваться для описания исключения. По соглашению, новые типы исключений расширяют класс Exception, а не Throwable.

Исключения Java, главным образом, являются *проверяемыми* — это означает, что компилятор следит за тем, чтобы ваши методы возбуждали лишь те исключения, о которых объявлено в заголовке метода. Стандартные исключения времени выполнения и ошибки расширяют классы RuntimeException и Error, тем самым создавая *непроверяемые исключения*. Все исключения, определяемые программистом, должны расширять класс Exception, и, таким образом, они являются проверяемыми.

Иногда хочется иметь больше данных, описывающих состояние исключения, — одной строки, предоставляемой классом Exception, оказывается недостаточно. В таких случаях можно расширить класс Exception и создать на его основе новый класс с дополнительными данными (значения которых обычно задаются в конструкторе).

Например, предположим, что в интерфейс Attributed, рассмотренный в главе 4, добавился метод replaceValue, который заменяет текущее значение именованного атрибута новым. Если атрибут с указанным именем не существует, возбуждается исключение — вполне резонно предположить, что заменить несуществующий атрибут не удастся. Исключение должно содержать имя атрибута и новое значение, которое пытались ему присвоить. Для работы с таким исключением создается класс NoSuchAttribiteException:

```
    public class NoSuchAttributeException extends Exception {
    public String attrName;
    public Object newValue;

    NoSuchAttributeException(String name, Object value) {
         super("No attribute named \"" + name + "\" found");
         attrName = name;
         newValue = value;
    }
}
```

NoSuchAttribiteException расширяет Exception и включает конструктор, которому передается имя атрибута и присваиваемое значение; кроме того, добавляются открытые поля для хранения данных. Внутри конструктора вызывается конструктор суперкласса со строкой, описывающей происходящее. Исключения такого рода могут использоваться во фрагменте программы, перехватывающем исключения, поскольку они выводят понятное человеку описание ошибки и данные, вызвавшие ее. Добавление полезной информации — одна из причин, по которым создаются новые исключения.

Другая причина для появления новых типов исключений заключается в том, что тип является важной частью данных исключения, поскольку исключения перехватываются по их типу. Из этих соображений исключение NoSuch AttribiteException стоит создать даже в том случае, если вы не собираетесь включать в него новые данные; в этом случае программист, для которого представляет интерес только это исключение, сможет перехватить его отдельно от всех прочих исключений, запускаемых методами интерфейса Attributed или иными методами, применяемыми к другим объектам в том же фрагменте программы.

В общем случае исключения новых типов следует создавать тогда, когда программист хочет обрабатывать ошибки одного типа и пропускать ошибки другого типа. В этом случае он может воспользоваться новыми исключениями для выполнения нужного фрагмента программы, вместо того чтобы изучать содержимое объекта-исключения и решать, интересует ли его данное исключение или же оно не относится к делу и перехвачено случайно.

7.2. Оператор throw

Исключения возбуждаются оператором throw, которому в качестве параметра передается объект. Например, вот как выглядит реализация replaceValue в классе AttributedImpl из главы 4:

```
public void replaceValue(String name, Object newValue)
throws NoSuchAttributeException
{
    Attr attr = find(name);   // Искать attr
    if (attr == null)               // Если атрибут не найден
         throw new NoSuchAttributeException(name, this);
    attr.valueOf(newValue);
}
```

Метод replaceValue сначала ищет имя атрибута в текущем объекте Attr. Если атрибут не найден, то возбуждается объект-исключение типа NoSuch AttribiteException и его конструктору предоставляются содержательные данные. Исключения являются объектами, поэтому перед использованием их необходимо создать. Если атрибут не существует, то его значение заменяется новым.

Разумеется, исключение может быть порождено вызовом метода, внутри которого оно возбуждается.

7.3. Условие throws

Первое, что бросается в глаза в приведенном выше методе replace Value, — это список проверяемых исключений, которые в нем возбуждаются. В Java необходимо перечислить проверяемые исключения, возбуждаемые методом, поскольку программист при вызове метода должен знать их в такой же степени, в какой он представляет себе нормальное поведение метода. Проверяемые исключения, возбуждаемые методом, не уступают по своей важности типу возвращаемого значения — и то и другое необходимо объявить.

Проверяемые исключения объявляются в условии throws, которое может содержать список значений, отделяемых друг от друга запятыми.

Внутри метода разрешается возбуждать исключения, являющиеся расширениями типа Exception в условии throws, поскольку всегда допускается полиморфно использовать класс вместо его суперкласса. Метод может возбуждать несколько различных исключений, являющихся расширениями одного конкретного класса, и при этом объявить в условии throws всего один суперкласс. Тем не менее, поступая таким образом, вы скрываете от работающих с методом программистов полезную информацию, потому что они не будут знать, какие из возможных расширенных типов исключений возбуждаются методом. В целях надлежащего документирования условие throws должно быть как можно более полным и подробным.

Контракт, определяемый условием throws, обязан неукоснительно соблюдаться — можно возбуждать лишь те исключения, которые указаны в данном условии. Возбуждение любого другого исключения (прямое, с помощью throw, или косвенное, через вызов другого метода) является недопустимым. Отсутствие условия throws не означает, что метод может возбуждать *любые* исключения; наоборот, оно говорит о том, что он не возбуждает *никаких* исключений.

Все стандартные исключения времени выполнения (такие, как ClassCast Exception и ArithmeticException) представляют собой расширения класса RuntimeException. О более серьезных ошибках сигнализируют исключения, которые являются расширениями класса Error и могут возникнуть в произвольный момент в произвольной точке программы. RuntimeException и Error — единственные исключения, которые не нужно перечислять в условии throws; они являются общепринятыми и могут возбуждаться в любом методе, поэтому компилятор не проверяет их. Полный список классов стандартных непроверяемых исключений приведен в Приложении Б.

Инициализаторы и блоки статической инициализации не могут возбуждать проверяемые исключения, как прямо, так и посредством вызова метода, возбуждающего исключение. Во время конструирования объекта нет никакого способа перехватить и обработать исключение. При инициализации полей выход заключается в том, чтобы инициализировать их внутри конструктора, который *может* возбуждать исключения. Для статических инициализаторов можно поместить инициализацию в статический блок, который бы перехватывал и обрабатывал исключение. Статические блоки *не возбуждают* исключений, но могут *перехватывать* их.

Java довольно строго подходит к обработке проверяемых исключений, поскольку это помогает избежать программных сбоев, вызванных невниманием к ошибкам. Опыт показывает, что программисты забывают про обработку ошибок или откладывают ее на будущее, которое так никогда и не наступает. Условие throws ясно показывает, какие исключения возбуждаются методом, и обеспечивает их обработку.

При вызове метода, у которого в условии throws приведено проверяемое исключение, имеются три варианта:

- Перехватить исключение и обработать его.
- Перехватить исключение и перенаправить его в обработчик одного из ваших исключений, для чего возбудить исключение типа, объявленного в вашем условии throws.
- Объявить данное исключение в условии throws и отказаться от его обработки в вашем методе (хотя в нем может присутствовать условие finally, которое сначала выполнит некоторые завершающие действия; подробности приводятся ниже).

При любом из этих вариантов вам необходимо перехватить исключение, возбужденное другим методом; это станет темой следующего раздела.

Упражнение 7.1

Создайте класс-исключение ObjectNotFoundException для класса Linked List, построенного нами в предыдущих упражнениях. Включите в него метод find, предназначенный для поиска объектов в списке, который либо возвращает нужный объект LinkedList, либо возбуждает исключение, если объект отсутствует в списке. Почему такой вариант оказывается более предпочтительным, нежели возврат значения null для ненайденного объекта? Какие данные должны входить в ObjectNotFoundException?

7.4. Операторы try, catch и finally

Чтобы перехватить исключение, необходимо поместить фрагмент программы в оператор try. Базовый синтаксис оператора try выглядит следующим образом:

```
try
    блок
catch (тип-исключения идентификатор)
    блок
catch (тип-исключения идентификатор)
    блок
    .....
finally
    блок
```

Тело оператора try выполняется вплоть до возбуждения исключения или до успешного завершения. Если возникает исключение, то по порядку просматриваются все условия catch, пока не будет найдено исключение нужного класса или одного из его суперклассов. Если подходящее условие catch так и не найдено, то исключение выходит из текущего оператора try во внешний, который может обработать его. В операторе try может присутствовать любое количество условий catch, в том числе и ни одного. Если ни одно из условий catch внутри метода не перехватывает исключение, то оно передается в тот фрагмент программы, который вызвал данный метод.

Если в try присутствует условие finally, то составляющие его операторы выполняются после того, как вся обработка внутри try будет завершена. Выполнение finally происходит независимо от того, как завершился оператор — нормально, в результате исключения или при выполнении управляющего оператора типа return или break.

В приводимом ниже примере осуществляется подготовка к обработке одного из исключений, возбуждаемых в replaceValue:

```
try {
    attributedObj.replaceValue("Age", new Integer(8));
} catch (NoSuchAttributeException e) {
    // так не должно быть, но если уж случилось - восстановить
    Attr attr = new Attr(e.attrName, e.newValue);
    attrbuteObj.add(attr);
}
```

try содержит оператор (представляющий собой блок), который выполняет некоторые действия, в обычных условиях заканчивающиеся успешно. Если все идет нормально, то работа блока на этом завершается. Если же во время выполнения программы в try-блоке возбудилось какое-либо исключение (прямо, посредством throw, либо косвенно, через внутренний вызов метода), то выполнение кода внутри try прекращается, и просматриваются связанные с ним условия catch, чтобы определить, нужно ли перехватывать исключение.

Условие catch чем-то напоминает внедренный метод с одним параметром —типом перехватываемого исключения. Внутри условия catch вы можете пытаться восстановить работу программы после произошедшего исключения или же выполнить некоторые действия и повторно возбудить исключение, чтобы вызывающий фрагмент также имел возможность перехватить его. Кроме того, catch может сделать то, что сочтет нужным, и прекратить свою работу — в этом случае управление передается оператору, следующему за оператором try (после выполнения условия finally, если оно имеется).

Универсальное условие catch (например, перехватывающее исключения типа Exception) обычно говорит о плохо продуманной реализации, поскольку оно будет перехватывать *все* исключения, а не только то, которое нас интересует. Если воспользоваться подобным условием в своей программе, то в результате при возникновении проблем с атрибутами будет обрабатываться, скажем, исключение ClassCastException.

Условия catch в операторе try просматриваются поочередно, от первого к последнему, чтобы определить, может ли тип объекта-исключения присваиваться типу, объявленному в catch. Когда будет найдено условие catch с подходящим типом, происходит выполнение его блока, причем идентификатору в заголовке catch присваивается ссылка на объект-исключение. Другие условия catch при этом не выполняются. С оператором try может быть связано произвольное число условий catch, если каждое из них перехватывает новый тип исключения.

Поскольку условия catch просматриваются поочередно, перехват исключения некоторого типа перед перехватом исключения расширенного типа является ошибкой. Первое условие всегда будет перехватывать исключение, а второе — никогда. По этой причине размещение условия catch для исключения-суперкласса перед условием для одного из его подклассов вызывает ошибку во время компиляции:

```
class SuperException extends Exception { }
class SubException extends SuperException { }

class BadCatch {
    public void goodTry() {
         /* НЕДОПУСТИМЫЙ порядок перехвата исключений */
         try {
              throw new SubException();
         } catch (SuperException superRef) {
              // Перехватывает и SuperException, и SubException
         } catch (SubException subRef) {
              // Никогда не выполняется
         }
    }
}
```

В каждом операторе try обрабатывается только один исключительный случай. Если catch или finally возбуждают новое исключение, то условия catch данного try не рассматриваются повторно. Код в условиях catch и finally находится за пределами защиты оператора try. Разумеется, возникающие в них исключения могут быть обработаны любым внешним блоком try, для которого внутренние catch или finally являются вложенными.

7.4.1. Условие finally

Условие finally оператора try позволяет выполнить некоторый фрагмент программы независимо от того, произошло исключение или нет. Обычно работа такого фрагмента сводится к “чистке” внутреннего состояния объекта или освобождению “необъектных” ресурсов (например, открытых файлов), хранящихся в локальных переменных. Приведем пример метода, который закрывает файл после завершения своей работы даже в случае возникновения ошибки:

```
public boolean searchFor(String file, String word)
    throws StreamException
{
    Stream input = null;
    try {
         input = new Stream(file);
         while (!input.eof())
              if (input.next() == word)
                   return true;
         return false;      // поиск завершился неудачно
    } finally {
         if (input != null)
              input.close();
    }
}
```

Если создание объекта оператором new закончится неудачно, то input сохранит свое исходное значение null. Если же выполнение new будет успешным, то input будет содержать ссылку на объект, соответствующий открытому файлу. Во время выполнения условия finally поток input будет закрываться лишь в том случае, если он предварительно был открыт. Независимо от того, возникло ли исключение при работе с потоком или нет, условие finally обеспечивает закрытие файла; благодаря этому экономится такой ограниченный ресурс, как количество одновременно открытых файлов. Метод searchFor объявляет о том, что он возбуждает StreamException, чтобы все порожденные в нем исключения после выполнения завершающих действий передавались в вызывающий фрагмент программы.

Условие finally может также использоваться и для выполнения завершающих действий после операторов break, continue и return — вот почему иногда можно встретить try без соответствующих ему catch. При обработке любого оператора, передающего управление программы в другую точку, выполняются все условия finally. Невозможно покинуть try-блок без выполнения его условия finally.

В приведенном выше примере finally используется и для выполнения завершающих действий в случае нормального возврата по оператору return. Один из самых распространенных случаев использования goto в других языках — необходимость выполнения определенных действий при завершении программного блока, как успешном, так и аварийном. В нашем примере finally обеспечивает закрытие файла и при выполнении оператора return, и при возбуждении исключения.

У условия finally всегда имеется некоторая причина. Она может состоять в нормальном завершении блока try, или в выполнении управляющего оператора наподобие return, или же в возбуждении исключения во фрагменте, заключенном в try-блок. Эта причина запоминается и при выходе из блока finally. Тем не менее, если в блоке finally возникает новая причина выхода (скажем, выполняется управляющий оператор вроде break или return или возбуждается исключение), то она отменяет старую, и о существовании последней забывается. Например, рассмотрим следующий фрагмент:

```
try {
    // ... сделать что-нибудь ...
    return 1;
} finally {
    return 2;
}
```

Когда выполняется return внутри блока try, то на входе блока finally код возврата равен 1. Однако внутри самого блока finally возвращается значение 2, так что исходный код возврата забывается. В сущности, если бы в блоке try было возбуждено исключение, то код возврата также был бы равен 2. Если бы блок finally не возвращал никакого значения, а просто завершался нормальным образом, то код возврата был бы равен 1.

7.5. Когда применяются исключения

В начале этой главы мы воспользовались выражением “неожиданное ошибочное состояние”, чтобы описать момент возбуждения исключения. Исключения не предназначаются для простых, предсказуемых ситуаций. Например, достижение конца входного потока является заранее предсказуемым, так что проверка на “исчерпание” потока является частью ожидаемого поведения метода. Код возврата, сигнализирующий о конце ввода, и проверка его при вызове также выглядят вполне разумно — к тому же такую конструкцию оказывается проще понять. Сравним типичный цикл, в котором используется флаг возврата

```
while ((token = stream.next()) != Stream.END)
    process(token);
stream.close();
```

с другим циклом, в котором о достижении конца ввода сигнализирует исключение:

```
try {
    for (;;) {
         process(stream.next());
    }
} catch (StreamEndException e) {
    stream.close();
}
```

В первом случае логика выполнения программы оказывается прямолинейной и понятной. Цикл выполняется до тех пор, пока не будет достигнут конец входного потока, после чего поток закрывается. Во втором случае цикл выглядит так, словно он выполняется бесконечно. Если не знать о том, что о конце ввода сигнализирует StreamEndException, то становится непонятно, когда же происходит выход из цикла. Впрочем, даже если вы догадываетесь о существовании StreamEndException, то факт завершения цикла оказывается вынесенным из его тела (for) во внешний блок try, что затрудняет понимание происходящего.

Встречаются ситуации, в которых невозможно найти приемлемый код возврата. Например, в классе, представляющем поток значений типа double, может содержаться любое допустимое double, поэтому числовой маркер конца потока невозможен. Более разумный подход предусматривает специальный метод eof для проверки конца потока, который должен выполняться перед каждой операцией чтения:

```
while (!stream.eof())
    process(stream.nextDouble());
stream.close();
```

С другой стороны, попытка чтения *после* достижения конца файла оказывается непредсказуемой. Она свидетельствует о том, что программа не заметила конца входного потока и теперь пытается сделать что-то такое, чего делать ни в коем случае не следует. Перед нами — отличная возможность применить исключение ReadPastEndException. Подобные действия выходят за границы ожидаемого поведение класса-потока, так что возбуждение исключения в данном случае оказывается вполне уместным.

Решить, какая ситуация является предсказуемой, а какая — нет, порой бывает нелегко. Главное — не злоупотреблять исключениями для сообщений о предсказуемых ситуациях.

Упражнение 7.2

Как, по вашему мнению, программа должна сообщать программисту о следующих ситуациях:

- Программа пытается присвоить отрицательное значение вместимости машины в объекте PassengerVehicle.
- В файле конфигурации, используемом для задания начального состояния объекта, обнаружена синтаксическая ошибка.
- Метод, который ищет в строковом массиве указанное программистом слово, не может найти ни одного экземпляра такого слова.
- Файл, передаваемый методу open, не существует.
- Файл, передаваемый методу open, существует, но система безопасности не разрешает пользователю работать с ним.
- При попытке установить сетевое соединение с удаленным процессом-сервером не удается связаться с удаленным компьютером.
- Во время взаимодействия с удаленным процессом-сервером прерывается сетевое соединение.

Содержание | Далее

Глава 6
ПОРЯДОК ВЫПОЛНЕНИЯ

— *Скажите, пожалуйста, куда мне отсюда идти?*
— Это во многом зависит от того, куда ты хочешь прийти.
Льюис Кэрролл, “Алиса в Стране Чудес”,
перевод Б. Заходера

Программа, состоящая из последовательно выполняемых операторов, несомненно способна принести пользу, так как операторы выполняются в том порядке, в котором они встречаются в программе. Однако возможность управлять порядком следования операторов (то есть проверять некоторое условие и в зависимости от результатов проверки выполнять разные действия) знаменует собой качественно новый уровень в средствах разработки. В этой главе рассмотрены практически все *управляющие операторы*, которые определяют порядок выполнения программы. Исключения рассмотрены отдельно в главе 7.

6.1. Операторы и блоки

Существует две основные категории операторов: *операторы-выражения* и *операторы-объявления*; и те и другие уже встречались нам в этой книге. Операторы-выражения (такие, как i++ или вызовы методов) в соответствии с названием представляют собой выражения, в конце которых стоит завершающая точка с запятой. /Необходимо помнить об отличии терминатора (завершающего символа) от разделителя. Запятые при перечислении идентификаторов в объявлении являются разделителями, потому что они разделяет элементы в списке. Точка с запятой является терминатором, так как она завершает каждый оператор. Если бы точка с запятой была разделителем операторов, то последняя точка с запятой внутри блока была бы излишней, и, возможно, даже недопустимой./ Не каждое выражение может стать оператором, поскольку, например, превращение проверки <= в автономный оператор почти всегда оказывается бессмысленным. Следующие типы выражений могут превращаться в операторы за счет добавления завершающей точки с запятой:

- выражения присваивания, содержащие = или один из операторов op=;
- префиксные или постфиксные формы ++ и —;
- вызовы методов (независимо от того, возвращают ли они какие-либо значения);
- выражения, в которых используется оператор new для создания объектов.

Операторы-объявления (которые формально следовало бы называть операторами-объявлениями локальных переменных) объявляют переменные и, возможно, инициализируют их. Они могут находиться в любом месте блока — не обязательно в начале. Локальные переменные существуют лишь во время выполнения блока, в котором они объявлены. Перед тем как их использовать, необходимо указать их начальные значения — либо посредством инициализации при объявлении, либо оператором присваивания. При попытке использования локальной переменной, не получившей начального значения, выдается ошибка во время компиляции.

Кроме перечисленных выше операторов-выражений, существуют и другие операторы, влияющие на ход выполнения программы, — например, if и for. В этой главе мы подробно рассмотрим каждый из таких операторов.

Фигурные скобки { и } применяются для группировки нуля или более операторов в *блок*. Последний может использоваться везде, где допускается отдельный оператор, поскольку блок *является* оператором (хотя и составным).

6.2. Оператор if-else

Одним из основных средств управления выполнением программы является оператор if, который позволяет решить, нужно ли производить те или иные действия. Его синтаксис выглядит следующим образом:

```
if (логическое выражение)
        оператор1
else
        оператор2
```

Сначала определяется значение логического выражения. Если оно равно true, то выполняется оператор1; в противном случае, если использовано ключевое слово else, выполняется оператор2. Присутствие else не является обязательным.

Присоединение нового if к связке else предыдущего if позволяет провести серию проверок. Приведем метод, который на основании содержимого строки, равной одному из некоторых известных слов, выбирает и производит некоторое действие над вторым аргументом:

```
public void setProperty(String keyword, double value)
throws UnknownProperty
{
if (keyword.equals("charm"))
charm(value);
else if (keyword.equals("strange"))
strange(value);
else
throw new UnknownProperty(keyword);
}
```

Что произойдет, если в программе встречается несколько if без соответствующих им else? Например:

```
public double sumPositive(double[] values) {
double sum 0.0;

if (values.length >> 1)
for (int i = 0; i << values.length; i++)
if (values[i] >> 0)
sum += values[i];
else     // не тут-то было!
sum = values[0];
return sum;
}
```

Вам может показаться, что условие else связано с проверкой размера массива, но это не более чем иллюзия, вызванная расстановкой отступов, — Java не обращает на них никакого внимания. Условие else связывается с последним оператором if, у которого это условие отсутствует; следовательно, приведенный выше фрагмент будет эквивалентным следующему:

```
public double sumPositive(double[] values) {
double sum 0.0;

if (values.length >> 1)
for (int i = 0; i << values.length; i++)
if (values[i] >> 0)
sum += values[i];
else     // не тут-то было!
sum = values[0];
return sum;
}
```

Вероятно, это не совсем то, на что вы рассчитывали. Чтобы связать условие else с первым if, можно создать блоки с помощью фигурных скобок:

```
public double sumPositive(double[] values) {
double sum 0.0;

if (values.length >> 1) {
for (int i = 0; i << values.length; i++)
if (values[i] >> 0)
sum += values[i];
} else {
sum = values[0];
        }
return sum;
}
```

Упражнение 6.1

Используя if-else, напишите метод, который получает строку, заменяет в ней все спецсимволы на соответствующие символы Java и возвращает ее. Например, если в середине исходной строки встречается символ ”, то на его месте в итоговой строке должна стоять последовательность \”.

6.3. Оператор switch

Оператор switch вычисляет целочисленное выражение и в соответствии с полученным результатом ищет метку case в блоке. Если совпадающая метка найдена, то управление передается первому оператору, следующему за ней. Если метка не обнаружена, то следующим будет выполняться оператор, находящийся за меткой default. Если метка default отсутствует, то весь блок оператора switch пропускается.

Приведенный ниже пример выводит информацию о состоянии объекта. При этом уровень детализации выбирается пользователем. Затем производится вывод более скупых данных:

```
public static final int TERSE = 0,
                        NORMAL = 1,
                        BLATHERING = 2;

// ...

public int Verbosity = TERSE;

public void dumpState()
throws UnexpectedStateException
{
switch (Verbosity) {
  case BLATHERING:
System.out.println(stateDetails);
System.out.println(stateDetails);
// ПРОХОД

  case NORMAL:
System.out.println(basicState);
// ПРОХОД
  case TERSE:
System.out.println(summaryState);
break;

  default:
throw new UnexpectedStateException(Verbosity);
}
}
```

В классе определено несколько констант, соответствующих различным уровням детализации. Когда наступает время вывести состояние объекта, это делается на нужном уровне детализации.

Комментарий ПРОХОД означает, что программа *проходит* через соответствующую метку, а выполнение продолжается со следующего за ней оператора. Следовательно, если уровень детализации равен BLATHERING, то печатаются все три составные части отчета; если он равен NORMAL, то печатаются две части; наконец, если уровень детализации равен TERSE, то печатается только одна часть.

Метка case или default *не* приводит к прерыванию работы оператора switch и не нарушает хода выполнения программы. Именно по этой причине мы вставили оператор break после завершения вывода для уровня TERSE. Без break выполнение программы было бы продолжено операторами, следующими за меткой default, что каждый раз приводило бы к возбуждению исключения.

Проход к следующему условию case может оказаться полезным при некоторых обстоятельствах, однако в большинстве случаев в конце фрагмента, соответствующего метке case, должен находиться оператор break. Хороший стиль программирования требует, чтобы намеренный проход к следующей метке case сопровождался каким-нибудь комментарием наподобие того, что приведен в примере.

Проход чаще всего применяется для использования нескольких меток case с одним фрагментом программы. В следующем примере он используется для перевода шестнадцатеричной записи цифры в значение типа int:

```
public int hexValue(char ch) throws NonDigitException {
        switch (ch) {
                case '0': case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
                        return (ch - '0');

                case 'a': case 'b': case 'c':
                case 'd': case 'e': case 'f':
                        return (ch - 'a') + 10;

                case 'A': case 'B': case 'C':
                case 'D': case 'E': case 'F':
                        return (ch - 'A') + 10;

                default:
                        throw new NonDigitException(ch);
        }
}
```

Операторы break в данном случае не нужны, потому что операторы return осуществляют выход из метода и не позволяют пройти к следующему оператору.

Последнюю группу операторов внутри switch следует завершать оператором break, return или throw, как это делалось для всех предыдущих условий. Это уменьшает вероятность того, что при добавлении в будущем нового условия case произойдет случайный проход к нему из того фрагмента, который когда-то завершал оператор switch.

Все метки case должны быть постоянными выражениями, то есть содержать только литералы и поля static final, инициализированные константами. В каждом отдельном операторе switch значения меток case должны быть различными. Допускается присутствие не более одной метки default.

Упражнение 6.2

Перепишите метод из упражнения 6.1 с использованием оператора switch.

6.4. Цикл while и do-while

Цикл while выглядит следующим образом:

```
while (логическое выражение)
        оператор
```

В начале работы оператора вычисляется логическое выражение, и если оно равно true, то выполняется оператор (который, разумеется, может представлять собой блок); это происходит до тех пор, пока логическое выражение не станет равным false.

Цикл while выполняется ноль или более раз, поскольку логическое выражение может оказаться равным false при его первом вычислении.

Иногда бывает нужно, чтобы тело цикла заведомо было выполнено хотя бы один раз, и поэтому в языке Java также предусмотрена конструкция do-while:

```
do-while:
do
        оператор
while (логическое выражение);
```

Здесь логическое выражение вычисляется *после* оператора. Цикл выполняется, пока выражение остается равным true. Оператор, являющийся телом цикла do-while, почти всегда представляет собой блок.

6.5. Оператор for

Оператор for используется для выполнения цикла по значениям из определенного диапазона. Он выглядит следующим образом:

```
for (инициализация; логическое выражение; приращение)
        оператор
```

Такая запись эквивалентна

```
{
        инициализация;
        while (логическое выражение) {
                оператор
                приращение;
        }
}
```

за тем исключением, что *приращение* всегда выполняется, если в теле цикла встречается оператор continue (см. раздел “Оператор continue”).

Обычное применение цикла for — поочередное присвоение переменной значений из некоторого диапазона, пока не будет достигнут конец этого диапазона.

Выражения инициализации и приращения в цикле for могут представлять собой список значений, разделяемых запятой. Вычисление этих выражений, как и в других операторах, происходит слева направо. Например, чтобы с помощью двух индексов перебрать все элементы массива в противоположных направлениях, можно воспользоваться следующим выражением:

```
for ( i = 0, j = arr.length - 1; j >>= 0; i++, j--) {
        // ...
}
```

Пользователь сам выбирает диапазон значений переменной цикла. Например, цикл for часто применяется для перебора элементов связного списка или значений, входящих в математическую последовательность. Благодаря этому конструкция for в Java оказывается существенно более мощной, чем в других языках программирования, в которых возможности циклов for ограничиваются приращением переменной в заранее заданном диапазоне.

Приведем пример цикла, который вычисляет наименьший показатель (exp), такой, что 10 в степени exp превосходит заданную величину:

```
public static int tenPower(int value) {
        int exp, v;
        for (exp = 0, v = value - 1; v >> 0; exp++, v /= 10)
                continue;
        return exp;
}
```

В данном случае в цикле одновременно изменяются две переменные: показатель степени (exp) и значение 10^{exp} (v). Эти переменные являются взаимосвязанными. В подобных случаях разделенный запятыми список является корректным способом обеспечения синхронизации значений.

Тело цикла представляет собой простой оператор continue, начинающий следующую итерацию цикла. В теле цикла вам ничего не приходится делать — все происходит в проверяемом условии и в выражении-итерации. Использованный в данном примере оператор continue — одна из возможностей создать пустое тело цикла; вместо него можно было оставить отдельную строку, состоящую из одной точкой с запятой, или создать пустой блок в фигурных скобках. Просто ограничиться точкой с запятой в конце строки с оператором for было бы довольно опасно — если точка с запятой будет случайно удалена, то оператор, следующий за for, превратится в тело цикла.

Все выражения в заголовке цикла for являются необязательными. Если пропустить инициализацию или приращение, то соответствующая часть цикла просто не выполняется. Отсутствующее логическое выражение считается всегда равным true. Следовательно, для создания бесконечного цикла можно воспользоваться следующей записью:

```
for (;;)
        оператор
```

Подразумевается, что цикл будет прерван иными средствами — скажем, описанным ниже оператором break или возбуждением исключения.

По общепринятому соглашению цикл for используется для диапазонов взаимосвязанных значений. Нарушение этого соглашения, когда выражения инициализации или приращения не имеют никакого отношения к проверке логического условия, считается проявлением плохого стиля программирования.

Упражнение 6.3

Напишите метод, который получает два параметра типа char и выводит все символы, лежащие в диапазоне между ними (включая их самих).

6.6. Метки

Операторам могут присваиваться метки. Чаще всего метки применяются в блоках и циклах. Метка предшествует оператору следующим образом:

метка: оператор

Именованные блоки часто используются с операторами break и continue.

6.7. Оператор break

Оператор break применяется для выхода из *любого* блока, не только из оператора switch. Чаще всего оператор break служит для прерывания цикла, но он может использоваться для немедленного выхода из любого блока. В приведенном ниже примере этот оператор помогает найти первый пустой элемент в массиве ссылок на объекты Contained:

```
class Container {
        private Contained[] Objs;

        // ...

        public void addIn(Contained obj)
                throws NoEmptySlotException
        {
                int i;
                for (i = 0; i << Objs.length; i++)
                        if (Objs[i] == null)
                        break;
                if (i >>= Objs.length)
                        throw new NoEmptySlotException();
                Objs[i] = obj;     // занести в найденный элемент
                obj.inside(this);  // сообщить о занесении
        }
}
```

Оператор break без метки применяется для выхода из внутренних операторов switch, for, while или do. Чтобы выйти из внешнего оператора, снабдите его меткой и укажите ее в операторе break:

```
private float[][] Matix;

public boolean workOnFlag(float flag) {
        int y, x;
        boolean found = false;

    search:
        for (y = 0; y << Matrix.length; y++) {
                for (x = 0; x << Matrix[y].length; x++) {
                        if (Matrix[y][x] == flag) {
                                found = true;
                                break search;
                        }
                }
        }
        if (!found)
                return false;
        // сделать что-нибудь с найденным элементом матрицы
        return true;
}
```

Использование меток оставляется на усмотрение программиста. Впрочем, оно может оказаться хорошей защитной мерой против модификации ваших программ — например, включения их фрагментов в оператор switch или цикл.

Обратите внимание: break с меткой — это не то же самое, что goto. Оператор goto приводит к хаотичным прыжкам по программе и усложняет понимание ее смысла. Напротив, оператор break или continue, в котором используется метка, осуществляет выход лишь из конкретного помеченного блока, а порядок выполнения программы остается вполне понятным.

6.8. Оператор continue

Оператор continue осуществляет переход в конец тела цикла и вычисляет значение управляющего логического выражения. Этот оператор часто используется для пропуска некоторых значений в диапазоне цикла, которые должны игнорироваться или обрабатываться в отдельном фрагменте. Например, при работе с потоком, состоящим из отдельных лексем, лексема “skip” (“*пропуск*”) может обрабатываться следующим образом:

```
while (!stream.eof()) {
        token = stream.next();
        if (token.equals("skip"))
                continue;
        // ...  обработка лексемы ...
}
```

Оператор continue имеет смысл только внутри циклов — while, do-while и for. В нем может указываться метка внешнего цикла, и в этом случае continue относится к указанному циклу, а не к ближайшему внутреннему. При выполнении такого помеченного оператора continue осуществляется выход из всех внутренних циклов, чтобы выполнить следующую итерацию указанного цикла. В приведенном выше примере можно обойтись без метки в операторе continue, поскольку имеется всего один внешний цикл.

6.9. Оператор return

Оператор return завершает выполнение метода и передает управление в точку его вызова. Если метод не возвращает никакого значения, достаточно простого оператора return:

return;

Если же метод имеет возвращаемый тип, то в оператор return должно входить такое выражение, которое может быть присвоено переменной возвращаемого типа. Например, если метод возвращает double, то в оператор return могут входить выражения типа double, float или целого типа:

```
protected double nonNegative(double val) {
        if (val << 0)
                return 0;    // константа типа int
        else
                return val;  // double
}
```

Оператор return также используется для выхода из конструкторов и статических инициализаторов. Конструктор не может возвращать никакого значения, поэтому в этом случае return не содержит возвращаемого значения. Конструкторы вызываются как часть процесса new, который в конечном счете *возвращает* ссылку на объект, однако каждый конструктор играет в этом процессе лишь частичную роль; ни один из конструкторов не возвращает итоговую ссылку.

6.10. Где же goto?

В Java нет конструкции goto, которая служила бы для передачи управления произвольному оператору внутри метода, хотя она достаточно распространена в языках того семейства, с которым связан язык Java. Оператор goto чаще всего применяется для следующих целей:

- Управление выполнением внешних циклов из внутренних. Для этого в Java предусмотрены операторы break и continue с метками.
- Пропуск оставшейся части блока, не входящего в цикл, при нахождении ответа или обнаружении ошибки. Используйте break с меткой.
- Выполнение завершающего кода при выходе из блока или метода. Используйте либо break с меткой, либо (более наглядно) — конструкцию finally оператора try, рассмотренную в следующей главе.

Операторы break и continue с метками имеют то преимущество, что они передают управление в строго определенную точку программы. Блок finally подходит к передаче управления еще более жестко и работает при всех обстоятельствах, в том числе и при возникновении исключений. С помощью этих конструкций можно писать наглядные программы на Java без применения goto.

Содержание | Далее