Учебное пособие

ПИТЕР

СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

А. Н. Васильев

Java

Объектно-ориентированное программирование

ДЛЯ
МАГИСТРОВ
И БАКАЛАВРОВ

БАЗОВЫЙ КУРС ПО ОБЪЕКТНО-ОРИЕНТИРОВАННОМУ ПРОГРАММИРОВАНИЮ

СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

А. Н. Васильев

Java

Объектно-ориентированное программирование

ДЛЯ МАГИСТРОВ И БАКАЛАВРОВ

Базовый курс
по объектно-ориентированному
программированию

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2011

ББК 32.972.2-018я7
УДК 004.43(075)
В19

Васильев А. Н.

В19 Java. Объектно-ориентированное программирование: Учебное пособие. — СПб.: Питер, 2011. — 400 с.

ISBN 978-5-49807-948-6

Учебное пособие предназначено для изучающих объектно-ориентированное программирование в вузе, а также для всех желающих самостоятельно изучить язык программирования Java. Книга охватывает все базовые темы, необходимые для эффективного составления программ на Java, в том числе базовые типы данных, управляющие инструкции, особенности описания классов и объектов в Java, создание пакетов и интерфейсов, перегрузку методов и наследование. Особое внимание уделяется созданию приложений с графическим интерфейсом.

В первой части книги излагаются основы синтаксиса языка Java. Материала первой части книги достаточно для написания простых программ. Во второй части описываются темы, которые будут интересны тем, кто хочет освоить язык на профессиональном уровне. Каждая глава книги содержит теоретический материал, иллюстрируемый простыми примерами, позволяющими подчеркнуть особенности языка программирования Java. В конце каждой главы первой части имеется раздел с примерами решения задач.

Учебное пособие соответствует Государственному образовательному стандарту 3-го поколения для специальностей «Информатика и вычислительная техника», «Информационные системы и технологии», «Прикладная информатика» и «Фундаментальная информатика и информационные технологии».

ББК 32.972.2-018я7
УДК 004.43(075)

ISBN 978-5-49807-948-6

Краткое оглавление

Оглавление

Вступление. О книге и не только

Вниманию читателя предлагается книга по языку программирования Java. В основу книги положены курсы лекций, прочитанные в разное время автором для магистров на физическом факультете Киевского национального университета имени Тараса Шевченко и бакалавров на медико-инженерном факультете Национального технического университета «Киевский политехнический институт». Курс адаптирован для всех желающих самостоятельно изучать язык программирования Java и поэтому может использоваться в качестве самоучителя.

Материал книги разбит на две части, в каждой из которой по шесть глав, хотя следует признать, что деление это во многом условное. В первой части излагаются основы синтаксиса языка Java. Этого материала вполне достаточно, чтобы начать писать простые программы. Во второй части описываются темы, интересные тем, кто желает освоить язык на профессиональном уровне. Вообще же книга охватывает все базовые темы, необходимые для эффективного составления программ на Java, в том числе в ней описываются базовые типы данных, управляющие инструкции, особенности создания классов и объектов в Java, способы создания пакетов и интерфейсов, перегрузка методов и наследование. Кроме того, отдельно рассматриваются вопросы создания приложений с графическим интерфейсом. Этой теме посвящена последняя глава книги.

Теоретический материал каждой главы иллюстрируется достаточно простыми примерами, позволяющими подчеркнуть особенности языка программирования Java. В конце каждой главы первой части имеется раздел с примерами решения задач. Это более сложные по сравнению с примерами основной части главы задачи и нередко для их решения приходится использовать специальные подходы или приемы. Разделы с примерами призваны продемонстрировать практические приемы применения программных средств Java для составления эффективных программных кодов. Для лучшего усвоения материала в конце каждой главы имеется краткое резюме.

Объектно-ориентированное программирование и Java

Язык Java является полностью объектно-ориентированным. Это означает, что любая программа, написанная на языке Java, должна поддерживать парадигму объектно-ориентированного программирования (сокращенно ООП). В отличие от традиционного процедурного программирования, объектно-ориентированные программы подразумевают описание классов и, как правило, создание объектов. На сегодняшний день существует несколько наиболее популярных языков программирования, поддерживающих концепцию ООП. В первую очередь это C++, C# и Java. Исторически первым появился язык C++, ставший существенно усовершенствованной версией языка C. Усовершенствования касались главным образом поддержки парадигмы ООП. Именно C++ стал в известном смысле родительским для языков C# и Java. В этом несложно убедиться, если сравнить синтаксисы языков — они очень схожи. Язык C++ в известном смысле является «переходным», поскольку позволяет писать программы как с использованием классов, так и без них. В то же время такие языки, как Java и C#, для составления даже самой простой программы требуют использовать концепцию классов.

Язык Java является продуктом компании Sun Microsystems (теперь эта компания поглощена корпорацией Oracle), язык C# поддерживается компанией Microsoft. Языки программирования Java и C# можно рассматривать как попытку «усовершенствовать» и «адаптировать» используемые в C++ подходы для эффективного создания программных кодов, ориентированных на Интернет. В данном случае задачи и проблемы, которые решали создатели языка C#, нас интересовать не будут, а вот об особенностях языка Java хочется сказать несколько слов.

«Написано однажды — работает везде!» — эти слова можно назвать главным принципом, положенным в основу технологии Java. Именно на создание универсальной технологии программирования были направлены усилия разработчиков компании Sun Microsystems, в результате чего и появился язык программирования Java. Периодом создания языка принято считать годы с 1991 по 1995. К тому времени остро встала проблема составления эффективных программ для работы в Интернете. В этом случае важное место занимают вопросы совместимости программного обеспечения, поскольку особенностью интернет-среды является принципиальное разнообразие используемых операционных систем и аппаратного обеспечения. Другими словами, задача состояла в том, чтобы эффективность и корректность написанных на Java программ не зависела (или почти не зависела) от типа процессора или операционной системы.

Решение задачи было найдено в рамках концепции виртуальной Java-машины. Так, если обычно при компиляции программы (например, написанной на C++) на выходе мы получаем исполнительный машинный код, то в результате компиляции Java-программы получают промежуточный байт-код, который выполняется не операционной системой, а виртуальной Java-машиной (Java Virtual Machine, JVM). Разумеется, предварительно виртуальная Java-машина должна быть установлена на компьютер пользователя. С одной стороны, это позволяет создавать достаточно универсальные программы (в том смысле, что они могут

использоваться с разными операционными системами). Однако, с другой стороны, платой за такую «универсальность» является снижение скорости выполнения программ.

Кроме того, следует четко понимать, что язык Java создавался для написания больших и сложных программ. Писать на Java консольные программы, которые выводят сообщения вроде «Hello, world!» — это все равно, что на крейсере отправиться на ловлю карасей. Тем не менее Java позволяет решать и такие задачи (имеются в виду программы, а не караси). Хотя большинство примеров в книге представляют собой как раз простые программные коды, в данном случае это оправдано, поскольку в учебе хороши любые приемы — главное, чтобы они были эффективными.

Различия между Java и C++

Следующее замечание предназначено специально для тех, кто программирует на C++. Вначале, особенно из первых глав книги, может сложиться впечатление, что различия между языками C++ и Java носят чисто внешний, косметический характер. На самом деле это не так. Чем глубже проникать в концепцию технологии Java, тем отчетливее будет вырисовываться непохожесть Java и C++. Первое проявление непохожести языков читатель встретит в главе 3, посвященной массивам. В отличие от языка C++ в Java все массивы являются динамическими с автоматической проверкой ситуации выхода за пределы массива. Поэтому если известно имя массива, можно достаточно просто узнать его размер. Более того, в Java существенно переработана концепция указателей. Внешний эффект связан с тем, что в Java указатели как таковые отсутствуют, хотя пытливый ум заметит их неявное присутствие. Например, в C++ имя массива является указателем на его первую ячейку. В Java имя массива является переменной, которая фактически служит ссылкой на массив. То есть, по большому счету, это тот же указатель, только надежно спрятанный от программиста. Таким образом, в Java переменная массива и сам массив — далеко не одно и то же. И хотя может показаться, что это неудобно, на практике все выглядит иначе. Вот самые простые примеры выгоды от такого подхода: в Java одной переменной массива можно присвоить значение другой переменной массива. При этом размеры соответствующих массивов могут и не совпадать — достаточно, чтобы совпадали размерности и тип. Нечто похожее можно сделать и в C++, но для этого придется немного потрудиться.

Аналогичная ситуация имеет место с объектами. Все объекты в Java создаются динамически, и объектная переменная является ссылкой на объект. Поэтому при присваивании объектов ссылка с одного объекта «перебрасывается» на другой объект. Данное обстоятельство постоянно следует иметь в виду при работе с объектами.

Благодаря специальным классам в Java намного удобнее работать с текстом, хотя это субъективная точка зрения автора, и читатель с ней может не согласиться.

Неприятным сюрпризом для поклонников C++ может стать невозможность перегрузки операторов в Java. Эта красивая и эффективная концепция, реализованная в C++, разработчиками Java была проигнорирована. Хотя с точки зрения стабильности программного кода это можно было бы и оправдать, с хорошими игрушками расставаться обидно.

В то же время утверждать, что C++ и Java — языки абсолютно разные, было бы некоторым преувеличением. Безусловно, для тех, кто знаком с C++, освоить Java особого труда не составит. Знание C++ является несомненным преимуществом, просто нужно иметь в виду упомянутые особенности языка Java.

Программное обеспечение

Необходимо отдать должное компании Sun Microsystems. Она не только предложила достаточно оригинальный и мощный язык программирования, но и создала широкий спектр программных средств, в основном распространяющихся на условиях лицензии с открытым кодом. Загрузить все (или практически все) необходимое для работы программное обеспечение можно на сайте www.java.com, посвященном технологии Java.

Для того чтобы программировать в Java, необходимо установить среды JDK (Java Development Kit — среда разработки Java) и JRE (Java Runtime Environment — среда выполнения Java). Обе свободно загружаются с сайта www.java.com (или www.oracle.com). В принципе, этого для работы достаточно. Однако лучше все же прибегнуть к помощи какой-нибудь интегрированной среды разработки. Лучшим выбором в этом случае будет среда NetBeans, которая доступна на сайте www.netbeans.org. Причем к услугам пользователей предоставляются полные версии среды, включая системы JDK и JRE. Можно также воспользоваться средой Eclipse, которая свободно доступна на сайте www.eclipse.org. Правда, работа с этой средой имеет свои особенности. Используемому при программировании в Java программному обеспечению посвящено приложение в конце книги.

Обратная связь

Полезную для себя информацию читатели могут найти на сайте автора www.vasilev.kiev.ua. Свои замечания, пожелания и предложения можно отправить по электронной почте на адрес vasilev@univ.kiev.ua или alex@vasilev.kiev.ua.

Программные коды

Рассмотренные в книге программные коды можно загрузить через Интернет с сайта издательства www.piter.com или персональной страницы автора www.vasilev.kiev.ua.

Благодарности

К чтению курса лекций по Java на медико-инженерном факультете Национального технического университета «Киевский политехнический институт» автора приобщил декан (на тот момент) факультета, заведующий кафедрой медицинской кибернетики и телемедицины профессор *Яценко Валентин Порфирьевич*. Эту приятную традицию поддержал нынешний декан медико-инженерного факультета, заведующий кафедрой биомедицинской инженерии, профессор *Максименко Виталий Борисович*. Автор считает своей приятной обязанностью выразить им за это свою искреннюю благодарность.

Автор выражает искреннюю признательность издательству «Питер» и лично *Андрею Юрченко* за профессиональную и эффективную работу по выпуску книги. Хочется также поблагодарить редактора *Алексея Жданова* за его полезные замечания, благодаря которым книга стала намного лучше.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Часть I. Введение в Java

Глава 1. Основы Java

Фандорин, у меня времени нет! Скажите по-человечески.
Я не понимаю этого языка.

Из к/ф «Статский советник»

Как отмечалось во вступлении, язык программирования Java является полностью объектно-ориентированным. Это означает, что для составления даже самой простой программы необходимо описать класс. Однако в языке программирования Java, кроме классов и объектов, есть на что обратить внимание.

Рассмотрение методов программирования в Java начнем с наиболее простых случаев. При этом нам все же придется использовать классы. Чтобы не загромождать самое начало книги довольно отвлеченными и не всегда понятными для новичков в программировании вопросами по созданию классов и объектов, используем следующий прием. Постулируем некоторые базовые синтаксические конструкции как основу создания программы в Java, а затем, в главе 4, посвященной классам и объектам, дадим более подробные объяснения по этому поводу, причем в контексте методов объектно-ориентированного программирования (сокращенно *ООП*). Думается, такой подход, с одной стороны, позволит читателю, не знакомому с концепцией ООП, легче и быстрее усваивать новый материал, а затем плавно перейти к созданию реальных объектно-ориентированных программ в Java. С другой стороны, практически не пострадают те, кто знаком с методами ООП (например, программирующие на C++), поскольку представленный далее материал в любом случае важен для понимания принципов программирования в Java.

Простые программы

Отлично, отлично! Простенько, и со вкусом!

Из к/ф «Бриллиантовая рука»

За исключением нескольких последних глав книги, в основном будут рассматриваться консольные программы. Памятуя о том, что лучше один раз увидеть, чем сто раз услышать, рассмотрим достаточно простую программу, выводящую на экран сообщение.

Листинг 1.1. Простая программа

```
class Intro{
public static void main(String[] args){
System.out.println("Мы программируем на Java!");
}
}
```

После компиляции и запуска программы (например, в среде NetBeans) в окне вывода появляется сообщение `Мы программируем на Java!`. Рассмотрим программный код подробнее. Приведенные далее комментарии о этому поводу предназначены в основном для тех, кто никогда не имел дела с таким языком программирования, как C++.

Во-первых, сразу отметим, что фигурными скобками в языке программирования Java (как и C++ и C#) отмечаются блоки программного кода. Программный код размещается между открывающей (символ {) и закрывающей (символ }) фигурными скобками. В данном случае использовано две пары фигурных скобок. Первая, внешняя, пара использована для определения программного кода класса, вторая — для определения метода этого класса.

Как неоднократно отмечалось, для создания даже самой простой программы необходимо описать класс. Описание класса начинается с ключевого слова `class`. После этого следует уникальное имя класса. Непосредственно программный код класса заключается в фигурные скобки. Таким образом, синтаксической конструкцией `class Intro{...}` определяется класс с названием `Intro`.

Программный код класса `Intro` состоит всего из одного метода с названием `main()` (здесь и далее названия методов будут указываться с круглыми скобками после имени, чтобы отличать их от переменных). Название метода стандартное. Дело в том, что выполнение Java-программы начинается с вызова метода с именем `main()`. Другими словами, в методе `main()` представлен код, который выполняется в результате вызова программы. Программа содержит один и только один метод с именем `main()` (исключение составляют апплеты — у них метода `main()` нет). Метод `main()` иногда называют главным методом программы, поскольку во многом именно с этим методом отождествляется сама программа.

Ключевые слова `public`, `static` и `void` перед именем метода `main()` означают буквально следующее: `public` — метод доступен вне класса, `static` — метод статический и для его вызова нет необходимости создавать экземпляр класса (то есть объект), `void` — метод не возвращает результат. Уровни доступа членов класса, в том числе открытый (`public`) доступ, детально описываются в главе 6, посвященной наследованию. Статические (`static`) члены класса и особенности работы с ними описываются в главе 4, посвященной созданию классов и объектов. Пояснения по поводу типа результата, возвращаемого методами (в том числе методом `main()`), даются в той же главе.

Инструкция `String[] args` в круглых скобках после имени метода `main()` означает тип аргумента метода: формальное название аргумента `args`, и этот аргумент является текстовым массивом (тип `String`). Типу `String` посвящена

отдельная глава книги (см. главу 8). Массивы описываются в следующей главе. Желающие побольше узнать о способах передачи аргументов методам могут обратиться к главе 4, посвященной созданию классов и объектов. Квадратные скобки можно указывать после ключевого слова `String` или после имени аргумента `args`.

Тем, кто программирует в C++, многое из приведенного уже знакомо. Для тех, кто ничего знакомого во всем этом не увидел, резюмируем: на ближайшее время все наши программы будут иметь следующую структуру:

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

Название класса (параметр `имя_класса`) задается пользователем — это, фактически, название программы. В месте, где указан параметр `программный_код`, указывается непосредственно выполняемый при вызове программы программный код.

В рассматриваемом примере программный код состоит всего из одной команды `System.out.println("Мы программируем на Java!")`. Команда заканчивается точкой с запятой — это стандарт для Java. Командой с помощью встроенного метода `println()` на консоль (по умолчанию консолью является экран) выводится сообщение `"Мы программируем на Java!"`. Текст сообщения указан аргументом метода. Метод вызывается через поле-объект `out` объекта потока стандартного вывода `System`. Подробнее система ввода-вывода обсуждается во второй части книги, после того как мы подробнее познакомимся с классами и объектами. Пока же следует запомнить, что для вывода информации на экран в консольных приложениях используется инструкция вида `System.out.println()`, где в круглых скобках указывается выводимый текст, числовые значения, имена переменных и т. д. — все то, что можно указывать аргументом метода `println()`, и каковы будут последствия, описано в главе 8, посвященной работе с объектами класса `String` и `StringBuffer`.

Если читатель испытывает трудности с компиляцией и запуском программы из листинга 1.1, рекомендуем ему обратиться к приложению в конце книги, посвященному методам практического использования среды разработки NetBeans.

Комментарии

Это мелочи. Но нет ничего важнее мелочей!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Очень часто в программном коде необходимо оставить комментарий — некоторые пояснения, напоминания или просто текст, предназначенный для

пользователя. Важно отметить, что комментарий предназначен не для компилятора, поэтому компилятором он игнорируется. В Java существует три типа комментариев.

1. Однострочный комментарий. Такой комментарий начинается с двойной косой черты (символ //). Все, что находится в строке кода справа от двойной косой черты, компилятором игнорируется.
2. Многострочный комментарий. Такой комментарий начинается последовательностью символов /* и заканчивается последовательностью символов */. Все, что находится между символами /* и */, компилятором игнорируется.
3. Многострочный комментарий документационной информации. Начинается последовательностью символов /** и заканчивается последовательностью символов */. Обычно используется для выделения в качестве комментария данных справочного характера.

Не следует недооценивать значения комментариев. Хороший комментарий существенно улучшает читабельность программного кода и позволяет избежать многих неприятностей. К сожалению, большинство программистов учатся этому правилу на своих горьких ошибках.

Простые типы данных и литералы

Все должно быть изложено так просто,
как только возможно, но не проще.

А. Энштейн

Обычно программы пишут для того, чтобы обрабатывать данные. Методы и возможности по обработке данных в значительной степени зависят от типа данных. Язык Java относится к строго типизованным языкам. Это означает, что любая переменная в программе относится к определенному типу данных — одному и только одному. В Java все данные можно разделить на простые и ссылочные. Ссылочные данные реализуются через иерархию классов. Простые данные — это скорее дань традиции. Забегая наперед, отметим, что для простых типов данных существуют ссылочные аналоги.

Разница между простыми и ссылочными типами на практике проявляется при передаче аргументов методам. Простые типы данных передаются по значению, ссылочные — через ссылку. Читателям, знакомым хотя бы с одним из современных языков программирования, эти термины должны быть знакомы. Способы передачи аргументов методам в языке Java подробно обсуждаются в главе 4, посвященной работе с классами и объектами. Пока же заметим, что простые типы данных являются, по сути, базовыми. Именно данные этих типов будут наиболее часто использоваться в первой части книги.

В Java существует четыре группы базовых типов: для работы с целыми числами, для работы с числами в формате с плавающей точкой (действительные числа),

символы и логический тип — таким образом, всего получается восемь базовых типов. Базовые типы Java перечислены в табл. 1.1.

Таблица 1.1. Базовые (простые) типы в Java

Тип данных (название)	Количество битов	Пояснение	Класс-оболочка
`byte`	8	Целые числа в диапазоне от –128 до 127	`Byte`
`short`	16	Целые числа в диапазоне от –32768 до 32767	`Short`
`int`	32	Целые числа в диапазоне от –2147483648 до 2147483647	`Integer`
`long`	64	Целые числа в диапазоне от –9223372036854775808 до 9223372036854775807	`Long`
`float`	32	Действительные числа. По абсолютной величине изменяются в диапазоне от $3{,}4 \times 10^{-38}$ до $3{,}4 \times 10^{38}$	`Float`
`double`	64	Действительные числа двойной точности. По абсолютной величине изменяются в диапазоне от $1{,}7 \times 10^{-308}$ до $1{,}7 \times 10^{308}$	`Double`
`char`	16	Символьный тип для представления символьных значений (букв). Диапазон значений от 0 до 65536 (каждое значение соответствует определенному символу)	`Character`
`boolean`	–	Логический тип данных. Переменная этого типа может принимать два значения: `true` (истина) и `false` (ложь)	`Boolean`

В этой же таблице приведены названия классов-оболочек для базовых типов. Классы-оболочки используются в тех случаях, когда переменную соответствующего типа необходимо рассматривать как объект. Далее изучим каждую группу базовых типов отдельно. В первую очередь стоит обратить внимание на целочисленные типы данных.

В Java существует четыре типа целочисленных данных: `byte`, `short`, `int` и `long`. Отличаются типы количеством битов, выделяемых для записи значения соответствующего типа. Размер в битах увеличивается от 8 для типа `byte` до 32 для типа `long` (с шагом дискретности 8 бит). На практике выбор подходящего типа осуществляется в соответствии с предполагаемым диапазоном изменения значения переменных. Разумеется, для надежности разумно использовать наиболее «широкий» тип данных, однако при этом не следует забывать и о том, что системные ресурсы даже самого производительного компьютера не безграничны.

Для работы с действительными числами используются типы `float` и `double`. С помощью этих типов реализуется формат числа с плавающей точкой. В этом формате действительное число задается посредством двух чисел: мантиссы

и показателя степени. Заданное таким образом число равно произведению мантиссы на десять в соответствующей второму числу степени. Поскольку размер в битах, выделяемый для типа double, в два раза больше размера для данных типа float, тип double называют типом действительных чисел двойной точности. На практике обычно используется тип double.

Поскольку в Java для символьных данных (тип char) выделяется 16 бит, такая широта размаха позволяет охватить практически все имеющиеся и использующиеся на сегодня символы, включая китайские иероглифы. Этот демократизм, свойственный далеко не каждому языку программирования, является следствием курса разработчиков Java на создание универсального языка программирования, ориентированного на работу в Интернете. Символам расширенного 8-разрядного набора ISO-Latin-1 соответствует интервал значений от 0 до 255, а интервал значений от 0 до 127 определяет стандартные символы ASCII.

Что касается логического типа boolean, то переменные этого типа могут принимать всего два значения: true и false. В свете этого обстоятельства говорить о размере (в битах) переменной типа boolean как-то не принято. В действительности ответ на этот вопрос зависит от типа используемой виртуальной Java-машины. Как правило, логические выражения применяются в условных инструкциях при создании точек ветвления программы.

Указать тип переменной недостаточно. Переменной рано или поздно придется присвоить значение. Делается это с помощью литералов. Литерал — это постоянное значение, предназначенное для восприятия человеком, которое не может быть изменено в программе. В рассмотренном ранее примере уже использовался строчный литерал — фраза "Мы программируем на Java!". Читатель, вероятно, не удивится, узнав, что целочисленные литералы вводятся с помощью арабских цифр от 0 до 9. Также вводятся действительные числа. При этом в качестве десятичного разделителя используется точка. Символы вводятся в одинарных кавычках (не путать с текстом, который заключается в двойные кавычки!), а для ввода логических значений указывают ключевые слова true и false.

Что касается непосредственно объявления переменных в Java, то выполняется оно по следующим правилам. В первую очередь при объявлении переменной перед ее именем в обязательном порядке указывается идентификатор типа. Например, инструкцией int n объявляется переменная n целочисленного типа int. Впоследствии этой переменной может быть присвоено значение. В качестве оператора присваивания в Java используется оператор =. Следующими командами объявляется целочисленная переменная, после чего ей присваивается значение 12:

```
int n;
n=12;
```

При этом всю означенную конструкцию из двух команд можно объединить в одну инструкцию вида int n=12. Более того, объявлять и инициализировать можно сразу несколько переменных, которые перечисляются через запятую

после идентификатора типа. Сразу при объявлении переменной допускается присваивать ей начальное значение, как показано далее:

```
long n, m;
int x, y=3, z=5;
char sym='a';
```

В приведенном фрагменте первой инструкцией объявляются две целочисленные переменные типа `long`, после чего следующей командой объявляются три переменных типа `int`, причем для двух из них указано начальное значение. Третьей командой инициализируется символьная переменная `sym` со значением `a` (символы-значения заключаются в одинарные кавычки). Что касается доступности переменных, то она определяется блоком, в котором эта переменная объявлена. Блок, в свою очередь, выделяется парой фигурных скобок (то есть { и }).

Инструкции объявления и инициализации переменных могут размещаться в любом месте программы. Самое главное, чтобы переменная в программе использовалась (вызывалась) после того, как эта переменная инициализирована (ей присвоено значение). Пример программы, в которой применяются переменные разных типов и литералы, приведен в листинге 1.2.

Листинг 1.2. Переменные и литералы

```
class VarDemo{
public static void main(String[] args){
// Инициализация переменных:
byte age=34;
char sex='м';
double weight=103.6;
int height=182;
// Вывод данных:
System.out.println("Персональные данные пользователя:");
System.out.println("Возраст: "+age+" лет");
System.out.println("Пол (м/ж): "+sex+".");
System.out.println("Вес: "+weight+" кг");
System.out.println("Рост: "+height+" см");
}
}
```

Результат выполнения этой программы:

```
Персональные данные пользователя:
Возраст: 34 лет
Пол (м/ж): м.
Вес: 103.6 кг
Рост: 182 см
```

В программе объявлено с одновременной инициализацией несколько переменных разных типов. Переменные предназначены для хранения персональных данных пользователя (таких как возраст, рост, вес и пол). Выражения в правой

части от операторов присваивания (присваиваемые переменным значения) являются примерами литералов.

Числовые литералы, кроме обычного десятичного представления, могут быть записаны в восьмеричной и шестнадцатеричной системах счисления. Восьмеричные литералы начинаются с нуля. Следующие цифры в позиционной записи восьмеричного литерала могут принимать значения в диапазоне от `0` до `7` включительно. Например, восьмеричный литерал `012` означает десятичное число `10`. Шестнадцатеричные литералы начинаются с префикса `0x`. Для позиционного представления шестнадцатеричного числа используются цифры от `0` до `9` и буквы от `A` до `F`. Например, шестнадцатеричный литерал `0x12` означает десятичное число 18.

Наконец, в формате `'\xxx'` задаются восьмеричные символы Unicode, а в формате `'\uxxx'` — шестнадцатеричные (символами `x` обозначены позиции кода).

Приведение типов

Я там столкнулся с одним очень нахальным типом.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Строгая типизация переменных вместе с очевидными преимуществами привносит и ряд не столь очевидных проблем. Поясним это на простом примере. Предположим, что в программе объявлены две числовые переменные: одна типа `int` и другая типа `double`. Переменным присвоены значения. Далее мы хотим к переменной типа `double` прибавить значение переменной типа `int` и результат записать в первую переменную. С формальной точки зрения здесь нет никакой проблемы, поскольку целые числа являются подмножеством множества действительных чисел. С точки зрения программной логики ситуация не такая простая, ведь складываются переменные разных типов. Понятно, что на самом деле здесь проблемы не возникает и описанную операцию можно выполнить (в том числе и в Java), причем возможность выполнения подобного рода операций достижима благодаря автоматическому приведению типов. Другими словами, если нужно вычислить выражение, в которое входят переменные разных типов, автоматически выполняется преобразование входящих в выражение переменных к общему формату. Процесс автоматического преобразования типов подчиняется нескольким базовым правилам. Вот они.

- Типы переменных, входящих в выражение, должны быть совместимыми. Например, целое число можно преобразовать в формат действительного числа, чего не скажешь о текстовой строке.
- Целевой тип (тип, к которому выполняется приведение) должен быть «шире» исходного типа. Другими словами, преобразование должно выполняться без потери данных.

- Перед выполнением арифметической операции типы byte, short и char расширяются до типа int.
- Если в выражении есть операнды типа long, то расширение осуществляется до типа long.
- Если в выражении есть операнды типа float, то расширение осуществляется до типа float.
- Если в выражении есть операнды типа double, то расширение осуществляется до типа double.

К этим правилам следует добавить не менее важные правила интерпретации литералов. Действительно, как следует рассматривать, например, число (литерал) 2? Как значение типа int, типа long или, например, типа double? Следующие правила дают ответы на подобные вопросы.

- Литералы, обозначающие целые числа, интерпретируются как значения типа int.
- Литералы, обозначающие действительные числа, интерпретируются как значения типа double.

Хотя эти правила представляются логичными и простыми, нередко автоматическое приведение типов приводит к непредсказуемым, на первый взгляд, результатам и ошибкам там, где их быть в принципе не должно. Например, следующая последовательность команд приводит к ошибке:

```
byte a=1,b=2,c;
// Ошибка:
c=a+b;
```

Ошибку вызывает последняя команда. Хотя все три переменные относятся к типу byte, при вычислении выражения a+b выполняется автоматическое преобразование к типу int. В результате имеет место попытка присвоить значение типа int переменной типа byte. Поскольку в Java преобразования с возможной потерей точности не допускаются, программа с таким кодом не скомпилируется.

Еще один пример ошибки, связанной с автоматическим преобразованием типов:

```
float x=2.7;
```

В данном случае проблема связана с тем, что литерал 2.7, использованный для инициализации переменной x типа float, интерпретируется как значение типа double.

Для обработок ошибок подобного рода, а также для ряда других целей в Java предусмотрено явное приведение типов и явное определение типа литерала с помощью суффиксов типа.

Для приведения выражения к нужному типу перед этим выражением указывается имя типа, заключенное в круглые скобки. Например, следующий код является корректным:

```
byte a=1,b=2,c;
// Нет ошибки - явное приведение типа:
c=(byte)(a+b);
```

Командой `(byte)(a+b)` вычисляется сумма значений переменных `a` и `b`, а результат преобразуется к типу `byte`. Поскольку в правой части от оператора присваивания стоит переменная того же типа, проблем не возникает. Тем не менее следует понимать, что явное приведение типа потенциально опасно, поскольку может приводить к потере значения. Такие ситуации должен отслеживать программист — системой они не отслеживаются.

Аналогичную процедуру можно применять и к литералам. Кроме того, изменять тип литералов можно с помощью суффиксов. Так, суффикс `L` у целочисленного литерала (например, `123L`) означает, что он принадлежит к типу `long`, а суффикс `F` у литерала, обозначающего действительное число (например, `12.5F`), означает, что этот литерал относится к типу `float`. В свете сказанного корректными являются такие команды:

```
float x=2.7F;
float x=(float)2.7;
```

Кроме прочего, явное приведение типов часто используется вместе с оператором деления.

В Java, как и в C++, допускается динамическая инициализация переменных. При динамической инициализации значение переменной присваивается при объявлении, причем значением является выражение, содержащее другие переменные. Пример динамической инициализации переменной:

```
int a=3,b=4;
int c=a*a+b*b;
```

В данном случае переменная `c` инициализируется выражением `a*a+b*b`, то есть получает значение 25. Главное и единственное условие для динамической инициализации — все переменные, входящие в соответствующее выражение, должны быть предварительно объявлены и им должны быть присвоены значения.

Основные операторы Java

> Мне кажется, давно уже пора приступить к разработке документа, в котором будет четко оговорено, что граждане могут делать в свое свободное время, а чего они делать не должны.
>
> *Из к/ф «Забытая мелодия для флейты»*

Все операторы Java можно разделить на четыре группы: арифметические, логические, побитовые и сравнения. Рассмотрим последовательно каждую группу операторов. Начнем с арифметических. Эти операторы перечислены в табл. 1.2.

Таблица 1.2. Арифметические операторы Java

Оператор	Название	Пояснение
+	Сложение	Бинарный оператор. Результатом команды a+b является сумма значений переменных a и b
-	Вычитание	Бинарный оператор. Результатом команды a-b является разность значений переменных a и b
*	Умножение	Бинарный оператор. Результатом команды a*b является произведение значений переменных a и b
/	Деление	Бинарный оператор. Результатом команды a/b является частное от деления значений переменных a и b. Для целочисленных операндов по умолчанию выполняется деление нацело
%	Остаток	Бинарный оператор. Результатом команды a%b является остаток от целочисленного деления значений переменных a и b
+=	Сложение (упрощенная форма с присваиванием)	Упрощенная форма оператора сложения с присваиванием. Команда a+=b является эквивалентом команды a=a+b
-=	Вычитание (упрощенная форма с присваиванием)	Упрощенная форма оператора вычитания с присваиванием. Команда a-=b является эквивалентом команды a=a-b
*=	Умножение (упрощенная форма с присваиванием)	Упрощенная форма оператора умножения с присваиванием. Команда a*=b является эквивалентом команды a=a*b
/=	Деление (упрощенная форма с присваиванием)	Упрощенная форма оператора деления с присваиванием. Команда a/=b является эквивалентом команды a=a/b
%=	Остаток (упрощенная форма)	Упрощенная форма оператора вычисления остатка с присваиванием. Команда a%=b является эквивалентом команды a=a%b
++	Инкремент	Унарный оператор. Команда a++ (или ++a) является эквивалентом команды a=a+1
--	Декремент	Унарный оператор. Команда a-- (или --a) является эквивалентом команды a=a-1

Эти операторы имеют некоторые особенности. В первую очередь обращаем внимание на оператор деления /. Если операндами являются целые числа, в качестве значения возвращается результат целочисленного деления. Рассмотрим последовательность команд:

```
int a=5,b=2;
double x=a/b;
```

В данном примере переменная x получает значение 2.0, а не 2.5, как можно было бы ожидать. Дело в том, что сначала вычисляется выражение a/b. Поскольку

операнды целочисленные, выполняется целочисленное деление. И только после этого полученное значение преобразуется к формату double и присваивается переменной x.

Для того чтобы при целочисленных операндах выполнялось обычное деление, перед выражением с оператором деления указывается в круглых скобках идентификатор типа double (или float). Например, так:

```
int a=5,b=2;
double x=(double)a/b;
```

Теперь значение переменной x равно 2.5.

В Java, как и в C++, есть группа упрощенных арифметических операторов с присваиванием. Если op — один из операторов сложения, умножения, деления и вычисления остатка, то упрощенная форма этого оператора с присваиванием имеет вид op=. Это тоже бинарный оператор, как и оператор op, а команда вида x op=y является эквивалентом команды x=x op y.

Еще два исключительно полезных унарных оператора — операторы инкремента (++) и декремента (--). Действие оператора декремента сводится к увеличению на единицу значения операнда, а оператор декремента на единицу уменьшает операнд. Другими словами, команда x++ эквивалентна команде x=x+1, а команда x-- эквивалентна команде x=x-1. У операторов инкремента и декремента есть не только представленная здесь постфиксная форма (оператор следует после операнда: x++ или x--), но и префиксная (оператор располагается перед операндом: ++x или --x). С точки зрения действия на операнд нет разницы в том, префиксная или постфиксная формы оператора использованы. Однако если выражение с оператором инкремента или декремента является частью более сложного выражения, различие в префиксной и постфиксной формах операторов инкремента и декремента существует. Если использована префиксная форма оператора, сначала изменяется значение операнда, а уже после этого вычисляется выражение. Если использована постфиксная форма оператора, сначала вычисляется выражение, а затем изменяется значение операнда. Рассмотрим небольшой пример:

```
int n,m;
n=10;
m=n++;
```

В этом случае после выполнения команд переменная n будет иметь значение 11, а переменная m — значение 10. На момент выполнения команды m=n++ значение переменной n равно 10. Поскольку в команде m=n++ использована постфиксная форма оператора инкремента, то сначала выполняется присваивание значения переменной m, а после этого значение переменной n увеличивается на единицу.

Иной результат выполнения следующих команд:

```
int n,m;
n=10;
m=++n;
```

Обе переменные (n и m) в этом случае имеют значение 11. Поскольку в команде m=++n использована префиксная форма инкремента, сначала на единицу увеличивается значение переменной n, а после этого значение переменной n присваивается переменной m.

Следующую группу образуют логические операторы. Операндами логических операторов являются переменные и литералы типа boolean. Логические операторы Java перечислены в табл. 1.3.

Таблица 1.3. Логические операторы Java

Оператор	Название	Пояснение
&	Логическое И	Бинарный оператор. Результатом операции A&B является true, если значения обоих операндов равны true. В противном случае возвращается значение false
&&	Сокращенное логическое И	Бинарный оператор. Особенность оператора, по сравнению с оператором &, состоит в том, что если значение первого операнда равно false, то значение второго операнда не проверяется
|	Логическое ИЛИ	Бинарный оператор. Результатом операции A|B является true, если значение хотя бы одного операнда равно true. В противном случае возвращается значение false
||	Сокращенное логическое ИЛИ	Бинарный оператор. Особенность оператора, по сравнению с оператором |, состоит в том, что если значение первого операнда равно true, то значение второго операнда не проверяется
^	Исключающее ИЛИ	Бинарный оператор. Результатом операции A^B является true, если значение одного и только одного операнда равно true. В противном случае возвращается значение false
!	Логическое отрицание	Унарный оператор. Результатом команды !A является true, если значение операнда A равно false. Если значение операнда A равно true, результатом команды !A является значение false

Логические операторы обычно используются в качестве условий в условных операторах и операторах цикла.

В табл. 1.4 перечислены операторы сравнения, используемые в Java.

Таблица 1.4. Операторы сравнения Java

Оператор	Название	Пояснение
==	Равно	Результатом операции A==B является значения true, если операнды A и B имеют одинаковые значения. В противном случае значением является false

Оператор	Название	Пояснение
<	Меньше	Результатом операции A<B является значения true, если значение операнда A меньше значения операнда B. В противном случае значением является false
<=	Меньше или равно	Результатом операции A<=B является значения true, если значение операнда A не больше значения операнда B. В противном случае значением является false
>	Больше	Результатом операции A>B является значения true, если значение операнда A больше значения операнда B. В противном случае значением является false
>=	Больше или равно	Результатом операции A>=B является значения true, если значение операнда A не меньше значения операнда B. В противном случае значением является false
!=	Не равно	Результатом операции A!=B является значения true, если операнды A и B имеют разные значения. В противном случае значением является false

Операторы сравнения обычно используются совместно с логическими операторами.

Для понимания принципов работы поразрядных операторов необходимо иметь хотя бы элементарные познания о двоичном представлении чисел. Напомним читателю некоторые основные моменты.

- В двоичном представлении позиционная запись числа содержит нули и единицы.
- Старший бит (самый первый слева) определяет знак числа. Для положительных чисел старший бит равен нулю, для отрицательных — единице.
- Перевод из двоичной системы счисления положительного числа с позиционной записью $\overline{b_n b_{n-1} ... b_2 b_1 b_0}$ (b_i могут принимать значения 0 или 1, старший бит для положительных чисел $b_n = 0$) в десятичную выполняется так:

$$b_0 2^0 + b_1 2^1 + b_2 2^2 + ... + b_{n-1} 2^{n-1} + b_n 2^n .$$

- Для перевода отрицательного двоичного числа в десятичное представление производится побитовое инвертирование кода (об операции побитового инвертирования — см. далее), полученное двоичное число переводится в десятичную систему, к нему прибавляется единица (и добавляется знак минус).
- Для перевода отрицательного числа из десятичной в двоичную систему от модуля числа отнимают единицу, результат переводят в бинарный код и затем этот код инвертируют.
- Умножение числа на два эквивалентно сдвигу влево на один бит позиционной записи числа (с заполнением первого бита нулем).

Побитовые операторы Java описаны в табл. 1.5.

Таблица 1.5. Побитовые операторы Java

Оператор	Название	Пояснение
&	Побитовое И	Бинарный оператор. Логическая операция И применяется к каждой паре битов операндов. Результатом является 1, если каждый из двух сравниваемых битов равен 1. В противном случае результат равен 0
|	Побитовое ИЛИ	Бинарный оператор. Логическая операция ИЛИ применяется к каждой паре битов операндов. Результатом является 1, если хотя бы один из двух сравниваемых битов равен 1. В противном случае результат равен 0
^	Побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ	Бинарный оператор. Логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ применяется к каждой паре битов операндов. Результатом является 1, если один и только один из двух сравниваемых битов равен 1. В противном случае результат равен 0
~	Побитовое отрицание	Унарный оператор. Выполняется инверсия двоичного кода: 0 меняется на 1, а 1 меняется на 0
>>	Сдвиг вправо	Бинарный оператор. Результатом является число, получаемое сдвигом вправо в позиционном представлении первого операнда (слева от оператора) на количество битов, определяемых вторым операндом (справа от оператора). Исходное значение первого операнда при этом не меняется. Младшие биты теряются, а старшие заполняются дублированием знакового бита
<<	Сдвиг влево	Бинарный оператор. Результатом является число, получаемое сдвигом влево в позиционном представлении первого операнда (слева от оператора) на количество битов, определяемых вторым операндом (справа от оператора). Исходное значение первого операнда при этом не меняется. Младшие биты заполняются нулями, а старшие теряются
>>>	Беззнаковый сдвиг вправо	Бинарный оператор. Результатом является число, получаемое сдвигом вправо в позиционном представлении первого операнда (слева от оператора) на количество битов, определяемых вторым операндом (справа от оператора). Исходное значение первого операнда при этом не меняется. Младшие биты теряются, а старшие заполняются нулями
&=	Упрощенная форма побитового оператора & с присваиванием	Команда вида `A&=B` является эквивалентом команды `A=A&B`
|=	Упрощенная форма побитового оператора | с присваиванием	Команда вида `A|=B` является эквивалентом команды `A=A|B`

Оператор	Название	Пояснение
^=	Упрощенная форма побитового оператора ^ с присваиванием	Команда вида A^=B является эквивалентом команды A=A^B
>>=	Упрощенная форма побитового оператора >> с присваиванием	Команда вида A>>=B является эквивалентом команды A=A>>B
<<=	Упрощенная форма побитового оператора << с присваиванием	Команда вида A<<=B является эквивалентом команды A=A<<B
>>>=	Упрощенная форма побитового оператора >>> с присваиванием	Команда вида A>>>=B является эквивалентом команды A=A>>>B

За редким исключением, побитовые операции используются в случаях, когда необходимо оптимизировать программу в отношении быстродействия.

Помимо перечисленных операторов, в Java есть единственный тернарный оператор (у оператора три операнда). Формально оператор обозначается как ?:. Синтаксис вызова этого оператора следующий:

```
условие?значение_1:значение_2
```

Первым операндом указывается условие — выражение, возвращающее в качестве значения логическое значение. Если значение выражения-условия равно true, в качестве значения тернарным оператором возвращается значение_1. Если значением выражения-условия является false, тернарным оператором в качестве значения возвращается значение_2.

Несколько замечаний по поводу оператора присваивания (оператор =). В Java оператор присваивания возвращает значение. Команда вида x=y выполняется следующим образом. Сначала вычисляется выражение y, после чего это выражение приводится к типу переменной x и затем записывается в эту переменную. Благодаря тому, что, в отличие от других операторов с равными приоритетами, присваивание выполняется справа налево, в Java допустимыми являются команды вида x=y=z. В этом случае значение переменной z присваивается сначала переменной y, а затем значение переменной y присваивается переменной x.

Еще одно замечание касается упрощенных форм операторов с присваиванием, то есть операторов вида op=. Хотя утверждалось, что команда вида A op=B эквивалента команде A=A op B, это не совсем так. При выполнении команды вида A op=B сначала вычисляется выражение A op B, затем полученное значение приводится к типу переменной A и только после этого присваивается переменной A. Поскольку приведение к типу переменной A выполняется, фактически, явно, а в команде A=A op B приведение типов неявное, может проявиться разница

в использовании полной и упрощенной форм команд присваивания. Рассмотрим простой пример:

```
byte a=10,b=20;
// Правильно:
a+=20;
// Неправильно:
a=a+b;
```

В данном случае команда `a+=20` является корректной, а команда `a=a+b` — нет. В первом случае литерал `20` типа `int` «насильственно» приводится к типу `byte` в силу особенностей оператора `+=`. Во втором случае результат вычисления выражения `a+b` автоматически расширяется до типа `int`, а автоматическое приведение типа `int` к типу `byte` запрещено.

Напоследок приведем в табл. 1.6 данные о приоритете различных операторов в Java.

Таблица 1.6. Приоритеты операторов в Java

Приоритет	Операторы
1	Круглые скобки `( )`, квадратные скобки `[ ]` и оператор «точка»
2	Инкремент `++`, декремент `--`, отрицания `~` и `!`
3	Умножение `*`, деление `/` и вычисление остатка `%`
4	Сложение `+` и вычитание `-`
5	Побитовые сдвиги `>>`, `<<` и `>>>`
6	Больше `>`, больше или равно `>=`, меньше или равно `<=` и меньше `<`
7	Равно `==` и неравно `!=`
8	Побитовое И `&`
9	Побитовое исключающее ИЛИ `^`
10	Побитовое ИЛИ `|`
11	Логическое И `&&`
12	Логические ИЛИ `||`
13	Тернарный оператор `?:`
14	Присваивание `=` и сокращенные формы операторов вида `op=`

Операторы равных приоритетов (за исключением присваивания) выполняются слева направо. В случаях когда возникают сомнения в приоритете операторов и последовательности вычисления выражений, рекомендуется использовать круглые скобки.

Примеры программ

Далее рассмотрим некоторые задачи, которые иллюстрируют возможности Java и специфику синтаксиса этого языка.

Полет брошенного под углом к горизонту тела

Составим программу для вычисления в указанный момент времени координат тела, брошенного под углом к горизонту. Полагаем, что известны масса тела m, начальная скорость V, угол a, под которым тело брошено к горизонту. Кроме того, считаем, что на тело действует сила сопротивления воздуха, по модулю пропорциональная скорости тела и направленная противоположно к направлению полета тела. Коэффициент пропорциональности для силы сопротивления воздуха g также считаем известным.

В программе используем известное аналитическое решение для зависимостей координат тела от времени. В частности, для горизонтальной координаты (расстояние от точки бросания до тела вдоль горизонтали) имеем зависимость:

$$x(t) = \frac{Vm\cos(\alpha)}{\gamma}(1 - \exp(-\gamma t/m)).$$

Аналогично для вертикальной координаты (высота тела над горизонтальной поверхностью) имеем зависимость:

$$y(t) = \frac{m(V\sin(\alpha)\gamma + mg)}{\gamma^2}(1 - \exp(-\gamma t/m)) - \frac{mgt}{\gamma}.$$

Здесь через g обозначено ускорение свободного падения. Этими соотношениями воспользуемся при создании программы. Программный код приведен в листинге 1.3.

Листинг 1.3. Вычисление координат тела

```
class BodyPosition{
public static void main(String args[]){
// Ускорение свободного падения:
double g=9.8;
// Число "пи":
double pi=3.141592;
// Угол к горизонту (в градусах):
double alpha=30;
// Масса тела (в килограммах):
double m=0.1;
// Коэффициент сопротивления воздуха (в Н*с/м):
double gamma=0.1;
// Скорость тела (в м/с):
double V=100.0;
// Время (в секундах):
double t=1.0;
// Координаты тела (в метрах):
double x,y;
// Перевод градусов в радианы:
alpha/=180/pi;
```

продолжение

Листинг 1.3 *(продолжение)*

```
// Вычисление координат:
x=V*m*Math.cos(alpha)/gamma*(1-Math.exp(-gamma*t/m));
y=m*(V*Math.sin(alpha)*gamma+m*gamma)/gamma/gamma*(1-Math.exp(-gamma*t/m))-m*g*t/
gamma;
// Вывод информации на экран:
System.out.println("Координаты тела для t="+t+" сек:\nx="+x+" м\ny="+y+" м");
System.out.println("Параметры:");
System.out.println("Угол alpha="+alpha/pi*180+" градусов");
System.out.println("Скорость V="+V+" м/с");
System.out.println("Коэффициент сопротивления gamma="+gamma+" Н*с/м");
System.out.println("Масса тела m="+m+" кг");
}}
```

В результате выполнения программы получаем последовательность сообщений:

```
Координаты тела для t=1.0 сек:
x=54.743249662890555 м
y=21.86923403403938 м
Параметры:
Угол alpha=30.0 градусов
Скорость V=100.0 м/с
Коэффициент сопротивления gamma=0.1 Н*с/м
Масса тела m=0.1 кг
```

При расчете параметров программы использовались математические функции для вычисления синуса, косинуса и экспоненты. Функции статические и описаны они в классе `Math`. Способ вызова статических функций в Java подразумевает указание класса, в котором они описаны, и, через точку, имя самой функции. Например, ссылка на функцию вычисления косинуса имеет вид `Math.cos()`. Аналогично, синус и экспонента вычисляются функциями `Math.sin()` и `Math.exp()` соответственно. Подробнее функции (методы), в том числе статические, описываются в следующих главах книги.

Сам программный код достаточно прост: объявляется несколько переменных, которым при объявлении сразу присваиваются значения (ускорение свободного падения `g`, начальная скорость `V`, угол в градусах `alpha`, под которым брошено тело, коэффициент сопротивления `gamma`, а также масса тела `m`). Кроме того, значение присваивается переменной `t`, определяющей момент времени, для которого вычисляются координаты тела. Переменные `x` и `y` предназначены для записи в них значений координат тела. После присваивания этим переменным значения результаты вычислений выводятся на экран вместе с дополнительной информацией о массе тела, начальной скорости и т. п.

Вычисление скорости на маршруте

Составим программу для вычисления скорости движения автомобиля на маршруте, если известно, что автомобиль движется с постоянной известной скоростью между пунктами *А* и *Б*, расстояние между которыми тоже известно. Далее

автомобиль движется от пункта *Б* до пункта *В* (расстояние между пунктами известно) с постоянной, но неизвестной скоростью. Ее необходимо вычислить, если известна средняя скорость движения автомобиля на маршруте от пункта *А* до пункта *В* (через пункт *Б*).

Если расстояние между пунктами *А* и *Б* обозначить через S_1, расстояние между пунктами *Б* и *В* — через S_2, скорость движения на этих участках — соответственно через V_1 и V_2, среднюю скорость движения на маршруте — через V, то неизвестную скорость V_2 движения на маршруте от *Б* до *В* можно вычислить по формуле:

$$V_2 = \frac{S_2}{(S_1 + S_2)/V - S_1/V_1}.$$

Проблема, однако, в том, что вычисленное по данной формуле значение для скорости может оказаться отрицательным. Это означает, на самом деле, невозможность для автомобиля иметь указанную среднюю скорость. Другими словами, даже если бы автомобиль мгновенно переместился из пункта *Б* в пункт *В*, он настолько медленно проехал первый участок, что средняя скорость никак не может оказаться равной указанному значению. Эту возможность учтем при составлении программного кода.

Некоторые замечания касаются самого процесса вычисления скорости. Удобнее пользоваться не сразу готовой функцией, а разбить процесс на несколько этапов. В частности, разумно предварительно вычислить время движения автомобиля по всему маршруту $T = (S_1 + S_2)/V$, а также время движения по первому участку $t = S_1/V_1$. Затем искомую скорость можно рассчитать по формуле:

$$V_2 = \frac{S_2}{T - t}.$$

В листинге 1.4 приведен программный код для вычисления скорости движения автомобиля.

Листинг 1.4. Вычисление скорости автомобиля

```
class FindV{
public static void main(String args[]){
// Расстояние между объектами (км):
double S1=100;
double S2=200;
// Скорость на первом участке (км/ч):
double V1=80;
// Средняя скорость (км/ч):
double V=48;
/* Скорость на втором участке, общее время движения
 и время движения на первом участке:*/
double V2,T,t;
// Общее время движения (час):
T=(S1+S2)/V;
```

продолжение

Листинг 1.4 *(продолжение)*

```
// Время движения на первом участке (час):
t=S1/V1;
// Скорость движения на втором участке (км/ч):
V2=T>t?(S1+S2)/(T-t):-1;
System.out.println("Скорость на втором участке:");
// Результат:
System.out.println(V2<0?"Это невозможно!":V2+" км/ч");}
}
```

Результат выполнения программы имеет вид:

```
Скорость на втором участке:
60.0 км/ч
```

Если изменить значение средней скорости (переменная `V`) на 240 или больше (при неизменных прочих параметрах), получим сообщение:

```
Скорость на втором участке:
Это невозможно!
```

Значение скорости на втором участке в программе определяется с помощью тернарного оператора командой:

```
V2=T>t?(S1+S2)/(T-t):-1
```

Тернарный оператор здесь необходим исключительно с одной целью: предотвратить возможное деление на нуль при условии, что значения переменных `T` и `t` совпадают. Если общее время движения превышает время движения по первому участку, значение скорости автомобиля на втором участке вычисляется по приведенной формуле. Если данное условие не выполняется, переменной `V2` для скорости на втором участке присваивается формальное отрицательное значение `-1`.

При выводе результата отображаются два сообщения. Первое содержит формальное сообщение о том, что вычислено значение скорости на втором участке. Второе сообщение, в зависимости от значения переменной `V2`, либо содержит информацию о фактическом значении скорости на втором участке, либо представляет собой сообщение `"Это невозможно!"`.

Второе сообщение выводится следующей командой:

```
System.out.println(V2<0?"Это невозможно!":V2+" км/ч")
```

Аргументом метода `println()` указано выражение `V2<0?"Это невозможно!":V2+" км/ч"`, в котором также использован тернарный оператор. При отрицательном значении переменной `V2` возвращается текстовое значение `"Это невозможно!"`, в противном случае возвращается текстовое значение, которое получается объединением (и преобразованием к текстовому формату) значения скорости и надписи `"км/ч"`.

Орбита спутника

Следующая задача иллюстрирует работу с большими числами. Состоит она в вычислении высоты орбиты спутника над поверхностью Земли, если известны

масса и радиус Земли, а также период обращения спутника вокруг Земли. В частности, используем значения универсальной гравитационной постоянной, массы Земли и радиуса Земли.

- Универсальная гравитационная постоянная:

$G \approx 6,672 \times 10^{-11}$ Нм^2/кг^2.

- Масса Земли:

$M \approx 5,96 \times 10^{24}$ кг.

- Радиус Земли:

$R = 6,37 \times 10^{6}$ м.

Если через T обозначить период обращения спутника (в секундах), то высоту H спутника над поверхностью Земли можно вычислить по формуле:

$$H = \sqrt[3]{\frac{GMT^2}{4\pi^2}} - R.$$

Соответствующий программный код приведен в листинге 1.5.

Листинг 1.5. Орбита спутника

```
class FindH{
public static void main(String args[]){
// Гравитационная постоянная (Нм^2/кг^2):
double G=6.672E-11;
// Масса Земли (кг):
double M=5.96e24;
// Радиус Земли:
double R=6.37E6;
// Период обращения спутника (часы):
double T=1.5;
// Высота над поверхностью:
double H;
// Перевод в секунды:
T*=3600;
// Высота в метрах:
H=Math.pow(G*M*T*T/4/Math.PI/Math.PI,(double)1/3)-R;
// Высота в километрах с точностью до тысячных:
H=(double)(Math.round(H))/1000;
// Вывод результата на экран:
System.out.println("Высота орбиты спутника: "+H+" км");}
}
```

В результате выполнения программы получаем сообщение:

```
Высота орбиты спутника: 277.271 км
```

При инициализации переменных, определяющих параметры Земли и значение гравитационной постоянной, используется формат представления чисел в виде мантиссы и после литеры `E` (или `e`) значения показателя степени десятки. Поскольку время периода обращения спутника (переменная `T`) задается в часах, для перевода в секунды используем команду `T*=3600`. Высота вычисляется с помощью команды:

```
H=Math.pow(G*M*T*T/4/Math.PI/Math.PI,(double)1/3)-R
```

В этой команде использована математическая функция `pow()` для возведения числа в степень. Первым аргументом указывается возводимое в степень число, вторым — показатель степени. При вызове функции `pow()` явно указывается класс `Math`, в котором описана функция. Также использована константа `PI` (полная ссылка на константу имеет вид `Math.PI`) для числа π. Кроме того, при вычислении второго аргумента-показателя степени делятся два целых числа, а по умолчанию такое деление выполняется нацело. Чтобы деление выполнялось «как надо», использована инструкция `(double)`.

После вычисления значения переменной `H` получаем значение высоты орбиты в метрах. Затем с помощью функции `Math.round()` это значение округляем и делим на 1000 для вычисления значения высоты орбиты в километрах. Поскольку функцией `Math.round()` возвращается целое число, при делении результата вызова этой функции на 1000 по умолчанию также выполняется деление нацело. В силу этой причины перед выражением указана инструкция `(double)`, в результате чего значение переменной `H` получаем в километрах с точностью до сотых, то есть точность орбиты вычисляется с точностью до метра.

Комплексные числа

Рассмотрим программу, в которой вычисляется целочисленная степень комплексного числа. Напомним, что комплексным называется число в виде $z = x + iy$, где x и y — действительные числа, а мнимая единица $i^2 = -1$. Величина $\mathrm{Re}(z) = x$ называется действительной частью комплексного числа, а величина $\mathrm{Im}(z) = y$ — мнимой. Модулем комплексного числа называется действительная величина $r = \sqrt{x^2 + y^2}$. Каждое комплексное число может быть представлено в тригонометрическом виде $z = r\exp(ij) = r\cos(j) + ir\sin(j)$, где модуль комплексного числа r и аргумент j связаны с действительной x и мнимой y частями комплексного числа соотношениями $x = r\cos(j)$ и $y = r\sin(j)$.

Если комплексное число $z = x + iy$ необходимо возвести в целочисленную степень n, результатом является комплексное число $zn = rn\exp(inj) = rn\cos(nj) + irn\sin(nj)$. Этим соотношением воспользуемся в программе для вычисления целочисленной степени комплексного числа. Программный код приведен в листинге 1.6.

Листинг 1.6. Возведение комплексного числа в степень

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

```
double x=1.0,y=-1.0;
int n=5;
double r,phi;
double Re,Im;
r=Math.sqrt(x*x+y*y);
phi=Math.atan2(y,x);
Re=Math.pow(r,n)*Math.cos(n*phi);
Im=Math.pow(r,n)*Math.sin(n*phi);
System.out.println("Re="+Re);
System.out.println("Im="+Im);}
}
```

В программе на основании действительной и мнимой частей исходного комплексного числа вычисляются модуль и аргумент этого числа. На основании полученных значений вычисляются действительная и мнимая части комплексного числа, возведенного в целочисленную степень.

Действительные переменные `x` и `y` определяют действительную и мнимую части исходного комплексного числа. Целочисленная переменная `n` содержит значение степени, в которую возводится комплексное число. В переменные `Re` и `Im` записываются соответственно действительная и мнимая части комплексного числа-результата возведения в степень.

Переменные `r` и `phi` типа `double` предназначены для записи в них модуля и аргумента комплексного числа. Для вычисления модуля используется функция вычисления квадратного корня `Math.sqrt()`.

Аргумент комплексного числа вычисляется с помощью функции `Math.atan2()`. Аргументом функции `atan2()` указываются ордината и орта точки, а в качестве результата возвращается полярный угол направления на эту точку. Для комплексного числа это означает, что результатом вызова функции, если первым аргументом указать мнимую часть, а вторым действительную, является его аргумент.

Возведение в целочисленную степень выполняется с помощью функции `Math.pow()`. Первым аргументом функции указывается возводимое в целочисленную степень число, вторым аргументом — степень, в которую возводится число.

После выполнения всех необходимых расчетов действительная и мнимая части комплексного числа-результата возведения в степень выводятся на экран. В результате выполнения программы получаем:

```
Re=-4.000000000000003
Im=4.000000000000001
```

Справедливости ради следует отметить, что работу с комплексными числами все же лучше реализовывать с помощью классов и объектов.

Прыгающий мячик

Рассмотрим такую задачу. Тело (упругий мячик) бросают под углом к горизонту с некоторой начальной скоростью. При падении мячика на ровную горизонтальную поверхность происходит упругое отбивание, так что горизонтальная

составляющая скорости мячика не меняется, а вертикальная меняется на противоположную. Необходимо написать программу, которая бы вычисляла положение (координаты) мячика в произвольный момент времени.

При составлении программы воспользуемся тем, что если в начальный момент времени (то есть при $t = 0$) скорость мячика по модулю равна V, а угол к горизонту составляет a, то закон движения для горизонтальной координаты имеет вид:

$$x(t) = tV\ cos(a).$$

Для вертикальной координаты соответствующая зависимость, исходя из постановки задачи, может быть записана так:

$$y(t) = (t - T)V\sin(\alpha) - g(t - T)^2/2.$$

Здесь через T обозначено время последнего на данный момент удара о землю. Поскольку время между ударами может быть определено как $T_0 = 2V\sin(\alpha)/g$, то:

$$T = T_0\left[\frac{t}{T_0}\right].$$

В данном случае квадратные скобки означают вычисление целой части от внутреннего выражения. Соответствующий программный код приведен в листинге 1.7.

Листинг 1.7. Полет брошенного под углом к горизонту тела

```
class FindCoords{
public static void main(String args[]){
// Ускорение свободного падения, м/с^2:
double g=9.8;
// Начальная скорость, м/с:
double V=10;
// Угол в градусах:
double alpha=30;
// Время в секундах:
double t=5;
// Расчетные параметры:
double T0,T,x,y;
// Перевод угла в радианы
alpha=Math.toRadians(alpha);
// Время полета до удара о поверхность:
T0=2*V*Math.sin(alpha)/g;
// Момент последнего удара о поверхность:
T=T0*Math.floor(t/T0);
// Горизонтальная координата:
x=V*Math.cos(alpha)*t;
```

```
// Высота над поверхностью:
y=V*Math.sin(alpha)*(t-T)-g*(t-T)*(t-T)/2;
// Округление значений:
x=Math.round(100*x)/100.0;
y=Math.round(100*y)/100.0;
// Вывод результатов на экран:
System.out.println("x("+t+")="+x+" м");
System.out.println("y("+t+")="+y+" м");}
}
```

В результате выполнения программы получаем следующее:

```
x(5.0)=43.3 м
y(5.0)=0.46 м
```

В начале программы задаются значения ускорения свободного падения (переменная `g`), начальная скорость мячика (переменная `V`), угол в градусах, под которым тело брошено к горизонту (переменная `alpha`), и момент времени, для которого вычисляются координаты положения мячика (переменная `t`). Переменные `T0` и `T` используются для записи в них значений времени полета мячика между ударами о поверхность и времени последнего удара соответственно. В переменные `x` и `y` записываются значения координат мячика в данный момент времени (эти значения и нужно вычислить в программе).

Поскольку угол задан в градусах, для вычислений его необходимо перевести в радианы. В данном случае для этого используем команду `alpha=Math.toRadians(alpha)`, в которой вызвана встроенная функция `toRadians()`, предназначенная именно для этих целей.

Время полета между двумя последовательными ударами мячика о поверхность вычисляется командой `T0=2*V*Math.sin(alpha)/g`. Момент времени для последнего удара определяется с помощью команды `T=T0*Math.floor(t/T0)`. При этом использована функция округления `floor()`, которой в качестве результата возвращается наибольшее целое число, не превышающее аргумент функции.

В соответствии с приведенными ранее формулами координаты мячика вычисляются с помощью команд:

```
x=V*Math.cos(alpha)*t
y=V*Math.sin(alpha)*(t-T)-g*(t-T)*(t-T)/2
```

Округление этих значений до сотых выполняется командами:

```
x=Math.round(100*x)/100.0
y=Math.round(100*y)/100.0
```

Поскольку результат для координат мячика вычисляется в метрах, то последнее означает точность вычисления положения тела до сантиметра по каждой из координат. Для округления использовалась функция `round()`, которая округляет к ближайшему целому значению. После округления результат вычисления координат тела выводится на экран.

Решение тригонометрического уравнения

Рассмотрим программу для решения уравнения вида:

$$a\cos(x) + b\sin(x) = c\,.$$

Это уравнение, как известно, сводится к уравнению вида:

$$\sin(x + \alpha) = \frac{c}{\sqrt{a^2 + b^2}}\,,$$

где $\sin(\alpha) = \frac{a}{\sqrt{a^2 + b^2}}$. Поэтому формально решением исходного уравнения для любого целого n является:

$$x = -\arcsin\left(\frac{a}{\sqrt{a^2 + b^2}}\right) + (-1)^n \arcsin\left(\frac{c}{\sqrt{a^2 + b^2}}\right) + \pi n\,.$$

В программе, представленной в листинге 1.8, по значениям параметров a, b и c для значения $n = 0$ вычисляется решение уравнения, то есть решение (разумеется, если оно существует):

$$x = \arcsin\left(\frac{c}{\sqrt{a^2 + b^2}}\right) - \arcsin\left(\frac{a}{\sqrt{a^2 + b^2}}\right).$$

При этом проверяется условие существования решения $a^2 + b^2 = c^2$. Если данное условие не выполняется, уравнение решений не имеет. Экзотический случай, когда $a = b = c = 0$ (при таких условиях решением является любое значение параметра x) в программе не отслеживается.

Листинг 1.8. Вычисление корня уравнения

```
class FindRoot{
public static void main(String args[]){
// Параметры уравнения:
double a=5;
double b=3;
double c=1;
// Вспомогательная переменная:
double alpha;
// Логическая переменная - критерий наличия решений:
boolean state;
// Значение вспомогательной переменной:
alpha=Math.asin(a/Math.sqrt(a*a+b*b));
// Вычисление критерия:
state=a*a+b*b>=c*c;
// Вывод на экран значений исходных параметров:
System.out.println("Уравнение a*cos(x)+b*sin(x)=c");
```

```
System.out.println("Параметры:");
System.out.println("a="+a);
System.out.println("b="+b);
System.out.println("c="+c);
System.out.print("Решение для x: ");
// Вычисление решения уравнения и вывод на экран:
System.out.println(state?Math.asin(c/Math.sqrt(a*a+b*b))-alpha:"решений нет!");
}}
```

Основное место в программе — использование тернарного оператора в последней команде вывода на экран значения для корня уравнения. Предварительно выводится справочная информация о значениях параметров уравнения.

В последней команде вывода аргументом метода `println()` указано выражение:

```
state?Math.asin(c/Math.sqrt(a*a+b*b))-alpha:"решений нет!"
```

Это результат вычисления тернарного оператора, проверяемым условием в котором указана логическая переменная `state`. Ранее значение этой переменной присваивается командой `state=a*a+b*b>=c*c`. Значение переменной равно `true` в том случае, если уравнение имеет решения, и `false` — если не имеет. В случае если значение переменной `state` равно `true`, тернарным оператором в качестве результата возвращается числовое значение `Math.asin(c/Math.sqrt(a*a+b*b))-alpha`, где переменной `alpha` предварительно с помощью команды `alpha=Math.asin(a/Math.sqrt(a*a+b*b))` присвоено значение. В этих выражениях использованы встроенные функции `asin()` и `sqrt()` для вычисления арксинуса и корня квадратного. Таким образом, при истинном первом операнде тернарного оператора в качестве значения возвращается решение уравнения. Если значение первого операнда тернарного оператора (переменная `state`) равно `false`, в качестве результата возвращается текст `"решений нет!"`. Хотя при разных значениях первого операнда возвращаются значения разного типа, поскольку вся конструкция указана аргументом метода `println()` за счет автоматического приведения типов, в обоих случаях результат преобразуется в текстовый формат. Результат выполнения программы имеет вид:

```
Уравнение a*cos(x)+b*sin(x)=c
Параметры:
a=5.0
b=3.0
c=1.0
Решение для x: -0.8580262366249893
```

Если поменять значения исходных параметров уравнения, можем получить такое:

```
Уравнение a*cos(x)+b*sin(x)=c
Параметры:
a=5.0
b=3.0
c=10.0
Решение для x: решений нет!
```

Хотя использование тернарного оператора может быть достаточно эффективным, обычно подобного рода задачи решаются с помощью условных операторов, которые описываются в следующей главе.

Кодирование символов числами

Рассмотрим простую иллюстративную программу, в которой для записи сразу двух символьных значений (типа `char`) используется одна переменная целочисленного типа (`int`).

В программе учитывается то обстоятельство, что тип `int` в Java имеет размер в 32 бита, а для записи основных символов кодировки Unicode вполне достаточно 16 бит. Таким образом, объем памяти, выделяемой переменной типа `int`, достаточен для записи, по меньшей мере, двух букв (значений типа `char`). Алгоритм записи буквенных значений в виде чисел используем следующий: начальные 16 бит содержат код первой буквы, следующие 16 — код второй буквы. Программный код, в котором реализован этот принцип, приведен в листинге 1.9.

Листинг 1.9. Кодирование символов

```
class CharsAndInts{
public static void main(String args[]){
// Кодовое число:
int number;
// Исходные буквы для кодирования:
char symbA='A',symbB='ы';
// Буквы после декодирования:
char SymbA,SymbB;
// Вычисление кода:
number=((int)symbB<<16)+((int)symbA);
// Вывод исходных данных и кода:
System.out.println("Исходные буквы: \'"+symbA+"\' и \'"+symbB+"\'.");
System.out.println("Кодовое число: "+number);
// Декодирование:
SymbB=(char)(number>>>16);
SymbA=(char)(number^((int)SymbB<<16));
// Вывод результата декодирования:
System.out.println("Обратное преобразование:");
System.out.println("Буквы \'"+SymbA+"\' и \'"+SymbB+"\'.");}
}
```

Целочисленная переменная `number` предназначена для записи в нее числового кода, который формируется на основе значений переменных `symbA` и `symbB` типа `char`. После того как код создан и записан в переменную `number`, выполняется обратная операция: на основании значения переменной `number` восстанавливаются исходные символы, а результат записывается в переменные `SymbA` и `SymbB` типа `char`.

Значение переменной number задается командой:

```
number=((int)symbB<<16)+((int)symbA)
```

В правой части соответствующего выражения стоит сумма из двух слагаемых. Первое слагаемое ((int)symbB<<16) представляет собой смещенный вправо на 16 позиций (битов) числовой код символа, записанного в переменную symbB. Для получения кода символа использована инструкция (int) явного приведения типов. Таким образом, инструкцией (int)symbB получаем код символа, после чего с помощью оператора сдвига << смещаем код на 16 бит влево с заполнением нулями младших 16 бит. В эти биты записывается код оставшегося символа, записанного в переменную symbA. Для этого к полученному на первом этапе коду прибавляется значение ((int)symbA) — то есть код первого символа.

Исходные символы и полученный на их основе числовой код выводятся на экран. Затем начинается обратная процедура по «извлечению» символов из числового кода. Для этого командой SymbB=(char)(number>>>16) «считывается» второй символ и записывается в переменную SymbB. Действительно, результатом инструкции number>>>16 является смещенный вправо на 16 бит код переменной number (с заполнением старшего бита нулем), то есть код второго символа (того, что записан в переменную symbB). Первый символ «считывается» немного сложнее. В частности, используется команда:

```
SymbA=(char)(number^((int)SymbB<<16))
```

Результатом инструкции (int)SymbB<<16) является код уже считанного второго символа, смещенный влево на 16 бит. По сравнению с кодом, записанным в переменную number, он отличается тем, что его младшие 16 бит нулевые, в то время как в переменной number эти биты содержат код первого символа. Старшие 16 бит при этом совпадают. Указанные два кода являются операндами в логической операции ^ (побитовое исключающее ИЛИ). Напомним, что результатом такой операции является единица, если один и только один из двух сравниваемых битов равен единице. Для совпадающих старших битов это означает «полное обнуление», а младшие единичные биты «выживают», поэтому на выходе получаем код, записанный в младшие 16 бит, то есть код первого символа. Сам символ получаем с помощью инструкции (char) явного приведения к символьному типу. После выполненного декодирования символы выводятся на экран. В результате выполнения программы получаем следующее:

```
Исходные буквы: 'A' и 'ы'.
Кодовое число: 72025104
Обратное преобразование:
Буквы 'A' и 'ы'.
```

В принципе, если для работы планируется использовать только небольшой набор символов, по описанному принципу можно «упаковать» в одной числовой переменной и большее количество символов. Однако для практических задач криптографии такой метод вряд ли можно считать приемлемым, поэтому пример следует рассматривать лишь как иллюстративный.

Расчет параметров цепи

Составим программу для решения следующей задачи. Предположим, что участок электрической цепи должен состоять из двух блоков, в каждом из которых располагаются два параллельно соединенных резистора. Блоки между собой соединены последовательно. Имеется три резистора известного сопротивления, которые можно свободно переставлять между блоками, и один основной резистор, который обязан находиться во втором блоке. Необходимо определить, какой резистор вставить во второй блок в дополнение к основному, чтобы общее сопротивление участка цепи было минимальным.

Если сопротивления трех переставляемых резисторов обозначить как R_1, R_2 и R_3, а сопротивление основного резистора как R, то при условии, что первые два резистора подключаются в первый блок, а третий резистор — во второй, общее сопротивление участка цепи будет составлять величину:

$$r = \frac{R_1 R_2}{R_1 + R_2} + \frac{R R_3}{R + R_3}.$$

Поэтому для определения оптимального способа подключения резисторов нужно проверить три варианта, когда каждый из трех резисторов включается во второй блок, и выбрать тот вариант подключения, когда общее сопротивление минимально. Соответствующий программный код приведен в листинге 1.10.

Листинг 1.10. Оптимальное подключение резисторов

```
class Resistors{
public static void main(String args[]){
// Сопротивление резисторов (Ом):
double R1=3,R2=5,R3=2,R=1;
// Расчетные значения для сопротивления участка цепи (Ом):
double r1,r2,r3;
// Логические значения для определения способа подключения:
boolean A,B;
// Вычисление сопротивления участка цепи для разных способов подключения:
r1=R2*R3/(R2+R3)+R1*R/(R1+R);
r2=R1*R3/(R1+R3)+R2*R/(R2+R);
r3=R2*R1/(R2+R1)+R3*R/(R3+R);
// Вычисление критериев для способа подключения:
A=(r1<=r2)&&(r1<=r3);
B=(r2<=r1)&&(r2<=r3);
// Вывод начальных значений:
System.out.println("Значения сопротивлений резисторов:");
System.out.println("Первый  R1="+R1+" Ом");
System.out.println("Второй  R2="+R2+" Ом");
System.out.println("Третий  R3="+R3+" Ом");
System.out.println("Основной R="+R3+" Ом");
```

```
// Вычисление и вывод результата:
System.out.print("Во второй блок подключается ");
System.out.print(A?"первый":B?"второй":"третий");
System.out.println(" резистор!");}
}
```

В результате выполнения программы получаем следующие сообщения:

```
Значения сопротивлений резисторов:

Первый R1=3.0 Ом
Второй R2=5.0 Ом
Третий R3=2.0 Ом
Основной R=2.0 Ом
Во второй блок подключается второй резистор!
```

В программе объявляются и инициализируются переменные `R1`, `R2`, `R3`, `R` типа `double`, определяющие сопротивления трех переставляемых резисторов и основного резистора соответственно. Переменные `r1`, `r2` и `r3` типа `double` предназначены для вычисления и записи в них значения сопротивления участка цепи для каждого из трех возможных способов подключения резисторов. Также в программе объявляются две логические переменные `A` и `B` (типа `boolean`). Значения этих переменных определяются командами `A=(r1<=r2)&&(r1<=r3)` и `B=(r2<=r1)&&(r2<=r3)`. Значение переменной `A` равно `true` в том случае, если при первом способе подключения резисторов (во втором блоке размещается первый резистор) общее сопротивление цепи не превышает сопротивление цепи для второго и третьего способов подключения резисторов. Значение переменной `B` равно `true` в том случае, если при втором способе подключения резисторов (во втором блоке размещается второй резистор) общее сопротивление цепи не превышает сопротивление цепи для первого и третьего способов подключения резисторов. Понятно, что если обе эти переменные равны `false`, то оптимальным является третий способ подключения резисторов.

После вычисления значений переменных `A` и `B` выполняется вывод результата. Сначала серией команд отображаются текущие значения, указанные при инициализации переменных для сопротивлений резисторов. Затем выводится начало фразы о способе подключения резисторов. Номер резистора (в текстовом формате) определяется непосредственно в аргументе метода `println()` командой `A?"первый":B?"второй":"третий"`, в которой использованы вложенные тернарные операторы. Если значение переменной `A` (первый операнд внешнего тернарного оператора) равно `true`, возвращается второй операнд внешнего тернарного оператора — текстовое значение `"первый"`. В противном случае вычисляется третий операнд внешнего тернарного оператора. Третьим операндом является тернарный оператор `B?"второй":"третий"`. При условии, что значение переменной `B` равно `true`, возвращается текст `"второй"`, в противном случае — текст `"третий"`. После того как нужное слово (название резистора) выведено на экран, следующими командами завершается выведение финальной фразы.

Резюме

1. Язык программирования Java является полностью объектно-ориентированным. Для создания даже самой простой программы необходимо описать, по крайней мере, один класс. Этот класс содержит метод со стандартным названием `main()`. Выполнение программы отождествляется с выполнением этого метода.
2. В методе `main()` можно объявлять переменные. Для объявления переменной указывается тип переменной и ее имя. Переменной одновременно с объявлением можно присвоить значение (инициализировать переменную). Переменная должна быть инициализирована до ее первого использования.
3. Существует несколько базовых типов данных. При вычислении выражений выполняется автоматическое приведение типов. Особенность приведения типов в Java состоит в том, что оно осуществляется без потери значений. Также можно выполнять явное приведение типов, для чего перед выражением в круглых скобках указывается идентификатор соответствующего типа.
4. Основные операторы Java делятся на арифметические, логические, побитовые и сравнения. Арифметические операторы предназначены для выполнения таких операций, как сложение, вычитание, деление и умножение. Логические операторы предназначены для работы с логическими операндами и позволяют выполнять операции *отрицания*, *ИЛИ*, *И*, *ИСКЛЮЧАЮЩЕГО ИЛИ*. Операторы сравнения используются, как правило, при сравнении (на предмет равенства или неравенства) числовых операндов. Результатом сравнения является логическое значение (значение логического типа). Побитовые операторы служат для выполнения операций (логических) на уровне битовых представлений чисел, а также побитовых сдвигов вправо и влево в побитовом представлении числа.
5. В Java для основных арифметических и побитовых операторов, используемых в комбинации с оператором присваивания для изменения значения одного из операндов, имеются упрощенные формы. В частности, команда вида `x =x op y` может быть записана как `x op=y`, где через `op` обозначен арифметический или побитовый оператор.
6. В Java есть тернарный оператор, который представляет собой упрощенную форму условного оператора. Первым его операндом указывается логическое выражение. В зависимости от его значения в качестве результата возвращается второй или третий операнд.

Глава 2. Управляющие инструкции Java

Мы никогда ничего не запрещаем! Мы только советуем!

Из к/ф «Забытая мелодия для флейты»

К управляющим инструкциям относят условные инструкции и инструкции цикла. В Java таких инструкций несколько, и каждая из них имеет свои особенности. Далее последовательно рассматриваются эти инструкции и приводятся примеры их использования.

Условная инструкция if()

Мой соперник не будет избран, если дела не пойдут хуже.
А дела не пойдут хуже, если его не выберут.

Дж. Буш-старший

Если не считать тернарного оператора, в Java существует две условных конструкции, которые позволяют выполнять разные операции в зависимости от некоторого условия. В первую очередь рассмотрим условную инструкцию `if()`. Синтаксис ее вызова имеет в общем случае вид:

```
if(условие){инструкции_1}
elase{инструкции_2}
```

Условная инструкция `if()` выполняется в следующей последовательности. Сначала проверяется условие, указанное в круглых скобках после ключевого слова `if`. Если условие верное (значение соответствующего выражения равно `true`), выполняется блок инструкций, указанный сразу после инструкции `if(условие)` (в данном случае это `инструкции_1`). В противном случае, то есть если значение выражения в круглых скобках после ключевого слова `if` равно `false`, выполняется блок инструкций, указанных после ключевого слова `else` (в данном случае это `инструкции_2`). После выполнения условной инструкции управление передается следующей после ней инструкции. Обращаем внимание читателя на несколько обстоятельств.

Во-первых, если любой из двух блоков инструкций состоит всего из одной команды, фигурные скобки для соответствующего блока можно не использовать.

Тем не менее лишними фигурные скобки никогда не бывают, поскольку на быстродействии они не сказываются, а читабельность программы значительно улучшают.

Во-вторых, ветвь `else` условной инструкции не является обязательной. Синтаксис вызова такой упрощенной формы условной инструкции `if()` имеет следующий вид:

```
if(условие){инструкции}
```

В этом случае сначала проверяется на истинность условие, указанное в скобках после ключевого слова `if`. Если условие истинно, выполняется следующий после условия `if` блок инструкций. Если условие ложно, указанные инструкции не выполняются.

На практике нередко используются вложенные инструкции `if()`. С точки зрения синтаксиса языка Java такая ситуация проста: в ветви `else` условной инструкции указывается другая условная инструкции и т. д. Синтаксическая конструкция имеет вид:

```
if(условие_1){инструкции_1}
else if(условие_2){инструкции_2}
else if(условие_3){инструкции_3}
...
else if(условие_N){инструкции_N}
elase{инструкции}
```

Последовательность выполнения такого блока вложенных условных инструкций такова. Сначала проверяется `условие_1`. Если оно истинно, выполняются `инструкции_1`. Если `условие_1` ложно, проверяется `условие_2`. При истинном условии выполняются `инструкции_2`. В противном случае проверяется `условие_3` и т. д. Если и последнее `условие_N` окажется ложным, будет выполнен блок инструкций `инструкции` финальной ветви `else`.

Для читателей, программирующих в C++, отдельно обращаем внимание на то обстоятельство, что условие, которое указывается для проверки в условной инструкции `if()`, должно быть выражением, возвращающим логическое значение (тип `boolean`). Здесь кроется принципиальное отличие от языка C++, в котором в качестве условия в аналогичной инструкции `if()` может указываться число. В языке Java автоматического приведения числовых значений к логическому типу нет! Это же замечание относится и к прочим управляющим инструкциям в Java.

В листинге 2.1 приведен пример достаточно простой программы, в которой используется условная инструкция.

Листинг 2.1. Использование условной инструкции if()

```
class UsingIf{
public static void main(String[] args){
int x=3,y=6,z;
```

```
// Условная инструкция:
if(x!=0){
    z=y/x;
    System.out.println("Значение z="+z);}
else System.out.println("Деление на нуль!");
}
}
```

В программе объявляются три целочисленные переменные x, y и z. Первым двум переменным сразу при объявлении присваиваются значения. В условной инструкции переменная x проверяется на предмет отличия ее значения от нуля (условие x!=0). Если значение переменной не равно нулю, переменной z присваивается результат деления значения переменной y на значение переменной x. После этого выводится сообщение с указанием значения переменой z.

Если проверяемое условие ложно (то есть значение переменной x равно нулю), выводится сообщение "Деление на нуль!". Для приведенных в листинге 2.1 значений переменных в результате выполнения программы появится сообщение Значение z=2.

Еще один пример использования условной инструкции if() в упрощенной форме приведен в листинге 2.2.

Листинг 2.2. Использование упрощенной формы инструкции if()

```
class UsingIf2{
public static void main(String[] args){
int x=3,y=6,z;
// Условная инструкция:
if(x!=0){
    z=y/x;
    System.out.println("Значение z="+z);
// Завершение программы:
    return;}
System.out.println("Деление на нуль!");
}
}
```

Функциональность программного кода, по сравнению с предыдущим примером, не изменилась. Однако механизм использования условной инструкции несколько изменился. Главное отличие состоит в том, что теперь отсутствует ветвь else условной инструкции.

Как и ранее, если значение переменной x отлично от нуля, командой z=y/x присваивается значение переменной z, после чего выводится сообщение "Значение z=" со значением этой переменной. Следующей командой в блоке условной инструкции является инструкция завершения работы программы return.

Если значение переменной x равно нулю, блок команд условной инструкции не выполняется, а выполняется команда System.out.println("Деление на нуль!"),

размещенная после условной инструкции. Таким образом, сообщение Деление на нуль! появляется только в том случае, если не выполнено условие в условной инструкции.

Для значений переменных, представленных в листинге 2.2, в результате выполнения программы на экране появляется сообщение Значение z=2.

Пример использования нескольких вложенных инструкций if() приведен в листинге 2.3.

Листинг 2.3. Использование вложенных инструкций if()

```
class UsingIf3{
public static void main(String[] args){
int a=0;
// Если a равно 0:
if(a==0){System.out.println("Нулевое значение переменной!");}
// Если a равно 1:
else if(a==1){System.out.println("Единичное значение переменной!");}
// Если a - четное (остаток от деления на 2 равен 0):
else if(a%2==0){System.out.println("Четное значение переменной!");}
// В прочих случаях:
else {System.out.println("Нечетное значение переменной!");}
System.out.println("Программа завершила работу!");
}
}
```

В методе main() объявляется целочисленная переменная a. В зависимости от значения этой переменной на экран выводятся разные сообщения. При нулевом значении переменной выводится сообщение Нулевое значение переменной!. Если значение переменной a равняется 1, выводится сообщение Единичное значение переменной!. Сообщение Четное значение переменной! появляется в том случае, если значение переменной a есть число четное. Для проверки четности значения переменной a вычисляется остаток от деления a на 2 (для четного числа остаток равен 0). В прочих случаях программой выводится сообщение Нечетное значение переменной!.

Перебор всех возможных вариантов реализован через блок вложенных условных инструкций. Перечисленные ранее условия проверяются по очереди, до первого выполненного условия. Если ни одно из условий не выполнено, командой System.out.println("Программа завершила работу!") в завершающей ветви else блока вложенных условных инструкций выводится соответствующее сообщение.

Хотя с помощью блоков вложенных условных инструкций if() можно реализовать практически любую схему ветвления алгоритма программы, нередко вместо вложенных инструкций if() используется условная инструкция выбора switch().

Условная инструкция switch()

— Утром деньги — вечером стулья. Вечером деньги — утром стулья.
— А можно так: утром стулья — вечером деньги?
— Можно! Но деньги вперед!

Из к/ф «Двенадцать стульев»

Обычно к услугам условной инструкции `switch()` прибегают в случае, когда при проверке условия альтернатив больше, чем две. Эту инструкцию еще называют инструкцией выбора. Синтаксис вызова инструкции `switch()` следующий:

```
switch(условие){
case значение_1:
//команды_1
brake;
case значение_2:
//команды_2
brake;
...
case значение_N:
//команды_N
brake;
default:
//команды
}
```

После ключевого слова `switch` в круглых скобках указывается переменная или выражение, значение которого проверяются (`условие`). Возможные значения, которые может принимать `условие`, перечисляются после ключевых слов `case`. Каждому значению соответствует свой блок `case`. Каждый блок `case` заканчивается инструкцией `break`. Последним блоком в инструкции `switch()` является блок команд, выполняемых по умолчанию. Блок выделяется инструкцией `default()`. Блок не является обязательным, и инструкцию `break` в конце этого блока размещать не нужно. Наконец, все блоки `case` и блок `default`, если он есть, заключаются в фигурные скобки. Именно эти фигурные скобки определяют тело инструкции `switch()`.

Алгоритм выполнения инструкции `switch()` следующий. Сначала вычисляется выражение или значение переменной, указанной в качестве условия. Затем вычисленное значение последовательно сравнивается со значениями, указанными после инструкций `case`, пока не будет найдено совпадение или не встретится блок `default` (если блок `default` отсутствует, то пока не будет достигнут конец тела инструкции `switch()`). Если совпадение найдено, начинает выполняться программный код соответствующего блока `case`. Код выполняется до конца тела инструкции `switch()` или `break()`. Собственно, инструкции `break` в блоках `case` и нужны для того, чтобы остановить выполнение программного кода инструк-

ции switch(). В противном случае продолжали бы выполняться следующие блоки case.

Выражение, которое указывается в качестве проверяемого условия, может возвращать в качестве значения целое число или символ. Значения, указываемые после инструкций case(), должны быть литералами или константами. Пример использования инструкции switch приведен в листинге 2.4.

Листинг 2.4. Использование инструкции switch()

```
class UsingSwitch{
public static void main(String[] args){
char s='П';
System.out.print("Фамилия пользователя: ");
// Инструкция выбора:
switch(s){
    case 'И':
    System.out.println("Иванов");
    break;
    case 'П':
    System.out.println("Петров");
    break;
    case 'С':
    System.out.println("Сидоров");
    break;
    default:
    System.out.println("Не определена");
}
System.out.println("Программа завершила работу!");
}
}
```

В методе main() класса UsingSwitch объявляется переменная s типа char. Значением переменной является начальная буква фамилии пользователя. Рассматривается три варианта: буква И соответствует фамилии Иванов, буква П соответствует фамилии Петров и буква С соответствует фамилии Сидоров.

Командой System.out.print("Фамилия пользователя: ") выводится сообщение, причем переход на следующую строку не осуществляется — в этом главное отличие метода print() от метода println(). Далее с помощью инструкции switch() осуществляется перебор значений переменной s. Если совпадение найдено, выводится соответствующая фамилия. Если совпадение не найдено, командой System.out.println("Не определена") выводится сообщение Не определена. В конце выполнения программы выводится сообщение об окончании работы. Для значения переменной s='П' результат выполнения программы будет иметь вид:

```
Фамилия пользователя: Петров
Программа завершила работу!
```

Обращаем внимание, что, во-первых, значением переменной s может быть кириллическая буква, во-вторых, регистр буквы имеет значение — если переменной s присвоить значение 'п', результат выполнения программы будет следующим:

```
Фамилия пользователя: Не определена
Программа завершила работу!
```

Чтобы фамилия определялась независимо от регистра буквы, рассмотренную программу нужно несколько видоизменить. Модифицированный ее вариант приведен в листинге 2.5.

Листинг 2.5. Пустые блоки в инструкции switch

```
class UsingSwitch2{
public static void main(String[] args){
char s='п';
System.out.print("Фамилия пользователя: ");
// Инструкция вывода:
switch(s){
    case 'И':
    case 'и':
    System.out.println("Иванов");
    break;
    case 'П':
    case 'п':
    System.out.println("Петров");
    break;
    case 'С':
    case 'с':
    System.out.println("Сидоров");
    break;
    default:
    System.out.println("Не определена");
}
System.out.println("Программа завершила работу!");
}
}
```

В данном случае использованы пустые блоки case. Пример:

```
case 'П':
case 'п':
...
```

В этом случае, если значение проверяемой переменной (в данном случае переменной s) совпадает с буквой 'П', выполняется код вплоть до первой инструкции break, то есть такой же код, как и для случая, когда переменная s имеет значение 'п'.

Инструкция цикла for()

Все кончается рано или поздно.

Из к/ф «Гараж»

Для выполнения однотипных многократно повторяющихся действий используют инструкции цикла. В Java существует несколько инструкций цикла. Рассмотрим инструкцию цикла `for()`.

Синтаксис вызова инструкции цикла `for()` следующий:

```
for(инициализация;условие;итерация){
// тело цикла
}
```

В круглых скобках после ключевого слова `for` указываются три группы, или блока, выражений. Блоки разделяются точкой с запятой. Первый блок обычно называется блоком инициализации. Как правило, в этом блоке размещается команда (или команды), которая перед выполнением цикла присваивает индексным переменным начальные значения. Второй блок — условие, при выполнении которого продолжается работа инструкции цикла. Третий блок содержит команды, которыми изменяются значения индексных переменных. Первый и третий блоки могут состоять из нескольких команд. Команды одного блока разделяются запятыми.

Далее, после ключевого слова `for` и круглых скобок в фигурных скобках идет блок команд, которые выполняются в рамках каждого цикла и фактически формируют тело инструкции цикла. Если тело инструкции цикла состоит из одной команды, фигурные скобки можно не ставить. Выполняется инструкция цикла по следующему алгоритму.

Начинается выполнение инструкции цикла с блока инициализации: последовательно выполняются все команды этого блока. Далее проверяется условие во втором блоке. Если оно истинно (значение `true`), выполняются команды тела инструкции цикла (команды в фигурных скобках). Далее выполняются команды третьего блока в круглых скобках и проверяется условие во втором блоке. Если условие истинно, выполняются команды основного тела инструкции цикла, команды блока изменения индексных переменных (третий блок), затем проверяется условие и т. д. Вся эта процедура продолжается до тех пор, пока при проверке условия его значение не становится равным `false`.

По завершении инструкции цикла выполняется следующая после него инструкция. Обращаем внимание читателя на то, что команды первого блока инициализации инструкции цикла выполняются только один раз на начальном этапе выполнения инструкции. Пример использования инструкции цикла приведен в листинге 2.6.

Листинг 2.6. Использование инструкции цикла for()

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

```
// Индексная переменная:
int i;
// Переменная для вычисления суммы:
int sum=0;
// Инструкция цикла:
for(i=1;i<=100;i++){
sum+=i;}
System.out.println("Сумма чисел от 1 до 100: "+sum);}
}
```

Программой вычисляется сумма натуральных чисел от 1 до 100. Для этого вводится целочисленная переменная `sum`, которая инициализируется с начальным нулевым значением — в эту переменную записывается значение суммы. Вычисление суммы осуществляется посредством инструкции цикла. В нем используется целочисленная индексная переменная `i`. Объявляется переменная вне инструкции цикла. В первом блоке (блок инициализации) индексной переменной присваивается значение 1. Проверяется условие `i<=100`, то есть инструкция цикла выполняется до тех пор, пока значение индексной переменной не превысит значение 100. В теле инструкции цикла переменная `sum` увеличивается на текущее значение индексной переменной `i`. В третьем блоке инструкции цикла командой `i++` значение индексной переменной увеличивается на единицу.

Последней командой программы выводится сообщение о значении суммы. В результате выполнения программы мы получаем сообщение:

```
Сумма чисел от 1 до 100: 5050
```

Чтобы посчитать сумму нечетных чисел в указанном диапазоне, в третьем блоке изменения индексной переменной команду `i++` достаточно заменить командой `i+=2`. Кроме того, индексную переменную можно инициализировать прямо в первом блоке инструкции цикла. Пример измененной программы для вычисления суммы нечетных чисел приведен в листинге 2.7.

Листинг 2.7. Вычисление суммы нечетных чисел

```
class UsingFor2{
public static void main(String[] args){
// Переменная для вычисления суммы:
int sum=0;
// Инструкция цикла:
for(int i=1;i<=100;i+=2){
sum+=i;}
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

В результате выполнения программы получаем:

```
Сумма нечетных чисел от 1 до 100: 2500
```

Как уже отмечалось, первый блок инициализации и третий блок изменения индексной переменной могут состоять из нескольких команд. В листинге 2.8

приведен пример программы для вычисления суммы нечетных чисел, в которой команды инициализации переменных и команда основного тела инструкции цикла включены соответственно в первый и третий блоки инструкции цикла.

Листинг 2.8. Сумма нечетных чисел

```
class UsingFor3{
public static void main(String[] args){
int sum,i;
// Инструкция цикла:
for(sum=0,i=1;i<=100;sum+=i,i+=2);
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

Индексная переменная `i` и переменная `sum` объявляются вне инструкции цикла. Инициализируются обе переменные в первом блоке. Условие выполнения инструкции цикла не изменилось. Третий блок состоит из двух команд: команды `sum+=i`, предназначенной для увеличения значения переменной `sum` на величину `i`, и команды `i+=2`, изменяющей значение индексной переменной. Тело инструкции цикла не содержит команд, поэтому после закрывающей круглой скобки стоит точка с запятой. Результат выполнения программы такой же, как и в предыдущем случае. Отметим несколько принципиальных моментов.

- Если переменные `sum` и `i` объявить в инструкции цикла, доступными они будут только в пределах этой инструкции. С индексной переменной `i` в этом случае проблем не возникает, а вот последняя команда программы, в которой имеется ссылка на переменную `sum`, оказывается некорректной.
- Имеет значение порядок следования команд в третьем блоке инструкции цикла. Если команды `sum+=i` и `i+=2` поменять местами, сначала будет изменяться значение индексной переменной, а затем на это новое значение увеличиваться переменная `sum`. В результате сумма будет вычисляться не от 1, а от 3.
- Хотя тело инструкции цикла не содержит команд, точку с запятой все равно ставить нужно. Если этого не сделать, телом цикла станет следующая после инструкции цикла команда — в данном случае это команда:

  ```
  System.out.println("Сумма нечетных чисел от 1 до 100: "+sum)
  ```

  При этом с формальной точки зрения синтаксис программы остается корректным и сообщение об ошибке не появляется. При выполнении программы на каждой итерации, за исключением последней, будет осуществляться вывод текущего значения переменой `sum`.

Блоки инструкции цикла могут быть пустыми. В листинге 2.9 приведен пример, в котором отсутствуют команды первого блока инициализации и третьего блока изменения значения индексной переменной. При этом программа работает корректно, и сумма нечетных чисел вычисляется правильно.

Листинг 2.9. Пустые блоки в инструкции цикла

```
class UsingFor4{
public static void main(String[] args){
```

```
int sum=0,i=1;
// Инструкция цикла с пустыми блоками:
for(;i<=100;){
sum+=i;
i+=2;}
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

Индексная переменная `i` и переменная для вычисления суммы `sum` инициализируются при объявлении вне инструкции цикла. Поэтому в блоке инициализации ничего инициализировать не нужно, и блок оставлен пустым (хотя точка с запятой все равно ставится). Второй блок с проверяемым условием остался неизменным. Третий блок также пустой. Команда `i+=2`, которая изменяет значение индексной переменной, вынесена в тело инструкции цикла.

Ситуацию можно усугубить, что называется, до предела, видоизменив программу так, чтобы все три блока инструкции цикла были пустыми. Пример приведен в листинге 2.10.

Листинг 2.10. В инструкции цикла все блоки пустые

```
class UsingFor5{
public static void main(String[] args){
int sum=0,i=1;
// Инструкция цикла со всеми пустыми блоками:
for(;;){
sum+=i;
i+=2;
// Выход из инструкции цикла:
if(i>100) break;}
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

Поскольку второй блок пустой, то формально мы имеем дело с бесконечным циклом. В таком цикле необходимо предусмотреть возможность выхода из цикла. Для этого последней в теле инструкции цикла размещена команда `if(i>100) break` с условной инструкцией. При выполнении условия, проверяемого в условной инструкции, командой `break` осуществляется завершение инструкции цикла.

Инструкция цикла while()

А вы все разлагаете молекулы на атомы,
пока тут разлагается картофель на полях.

В. Высоцкий

Для организации циклов кроме инструкции `for()` часто используется инструкция `while()` (или ее модификация — инструкция `do-while()`, которая рассматри-

вается в следующем разделе). Далее приведен синтаксис вызова инструкции while():

```
while(условие){
// команды цикла
}
```

После ключевого слова while в круглых скобках указывается условие. В начале выполнения инструкции проверяется это условие. Если условие истинно, выполняются команды цикла — они заключаются в фигурные скобки. После этого снова проверяется условие и т. д.

От инструкции for() инструкция while() принципиально отличается тем, что инициализация индексной переменной, если такая имеется, выполняется до вызова инструкции, а команда изменения этой переменной размещается в теле цикла. Поэтому инструкция while() более гибкая в плане возможных вариантов ее использования. Все, что запрограммировано с помощью инструкции for(), может быть запрограммировано и с помощью инструкции while(). Например, в листинге 2.11 приведен пример программы для вычисления суммы нечетных натуральных чисел с использованием инструкции while().

Листинг 2.11. Вычисление суммы с помощью инструкции while()

```
class UsingWhile{
public static void main(String[] args){
int sum=0,i=1;
// Инструкция цикла:
while(i<=100){
sum+=i;
i+=2;}
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

Смеем надеяться, что приведенный код особых комментариев не требует.

Инструкция do-while()

— Что же делать? — Ждать!
— Чего? — Пока не похудеет!

Из м/ф «Винни-Пух и Пятачок»

Инструкция do-while() является модификацией инструкции while(). Синтаксис ее вызова такой:

```
do{
// команды цикла
}while(условие);
```

Выполнение инструкции начинается с блока команд цикла, размещенных в фигурных скобках после ключевого слова do. Затем проверяется условие, указанное в круглых скобках после ключевого слова while. Если условие истинно, выполняются команды цикла и снова проверяется условие и т. д. Таким образом, хотя бы один раз команды цикла будут выполнены — в этом отличие инструкции do-while() от инструкции while(). В листинге 2.12 приведен пример использования инструкции do-while() в программе для вычисления суммы нечетных натуральных чисел.

Листинг 2.12. Использование инструкции do-while()

```
class UsingDoWhile{
public static void main(String[] args){
int sum=0,i=1;
// Инструкция цикла:
do{
sum+=i;
i+=2;}while(i<=100);
System.out.println("Сумма нечетных чисел от 1 до 100: "+sum);}
}
```

Результат выполнения этой программы точно такой же, как и рассмотренных ранее аналогичных программ по вычислению суммы нечетных натуральных чисел в диапазоне от 1 до 100.

Метки и инструкции break() и continue()

Это лирическое отступление пора бы заканчивать.

Из к/ф «Гараж»

В Java, в отличие от языка C++, нет инструкции перехода goto(). Тем не менее в Java могут использоваться метки. Обычно для этого применяют инструкции break() и continue().

С инструкцией break() мы уже сталкивались. В общем случае она завершает работу инструкции цикла или инструкции выбора. Если после инструкции break() указать метку, то управление передается команде, размещенной за помеченной этой меткой инструкцией цикла, выбора или блоком команд (напомним, блок команд заключается в фигурные скобки).

Инструкция continue() используется для завершения текущего цикла в инструкции цикла и переходу к выполнению следующего цикла. Если после инструкции continue() указать метку, то выполняется переход для выполнения итерации помеченного меткой внешнего цикла. Таким образом, инструкция continue() с меткой применяется только тогда, когда имеются вложенные циклы.

Что касается непосредственно меток, то в качестве метки применяется идентификатор, который не может начинаться с цифры, заканчивается двоеточием и предназначен для выделения места в программном коде. Никакого дополнительного описания метка не требует.

Пример использования меток приведен в листинге 2.13.

Листинг 2.13. Использование меток

```
class LabelsDemo{
public static void main(String[] args){
MyLabel:
for(int i=1;i<=100;i++){
for(int j=1;j<=100;j++){
if(i!=j) continue;
if((j%3==0)||(i%2==0)) break;
if(i+j>20) break MyLabel;
System.out.println(i+":"+j);}
}}}
```

В программе имеется блок из вложенных инструкций цикла. В каждом цикле индексная переменная пробегает значения от 1 до 100 включительно. Внешняя инструкция цикла помечена меткой `MyLabel`.

В теле внутреннего цикла размещено три условных инструкции и команда `System.out.println(i+":"+j)`, предназначенная для вывода текущих значений индексных переменных `i` и `j` для вложенных инструкций цикла. В первой условной инструкции проверяется условие `i!=j`. Если индексные переменные принимают разные значения, командой `continue` досрочно завершается текущий цикл внутренней инструкции. В результате действия такого «фильтра» на экран выводятся только одинаковые значения индексных переменных, да и то не все. Преградой служат вторая и третья условные инструкции. Так, во второй условной инструкции проверяется условие `(j%3==0)||(i%2==0)`. Оно истинно, если индексная переменная `j` делится на 3 или индексная переменная `i` делится на 2. В этом случае командой `break` досрочно завершает работу внутренняя инструкция цикла. Внешняя индексная переменная увеличивается на единицу, и внутренняя инструкция цикла запускается снова. Наконец, если выполнено условие `i+j>20` (третья условная инструкция), командой `break MyLabel` выполняется досрочное завершение блока команд, помеченных меткой `MyLabel`, то есть в данном случае завершается работа внешней инструкции цикла. Результат выполнения программы имеет вид:

```
1:1
5:5
7:7
```

На экран выводятся пары одинаковых индексов, которые не делятся на 3 и 2 и сумма которых не превышает 20.

Примеры программ

Далее рассматриваются некоторые программы, в которых используются условные инструкции и инструкции цикла.

Вычисление экспоненты

В Java существует встроенная экспоненциальная функция `Math.exp()`, результатом выполнения которой по аргументу x является значение ex, где e = 2,718281828 — постоянная Эйлера. Для вычисления экспоненты используется сумма, которая представляет собой разложение экспоненциальной функции в ряд Тейлора в окрестности нуля:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$$

Разумеется, на практике вычислить бесконечную сумму невозможно, поэтому ограничиваются вычислением суммы для конечного количества слагаемых — чем их больше, тем выше точность. В листинге 2.14 приведен пример программы, в которой на основе приведенной функции вычисляется значение экспоненты. В этом случае используется инструкция цикла.

Листинг 2.14. Вычисление экспоненты

```
class MyExp{
public static void main(String args[]){
// Верхняя граница ряда и индексная переменная:
int N=100,k;
// Аргумент экспоненты, переменная для записи суммы
// и итерационная добавка:
double x=1,s=0,q=1;
// Вычисление экспоненты:
for(k=0;k<=N;k++){
s+=q;
q*=x/(k+1);}
// Вывод результата:
System.out.println("exp("+x+")="+s);}
}
```

Поскольку значением для аргумента экспоненты указана единица, в результате выполнения программы получаем приближенное значение для постоянной Эйлера (причем с довольно неплохой точностью):

```
exp(1.0)=2.7182818284590455
```

Как уже отмечалось, основу программы составляет инструкция цикла. В этой инструкции индексная переменная пробегает значения от 0 до `N` (значение этой переменной установлено равным 100). В теле цикла всего две команды. Первой командой `s+=q` в переменную `s` (ее начальное значение равно нулю) записывается сумма для экспоненты. Каждый раз значение переменной `s`

увеличивается на величину q, после чего командой q*=x/(k+1) изменяется значение переменной-добавки. Переменная q умножается на x и делится на (k+1). Изменение переменной-добавки выполняется так, чтобы на следующем шаге эта добавка давала «правильное» приращение ряда Тейлора. Действительно, в программе вычисляется сумма $\exp(x) \approx \sum_{k=0}^{N} \frac{x^k}{k!}$, поэтому приращение суммы для k-го индекса равняется $q_k = \frac{x^k}{k!}$. Для $(k + 1)$-го индекса добавка равняется $q_{k+1} = \frac{x^{k+1}}{(k+1)!}$. В соответствии с соотношением $\frac{q_{k+1}}{q_k} = \frac{x}{(k+1)}$ на основе добавки на k-м шаге для следующей итерации добавку необходимо умножить на x и разделить на $(k + 1)$.

Числа Фибоначчи

Числами Фибоначчи называется последовательность натуральных чисел, первые два из которых равны единице, а каждое следующее число в последовательности равняется сумме двух предыдущих. В листинге 2.15 приведен пример программного кода, в котором на экран выводятся числа из последовательности Фибоначчи. Как и в предыдущем случае, основу этой программы составляет инструкция цикла.

Листинг 2.15. Числа Фибоначчи

```
class Fibonacci{
public static void main(String args[]){
// Количество чисел последовательности и начальные члены:
int N=15,a=1,b=1;
// Индексная переменная:
int i;
System.out.println("Числа Фибоначчи:");
// Вывод на экран двух первых членов последовательности:
System.out.print(a+" "+b);
// Вычисление последовательности Фибоначчи:
for(i=3;i<=N;i++){
b=a+b;
a=b-a;
System.out.print(" "+b);}}
}
```

В результате выполнения программы получаем следующее:

```
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

В программе объявляются целочисленные переменные N (количество вычисляемых чисел в последовательности Фибоначчи), а также переменные a и b, которые предназначены для записи предпоследнего и последнего вычисленных на данный момент чисел в последовательности. Этим переменным присвоены начальные единичные значения. Эти значения сразу выводятся на экран. Далее

в инструкции цикла выполняется вычисление и вывод на экран последующих членов. В частности, для вычисления следующего числа в последовательности, если известны последнее (b) и предпоследнее (a) значения, используется команда b=a+b — каждое новое число равняется сумме двух предыдущих. После этого необходимо в переменную a записать значение, которое до этого было записано в переменную b. Поскольку значение b уже изменилось и содержит сумму «старого» значения переменной b и текущего значения переменной a, от текущего значения переменной b необходимо отнять текущее значение переменной a и записать результат в переменную a. Поэтому после первой упомянутой команды в инструкции цикла выполняется команда a=b-a. Новое вычисленное число b выводится на экран командой System.out.print(" "+b).

Вычисление числа π

Воспользуемся модифицированным методом Монте-Карло для вычисления числа π. В частности, проведем следующий мысленный эксперимент. Впишем круг в квадрат с единичной стороной. Площадь такого квадрата равна, очевидно, единице. Радиус круга равен 1/2, а площадь — π/4. Эксперимент состоит в том, что внутри квадрата случайным образом выбираются точки. Точек много, и они равномерно распределены по квадрату. Некоторые из них попадают внутрь круга, другие — нет. Вероятность попадания точки внутрь круга равна отношению площади круга к площади квадрата, то есть равна π/4. В то же время, если точек достаточно много, то отношение числа попавших внутрь круга точек к общему числу точек внутри квадрата должно быть близко к вероятности попадания точки внутрь круга. Чем больше выбрано точек, тем точнее совпадение. Поэтому для расчета числа π = 3,14159265 случайным (или не очень случайным) образом выбираем внутри квадрата какое-то количество точек (чем больше — тем лучше), подсчитываем, сколько из них попадает внутрь круга, находим отношение количества точек внутри круга к общему количеству точек, умножаем полученное значение на 4 и получаем, таким образом, оценку для числа π.

Для решения этой задачи нужен «хороший» генератор случайных чисел — такой, чтобы генерировал случайное число с постоянной плотностью распределения на интервале значений от 0 до 1. Этого не так просто добиться, как может показаться на первый взгляд. Поэтому вместо генерирования случайных чисел покроем область квадрата сеткой, узлы которой будут играть роль случайных точек. Чем меньше размер ячейки сетки, тем выше точность вычислений. В листинге 2.16 приведен программный код, в котором решается эта задача.

Листинг 2.16. Вычисление числа π

```
class FindPi{
public static void main(String args[]){
// Количество базовых линий сетки:
int N=100000;
// Индексные переменные:
int i,j;
```

продолжение

Листинг 2.16 *(продолжение)*

```
// Счетчик попавших в круг точек:
long count=0;
// Координаты точек и число "пи":
double x,y,Pi;
// Подсчет точек:
for(i=0;i<=N;i++){
    for(j=0;j<=N;j++){
    x=(double)i/N;
    y=(double)j/N;
    if((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)<=0.25) count++;
    }
}
// Число "пи":
Pi=(double)4*count/(N+1)/(N+1);
// Вывод на экран результата:
System.out.println("Вычисление значения по "+(long)(N+1)*(N+1)+" точкам:");
System.out.println(Pi);}
}
```

Хотя используется достаточно большое количество точек, результат оставляет желать лучшего:

```
Вычисление значения по 10000200001 точкам:
3.141529585494137
```

В программе инициализируется целочисленная переменная `N`, которая определяет количество базовых линий сетки по каждой из координат. Общее количество точек в этом случае внутри квадрата равно `(N+1)*(N+1)`. Это число может быть достаточно большим. Сравнимое с ним число — количество точек, которые попадают внутрь круга. Поэтому переменная `count`, которая предназначена для подсчета количества попавших внутрь круга точек, объявляется как принадлежащая типу `long`. Кроме целочисленных индексных переменных `i` и `j`, в программе объявляются переменные `x` и `y` для вычисления координат точек и переменная `Pi` для записи вычисляемого значения числа π.

Внутри вложенной инструкции цикла командами `x=(double)i/N` и `y=(double)j/N` вычисляются координаты точки, находящейся в узле на пересечении `i`-й и `j`-й линий. Поскольку при делении оба операнда целочисленные, для вычисления результата в формате с плавающей точкой используется инструкция `(double)` явного приведения типа. Поскольку центр вписанного в квадрат круга имеет координаты (0,5, 0,5), а радиус круга равен 0,5, то критерий того, что точка с координатами (x,y) попадает внутрь круга (или на его границу) имеет вид $(x-0,5)^2+(y-0,5)^2 \le 0,5^2$. Именно это условие проверяется в условной инструкции, и если условие выполнено, значение переменной `count` увеличивается на единицу.

Число π вычисляется командой `Pi=(double)4*count/(N+1)/(N+1)`. Это значение выводится на экран. При выводе на экран значения `(N+1)*(N+1)`, определяющего

общее количество точек, для приведения к соответствующему формату использована команда (long). Как уже отмечалось, даже если значение переменной N не выходит за допустимые границы диапазона для типа int, это может произойти при вычислении значения (N+1)*(N+1).

Следует отметить, что предложенный способ вычисления числа π является не самым оптимальным. Дело в том, что для достижения мало-мальски приемлемой точности приходится использовать достаточно большое количество точек, что существенно сказывается на времени выполнения программы. Существуют и другие подходы.

Другой метод вычисления числа π, который мы здесь рассмотрим, базируется на применении ряда Фурье. В частности, можно воспользоваться тем, что на интервале от 0 до 2π имеет место разложение в ряд Фурье:

$$x = 4\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\sin\left(\frac{nx}{2}\right).$$

Если в этом разложении положить $x = \pi$, получим $\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$. В программном коде, приведенном в листинге 2.17, для получения числа π вычисляется соответствующая сумма.

Листинг 2.17. Вычисление числа π на основе ряда Фурье

```
class FindPi2{
public static void main(String args[]){
// Количество слагаемых и индексная переменная:
int N=5000000,k;
// Начальное значение и добавка:
double Pi=0,q=4;
// Вычисление числа "пи":
for(k=0;k<=N;k++){
Pi+=q/(2*k+1);
q*=(-1);
}
// Вывод результата на экран:
System.out.println("Вычисление по "+N+" слагаемым ряда:");
System.out.println(Pi);}
}
```

В результате выполнения программы получаем:

```
Вычисление по 5000000 слагаемым ряда:
3.1415928535897395
```

Точность по сравнению с предыдущим способом вычисления выше, хотя количество слагаемых в сумме, которые при этом пришлось учесть, достаточно велико.

Еще один способ вычисления числа π основан на получении произведения. В частности, используем соотношение:

$$\pi = 2 \times \frac{2}{\sqrt{2}} \times \frac{2}{\sqrt{2+\sqrt{2}}} \times \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \ldots$$

В листинге 2.18 представлен программный код, в котором данное бесконечное произведение используется для вычисления значения π.

Листинг 2.18. Вычисление числа π на основе произведения

```
class FindPi3{
public static void main(String args[]){
// Количество множителей и индексная переменная:
int N=20,k;
// Начальное значение и итерационный множитель:
double Pi=2,q=Math.sqrt(2);
// Вычисление числа "пи":
for(k=1;k<=N;k++){
Pi*=2/q;
q=Math.sqrt(2+q);}
// Вывод результата на экран:
System.out.println("Вычисление по "+N+" множителям:");
System.out.println(Pi);}
}
```

В этом случае получаем для значения числа π:

```
Вычисление по 20 множителям:
3.1415926535886207
```

Обращаем внимание, что такой достаточно неплохой по точности результат получен на основе относительно малого количества множителей. Что касается непосредственно алгоритма получения значения числа π, то его основу составляет инструкция цикла, в которой вычисляется произведение, используемое как оценка для числа π. Результат записывается в переменную `Pi`, начальное значение которой равно 2. При вычислении произведения учтено то свойство, что каждый новый множитель представляет собой дробь. В числителе дроби двойка, а знаменатель дроби может быть получен на основе знаменателя предыдущего множителя, если к этому знаменателю добавить 2 и извлечь из результата квадратный корень. Для записи значения знаменателя на каждом из итерационных шагов используется переменная `q` с начальным значением $\sqrt{2}$. В теле инструкции цикла всего две команды. Командой `Pi*=2/q` на основе данного значения множителя изменяется значение переменной-результата `Pi`, а затем командой `q=Math.sqrt(2+q)` изменяется знаменатель для следующего множителя.

Метод последовательных итераций

В следующей программе с помощью инструкция цикла методом последовательных итераций решается алгебраическое уравнение. Для уравнения вида $x = f(x)$, решаемого относительно переменной x, применение метода последовательных итераций подразумевает выполнение следующих действий. Для переменной x задается начальное приближение $x0$, то есть $x = x0$. Каждое следующее приближение вычисляется на основании предыдущего. Если на n-м шаге приближение для корня уравнения есть $x\,n$, то приближение $x\,n + 1$ на следующем шаге вычисляется как $x\,n + 1 = f(x\,n)$.

Для того чтобы соответствующая итерационная процедура сходилась с корнем уравнения, необходимо, чтобы на области поиска корня выполнялось условие:

$$\left|\frac{df(x)}{dx}\right| < 1.$$

В листинге 2.19 приведен программный код, с помощью которого методом последовательных итераций решается уравнение $x = \frac{x^2 + 10}{7}$ с заданным начальным приближением. Корнями этого квадратного уравнения являются значения $x = 2$ и $x = 5$. В данном случае уравнение представлено в виде $x = f(x)$, где функция $f(x) = \frac{x^2 + 10}{7}$. Поскольку $\frac{df(x)}{dx} = \frac{2x}{7}$, то для такого представления уравнения методом последовательных итераций можно искать корень, попадающий в интервал значений $-2{,}5 < x < 2{,}5$, то есть корень $x = 2$.

Листинг 2.19. Решение уравнения методом последовательных итераций

```
class MyEquation{
public static void main(String args[]){
// Начальное приближение:
double x0=0;
// Переменные для корня и функции:
double x,f;
// Погрешность:
double epsilon=1E-10;
// Ограничение на количество итераций:
int Nmax=1000;
// Итерационная переменная:
int n=0;
// Начальное значение для функции:
f=x0;
do{
// Изменение индексной переменной:
n++;
```

продолжение

Листинг 2.19 *(продолжение)*

```
// Новое приближение для корня:
x=f;
// Новое значение для функции (корня):
f=(x*x+10)/7;}
// Проверяемое условие:
while((n<=Nmax)&&(Math.abs(x-f)>epsilon));
// "Последняя" итерация:
x=f;
// Вывод результатов на экран:
System.out.println("Решение уравнения:");
System.out.println("x="+x);
System.out.println("Количество итераций: "+(n+1));}
}
```

Используемая в программе итерационная процедура выполняется до тех пор, пока не будет достигнута необходимая точность вычислений либо общее количество итераций превысит установленную верхнюю границу. Верхняя граница для значения итерационной переменой определяется значением переменной `Nmax`. Погрешность корня задается значением переменной `epsilon`. В принципе, можно применять более точные оценки погрешности вычисляемого корня. Здесь в качестве оценки для погрешности учитывается разница между значениями корней на разных итерациях $|x_{n+1} - x_n|$. Другими словами, это не точность корня, а значение приращения корня (по абсолютной величине). Чтобы контролировать эту величину в программе, необходимо иметь две переменных: текущее значение корня (переменная `x`) и следующее значение корня (переменная `f`). В теле инструкции цикла каждое новое значение переменной `x` вычисляется командой `x=f`, а следующее значение для корня, записываемое в переменную `f`, — командой `f=(x*x+10)/7`. В инструкции цикла проверяется условие `(n<=Nmax)&&(Math.abs(x-f)>epsilon)`. Значение этого выражения равняется `true`, если индексная переменная `n` не превышает значение `Nmax` и если разность `x-f` по абсолютной величине превышает значение переменной `epsilon`.

В случае приведенных в программном коде значений для начального приближения корня и точности вычислений получаем следующий результат:

```
Решение уравнения:
x=1.9999999999205635
Количество итераций: 42
```

В данном случае работа программы завершена из-за того, что приращение корня стало меньше значения переменной `epsilon`.

Решение квадратного уравнения

Рассмотрим программу, в которой решается квадратное уравнение, то есть уравнение вида $a\,x2 + b\,x + c = 0$. В известном смысле задача банальная — корнями уравнения являются значения $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ при условии, что соответствующие

значения вычислимы. Рассмотрим наиболее общую ситуацию, когда параметры a, b и c могут принимать любые действительные значения. Можно выделить следующие особые ситуации, в которых формальное применение приведенных решений невозможно.

- Параметр $a = 0$. В этом случае уравнение не является квадратным — отсутствует слагаемое с x 2. Имеем дело с линейным уравнением вида $bx + c = 0$. Несмотря на кажущуюся простоту, это уравнение также имеет свои «подводные камни».
- Если параметр b отличен от нуля (при условии, что $a = 0$), то уравнение имеет решение $x = -c/b$. Если же $b = 0$, то возможны два варианта: отсутствие решения при $c \neq 0$ или любое число, если $c = 0$.
- В случае если параметр $a \neq 0$, выделяем три ситуации, определяемые знаком дискриминанта $D = b^2 - 4ac$. При $D < 0$ квадратное уравнение на множестве действительных чисел решений не имеет. Если $D = 0$, квадратное уравнение имеет единственный корень $x = -\frac{b}{2a}$. Наконец, при $D > 0$ уравнение имеет два решения — это $x = -\frac{b \pm \sqrt{D}}{2a}$.

Все эти варианты обрабатываются в программе, представленной в листинге 2.20.

Листинг 2.20. Решение квадратного уравнения

```
class SqEquation{
public static void main(String args[]){
// Параметры уравнения:
double a=2,b=-3,c=1;
// Корни и дискриминант:
double x1,x2,D;
// Вывод параметров уравнения на экран:
System.out.println("Уравнение вида ax^2+bx+c=0. Параметры:");
System.out.println("a="+a+"\nb="+b+"\nc="+c);
// Если a равно 0:
if(a==0){System.out.println("Линейное уравнение!");
         // Если a равно 0 и b не равно 0:
         if(b!=0){System.out.println("Решение x="+(-c/b)+".");}
         // Если a, b, и c равны нулю:
         else{if(c==0){System.out.println("Решение - любое число.");}
         // Если a и b равны нулю, а c - нет:
              else{System.out.println("Решений нет!");}
             }
         }
// Если a не равно 0:
else{System.out.println("Квадратное уравнение!");
     // Дискриминант (значение):
     D=b*b-4*a*c;
```

продолжение

Листинг 2.20 *(продолжение)*

```
    // Отрицательный дискриминант:
    if(D<0){System.out.println("Действительных решений нет!");}
    // Нулевой дискриминант:
    else{if(D==0){System.out.println("Решение x="+(-b/2/a));}
    // Положительный дискриминант:
         else{x1=(-b-Math.sqrt(D))/2/a;
              x2=(-b+Math.sqrt(D))/2/a;
              System.out.println("Два решения: x="+x1+" и x="+x2+".");
              }
         }
    }
// Завершение работы программы:
System.out.println("Работа программы завершена!");}
}
```

Основу программы составляет несколько вложенных условных инструкций, в которых последовательно проверяются условия наличия у уравнения решений. Для представленных в тексте программы значений параметров результат выполнения программы имеет вид:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=2.0
b=-3.0
c=1.0
Квадратное уравнение!
Два решения: x=0.5 и x=1.0.
Работа программы завершена!
```

В случае если квадратное уравнение имеет одно решение, результат выполнения программы может быть таким:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=1.0
b=-2.0
c=1.0
Квадратное уравнение!
Решение x=1.0
Работа программы завершена!
```

Если квадратное уравнение не имеет решений на множестве действительных чисел, результат может быть таким:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=2.0
b=-3.0
c=6.0
Квадратное уравнение!
Действительных решений нет!
Работа программы завершена!
```

Если значение переменной a равно нулю, получаем линейное уравнение:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=0.0
b=-3.0
c=6.0
Линейное уравнение!
Решение x=2.0.
Работа программы завершена!
```

Линейное уравнение может не иметь решений:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=0.0
b=0.0
c=1.0
Линейное уравнение!
Решений нет!
Работа программы завершена!
```

Решением может быть и любое число:

```
Уравнение вида ax^2+bx+c=0. Параметры:
a=0.0
b=0.0
c=0.0
Линейное уравнение!
Решение - любое число.
Работа программы завершена!
```

Правда, последний случай достаточно экзотический и реализуется, когда все параметры уравнения равны нулю — в этом случае уравнение превращается в тождество.

Полет в атмосфере

Рассмотрим еще одну задачу, в которой вычисляется траектория движения тела, брошенного под углом к горизонту с известной начальной скоростью при условии, что на тело, кроме силы тяжести, действует еще и сила сопротивления воздуха. Предполагаем, что в атмосфере в зависимости от высоты над поверхностью сила сопротивления воздуха различна: в первом нижнем слое она пропорциональна квадрату скорости тела (и направлена против вектора скорости), во втором, центральном слое, сила сопротивления воздуха пропорциональна скорости тела, а в третьем, верхнем слое атмосферы, сила сопротивления воздуха отсутствует.

В программе задаются начальная скорость тела V, угол к горизонту α, под которым тело брошено, масса тела m, высота H_1 (на этой высоте заканчивается первый слой), высота H_2 (на ней заканчивается второй слой), ускорение свободного падения g, коэффициенты пропорциональности γ_1 и γ_2 для силы

сопротивления воздуха в первой и второй зонах соответственно. По этим параметрам рассчитываются максимальная высота подъема H_{max}, дальность S_{max} и время полета тела T_{max}. Для вычислений используется дискретная модель — самый простой ее вариант.

В рамках дискретной модели исходными являются дифференциальные уравнения второго порядка (уравнения Ньютона), описывающие движение тела по каждой из координатных осей (вдоль горизонтали и вертикали). Не вдаваясь в физические детали, отметим, что это уравнения вида $m\ddot{x} = -F_x(\dot{x}, \dot{y})$ и $m\ddot{y} = -mg - F_y(\dot{x}, \dot{y})$. Точка означает производную по времени, x и y — координаты тела как функции времени, $F_x(\dot{x}, \dot{y})$ и $F_y(\dot{x}, \dot{y})$ — модули проекции силы сопротивления воздуха, которая, в силу условия, зависит только от скорости тела и неявно — от высоты тела, то есть от координаты y.

В рамках дискретной модели предполагаем, что время изменяется дискретно с интервалом Δt. В этом случае моменты времени можно нумеровать целыми числами. Для n-го момента времени $t_n = n\Delta t$ обозначим координаты тела как x_n и y_n, а проекции скорости на координатные оси — соответственно как V_n и U_n. Задача состоит в том, что по известным значениям для координат и скорости на n-м шаге определить эти параметры на $(n + 1)$-м шаге. Несложно показать, что для этого можно воспользоваться соотношениями:

$$x_{n+1} = x_n + V_n \Delta t,$$

$$y_{n+1} = y_n + U_n \Delta t,$$

$$V_{n+1} = V_n - \frac{F_x(\dot{x}, \dot{y})}{m} \Delta t,$$

$$U_{n+1} = U_n - g\Delta t - \frac{F_y(\dot{x}, \dot{y})}{m} \Delta t.$$

В начальный момент, то есть при $n = 0$, $x_0 = 0$, $y_0 = 0$, $V_0 = V\cos(\alpha)$ и $U_0 = V\sin(\alpha)$, где V — модуль вектора начальной скорости, а α — угол горизонта, под которым тело брошено.

Что касается проекций силы сопротивления воздуха, то для первой воздушной зоны (первый слой, определяется условием $y < H_1$) проекции силы сопротивления воздуха определяются соотношениями:

$$F_x = \gamma_1 V_n \sqrt{V_n^2 + U_n^2},$$

$$F_y = \gamma_1 U_n \sqrt{V_n^2 + U_n^2}.$$

Для второй зоны (второй слой определяется условием $H_1 \le y < H_2$) проекции силы сопротивления воздуха определяются соотношениями $F_x = \gamma_2 V_n$

и $F_y = \gamma_2 U_n$. Наконец, для третьей зоны (третий слой определяется условием $H_2 \le y$) $F_x = 0$ и $F_y = 0$. Этой информации вполне достаточно для составления программы. Ее код приведен в листинге 2.21.

Листинг 2.21. Полет тела в атмосфере

```
class BodyFlight{
public static void main(String args[]){
// Ускорение свободного падения (м/с^2):
double g=9.8;
// Масса (кг):
double m=0.1;
// Начальная скорость (м/с):
double V=100;
// Угол в градусах:
double alpha=60;
// Уровни воздушных зон (м):
double H1=100,H2=300;
// Коэффициенты силы сопротивления (Нс/м и Нс^2/м^2):
double gamma1=0.0001,gamma2=0.0001;
// Интервал времени (сек):
double dt=1E-6;
// Координаты и скорость (м и м/с)
double Xn=0,Yn=0,Vn,Un;
// Проекция силы сопротивления (Н):
double Fx,Fy;
// Время полета (сек), дальность (м) и высота (м):
double Tmax,Smax,Hmax=0;
// Индикатор высоты (номер зоны):
int height;
// Перевод угла в радианы:
alpha=Math.toRadians(alpha);
// Проекции начальной скорости:
Vn=V*Math.cos(alpha);
Un=V*Math.sin(alpha);
for(int n=1;true;n++){
// Координата по вертикали:
Yn+=Un*dt;
// Критерий завершения вычислений и расчетные параметры:
if(Yn<0){
Tmax=Math.round((n-1)*dt*100)/100.0;
Smax=Math.round(Xn*100)/100.0;
Hmax=Math.round(Hmax*100)/100.0;
break;}
// Координата по горизонтали:
Xn+=Vn*dt;
```

продолжение

Листинг 2.21 *(продолжение)*

```
// Максимальная высота:
if(Yn>Hmax) Hmax=Yn;
// Вычисление номера зоны:
height=Yn<H1?1:Yn<H2?2:3;
// Сила сопротивления:
switch(height){
    // Первая зона:
    case 1:
    Fx=gamma1*Vn*Math.sqrt(Vn*Vn+Un*Un);
    Fy=gamma1*Un*Math.sqrt(Vn*Vn+Un*Un);
    break;
    // Вторая зона:
    case 2:
    Fx=gamma2*Vn;
    Fy=gamma2*Un;
    break;
    // Третья зона:
    default:
    Fx=0;
    Fy=0;
}
// Проекция скорости по горизонтали:
Vn+=-Fx*dt/m;
// Проекция скорости по вертикали:
Un+=-g*dt-Fy*dt/m;}
// Вывод на экран результатов:
System.out.println("Время полета тела Tmax="+Tmax+" секунд.");
System.out.println("Дальность полета тела Smax="+Smax+" метров.");
System.out.println("Максимальная высота подъема тела Hmax="+Hmax+" метров.");}
}
```

В результате выполнения этой программы получаем следующее:

```
Время полета тела Tmax=15.97 секунды.
Дальность полета тела Smax=705.95 метра.
Максимальная высота подъема тела Hmax=312.31 метра.
```

Назначение переменных, объявленных и использованных в программе, описано в табл. 2.1.

Таблица 2.1. Назначение переменных программы

Переменная	Назначение
g	Переменная, содержащая значение для ускорения свободного падения
m	Масса тела
V	Начальная скорость тела (модуль)
alpha	Угол к горизонту в градусах, под которым брошено тело

Переменная	Назначение
H1	Высота, на которой заканчивается первая воздушная зона. Ниже этой высоты сила сопротивления пропорциональна квадрату скорости
H2	Высота, на которой заканчивается вторая воздушная зона. Ниже этой высоты (но выше первой) сила сопротивления воздуха пропорциональна скорости тела. Выше этого уровня сила сопротивления воздуха отсутствует
gamma1	Коэффициент пропорциональности в выражении для силы сопротивления воздуха в первой воздушной зоне
gamma2	Коэффициент пропорциональности в выражении для силы сопротивления воздуха во второй воздушной зоне
dt	Интервал дискретности времени. Чем меньше значение этой переменной, тем точнее дискретная модель. С другой стороны, это же приводит к увеличению времени расчетов
Xn	Координата тела вдоль горизонтали. Она же определяет расстояние, которое пролетело тело на данный момент времени. В начальный момент времени координата равна нулю
Yn	Координата тела по вертикали. Она же определяет высоту, на которой находится тело в данный момент времени. В начальный момент значение равно нулю. Критерием прекращения вычислений является отрицательность значения этой координаты (вычисления прекращаются, когда координата становится меньше нуля)
Vn	Переменная, в которую записывается проекция скорости тела на горизонтальную ось в данный момент времени
Un	Переменная, в которую записывается проекция скорости тела на вертикальную ось в данный момент времени
Fx	Переменная, в которую записывается проекция силы сопротивления воздуха на горизонтальную ось в данный момент времени
Fy	Переменная, в которую записывается проекция силы сопротивления воздуха на вертикальную ось в данный момент времени
height	Целочисленная переменная, в которую записывается номер воздушной зоны, в которой в данный момент находится тело
Tmax	Переменная, в которую записывается значение времени полета тела
Hmax	Переменная, в которую записывается значение максимальной высоты подъема тела
Smax	Переменная, в которую записывается дальность полета тела

Общая идея, положенная в основу алгоритма вычисления параметров траектории тела, достаточно проста. На начальном этапе координатам и проекциям скорости на координатные оси, исходя из начальных условий, присваиваются значения. Затем запускается инструкция цикла, в рамках которой последовательно в соответствии с приведенными соотношениями изменяются значения для координат и проекций скорости тела. Инструкция цикла выполняется до тех пор, пока вертикальная координата не станет отрицательной. При этом каждый раз

при вычислении вертикальной координаты она сравнивается с текущим значением переменной, в которой записано значение максимальной высоты подъема. Если вычисленная координата больше текущего значения максимальной высоты подъема, вычисленная координата заменяет значение высоты подъема. Поскольку для вычисления новых значений координат и проекций скорости необходимо знать силу сопротивления воздуха, которая зависит от того, в какой зоне находится объект и с какой скоростью движется, в каждом итерационном цикле размещается специальный блок из условных инструкций для вычисления проекций силы сопротивления воздуха на координатные оси. По завершении вычислений результат выводится на экран.

Таким образом, основу программы составляет инструкция цикла. В данном случае она формально бесконечна, поскольку в качестве проверяемого условия в ней указано значение `true`. Первой командой в инструкции цикла на основе текущего значения проекции скорости по вертикальной оси вычисляется новая вертикальная координата. После этого проверяется условие ее отрицательности. Это — критерий завершения работы инструкции цикла. Если условие выполнено, вычисляются характеристики траектории и командой `break` завершается работа инструкции цикла. В частности, для времени полета `Tmax` используется предыдущий момент, когда вертикальная координата еще была неотрицательной. В качестве дальности полета учитывается текущее значение горизонтальной координаты `Xn` (это значение еще осталось «старым» в отличие от измененного значения `Yn`). Также применяется текущее значение переменной `Hmax`. Все три переменные округляются до сотых значений: значение умножается на `100`, округляется с помощью функции `round()`, а затем снова делятся на `100.0` (литерал типа `double`, чтобы не выполнялось целочисленное деление).

Если координата `Yn` неотрицательна, работа инструкции цикла продолжается. В частности, вычисляется новое значение горизонтальной координаты и с помощью условной инструкции проверяется необходимость изменения значения переменной `Hmax`. Командой `height=Yn<H1?1:Yn<H2?2:3` вычисляется номер воздушной зоны, в которой находится тело. Далее по номеру зоны с помощью инструкции `switch()` определяются проекции силы сопротивления воздуха на координатные оси. Разумеется, всю эту процедуру можно было реализовать с помощью вложенных инструкции цикла без непосредственного вычисления номера воздушной зоны, но так получается нагляднее.

После вычисления компонентов вектора силы сопротивления воздуха вычисляются новые значения для проекций скорости на каждую из координатных осей.

Резюме

1. Для создания точек ветвления в алгоритмах используют инструкции цикла и условные инструкции. В Java применяются инструкции цикла `for()`, `while()` и `do-while()`. Условные инструкции Java: `if()` и `switch()`. Последнюю обычно называют инструкцией выбора.

2. Синтаксис вызова условной инструкции `if()`: после ключевого слова `if` в круглых скобках указывается условие (выражение, результатом которого является значение типа `boolean`). Если условие истинно (значение `true`), выполняется блок команд, указанный далее в фигурных скобках. Если условие ложно, выполняется блок команд, размещенный после ключевого слова `else`. Эта часть условной инструкции не является обязательной.
3. В условной инструкции (инструкции выбора) `switch()` в качестве аргумента после ключевого слова `switch` в круглых скобках указывается выражение, значением которого может быть число или символ. В зависимости от значения этого выражения выполняется один из блоков `case` инструкции. Такой блок состоит из ключевого слова `case` и значения, которое может принимать выражение. Выполняется блок команд от соответствующей инструкции `case` до конца инструкции или до появления команды `break`. В конце инструкции может также указываться команда `default`, после которой указывается блок команд, выполняемых, если ни одного совпадения не найдено.
4. Синтаксис вызова инструкции цикла `for()` следующий. В круглых скобках после ключевого слова `for` указывается три блока команд. Блоки разделяются точкой с запятой. В первом блоке (блок инициализации) размещаются команды, выполняемые один раз в начале выполнения инструкции цикла. Второй блок содержит условие. Инструкция цикла выполняется, пока истинно условие. В третьем блоке обычно размещаются команды для изменения индексной переменной. После этого в фигурных скобках указывается блок команд тела цикла, выполняемых за каждую итерацию. Первый и третий блоки могут содержать по несколько команд. Команды в одном блоке разделяются запятыми. Допускается использование пустых блоков. Выполняется инструкция цикла по следующей схеме: сначала один раз выполняются команды первого блока, затем проверяется условие, выполняются команды тела цикла, затем выполняются команды третьего блока, проверяется условие во втором блоке и т. д.
5. Инструкция цикла `while()` работает по следующей схеме. Сначала проверяется условие, указанное в круглых скобках после ключевого слова `while`. Если условие истинно (значение `true`), выполняются команды в теле инструкции цикла (в фигурных скобках). Затем снова проверяется условие и т. д. Как только при проверке условия оказывается, что оно не выполнено (значение `false`), управление передается следующей инструкции после условной.
6. При вызове инструкции `do-while()` используется следующий синтаксис. После ключевого слова `do` в фигурных скобках указывается блок команд основного тела инструкции. После этого указывается ключевое слово `while` и в круглых скобках проверяемое условие. Заканчивается инструкция точкой с запятой. Принцип выполнения этой инструкции такой же, как инструкции `while()`, с той лишь разницей, что сначала выполняются команды основного тела цикла, а затем проверяется условие.

Глава 3. Массивы

Ну и что вы скажете обо всем этом, Ватсон?

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как известно, под массивом подразумевают набор однотипных данных (переменных), к которым можно обращаться по общему имени. Массивы бывают статическими и динамическими. Под статические массивы память выделяется при компиляции программы. Для динамических массивов память выделяется в процессе выполнения программы. В Java все массивы динамические!

Создание одномерного массива

Это дело очень интересное. И простое!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Переменные, относящиеся к одному массиву, называются элементами этого массива. Чтобы однозначно идентифицировать элемент массива, необходимо знать имя массива и позицию (размещение) элемента в массиве. Позиция элементов в массиве определяется с помощью целочисленных индексов. Количество индексов, необходимых для идентификации элемента массива, называется размерностью массива. Одномерный массив — это такой массив, в котором идентификация элементов осуществляется с помощью одного индекса.

Для объявления одномерного массива необходимо задать тип, к которому относятся элементы массива, название массива, а также количество элементов, входящих в массив. Синтаксис объявления одномерного массива имеет вид:

```
тип[] имя=new тип[размер];
```

Вначале указывается `тип` элементов массива. Обращаем внимание, что для массива после идентификатора типа ставятся пустые квадратные скобки. Далее следует `имя` массива, оператор присваивания, инструкция динамического выделения памяти `new`, снова `тип` элементов массива и в квадратных скобках `размер` массива

(количество элементов в массиве). Например, командой `int nums=new int[20]` объявляется целочисленный массив `nums` из 20 элементов.

Строго говоря, представленная здесь команда объявления массива является симбиозом двух команд: команды `int[] nums` объявления переменной `nums` типа «целочисленный массив» и команды `nums=new int[20]` выделения памяти под массив и присваивания ссылки на этот массив переменной `nums`. Другими словами, процесс объявления массива можно выполнить двумя командами:

```
int[] nums;
nums=new int[20];
```

Допускается указывать квадратные скобки либо после имени типа массива, либо после имени массива. Например, вместо команды `int[] nums` можно использовать команду `int nums[]`.

Обращение к элементу одномерного массива осуществляется через имя массива с указанием в квадратных скобках индекса элемента. Индексация элементов массива начинается с нуля. Таким образом, ссылка на первый элемент массива `nums` будет иметь вид `nums[0]`. Если в массиве 20 элементов, то последний элемент массива имеет индекс 19, то есть `nums[19]`.

Длину массива можно узнать с помощью свойства `length`. Это такая переменная, которая создается при объявлении массива, и ее значением является количество элементов массива. Поскольку для каждого массива создается своя переменная `length`, обращение к таким переменным осуществляется с одновременным указанием имени массива. В частности, сначала указывается имя массива, а затем, через точку, имя переменной `length`. Например, чтобы в программе узнать значение длины массива `nums`, можно воспользоваться инструкцией `nums.length`. Ссылка на последний элемент массива может быть записана как `nums[nums.length-1]`, поскольку индекс последнего элемента на единицу меньше длины массива.

Здесь уместно будет обратить внимание читателей, знакомых с языком программирования C++, что в Java, в отличие от C++, выполняется автоматическая проверка факта выхода за пределы массива. Поэтому если в программном коде по ошибке выполняется обращение к несуществующему элементу массива, программа не скомпилируется.

При объявлении массива для него выделяется память. В Java элементы массива автоматически инициализируются с «нулевыми» значениями — выделенные ячейки обнуляются, а значения этих обнуленных ячеек интерпретируются в зависимости от типа массива. Тем не менее на такую автоматическую инициализацию полагаться не стоит. Разумно инициализировать элементы массива в явном виде. Для этого используют инструкцию цикла или задают список значений элементов при объявлении массива.

При инициализации массива списком значений при объявлении переменной массива после нее указывается (через оператор присваивания) заключенный в фигурные скобки список значений. Например:

```
int[] data={3,8,1,7};
```

Если в квадратных скобках размер массива не указан, он определяется автоматически в соответствии с количеством элементов в списке значений. В данном случае создается целочисленный массив data из четырех элементов со значениями data[0]=3, data[1]=8, data[2]=1 и data[3]=7. Того же результата можно добиться, воспользовавшись такими командами:

```
int[] data;
data=new int[]{3,8,1,7};
```

Первой командой int[] data объявляется переменная массива. Командой new int[]{3,8,1,7} создается массив из четырех целых чисел, а ссылка на этот массив присваивается в качестве значения переменной data командой data=new int[]{3,8,1,7}.

Пример объявления, инициализации и использования массивов приведен в листинге 3.1.

Листинг 3.1. Объявление и инициализация одномерного массива

```
class MyArray{
public static void main(String[] args){
// Индексная переменная и размер массива:
int i,n;
// Объявление переменной массива:
int[] data;
// Инициализация массива:
data=new int[]{3,8,1,7};
// Длина второго массива:
n=data.length;
// Объявление второго массива:
int[] nums=new int[n];
// Заполнение второго массива:
for(i=0;i<n;i++){
nums[i]=2*data[i]-3;
System.out.println("nums["+i+"]="+nums[i]);}
}
}
```

В программе объявляется и инициализируется массив data из четырех элементов. Длина массива, возвращаемая инструкцией data.length, присваивается в качестве значения целочисленной переменной n (команда n=data.length). Далее командой int[] nums=new int[n] объявляется целочисленный массив nums. Количество элементов в этом массиве определяется значением переменной n, поэтому совпадает с размером массива data. Заполнение массива nums выполняется с помощью инструкции цикла. Значения элементов массива nums заполняются на основе значений элементов массива data (командой nums[i]=2*data[i]-3). Вычисленные значения выводятся на экран командой System.out.println("nums["+i+"]="+nums[i]).

В результате выполнения программы получаем:

```
nums[0]=3
nums[1]=13
nums[2]=-1
nums[3]=11
```

Еще раз обращаем внимание читателя на то, что индексация элементов массива начинается с нуля. Поэтому в инструкции цикла индексная переменная `i` инициализируется с начальным нулевым значением, а в проверяемом условии (`i<n`) использован оператор строгого неравенства.

Немаловажно и то обстоятельство, что при создании массива `nums` его размер определяется с помощью переменной `n`, значение которой вычисляется в процессе выполнения программы. Такой способ определения размера массива возможен исключительно благодаря тому, что массивы в Java динамические.

Двухмерные и многомерные массивы

— Вы хотите сказать, что вам уже все ясно?
— Не хватает некоторых деталей. Но не в этом суть!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В Java массивы могут иметь размерность выше единичной. Но на практике массивы размерности выше второй используют редко. Вначале рассмотрим способы объявления, инициализации и использования двухмерных массивов.

Двухмерный массив в Java с технической точки зрения является одномерным массивом, элементами которого являются также одномерные массивы. Это на первый взгляд несущественное обстоятельство приводит к исключительной гибкости в использовании таких структур, как двухмерные массивы.

Объявляются двухмерные массивы практически так же, как и одномерные, с той лишь разницей, что при этом используются две пары квадратных скобок (как при объявлении переменной массива, так и при выделении для массива области памяти). При этом размер массива указывается по каждому из индексов. Синтаксис объявления двухмерного массива может быть следующим:

```
тип[][] имя=new тип[размер_1][размер_2];
```

Как и в случае одномерного массива, данная команда представляет собой объединение двух отдельных команд:

```
тип[][] имя;
имя=new тип[размер_1][размер_2];
```

Первой из них объявляется переменная двухмерного массива `имя`. Второй командой создается двухмерный массив с размерами `размер_1` и `размер_2`, а ссылка

на этот массив присваивается в качестве значения переменной массива имя. Например, командой double[][] data=new double[3][4] создается массив с именем data. Элементами массива являются значения типа double. Размер массива по первому индексу равен 3, а по второму — 4. К тому же результату приведет выполнение команд

```
double[][] data;
data=new double[3][4];
```

Обращение к элементам двухмерного массива выполняется в следующем формате: указывается имя массива, в квадратных скобках первый индекс элемента и в квадратных же скобках второй элемент массива. Индексация по всем размерностям начинается с нуля. Например, ссылка data[0][3] является обращением к элементу массива data с индексами 0 и 3.

Для инициализации двухмерного массива используют вложенные инструкции цикла или список значений, заключенный в фигурные скобки. Элементами списка являются заключенные в фигурные скобки списки значений элементов по каждому из индексов. Пример инициализации двухмерного массива с помощью списка значений:

```
double data[][]={{0.1,0.2,0.3},{0.4,0.5,0.6}};
int nums[][]={{1,2,3},{4,5}};
```

Первой командой создается и инициализируется двухмерный массив data размерами 2 на 3 (по первому индексу размер массива 2, по второму индексу — 3). Другими словами, массив data — это массив из двух элементов, которые, в свою очередь, являются массивами из трех элементов. Так, элемент data[0][0] получает значение 0.1, элемент data[0][2] — значение 0.3, элемент data[1][0] — значение 0.4, а элемент data[1][2] — значение 0.6.

Интереснее вторая команда. Этой командой создается целочисленный массив nums, который состоит из двух элементов-массивов. Однако первый массив имеет размерность 3, а второй — 2! Здесь мы находим подтверждение того, что в Java двухмерные массивы не обязаны быть прямоугольными, то есть иметь такую же размерность по второму индексу. В данном случае элемент nums[0][0] имеет значение 1, элемент nums[0][1] — значение 2, элемент nums[0][2] — значение 3, элемент nums[1][0] — значение 4, а элемент nums[1][1] — значение 5. Элемента nums[1][2] не существует вообще!

В листинге 3.2 приведен пример программы, в которой создается двухмерный массив и инициализируется с помощью вложенных инструкций цикла.

Листинг 3.2. Объявление и инициализация двухмерного массива

```
class MyDArray{
public static void main(String[] args){
// Индексные переменные и размерность массива:
int i,j,n=3;
```

```
// Создание двухмерного массива:
int[][] nums=new int[n-1][n];
// Вложенные инструкции цикла:
for(i=0;i<n-1;i++){
for(j=0;j<n;j++){
// Заполнение элементов массива:
nums[i][j]=10*(i+1)+j+1;
// Вывод значений в одну строку:
System.out.print(nums[i][j]+" ");}
// Переход на новую строку
System.out.println();}
}
}
```

Командой `int[][] nums=new int[n-1][n]` создается целочисленный массив `nums` с размерами `n-1` по первому индексу и `n` по второму. Переменная `n` предварительно инициализирована со значением 3. Заполняется массив с помощью вложенных инструкций цикла (команда `nums[i][j]=10*(i+1)+j+1`). Значения элементов массива выводятся на экран. В результате выполнения программы получаем:

```
11 12 13
21 22 23
```

В листинге 3.3 приведен код программы, в которой создается двухмерный «непрямоугольный» массив.

Листинг 3.3. Создание непрямоугольного массива

```
class ArrayDemo{
public static void main(String[] args){
// Индексные переменные и размер массива:
int i,j,n;
// Создание массива (второй размер не указан):
int[][] nums=new int[5][];
// Определение первого размера массива:
n=nums.length;
// Цикл для создания треугольного массива:
for(i=0;i<n;i++){
nums[i]=new int[i+1];}
// Вложенные циклы для заполнения элементов массива:
for(i=0;i<n;i++){
for(j=0;j<nums[i].length;j++){
// Присваивание значения элементу массива:
nums[i][j]=10*(i+1)+j+1;
// Вывод значения на экран:
System.out.print(nums[i][j]+" ");}
```

продолжение

Листинг 3.3 *(продолжение)*

```
// Переход на новую строку:
System.out.println();}
}
}
```

Обращаем внимание читателя на команду `int[][] nums=new int[5][]`, которой создается двухмерный целочисленный массив `nums`. Этот массив состоит из пяти элементов, каждый из которых также является массивом. Однако размер этих массивов не указан — вторая пара квадратных скобок в конце программы пуста! Определяются размеры каждого из элементов-массивов в рамках инструкции цикла, но предварительно переменной `n` командой `n=nums.length` присваивается значение размера массива по первому индексу. Ранее уже упоминалось, что двухмерный массив является массивом массивов. Поэтому ссылка `nums.length` возвращает размер массива `nums`, то есть число 5 в данном случае.

В первой инструкции цикла индексная переменная `i` получает значения от `0` до `n-1`. Командой `nums[i]=new int[i+1]` определяются размеры каждого из массивов — элементов массива `nums`. Учитывая, что `nums` является двухмерным массивом, инструкция вида `nums[i]` является ссылкой на `i`-й одномерный элемент-массив массива `nums`. Командой `new int[i+1]` выделяется место в памяти для массива, размер этого массива устанавливается равным `i+1`, а ссылка на массив записывается в переменную `nums[i]`. В результате мы получаем двухмерный массив «треугольного» вида: в первой «строке» массива один элемент, во второй — два элемента и т. д., до пятой «строки» массива.

С помощью вложенных инструкций цикла выполняется заполнение созданного массива. Внешняя индексная переменная `i` получает значения от 0 до `n-1` и определяет первый индекс двухмерного массива `nums`. Верхняя граница диапазона изменения второй индексной переменной `j`, определяющей второй индекс элемента массива `nums`, зависит от текущего значения переменной `i`. Для определения размера массива `nums[i]` используется инструкция `nums[i].length`. Индекс `j` изменяется от 0 до `nums[i].length-1`. Значение элементам массива присваивается командой `nums[i][j]=10*(i+1)+j+1`. Результат выполнения программы имеет вид:

```
11
21 22
31 32 33
41 42 43 44
51 52 53 54 55
```

Практически также создаются многомерные (размерности выше второй) массивы. В листинге 3.4 приведен пример создания трехмерного массива размером три по каждому из индексов, определяющего тензор Леви–Чевита. Компоненты этого тензора имеют три индекса и отличны от нуля, только если все индексы различны. Элемент с индексами 0, 1 и 2 равен единице. Любой элемент, который получается циклической перестановкой этих индексов, также равен 1. Прочие

элементы равны –1. Таким образом, всего три единичных элемента и три элемента со значением –1, остальные равны нулю.

Листинг 3.4. Создание трехмерного массива

```
class MyTArray{
public static void main(String[] args){
// Индексные переменные:
int i,j,k;
// Объявление трехмерного массива:
byte[][][] epsilon=new byte[3][3][3];
// Обнуление элементов массива:
for(i=0;i<3;i++)
for(j=0;j<3;j++)
for(k=0;k<3;k++)
epsilon[i][j][k]=0;
// Единичные элементы массива:
epsilon[0][1][2]=epsilon[1][2][0]=epsilon[2][0][1]=1;
// Элементы со значением -1:
epsilon[1][0][2]=epsilon[0][2][1]=epsilon[2][1][0]=-1;
}
}
```

Объявляется трехмерный массив `epsilon` командой `byte[][][] epsilon=new byte[3][3][3]`. Для надежности всем элементам массива присваиваются нулевые значения, для чего используется три вложенных инструкции цикла. Далее командой `epsilon[0][1][2]=epsilon[1][2][0]=epsilon[2][0][1]=1` задаются единичные значения для трех элементов массива и еще для трех элементов значение `-1` (командой `epsilon[1][0][2]=epsilon[0][2][1]=epsilon[2][1][0]=-1`).

Символьные массивы

— Что вы делаете?
— Не видите? Стреляю!
— Странный способ украшать дом
монограммой королевы.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В языке программирования C++ символьные массивы имеют особый ареол, поскольку там символьные массивы — одно из средств реализации текстовых строк. В Java в отношении символьных массивов все обстоит намного проще — это обычный, с точки зрения создания и применения, массив, элементами которого являются символы. Правда, и в Java в использовании символьных массивов есть некоторые особенности. Однако эти особенности скорее связаны

со спецификой типа char, чем со структурой или свойствами непосредственно массива. В листинге 3.5 приведен простой пример.

Листинг 3.5. Символьный массив

```
class CharArray{
public static void main(String[] args){
char[] words=new char[]
{'С','и','м','в','о','л','ь','н','ы','й',' ','м','а','с','с','и','в'};
System.out.println(words);
}
}
```

Символьный массив создается стандартным способом: одновременно с объявлением переменной массива words списком символов инициализируются элементы массива. В результате выполнения команды System.out.println(words) на экран выводится сообщение Символьный массив. Интересно здесь то, что для вывода значений элементов символьного массива аргументом метода println() указано имя массива (переменная массива words). Причина такого удобства кроется в способах автоматического преобразования разных объектов (в том числе символьного массива) в текстовый формат. Эта тема рассматривается в главе 8, посвященной работе с текстом (классы String и StringBuffer).

Другой пример объявления и использования символьных массивов приведен в листинге 3.6.

Листинг 3.6. Кодирование слова

```
class CharArray2{
public static void main(String[] args){
char[] words=new char[]{'С','л','о','в','о'};
char[] code=new char[words.length];
for(int i=0;i<words.length;i++)
code[i]=(char)(words[i]+i+1);
System.out.println(words);
System.out.println(code);
}
}
```

В программе выполняется достаточно простое кодирование слова, записанного по буквам в массив words. Этот массив объявляется и инициализируется значениями элементов, соответствующими слову Слово. Далее объявляется еще один символьный массив code. Его размер равен размеру массива words. Заполнение элементов массива code выполняется в рамках инструкции цикла. Для этого в теле цикла использована команда code[i]=(char)(words[i]+i+1). В данном случае при вычислении выражения words[i]+i+1 символ words[i] преобразуется в числовой формат (код символа) и к нему прибавляется значение i+1. Полученное число благодаря инструкции явного приведения типа

преобразуется в символ. Этот символ записывается в массив code. Далее оба массива выводятся на экран. В результате выполнения программы получаем следующее:

```
Слово
Тнсжу
```

Для «расшифровки» полученного «слова» можно применить обратную процедуру. Предлагаем читателю сделать это самостоятельно.

Присваивание и сравнение массивов

Простые вещи разучились делать!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В принципе, один массив может быть присвоен в качестве значения другому массиву, если их типы совпадают. Массивы также можно сравнивать. Правда, все означенные операции выполняются довольно специфически и в некоторой степени могут удивить. Чтобы свести такое удивление к минимуму, имеет смысл еще раз остановиться на том, что же такое массив и как его техническая реализация связана синтаксическими конструкциями языка Java.

Напомним наиболее существенные моменты, связанные с объявлением массивов в Java.

Переменная, обозначающая массив (переменная массива), объявляется независимо от фактического выделения памяти под массив. Другими словами, непосредственно массив и переменная массива — это далеко не одно и то же. В этом смысле показательным является двухэтапный (двумя командами) процесс создания массива. Например:

```
int[] nums;
nums=new int[]{1,2,3,4};
```

В данном случае команда int[] nums есть не что иное, как объявление переменной nums. Тип этой переменной — «массив целых чисел». Значением переменной может быть ссылка (адрес) на какой-нибудь массив, состоящий из целых чисел.

Оператор new в общем случае служит для динамического выделения памяти под различные объекты, в том числе массивы. Командой new int[]{1,2,3,4} в памяти выделяется место для целочисленного массива из четырех элементов, соответствующие значения присваиваются элементам массива. У этого вновь созданного массива есть адрес (ссылка на массив). В качестве значения оператор new возвращает ссылку на созданный объект. В данном случае возвращается ссылка на массив. Эта ссылка в качестве значения присваивается пере-

менной nums. Теперь несложно догадаться, каким будет результат выполнения следующих команд:

```
int[] nums,data;
nums=new int[]{1,2,3,4};
data=nums;
```

Все достаточно просто. Первой командой int[] nums,data объявляются две переменные массива nums и data. Второй командой nums=new int[]{1,2,3,4} создается массив, а ссылка на него присваивается в качестве значения переменной nums. Далее командой data=nums значение переменной nums присваивается переменной data. Однако значение переменой nums — это ссылка на массив. Поэтому после присваивания переменная data ссылается на тот же массив! Например, элемент data[1] имеет такое же значение, что и nums[1] (значение 2). Точнее, это один и тот же элемент. Более того, если теперь изменить какой-нибудь элемент массива data (например, data[3]=-1), автоматически изменится и соответствующий элемент массива nums. Причина та же — массив на самом деле один, просто на него ссылаются две переменные.

При сравнении массивов с помощью операторов равно == и не равно != (например, nums==data или nums!=data) сравниваются значения переменных массива, а не элементы в этих массивах. Поэтому результатом выражения nums==data является true, если обе переменные массива nums и data ссылаются на один и тот же массив.

Пример программы, в которой имеет место присваивание массива, приведен в листинге 3.7.

Листинг 3.7. Присваивание массива

```
class MyArrayDemo{
public static void main(String[] args){
int i;
int[] nums=new int[10];
int[] data=new int[20];
for(i=0;i<10;i++){
nums[i]=2*i+1;
data[i]=2*i;
data[i+10]=2*(i+10);}
data=nums;
for(i=0;i<data.length;i++)
System.out.print(data[i]+" ");}
}
```

В программе объявляются два целочисленных массива: массив nums из 10 элементов и массив data из 20 элементов. С помощью инструкции цикла эти массивы заполняются: массив nums заполняется нечетными числами, массив data — четными. После этого командой data=nums массиву data в качестве значения присваивается массив nums. Обращаем внимание, что хотя эти массивы имеют

одинаковый тип, у них разные размеры. Далее с помощью еще одной инструкции цикла элементы массива data выводятся с интервалом в одну строку (для вывода значений без перехода к новой строке используем метод print()). В результате мы получаем числовой ряд:

```
1 3 5 7 9 11 13 15 17 19
```

Это те значения, которыми инициализировался массив nums. Интерес в данном случае представляет то обстоятельство, что в инструкции цикла, обеспечивающей вывод значений массива data, верхняя граница для индексов элементов массива определяется через свойство length массива data. Массив инициализировался с размером 20, а в конечном итоге его размер оказался равным 10! Причина очевидна. После выполнения команды data=nums переменная массива data начинает ссылаться на тот же массив, что и переменная массива nums.

Особенности сравнения массивов на предмет равенства (неравенства) иллюстрируются программным кодом листинга 3.8.

Листинг 3.8. Сравнение массивов

```
class MyArrayDemo2{
public static void main(String[] args){
// Объявление массивов:
int[] nums=new int[]{1,2,3,4,5};
int[] data=new int[]{1,2,3,4,5};
// Комментирование следующей команды можно отменить:
// data=nums;
// Проверка совпадения ссылок:
if(data==nums){
System.out.println("Совпадающие массивы!");
return;}
// Проверка размеров массивов:
if(data.length!=nums.length){
System.out.println("Разные массивы!");
return;}
// Поэлементная проверка массивов:
for(int i=0;i<data.length;i++){
if(data[i]!=nums[i]){
System.out.println("Несовпадающие элементы!");
return;}}
System.out.println("Одинаковые массивы!");
}}
```

Программа предназначена для сравнения двух целочисленных массивов. В программе объявляются два целочисленных массива nums и data и инициализируются одинаковыми наборами значений. Далее непосредственно выполняется проверка. Состоит она из трех этапов. Сначала выполняется проверка равенства переменных массивов nums и data. Если ссылки равны, то, очевидно, массивы одинаковы (совпадают). Проверка равенства ссылок на массивы выполняется

с помощью условной инструкции `if()` с условием `data==nums`. При выполненном условии выводится сообщение `Совпадающие массивы!`. При этом работа программы завершается, для чего используется команда `return`.

Если ссылки различны, выполняется поэлементная проверка массивов. Массивы считаются одинаковыми, если у них совпадают соответствующие элементы. Но прежде необходимо проверить, совпадают ли размеры массивов. Проверка равенства размеров массивов также выполняется с помощью условной инструкции `if()`, при этом проверяется условие `data.length!=nums.length`. Условие является истинным, если массивы имеют разные размеры. В этом случае выводится сообщение `Несовпадающие элементы!`, и работа программы завершается.

При совпадающих размерах массивов запускается цикл, в рамках которого сравниваются элементы двух массивов. Для этого использована условная инструкция `if()` с проверяемым условием `data[i]!=nums[i]` (`i` — индексная переменная). Если встречаются несовпадающие элементы, выводится сообщение `Несовпадающие элементы!`, и работа программы завершается.

В случае если два массива имеют только совпадающие элементы, цикл заканчивается без последствий, и, самое главное, работа программы продолжается, поэтому в конце выводится сообщение `Одинаковые массивы!`.

В данном случае для списков инициализации массивов, приведенных в листинге 3.8, в результате выполнения программы появляется сообщение `Одинаковые массивы!`. Программа содержит закомментированную команду `data=nums`. Если отменить комментирование, результатом будет сообщение `Совпадающие массивы!`. Чтобы увидеть прочие сообщения, следует, не отменяя комментирование, внести изменения в списки инициализации массивов.

Примеры программ

В этом разделе рассматриваются некоторые программы, в которых в том или ином виде используются массивы. Для удобства программы разбиты по тематическим группам. Кроме того, некоторые программы, в которых совместно с массивами используются функции (методы) и классы, рассматриваются в следующих главах.

Умножение векторов

Через массивы очень удобно реализовывать в программах операции с векторами. В частности, рассмотрим программу, в которой вычисляется скалярное и векторное произведения векторов. Напомним, что если $\vec{a}$ и $\vec{b}$ — это векторы в Декартовом пространстве, скалярным произведением этих векторов называется число:

$$\vec{a} \times \vec{b} = \sum_{k=1}^{3} a_k b_k .$$

Здесь через a_k и b_k ($k = 1,2,3$) обозначены компоненты этих векторов. Векторным произведением двух векторов $\vec{a}$ и $\vec{b}$ называется вектор:

$$\vec{c} = [\vec{a} \times \vec{b}] = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \vec{i}(a_2 b_3 - a_3 b_2) + \vec{j}(a_3 b_1 - a_1 b_3) + \vec{k}(a_1 b_2 - a_2 b_1).$$

Здесь через $\vec{i}$, $\vec{j}$ и $\vec{k}$ обозначены орты ортогональной системы координат.

При составлении программы векторы реализуются в виде массивов, состоящих из трех компонентов. При вычислении скалярного произведения получаем число. При вычислении векторного произведения создается и заполняется еще один массив из трех элементов. Элементы c_k этого массива могут быть вычислены на основе элементов a_k и b_k исходных массивов по формуле:

$$c_k = a_{k+1} b_{k+2} - a_{k+2} b_{k+1}.$$

Причем здесь выполняется циклическая индексация символов: если индекс выходит за верхнюю допустимую границу диапазона индексации, индекс циклически перемещается в начало диапазона. Например, индексы компонентов вектора изменяются от 1 до 3, поэтому $a_4 \equiv a_1$ и $a_5 \equiv a_2$. Поскольку индексы массива изменяются от 0 до 2, то элементу с индексом 3 в формуле соответствует элемент с индексом 0, а элементу с индексом 4 — элемент с индексом 1.

В листинге 3.9 приведен код программы, в которой вычисляются скалярное и векторное произведения двух векторов.

Листинг 3.9. Произведение векторов

```
class VectProduct{
public static void main(String args[]){
// Объявление массивов для записи векторов:
double[] a,b,c;
// Создание массивов:
a=new double[]{1,2,-1};
b=new double[]{3,-1,2};
c=new double[3];
// Индексная переменная:
int k;
// Переменная для записи скалярного произведения:
double s=0;
System.out.print("Произведение векторов:\n[a.b]=<");
// Вычисление и вывод на экран результата:
for(k=0;k<3;k++){
s+=a[k]*b[k];
c[k]=a[(k+1)%3]*b[(k+2)%3]-a[(k+2)%3]*b[(k+1)%3];
```

продолжение

Листинг 3.9 *(продолжение)*

```
System.out.print(c[k]+(k!=2?";":">\n"));}
System.out.println("a.b="+s);}
}
```

Командой `double[] a,b,c` в программе объявляются три переменных массива. Командами `a=new double[]{1,2,-1}` и `b=new double[]{3,-1,2}` создаются и инициализируются базовые массивы. Командой `c=new double[3]` создается третий массив из трех элементов. В этот массив будет записан результат векторного произведения векторов.

Целочисленная индексная переменная `k` предназначена для использования в инструкции цикла, а для записи скалярного произведения служит переменная `s` типа `double`, которой присваивается начальное нулевое значение.

В инструкции цикла параллельно вычисляются скалярное и векторное произведения. Командой `s+=a[k]*b[k]` к переменной, определяющей результат скалярного произведения, добавляется произведение соответствующих элементов базовых массивов, а командой `c[k]=a[(k+1)%3]*b[(k+2)%3]-a[(k+2)%3]*b[(k+1)%3]` вычисляется соответствующий элемент вектора-результата векторного произведения векторов. Обращаем внимание, что в последнем случае индексы элементов базовых массивов вычисляются как остаток от деления на 3 — это позволяет реализовать процедуру циклической перестановки индекса в случае, если значение индекса выходит за допустимые границы.

Перед инструкцией цикла командой `System.out.print("Произведение векторов:\n[a.b]=<")` выводится начальное сообщение и первая часть текста с результатом вычисления векторного произведения. В текстовом аргументе метода `print()` использована инструкция `\n` для перехода на новую строку, что позволяет разбить выводимый на экран текст на две строки.

В инструкции цикла вывод очередного вычисленного элемента массива-результата векторного произведения реализуется командой `System.out.print(c[k]+(k!=2?";":">\n"))`. В этой команде в аргументе метода `print()` кроме непосредственно элемента массива указывается текстовый фрагмент, отображаемый после этого элемента. Если индекс элемента отличен от 2 (то есть это — не последний элемент массива), отображается точка с запятой. Если выводится последний элемент массива (индекс равен 2), отображается угловая скобка и выполняется переход на новую строку. Для реализации такого текстового разнообразия задействован тернарный оператор, в котором проверяемым условием является отличие индексной переменной от значения 2. Если условие истинно, в качестве результата возвращается текст с точкой с запятой, в противном случае возвращается текст с угловой скобкой и инструкцией перехода на новую строку.

После выполнения инструкции цикла на экран командой `System.out.println("a. b="+s)` выводится результат скалярного произведения векторов. Результат выполнения программы имеет вид:

```
Произведение векторов:
[a.b]=<3.0;-5.0;-7.0>
a.b=-1.0
```

Следует отметить, что подобные операции с векторами лучше все же реализовывать с помощью специальных классов или, по крайней мере, создать для этого несколько методов.

Числа Фибоначчи

В предыдущей главе рассматривалась программа, в которой вычислялись числа из последовательности Фибоначчи. Здесь мы рассмотрим программу, в которой числами Фибоначчи заполняется массив. Программный код приведен в листинге 3.10.

Листинг 3.10. Числа Фибоначчи

```
class FibonacciArray{
public static void main(String args[]){
// Индексная переменная и размер массива:
int k,n=20;
// Массив для чисел Фибоначчи:
int[] Fib=new int[n];
// Первые два числа последовательности:
Fib[0]=1;
Fib[1]=1;
// Вывод первых двух значений на экран:
System.out.print(Fib[0]+" "+Fib[1]);
// Вычисление последовательности и вывод на экран:
for(k=2;k<n;k++){
// Элемент массива вычисляется на основе двух предыдущих:
Fib[k]=Fib[k-1]+Fib[k-2];
// Вывод на экран:
System.out.print(" "+Fib[k]);}}
}
```

Результат выполнения программы имеет вид:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Программа достаточно проста, так что думается, особых комментариев не требует. Стоит лишь, пожалуй, обратить внимание на способ заполнения элементов массива. Значения для первых двух элементов вычисляются в явном виде (у них единичные значения). Затем в инструкции цикла заполняются прочие элементы массива. Каждый новый элемент вычисляется на основе двух предыдущих (это их сумма). Если сравнить этот алгоритм с тем, что использовался в программе из предыдущей главы, то здесь общая схема немного проще. Однако в данном случае приходится запоминать все элементы последовательности и записывать их в массив, в то время как в примере из предыдущей главы запоминались только два последних элемента последовательности.

Работа с полиномами

Массивы могут быть полезными при работе с выражениями полиномиального вида. Напомним, что полиномом степени n называется функция вида:

$$P_n(x) = \sum_{k=0}^{n} a_k x^k.$$

Степенью полинома называется наибольшая степень аргумента x, входящая в полиномиальное выражение. Основная информация о полиноме заложена в коэффициентах a_k, $k = 0,1,...,n$. Фактически, для того чтобы вычислить значение полинома для аргумента x, нужно знать массив коэффициентов a_k.

Далее, при известном массиве коэффициентов полинома можно вычислить и производную от полинома. Производная от полинома также является полиномом и дается соотношением:

$$P'_n(x) = \sum_{k=0}^{n-1} (k+1) a_{k+1} x^k.$$

Это полином степени $n-1$, а коэффициенты полинома-производной определяются на основе коэффициентов исходного полинома. Коэффициент полинома-производной равен $b_k = (k+1)a_{k+1}$ для $k = 0,1,...,n-1$ и $b_n = 0$. В листинге 3.11 приведен программный код, в котором по массиву коэффициентов полинома для заданного аргумента вычисляется значение полинома и производной от этого полинома в этой же точке.

Листинг 3.11. Полином и его производная

```
class Polynom{
public static void main(String args[]){
// Коэффициенты полинома:
double[] a=new double[]{1,-3,2,4,1,-1};
// Массив коэффициентов производной:
double[] b=new double[a.length-1];
// Аргумент и множитель:
double x=2.0,q=1;
// Индексная переменная:
int k;
// Значения полинома и производной:
double P=0,Q=0;
// Вычисление результата:
for(k=0;k<b.length;k++){
// Полином:
P+=a[k]*q;
// Коэффициент производной:
b[k]=(k+1)*a[k+1];
```

```
// Производная:
Q+=b[k]*q;
// Изменение множителя:
q*=x;}
// Последнее слагаемое полинома:
P+=a[a.length-1]*q;
// Вывод результата:
System.out.println("Полином P(x)="+P);
System.out.println("Производная P'(x)="+Q);}
}
```

В программе командой `double[] a=new double[]{1,-3,2,4,1,-1}` объявляется, создается и инициализируется массив с коэффициентами полинома. Командой `double[] b=new double[a.length-1]` объявляется и создается массив для записи коэффициентов полинома-производной. Размер этого массива на единицу меньше размера первого массива. Здесь принята во внимание та особенность, касающаяся производной, что этот полином степени на единицу меньше степени исходного полинома. Размер первого полинома возвращается инструкцией `a.length`.

Переменная `x` типа `double` содержит значение аргумента полинома, а переменная `q` того же типа представляет собой степенной множитель, используемый в дальнейшем при вычислении значений полинома и производной от него. Начальное значение этой переменной равно 1. Целочисленная переменная `k` служит в инструкции цикла для индексации элементов массива, а переменные `P` и `Q` типа `double` с нулевыми начальными значениями — для записи вычисляемых в программе значений полинома и производной соответственно.

В инструкции цикла переменная `k` получает значения от `0` до `b.length-1`, что соответствует диапазону индексации элементов массива `b` с коэффициентами для полинома-производной. В рамках каждого цикла выполняется несколько команд. Первой командой `P+=a[k]*q` изменяется значение полинома. Каждый раз добавляется соответствующий полиномиальный коэффициент `a[k]`, умноженный на аргумент в соответствующей степени. Это значение вычисляется и записывается в переменную `q`, начальное значение которой, напомним, равно единице. Командой `b[k]=(k+1)*a[k+1]` вычисляется коэффициент полинома-производной. Значение производной модифицируется командой `Q+=b[k]*q`, в которой использован вычисленный на предыдущем этапе коэффициент полинома-производной, а аргумент в нужной степени записан, как и ранее, в переменную `q`. Наконец, сама эта переменная модифицируется командой `q*=x` (на следующем цикле степень аргумента увеличивается на единицу).

После завершения цикла значение производной оказывается вычисленным, хотя в полиноме не учтено еще одно последнее слагаемое. Эта ситуация исправляется командой `P+=a[a.length-1]*q` после инструкции цикла. В этой команде используется последний элемент массива коэффициентов полинома и переменная `q`, которая после выполнения инструкции цикла содержит значение аргумента в нужной степени.

После вычисления значений для полинома и производной результат выводится на экран:

```
Полином P(x)=19.0
Производная P'(x)=5.0
```

Как и в предыдущем случае, операции с полиномами на основе массивов коэффициентов разумнее реализовывать на основе специальных методов или классов.

Сортировка массива

Существует несколько алгоритмов сортировки массивов. Достаточно популярным и простым, хотя и не очень оптимальным, является пузырьковая сортировка массива. Идея метода достаточно проста. Перебираются все элементы массива, причем каждый раз сравниваются два соседних элемента. Если элемент с меньшим индексом больше элемента с бóльшим индексом, элементы меняются местами. После перебора всех элементов самый большой элемент оказывается последним. После следующей серии с перебором и сравнением соседних элементов на «правильном» месте оказывается второй по величине элемент и т. д. В результате элементы массива оказываются упорядоченными в порядке возрастания. Если нужно сортировать массив в порядке убывания, при переборе и сравнении массива элементы меняются местами, если элемент с меньшим индексом меньше элемента с бóльшим индексом.

В листинге 3.12 приведен пример программы, в которой выполняется пузырьковая сортировка целочисленного массива.

Листинг 3.12. Сортировка массива

```
class Bubble{
public static void main(String args[]){
// Индексные переменные и размер массива:
int m,k,s,n=15;
// Создание массива:
int[] nums=new int[n];
System.out.println("Исходный массив:");
// Заполнение массива и вывод на экран:
for(k=0;k<n;k++){
// Элементы - случайные числа:
nums[k]=(int)(5*n*Math.random());
System.out.print(nums[k]+" ");}
// Сортировка массива:
for(m=1;m<n;m++){
for(k=0;k<n-m;k++){
if(nums[k]>nums[k+1]){
s=nums[k];
nums[k]=nums[k+1];
nums[k+1]=s;}
}}
```

```
// Результат:
System.out.println("\nМассив после сортировки:");
for(k=0;k<n;k++){
System.out.print(nums[k]+" ");}
}}
```

В программе объявляется целочисленный массив `nums`, и с помощью инструкции цикла элементы массива заполняются случайными целыми числами. Для генерирования случайных чисел служит функция `Math.random()`, которая возвращает действительное число в диапазоне от 0 до 1. Для получения случайного целого числа генерированное действительное число умножается на 5 и на размер массива (переменная `n`), после чего с помощью инструкции `(int)` явного приведения типов результат путем отбрасывания дробной части приводится к целочисленному формату. После вычисления очередного элемента массива он выводится на экран. Элементы через пробел выводятся в одну строку.

Сортировка элементов массива выполняется с помощью вложенных инструкций цикла. Индексная переменная `m` нумерует «проходы» — полный цикл перебора и сравнения двух соседних элементов. После каждого такого «прохода», по меньшей мере, один элемент оказывается на «правильном» месте. При этом нужно учесть, что когда предпоследний элемент занимает свою позицию, последний автоматически тоже оказывается в нужном месте. Поэтому количество «проходов» на единицу меньше количества элементов в массиве. Внутренняя индексная переменная `k` нумерует элементы массива. Она изменяется в пределах от 0 до `n-m-1`. Здесь принято во внимание, во-первых, то обстоятельство, что при фиксированном значении `k` сравниваются элементы с индексами `k` и `k+1`, поэтому нужно учитывать, что индекс последнего проверяемого элемента на единицу больше верхней границы изменения индекса `k`. Во-вторых, если какое-то количество «проходов» уже выполнено, то такое же количество последних элементов массива можно не проверять.

После того как сортировка массива выполнена, с помощью инструкции цикла результат выводится на экран. Этот результат может иметь следующий вид:

```
Исходный массив:
63 18 5 30 70 13 21 42 47 38 52 43 51 44 34
Массив после сортировки:
5 13 18 21 30 34 38 42 43 44 47 51 52 63 70
```

Поскольку массив заполняется случайными числами, от запуска к запуску результаты (значения элементов массива) могут быть разными. Неизменным остается одно — после сортировки элементы массива располагаются в порядке возрастания.

Транспонирование квадратной матрицы

Транспонирование матрицы подразумевает взаимную замену строк и столбцов матрицы. Для простоты рассмотрим процедуру транспонирования квадратной матрицы, реализованной в виде двухмерного массива. Результат транспонирования записывается в тот же массив.

Если элементами исходной квадратной матрицы A ранга n являются a_{ij}, где индексы $i, j = 1, 2, ..., n$, то транспонированная матрицы A^+ состоит из элементов $a_{ij}^+ = a_{ji}$. Образно выражаясь, для того чтобы транспонировать квадратную матрицу, необходимо зеркально отобразить ее относительно главной диагонали. В листинге 3.13 приведен пример программы, в которой выполнена такая процедура.

Листинг 3.13. Транспонирование матрицы

```
class MatrTrans{
public static void main(String args[]){
// Ранг матрицы:
int n=4;
// Двухмерный массив:
int[][] A=new int[n][n];
// Индексные и "рабочие" переменные:
int i,j,tmp;
System.out.println("Матрица до транспонирования:");
// Заполнение матрицы случайными числами:
for(i=0;i<n;i++){
for(j=0;j<n;j++){
A[i][j]=(int)(10*Math.random());
System.out.print(A[i][j]+(j!=n-1?"  ":"\n"));}}
// Транспонирование матрицы:
for(i=0;i<n;i++){
for(j=i+1;j<n;j++){
tmp=A[i][j];
A[i][j]=A[j][i];
A[j][i]=tmp;}}
// Вывод результата на экран:
System.out.println("Матрица после транспонирования:");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
System.out.print(A[i][j]+(j!=n-1?"  ":"\n"));}}
}}
```

Результат выполнения программы может иметь следующий вид:

```
Матрица до транспонирования:
0 8 6 5
8 2 2 4
6 2 0 3
8 1 9 0
Матрица после транспонирования:
0 8 6 8
8 2 2 1
6 2 0 9
5 4 3 0
```

В программе командой `int[][] A=new int[n][n]` объявляется двухмерный массив `A`, в который записываются элементы исходной матрицы. Заполнение матрицы и вывод значений элементов матрицы (массива) выполняется с помощью вложенных инструкций цикла. Случайное число, которое в качестве значения присваивается элементу матрицы, вычисляется командой `(int)(10*Math.random())`. Результатом является целое число в диапазоне от 0 до 9 включительно. На экран элемент выводится командой `System.out.print(A[i][j]+(j!=n-1?" ":"\n"))`. Аргументом метода `print()` указано выражение, представляющее собой сумму выводимого на экран элемента и результата вызова тернарного оператора. Тернарным оператором в качестве результата возвращается текст из двух пробелов, если значение второго индекса не равно `n-1` (максимально возможное значение индекса). В противном случае возвращается текст из инструкции перехода к новой строке. Поэтому элементы двухмерного массива выводятся так же, как элементы матрицы — построчно.

В следующей инструкции цикла выполняется транспонирование матрицы. Эта процедура также реализуется с помощью вложенной инструкции цикла. Однако в данном случае перебираются не все элементы, а только те, что лежат выше главной диагонали. Поэтому первый индекс `i`, которым перебираются строки матрицы, изменяется в пределах от 0 до `n-1`, а второй индекс, связанный с нумерацией столбцов, изменяется от `i+1` до `n-1`. Тело внутренней инструкции цикла состоит из трех команд, которыми меняются местами элементы, расположенные симметрично относительно главной диагонали (эти элементы отличаются порядком индексов).

Третья вложенная инструкции цикла в программе служит для вывода элементов матрицы после транспонирования.

Произведение квадратных матриц

Если A и B — квадратные матрицы ранга n с элементами a_{ij} и b_{ij} соответственно (индексы $i, j = 1, 2, ..., n$), то произведением этих матриц является матрица $C = AB$ с элементами:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

В листинге 3.14 приведен пример программы, в которой вычисляется произведение квадратных матриц.

Листинг 3.14. Произведение квадратных матриц

```
class MatrProd{
public static void main(String args[]){
// Ранг квадратных матриц:
int n=4;
// Массивы для реализации матриц:
int[][] A,B,C;
A=new int[n][n];
```

продолжение

Листинг 3.14 *(продолжение)*

```
B=new int[n][n];
C=new int[n][n];
// Индексные переменные:
int i,j,k;
// Заполнение матрицы A:
System.out.println("Матрица A:");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
A[i][j]=(int)(20*Math.random()-9);
System.out.print(A[i][j]+(j!=n-1?"\t":"\n"));}}
// Заполнение матрицы B:
System.out.println("Матрица B:");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
B[i][j]=(int)(20*Math.random()-9);
System.out.print(B[i][j]+(j!=n-1?"\t":"\n"));}}
// Вычисление матрицы C - произведение матриц A и B:
System.out.println("Матрица C=AB:");
for(i=0;i<n;i++){
for(j=0;j<n;j++){
for(k=0;k<n;k++){
C[i][j]+=A[i][k]*B[k][j];}
System.out.print(C[i][j]+(j!=n-1?"\t":"\n"));}}
}}
```

Результат выполнения программы может иметь вид:

```
Матрица A:
4   -2   8   -2
2   -1   4   -8
7    0   1   -5
-5   2   0   -3
Матрица B:
9    -6   -4    -6
-6    8    0    7
-1    7    3    -8
0    -6    0    -1
Матрица C=AB:
40   28   8    -100
20   56   4    -43
62   -5   -25  -45
-57  64   20    47
```

В программе объявляются и инициализируются три двухмерных массива. Первые два заполняются случайными числами. Третий массив заполняется в тройной вложенной инструкции цикла на основе первых двух. При заполнении третьего массива использована особенность языка Java, касающаяся того, что при создании числового массива элементам автоматически присваиваются нулевые

значения. При выводе элементов массивов на экран применяется символ \t табуляции, позволяющий сделать формат вывода данных более эстетичным.

Задача перколяции

Задача перколяции в том или ином виде имеет отношение к решению целого ряда прикладных вопросов. У этой задачи есть несколько формулировок. Рассмотрим один из наиболее простых вариантов. Итак, имеется сетка из полых трубочек, которая может пропускать жидкость. Затем случайным образом выбирают какое-то количество узлов сетки и перекрывают их, так что они больше не могут пропускать жидкость. Необходимо определить, поступит ли жидкость, поданная с одного края сетки, на другой ее край.

Воспользуемся при составлении программы следующим подходом. Создадим в программе квадратную матрицу (двухмерный массив), соответствующую сетке, на которой изучается перколяция. Элементы матрицы соответствуют узлам сетки. На начальном этапе элементы могут принимать два значения: 0 — если узел может пропускать жидкость, и 1 — в противном случае. Заполнение элементов массива выполняется с помощью генератора случайных чисел. В частности, генерируется случайное число в диапазоне от 0 до 1. Если это число больше определенного значения p, элемент матрицы получает значение 1. В противном случае значением элемента матрицы является 0. Таким образом, параметр p представляет собой вероятность того, что узел будет пропускать жидкость.

После заполнения перколяционной матрицы начинается «заливка» жидкости. Узел, в который попала жидкость, мы будем отмечать в перколяционной матрице значением 2. На начальном этапе перебираются элементы первого столбца матрицы, и если значение элемента равняется 0, оно меняется на значение 2. Здесь принято во внимание, что после «выключения» части узлов сетки некоторые из них могут оказаться в первом столбце.

Основная часть программы обеспечивает последовательный перебор всех элементов перколяционной матрицы. Если значение элемента равно 2 (в этом узле уже есть жидкость), соседним элементам, если их текущее значение равно 0 (узел может пропускать жидкость), присваивается значение 2. После перебора всех элементов они снова начинают перебираться и т. д., пока за весь перебор элементов перколяционной матрицы ни один из элементов не изменится.

Для проверки того, достигла ли жидкость конечной части сетки, просматривается последний столбик перколяционной матрицы. Если значение хотя бы одного элемента в этом столбце равно 2, имеет место протекание жидкости.

Вся описанная процедура позволяет определить, имеет ли место протекание жидкости для данной конфигурации выведенных из строя узлов. В общем случае желательно иметь более надежные и объективные показатели. Обычно изучают зависимость вероятности того, что сетка пропускает жидкость, от вероятности того, что выбранный случайным образом узел сетки пропускает жидкость (упоминавшийся параметр p). Для вычисления такой зависимости необходимо провести статистические усреднения: при данном фиксированном значении p

проделать описанную процедуру несколько раз (чем больше — тем лучше) и вычислить вероятность пропускания сеткой жидкости как относительное значение случаев, когда сетка пропускала жидкость, к общему количеству случаев исследования сетки на предмет пропускания жидкости.

В программе, представленной в листинге 3.15 для нескольких значений параметра p (вероятность пропускания жидкости отдельным узлом сетки), вычисляется вероятность пропускания жидкости всей сеткой. Полученные значения заносятся в массив. В результате выполнения программы данные из этого массива в виде импровизированной таблицы выводятся на экран.

Листинг 3.15. Задача перколяции

```
class Percolation{
public static void main(String args[]){
// Количество запусков для усреднения:
int N=100;
// Количество точек вычисления вероятности:
int M=5;
// Размер сетки:
int n=200;
// Переменная-счетчик:
int count;
// Начальная вероятность и ее приращение:
double q=0.57,dq=0.01;
// Матрица перколяционной сетки:
int[][] A=new int[n][n];
// Массив со значениями вероятностей:
double[][] P=new double[2][M+1];
// Индексные переменные:
int i,j,k,m;
// Заполнение массива вероятностей:
for(m=0;m<=M;m++) P[0][m]=q+dq*m;
// Вычисление вероятностей протекания:
for(m=0;m<=M;m++){
// Начальное значение вероятности протекания:
P[1][m]=0;
for(k=1;k<=N;k++){
for(i=0;i<n;i++){
for(j=0;j<n;j++){
// Заполнение перколяционной матрицы:
if(Math.random()>P[0][m]) A[i][j]=1;
else A[i][j]=0;}}
// "Заливка" жидкости (заполнение первого столбца):
for(i=0;i<n;i++){
if(A[i][0]==0) A[i][0]=2;}
```

```
// Определение протекания:
do{
// Начальное состояние счетчика:
count=0;
// Изменение состояния узлов сетки:
for(i=0;i<n;i++){
for(j=0;j<n;j++){
if(A[i][j]==2){
if(i>0&&A[i-1][j]==0) {A[i-1][j]=2; count++;}
if(i<n-1&&A[i+1][j]==0) {A[i+1][j]=2; count++;}
if(j<n-1&&A[i][j+1]==0) {A[i][j+1]=2; count++;}
if(j>0&&A[i][j-1]==0) {A[i][j-1]=2; count++;}
}}}
}while(count>0);
// Проверка последнего столбца перколяционной матрицы:
for(i=0;i<n;i++){
if(A[i][n-1]==2){
    P[1][m]+=(double)1/N;
    break;}
}}}
// Вывод результата на экран:
System.out.print("Протекание узла \t");
for(m=0;m<=M;m++){
System.out.print(Math.round(P[0][m]*100)/100.0+(m!=M?"\t":"\n"));}
System.out.print("Протекание сетки\t");
for(m=0;m<=M;m++){
System.out.print(Math.round(P[1][m]*100)/100.0+(m!=M?"\t":"\n"));}
}}
```

Переменные, использованные в программе, описаны в табл. 3.1.

Таблица 3.1. Переменные в задаче перколяции

Переменная	Описание
`N`	Целочисленная переменная, определяющая количество измерений, на основе которых вычисляется оценка для вероятности пропускания сетки при данном значении вероятности пропускания узла. При увеличении значения этой переменной точность оценок повышается, равно как и время расчетов
`M`	Целочисленная переменная, определяющая количество значений (`M+1`) вероятности пропускания узлов, для которых вычисляется вероятность пропускания сетки
`n`	Целочисленная переменная, определяющая размер перколяционной сетки и, соответственно, размер матрицы `A`, в которую записывается состояние узлов сетки

продолжение

Таблица 3.1 *(продолжение)*

Переменная	Описание
count	Целочисленная переменная-счетчик. Используется для подсчета количества элементов матрицы A, которым при определении пропускания сетки было присвоено значение 2. При определении протекания сетки запускается цикл, значение переменной count обнуляется, после чего проверяются все элементы матрицы A. При изменении значения элемента матрицы значение переменной count увеличивается на единицу. После перебора всех элементов значение count снова обнуляется, перебираются все элементы матрицы A и т. д. Процесс продолжается до тех пор, пока после перебора всех элементов матрицы значение переменой count не изменится (останется нулевым)
q	Первое из набора значений для вероятности протекания узла. В программе вычисляется вероятность протекания сетки для нескольких значений вероятности (точнее, (M+1)-го значения) протекания выбранного случайно узла. Первое из этих значений равно q, последнее — q+M*dq
dq	Переменная, которая определяет интервал дискретности для вероятности протекания узла
A	Целочисленный двухмерный массив, соответствующий перколяционной сетке. Элементы массива (матрицы) A могут принимать значения 0 (узел пропускает жидкость) и 1 (узел не пропускает жидкость). В процессе выполнения программы элемент, имеющий значение 0, может получить значение 2. Это означает, что узел заполнен жидкостью
P	Двухмерный массив размером 2 на M+1 с элементами типа double. Строка P[0] содержит значения для вероятностей пропускания узлов перколяционной сетки, для которой вычисляется вероятность пропускания сетки. Строка P[1] содержит вычисленные вероятности для пропускания сетки, соответствующие значениям из строки P[0]
i	Целочисленная индексная переменная. Используется в инструкциях цикла
j	Целочисленная индексная переменная. Используется в инструкциях цикла
k	Целочисленная индексная переменная. Используется в инструкциях цикла
m	Целочисленная индексная переменная. Используется в инструкциях цикла

После объявления всех переменных и массивов в программе запускается цикл, в теле которого командой P[0][m]=q+dq*m (индексная переменная m получает значения от 0 до M включительно) заполняется первая строка P[0] массива P значениями вероятности протекания узлов, для которых затем вычисляются значения вероятностей протекания сетки. Значения вероятностей протекания сетки вычисляются в следующем цикле (переменная m изменяется в тех же пределах) и после вычисления записываются в строку P[1]. Для начала эти элементы командой P[1][m]=0 обнуляются (для надежности, хотя этого можно и не делать — при создании массива его элементы уже получили нулевые

значения). Затем все в том же цикле вызывается еще один цикл (индексная переменная k изменяется от 1 до N включительно), который обрабатывает процесс проверки протекания перколяционной сетки при фиксированном значении вероятности протекания узлов соответствующее количество раз. На основании результатов работы этого цикла определяется оценка для вероятности протекания сетки. В начале цикла случайным образом, в соответствии с текущим значением вероятности протекания узлов сетки, элементы массива A заполняются нулями и единицами. Для этого служит двойной цикл. Затем с помощью еще одного цикла элементам массива A в первом столбце, значения которых равны 0, присваиваются значения 2 — это означает, что в соответствующие узлы поступила жидкость. После этого начинается обработка процесса заполнения сетки жидкостью. В частности, запускается цикл do-while(). В начале цикла переменной-счетчику count присваивается нулевое значение, а проверяемым в цикле условием является count>0. В цикле do-while() перебираются все элементы массива A. Если значение элемента равно 2, соседним элементам, имеющим нулевые значения, также присваивается значение 2. Соседние элементы — это те, у которых один и только один индекс отличается на единицу от индексов текущего элемента. При этом нужно учесть, что текущий узел может находиться не в центре сетки, а на ее границе. Поэтому для соответствующего элемента операция смещения индекса на одну позицию может привести к выходу за пределы массива A. В силу этого обстоятельства проверяемое условие состоит не только в том, что значение элемента, расположенного рядом с текущим, равно 0, но и в том, что текущий элемент не является «граничным» и операция обращения к соседнему элементу корректна. Указанное условие проверяется первым, а в качестве логического оператора *И* используется сокращенный оператор &&. Напомним, что в этом случае вычисляется первый операнд, и если он равен false, второй операнд не вычисляется. Здесь это — важное обстоятельство, благодаря которому код выполняется без ошибки.

Если хотя бы для одного элемента массива A значение изменено на 2, переменная count увеличивается на единицу. Таким образом, цикл do-while() выполняется до тех пор, пока при переборе всех элементов массива A ни одно значение не будет изменено.

После этого необходимо проверить результат — есть или нет протекание сетки. Если протекание есть, это означает, что жидкость дошла до правого конца сетки, а это, в свою очередь, означает, что последний столбец массива A содержит хотя бы одно значение 2. Поиск этого значения осуществляется в еще одном цикле. Если значение 2 найдено, вероятность P[1][m] увеличивается на величину 1/N, после чего работа инструкции цикла заканчивается (командой break). На этом основная, расчетная часть программы заканчивается. Далее результаты с помощью двух инструкций цикла выводятся на экран. В частности, они могут выглядеть следующим образом:

```
Протекание узла  0.57  0.58  0.59  0.6   0.61  0.62
Протекание сетки 0.0   0.07  0.4   0.78  0.95  1.0
```

С учетом того, что в программе используется процедура генерирования случайных чисел, от запуска к запуску результат может изменяться. Однако если количество запусков, на основании которых выполняется усреднение, достаточно большое, результаты должны изменяться незначительно.

Резюме

1. Массивом называется совокупность переменных одного типа, к которым можно обращаться по общему имени и индексу (или индексам). В Java все массивы динамические — память под них выделятся в процессе выполнения программы.
2. Создание массива можно условно разделить на два этапа. Во-первых, объявляется переменная массива, которой впоследствии в качестве значения присваивается ссылка на массив. Во-вторых, с помощью оператора `new` для массива выделяется место. Результат (ссылка на массив) записывается в переменную массива.
3. При объявлении переменной массива после идентификатора типа данных указываются пустые квадратные скобки. Количество пар пустых скобок соответствует размерности массива. При создании массива после оператора `new` указывается тип элементов массива и в квадратных скобках — размер массива по каждой из размерности.
4. При создании массива его элементы можно инициализировать (по умолчанию элементы созданного массива обнуляются). Список значений элементов массива (список инициализации) указывается в фигурных скобках через запятую. Этот список может указываться в команде объявления переменной массива после имени переменной (через оператор присваивания). Можно также указать список значений сразу после квадратных скобок после идентификатора типа в инструкции выделения памяти под массив (оператором `new`). В этом случае в квадратных скобках размер массива не указывается (он определяется автоматически по количеству значений в списке инициализации).
5. Обращение к элементу массива выполняется в следующем формате: имя массива и в квадратных скобках индекс элемента. Индекс по каждой из размерностей указывается в отдельных квадратных скобках. Индексация элементов массива всегда начинается с нуля.
6. В Java выполняется проверка на предмет выхода индекса элемента массива за допустимые границы. Длину массива (количество элементов) можно получить с помощью свойства `length` (указывается через точку после имени массива).

7. Массивы можно присваивать друг другу в качестве значения и сравнивать. При присваивании массивов имеет место присваивание значений переменных массива, то есть присваивание выполняется на уровне операций со ссылками на массив. При сравнении массивов выполняется сравнение переменных массива. Они считаются равными, если ссылаются на один и тот же массив.
8. Если аргументом метода `print()` или `println()` указано имя символьного массива (массива, элементом которого являются символы), отображается все содержимое символьного массива.

Глава 4. Классы и объекты

Ну, Ватсон, это уж такая простая дедукция!
Могли бы сами догадаться!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Язык программирования Java является особым в силу нескольких причин. Понятно, что для успешной работы в Java необходимо, как минимум, неплохо знать синтаксис языка, но все же не это самое главное. Успешное использование Java на практике невозможно без глубокого понимания принципов объектно-ориентированного программирования (сокращенно ООП). Основные идеи, заложенные в ООП, далее объясняются на простых примерах из повседневной жизни.

Знакомство с ООП

Все правильно и очень просто — после того,
как вы мне объяснили!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Язык программирования Java является полностью объектно-ориентированным. Это означает, что программа, написанная на языке Java, должна строго соответствовать парадигме объектно-ориентированного программирования (ООП). Следует понимать, что принципы ООП не просто определяют структуру программы. Это некий фундаментальный подход, если угодно, философия программирования, на которой имеет смысл остановиться подробнее перед непосредственным изучением основ языка Java.

Принципы, на которых базируется ООП, во многом объясняются причинами, приведшими к появлению ООП как такового. Не вдаваясь в детали, отметим лишь, что в свое время в развитии принципов программирования и программных языков наступил момент, когда сложность прикладных программ достигла уровня, критического для понимания программистами. Традиционный подход,

который получил название процедурного программирования, неприменим для составления больших и сложных программ. Любая программа связана с данными и методами обработки этих данных. Если данных и методов много, в рамках одной программы достаточно сложно разумно структурировать программный код. Такие коды на профессиональном сленге называют «спагетти»-кодами, поскольку отдельные ветви алгоритма программы переплетаются, образуя нечто наподобие запутанного клубка, в котором крайне сложно разобраться. Проблема была принципиальной. Из такой критической ситуации необходимо было искать принципиальный выход, и такой выход был найден в рамках нового подхода, который получил название объектно-ориентированного программирования. Объектно-ориентированный подход в программировании основан на нескольких принципах, достаточно простых и прагматичных. Обычно выделяют три фундаментальных принципа, которые, как три слона, поддерживающие в представлении древних землю на своих спинах, составляют основу ООП: *инкапсуляция*, *полиморфизм* и *наследование*. Кроме этих принципов, вводятся также два важных понятия — это понятия *класса* и *объекта*. Именно с класса и объекта мы и начнем.

Классы и объекты

Чтобы решить проблему упорядочивания программного кода, было принято решение ввести четкое разграничение данных и методов обработки этих данных. Более того, данные и соответствующие им методы объединили в одну структуру, которая в ООП называется объектом.

Такой на первый взгляд искусственный прием позволяет четко разграничить область применимости методов. Вся программа при этом имеет блочную структуру, что существенно упрощает анализ программного кода. Но даже в таком подходе было бы мало пользы, если бы каждый объект был абсолютно уникальным. Практика же такова, что каждый объект определяется некоторым общим шаблоном, который называется классом. В рамках класса задается общий шаблон, то есть структура, на основе которой затем создаются объекты. Данные, относящиеся к классу, называются полями класса, а программный код для их обработки — методами класса.

В классе описывается, какого типа данные относятся к классу (данные называются полями класса), а также то, какие методы применяются к этим данным. Затем в программе на основе того или иного класса создается экземпляр класса (объект), в котором указываются конкретные значения полей и выполняются необходимые действия над ними. Различие между классом и объектом поясним на простом примере, не имеющем никакого отношения к программированию. Поговорим о домашних животных, таких как коты и собаки. Проводя аналогию с программированием, можем определить класс *Кот* и класс *Собака*. Определение класса производится через указание полей (данных) и методов класса. Для класса *Кот* в качестве полей укажем *имя* (кличку кота) и *окрас* (цвет). Для класса *Собака* задаем поля *имя* (кличка собаки), *окрас* и *породу*. Помимо полей, определим методы для этих классов. По большому счету метод — это то, что

может делать объект соответствующего класса (или что можно делать с объектом). Коты будут мяукать и ловить мышей, а собаки — лаять и вилять хвостом. Отсюда методами класса *Кот* являются *мяукать* и *ловить мышей*, а класса *Собака* — *лаять* и *вилять хвостом*. Таким образом, мы определили шаблоны, на основании которых впоследствии будут создаваться экземпляры классов или объекты. Разница между классом и объектом такая же, как между абстрактным понятием и реальным объектом. При создании объекта класса задаем конкретные значения для полей. Когда мы говорим о собаке вообще, как понятии, мы имеем в виду домашнее животное, у которого есть имя, окрас, порода и которое умеет лаять и вилять хвостом. Точно так же, понятие кот означает, что он мяукает и ловит мышей, к нему можно обратиться по имени, и шубка у него может быть какого-то цвета. Это — абстрактные понятия, которые соответствуют классу. А вот если речь идет о конкретном Шарике или Мурзике, то это уже объекты, экземпляры класса.

Представим, что во дворе живут три собаки и два кота: Шарик (дворняжка коричневого окраса), Джек (рыжий спаниель), Ричард (черная немецкая овчарка), Мурзик (белый и пушистый кот) и Барсик (черный кот с белой манишкой). Каждый из пяти этих друзей представляет собой объект. В то же время они относятся к двум классам: Шарик, Джек и Ричард являются объектами класса *Собака*, а Мурзик и Барсик — объектами класса *Кот*. Каждый объект в пределах класса характеризуется одинаковым набором полей и методов. Одновременно с этим каждый объект уникален. Хотя Шарик, Джек и Ричард являются объектами одного класса, они уникальны, поскольку у них разные имена, породы и окрасы. Лают и виляют хвостом они тоже по-разному. Но даже если бы мы смогли клонировать, например, Шарика и назвать пса тем же именем, у нас, несмотря на полную тождественность обеих Шариков, было бы два объекта класса *Собака*. Каждый из них уникален, причем не в силу каких-то физических различий, а по причине того, что один пес существует независимо от другого.

Инкапсуляция, полиморфизм и наследование

Концепция процедурного программирования могла бы быть сформулирована как система составления программного кода, действующего на данные. В этом случае приоритет остается за программным кодом. В ООП предпочтение отдается данным. Именно данные управляют доступом к программному коду. В зависимости от того, какие данные обрабатываются, определяются методы для их обработки. В объектно-ориентированных языках программирования эта концепция реализуется через уже упоминавшиеся механизмы инкапсуляции, полиморфизма и наследования.

Инкапсуляция позволяет объединить данные и код обработки этих данных в одно целое. В результате получаем нечто наподобие «черного ящика», в котором содержатся все необходимые данные и код. Указанным способом создаются обсуждавшиеся уже объекты. Объект является именно той конструкцией, которая поддерживает и через которую реализуется механизм инкапсуляции.

Забегая наперед, отметим, что данные и код внутри объекта могут быть открытыми, доступными вне объекта, и закрытыми. В последнем случае доступ к данным и коду может осуществляться только в рамках объекта.

С точки зрения указанного подхода класс является базовой единицей инкапсуляции. Класс задает формат объекта, определяя тем самым новый тип данных в широком смысле этого термина, включая и методы, то есть программный код для обработки данных. Через концепцию класса данные связываются с программным кодом. В пределах класса данные представляются в виде переменных, а программный код — в виде функций (подпрограмм). Функции и переменные класса называются членами класса (соответственно, методами и полями).

Полиморфизм позволяет использовать один и тот же интерфейс для выполнения различных действий. Здесь действует принцип «Один интерфейс — много методов». Благодаря полиморфизму программы становятся менее сложными, так как для определения и выполнения однотипных действий служит единый интерфейс. Такой единый интерфейс применяется пользователем или программистом к объектам разного типа, а выбор конкретного метода для реализации соответствующей команды осуществляется компьютером в соответствии с типом объекта, для которого выполняется команда.

Важнейшим механизмом в ООП является наследование. Именно наследование позволяет усовершенствовать эволюционным способом программный код, сохраняя при этом на приемлемом уровне сложность программы. Наследование — это механизм, с помощью которого один объект может получить свойства другого объекта, что позволяет создавать на основе уже существующих объектов новые объекты с новыми свойствами, сохраняя при этом свойства старых объектов. Например, если мы решим создать новый класс *Породистая собака*, который от класса *Собака* отличается наличием поля *награды на выставках*, то в общем случае пришлось бы заново создавать класс, описывая в явном виде все его поля и методы. В рамках ООП с помощью механизма наследования можно создать новый класс *Породистая собака* на основе уже существующего класса *Собака*, добавив в описании класса только новые свойства — старые наследуются автоматически. Наследование — удобный и полезный механизм, который детально описан в следующих главах книги.

Преимущества ООП

Применение концепции ООП позволило существенно упростить процесс написания программ и расширило возможности в составлении сложных программных кодов. Среди основных преимуществ ООП выделим следующие.

- В ООП благодаря механизму наследования можно многократно использовать созданный единожды программный код. Это позволяет существенно экономить время на создание нового кода.
- Все ООП-программы достаточно хорошо структурированы, что улучшает их читабельность, да и работать с таким структурированным программным кодом намного приятнее.

- ООП-программы легко редактировать и тестировать, поскольку работа может выполняться с отдельными блоками программы.
- ООП-программы в случае необходимости легко дорабатываются и расширяются. Данная особенность крайне важна при создании больших проектов.
- Несмотря на все это, следует четко понимать, что концепция ООП эффективна лишь в том случае, когда речь идет о больших и сложных программах. Для создания простых программ лучше использовать простые приемы.

Создание классов и объектов

Ну кто так строит?! Кто так строит?!

Из к/ф «Чародеи»

Рассмотрим синтаксис описания классов в Java. Описание класса начинается с ключевого слова `class`. После этого следует имя класса и в фигурных скобках тело класса. Тело класса состоит из описания членов класса: полей и методов. По большему счету поля — это рассмотренные в предыдущих главах переменные (на данном этапе ограничимся таким определением). Таким образом, синтаксис объявления класса имеет следующий вид:

```
class имя_класса{
// тело класса
}
```

В теле класса перечисляются с указанием типа переменные — поля класса (это в том числе могут быть массивы и объекты). Что касается методов, то это — обычные функции, только описанные в рамках класса. Для тех, кто не знаком с азами программирования, заметим, что функция — это именованный фрагмент кода, к которому можно обращаться чрез имя из других частей программы.

Описание метода состоит из сигнатуры и тела метода. Сигнатура метода, в свою очередь, состоит из ключевого слова, которое обозначает тип возвращаемого методом результата, имени метода и списка аргументов в круглых скобках после имени метода. Аргументы разделяются запятыми, для каждого аргумента перед формальным именем аргумента указывается его тип.

Тело метода заключается в фигурные скобки и содержит код, определяющий функциональность метода. В качестве значения методы в Java могут возвращать значения простых (базовых) или ссылочных типов (объекты). Если метод не возвращает результат, в качестве идентификатора типа указывается ключевое слово `void`. Синтаксис объявления метода имеет вид:

```
тип_результат имя_метода(аргументы){
// тело метода
}
```

Значение, возвращаемое методом, указывается после инструкции return. Другими словами, выполнение инструкции return приводит к завершению выполнения метода, и если после этой инструкции указано некоторое выражение, то значение этого выражения возвращается в качестве результата метода. Само собой разумеется, что тип указанного после инструкции return выражения должен совпадать с типом результата, задекларированным в сигнатуре метода.

Отметим, что сигнатура метода может содержать и другие ключевые слова (например, идентификатор уровня доступа), но о них речь будет идти позже.

Программы, которые рассматривались до этого, содержали один класс. Точнее, все эти программы состояли из одного класса. В этом классе описывался всего один метод — главный метод программы main(). Для удобства будем называть класс, содержащий основной метод main(), основным классом программы. В основном классе ранее никаких полей и дополнительных методов не объявлялось (хотя это допустимо и нередко так и поступают). Мы продолжим придерживаться этой традиции (пока, во всяком случае).

Сказанное, в свою очередь, означает, что кроме базового класса придется создать еще один класс и, соответственно, объект этого класса в методе main(). Таким образом, мы плавно подходим к способам создания объектов. Рассмотрим наиболее простой вариант без особого углубления в подробности. В некотором смысле создание объекта напоминает создание массива.

Объект создается в два этапа, которые обычно объединяют. На первом этапе объявляется объектная переменная или переменная объекта — формально, это имя объекта. С технической точки зрения объектная переменная содержит в качестве значения адрес объекта. Второй этап состоит в выделении в памяти места под объект (непосредственно создание объекта) и присваивание в качестве значения объявленной на предыдущем этапе объектной переменной ссылки на созданный объект. Синтаксис объявления объектной переменной мало отличается от объявления переменной базового типа с той лишь разницей, что в качестве типа переменной указывается имя класса, для которого создается объект. Создание объекта (выделение памяти под объект) выполняется с помощью оператора new, после которого указывается имя класса с пустыми круглыми скобками. На самом деле в этом месте указывается конструктор с аргументами, но поскольку конструкторы рассматриваются в следующей главе, пока что примем на веру, что объекты создаются именно так. Таким образом, синтаксис создания объекта имеет вид:

```
имя_класса имя_объекта; // объектная переменная
имя_объекта=new имя_класса(); // выделение памяти
```

Обычно эти две команды объединяют:

```
имя_класса имя_объекта=new имя_класса();
```

Пример объявления класса, содержащего два поля и метод:

```
class MyClass{
double x;
int m;
void set(double z, int n){
x=z;
m=n;}
}
```

Класс имеет название `MyClass` и содержит два поля (поле `x` типа `double` и поле `m` типа `int`), а также метод с названием `set()`. Метод не возвращает результат, поэтому в сигнатуре метода в качестве типа возвращаемого результата указано ключевое слово `void`. У метода два аргумента: один типа `double` и второй типа `int`. Первый аргумент присваивается в качестве значения полю `x`, второй определяет значение поля `m`.

Обращаем внимание, что описание класса к созданию объектов не приводит. Другими словами, описывающий класс код — это всего лишь шаблон, по которому впоследствии можно создавать объекты, а можно и не создавать. В данном случае команды по созданию объекта класса `MyClass` могут выглядеть так:

```
MyClass obj; // Объектная переменная
obj=new MyClass(); // Создание объекта
```

Или так:

```
MyClass obj=new MyClass();
```

В последнем случае объединены две команды: команда объявления объектной переменной и команда создания объекта.

Как уже упоминалось, Java-программа может состоять из нескольких классов. Классы можно описывать в разных файлах, но каждый класс должен быть описан только в одном файле.

Еще одно замечание предназначено для тех, кто программирует в C++. В отличие от этого языка программирования, в Java описание метода и его реализация должны размещаться вместе в теле класса.

Поскольку все объекты класса создаются по единому шаблону, очевидно, что они имеют одинаковый набор полей и методов. Если в программе используется несколько объектов одного класса, необходимо как-то различать, поле или метод какого объекта вызывается — ведь только по названию метода или поля этого не определишь. В Java, как и в прочих объектно-ориентированных языках, применяют так называемый точечный синтаксис. Основная его идея состоит в том, что при обращении к полю или методу объекта сначала указывается имя этого объекта, затем ставится оператор «точка» и после этого имя поля или метода. Забегая наперед, заметим, что кроме обычных существуют так называемые статические члены класса (обычно это поля). Статический член класса один для всех объектов класса. Для использования статического члена класса объект

создавать не обязательно. К статическому члену обычно обращаются тоже через точечный синтаксис, но вместо имени объекта указывается имя класса (хотя можно задействовать и стандартный способ обращения через объект). Подробнее статические члены класса обсуждаются позже.

В листинге 4.1 приведен пример программы; в ней, кроме основного класса, описан еще один класс, в котором объявляются несколько полей и два метода, а также показано, как эти поля и метод используются в программе.

Листинг 4.1. Класс с полями

```
class Coords{
// Координаты точки:
double x;
double y;
double z;
// Метод для присваивания значений полям:
void set(double a,double b,double c){
x=a;
y=b;
z=c;
}
// Методом вычисляется расстояние до точки:
double getDistance(){
return Math.sqrt(x*x+y*y+z*z);}
}
class CoordsDemo{
public static void main(String[] args){
// Создание объекта:
Coords obj=new Coords();
// Вызов метода:
obj.set(5.0,0,2.5);
// Обращение к полю объекта:
obj.y=-4.3;
// Обращение к методу объекта:
System.out.println("Расстояние до точки: "+obj.getDistance());
}
}
```

В программе объявляется класс `Coords`, который имеет три поля `x`, `y` и `z` — все типа `double`. Поля являются аналогом координат точки в трехмерном пространстве. Кроме этого, у класса есть два метода. Метод `set()` не возвращает результат (использован идентификатор типа `void`) и предназначен для присваивания значений полям. У метода три аргумента — значения, присваиваемые полям `x`, `y` и `z` соответственно.

Метод `getDistance()` не имеет аргументов и возвращает в качестве результата значение типа `double`, которым определяется расстояние от начала координат

системы до точки (напомним, что для точки с координатами *x*, *y* и *z* расстояние определяется выражением $\sqrt{x^2 + y^2 + z^2}$). Для вычисления квадратного корня использована статическая функция `sqrt()` встроенного Java-класса `Math`. При вызове этой функции необходимо указать имя класса, то есть в данном случае инструкция вызова функции имеет вид `Math.sqrt()`.

В методе `main()` класса `CoordsDemo` командой `Coords obj=new Coords()` создается объект `obj` класса `Coords`. Командой `obj.set(5.0,0,2.5)` для вновь созданного объекта вызывается метод `set()`, которым задаются значения полей объекта. Далее с помощью инструкции `obj.y=-4.3` значение поля `y` меняется. Наконец, еще один метод `getDistance()` вызывается через инструкцию `obj.getDistance()` прямо в аргументе метода `println()`. Здесь использовано то обстоятельство, что метод возвращает значение. В результате выполнения программы получаем сообщение:

```
Расстояние до точки: 7.052659073002181
```

Сделаем несколько замечаний относительно компиляции программы. В Java для каждого класса программы в результате компиляции создается отдельный файл. Каждый такой файл имеет расширение .class, а его название совпадает с именем класса. Запускать на выполнение следует файл, соответствующий основному классу программы, то есть классу, в котором описан метод `main()`.

Статические элементы

— Что-то еще, джентльмены?
— Одну сигару на всех, сэр!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Как уже отмечалось, помимо обычных полей и методов, в классе могут быть статические члены. От нестатических членов статические принципиально отличаются тем, что они общие для всех объектов данного класса. Например, если речь идет о нестатическом поле, то у каждого объекта класса это поле имеет свое уникальное для объекта значение. Если поле является статическим, то у всех объектов значение этого поля одно и то же. В некотором смысле статические поля напоминают глобальные переменные языка C++, хотя аналогия достаточно натянутая. Со статическими методами ситуация несколько сложнее, поэтому сначала рассмотрим особенности объявления и использования статических полей.

Для объявления статического члена класса, в том числе статического поля, используется идентификатор `static`. Синтаксис описания статического поля следующий:

```
static тип_поля имя_поля;
```

Перед ключевым словом `static` также может следовать спецификатор доступа к полю (см. далее в этой главе). Инициализация значения статического поля

выполняется в специальном статическом блоке. Статический блок также выделяется ключевым словом static, после которого в фигурных скобках следуют команды инициализации статических полей класса. Статический блок не является обязательным. Синтаксис описания статического блока такой:

```
static{  // начало статического блока
// статический блок
}     // завершение статического блока
```

Вне класса обращение к статическим элементам может выполняться обычным способом (через объект), однако, как правило, к ним обращаются непосредственно через класс: после имени класса через точку указывается имя статического поля.

Статические методы класса от обычных отличаются в первую очередь тем, что могут обращаться только к статическим полям класса. Причина достаточно очевидна: метод, который не связан с конкретным экземпляром класса, может, естественно, оперировать с полями, тоже не связанными с конкретным экземпляром, поскольку действие метода должно затрагивать сразу все объекты, относящиеся к классу. Кроме того, в статических методах недопустимо использование ключевых слов this (ссылка на экземпляр класса, из которого вызывается метод) и super (ссылка на экземпляр суперкласса, на основе которого создан вызывающий метод производный класс). Причина та же, что и в первом случае.

Объявляется статический метод так же, как и статическое поле. В частности, в сигнатуре метода указывается ключевое слово static:

```
static тип_результата имя_метода(список аргументов){
// тело метода
}
```

Вызов статического метода вне класса, в котором он описан, осуществляется в формате имя_класса.имя_метода. В листинге 4.2 приведен пример объявления и использования статических полей и статического метода.

Листинг 4.2. Класс со статическими элементами

```
class UsingStatic{
// Статические поля:
static int a=3;
static int b;
// Статический метод:
static void meth(int x){
System.out.println("x="+x);
System.out.println("a="+a);
System.out.println("b="+b);}
// Статический блок:
static{
System.out.println("Статический блок:");
b=a*4;}
```

продолжение

Листинг 4.2 *(продолжение)*

```
// Вызов статического метода:
public static void main(String args[]){
meth(123);
}}
```

В классе UsingStatic описано два статических поля a и b типа int и статический метод meth(). Для статического поля a указано значение 3. Статическое поле b значением не инициализировано. Инициализация этого поля выполняется в статическом блоке командой b=a*4, причем предварительно выводится сообщение Статический блок.

Статический метод meth() результат не возвращает. Ему передается целочисленный аргумент. Действие метода состоит в последовательном выводе сообщения со значениями аргумента метода и статических полей класса.

В этом же классе UsingStatic описан метод main(), в котором всего одна команда вызова метода meth() с аргументом 123. Обращаем внимание, что поскольку статический метод вызывается из того же класса, в котором он описан, для вызова метода достаточно указать его имя. В результате выполнения программы мы получим следующее:

```
Статический блок:
x=123
a=3
b=12
```

Выполнение статического блока происходит при загрузке программного кода класса, поэтому еще до вызова метода meth() в главном методе программы статическая переменная b получает значение благодаря предварительному выполнению команд статического блока. Еще один пример использования статических элементов приведен в листинге 4.3.

Листинг 4.3. Использование статических элементов

```
class MyStatic{
// Статические поля:
static int a=50;
static int b=120;
// Статический метод:
static void callme(){
System.out.println("a="+a);
}}
// Использование статических элементов:
class StaticDemo{
public static void main(String args[]){
// Вызов статического метода через класс:
MyStatic.callme();
// Объекты класса:
MyStatic obj1=new MyStatic();
MyStatic obj2=new MyStatic();
```

```
// Вызов статического поля через объект:
obj1.a=-25;
// Вызов статического поля через объект:
System.out.println("a="+obj2.a);
// Вызов статического поля через класс:
System.out.println("b="+MyStatic.b);
}}
```

Результатом выполнения программы является следующая серия сообщений:

```
a=50
a=-25
b=120
```

В данном случае описано два класса. В классе `MyStatic` описаны два статических целочисленных поля `a` и `b` с предопределенными значениями 50 и 120 соответственно. Кроме того, в классе описан статический метод `callme()`. Метод не имеет аргументов и не возвращает результат, а его действие состоит в выводе значения статического поля `a`. В главном методе программы в классе `StaticDemo` создается два объекта `obj1` и `obj2` класса `MyStatic`. Однако предварительно с помощью команды `MyStatic.callme()` вызывается метод `callme()`. Другими словами, статический метод вызывается еще до того, как созданы объекты класса — в них для вызова статического метода или обращения к статическому полю нет необходимости. Далее командой `obj1.a=-25` полю `a` присваивается новое значение. Формально в данном случае речь идет о поле объекта `obj1`, но поскольку поле статическое, то эффект «глобальный». В этом несложно убедиться, проверив результат выполнения команды `System.out.println("a="+obj2.a)`. Наконец, пример обращения к статическому полю через ссылку на класс представлен командой `System.out.println("b="+MyStatic.b)`.

Хочется особо обратить внимание на несколько важных обстоятельств. Так, в приведенном примере отсутствует статический блок. Обращение к статическим элементам выполняется как через ссылку на класс, так и через ссылки на конкретные экземпляры класса. В последнем случае изменение значения поля для одного объекта означает такое же изменение этого поля для прочих объектов (на самом деле речь идет об одном общем для всех объектов поле). Кроме того, для использования статических элементов вообще нет необходимости создавать экземпляры класса (объекты).

Доступ к членам класса

Нормальные герои всегда идут в обход!

Из к/ф «Айболит-66»

В рассматривавшихся ранее примерах доступ к полям и методам вне класса, в котором они определены, осуществлялся без особых проблем. На самом деле

далеко не всегда к членам класса можно получить доступ вне пределов класса, то есть из кода, не входящего в тело класса.

В Java в зависимости от доступности все члены класса можно разделить на три группы: открытые, закрытые и защищенные. Во всех рассматривавшихся ранее примерах все члены классов были открытыми и, в силу этого обстоятельства, доступными за пределами класса. Таким образом, открытые члены класса — это члены класса, которые доступны вне этого класса. Если в каком-то месте программы создается объект класса, то к полям и методам этого объекта можно обращаться способом, описанным ранее (например, в формате `объект.метод` или `объект.поле`).

Если поле или метод является закрытым, ничего из упомянутого сделать не удастся. К закрытым членам класса доступ осуществляется только в пределах класса. К закрытому полю можно обращаться в методах класса, но нельзя обратиться к полю вне класса. Закрытые методы класса могут вызываться только методами класса, но не могут вызываться извне класса.

Разница между закрытыми и защищенными членами класса проявляется только при наследовании. Если о наследовании речь не идет, то можно полагать, что защищенный член класса — это аналог закрытого члена.

Нельзя сказать, что все сказанное полностью описывает ситуацию с закрытыми и открытыми членами класса. К этому вопросу мы еще вернемся при изучении наследования, после того как познакомимся с пакетами. Пока же ограничимся таким несколько упрощенным взглядом на предмет.

Для определения уровня доступности используются три идентификатора: `public`, `private` и `protected` — идентификаторы доступа соответственно для открытых, закрытых и защищенных членов. Идентификатор доступа указывается для каждого члена класса отдельно. Здесь проявляется отличие языка Java от языка C++: в C++ идентификаторы доступа указываются для групп членов.

Если идентификатор доступа не указан вовсе, соответствующий член считается открытым. Именно этой особенностью мы пользовались ранее, когда не указывали для членов класса идентификаторы доступа. Обращаем внимание читателей, знакомых с C++: там по умолчанию члены класса считаются закрытыми.

В листинге 4.4 приведен пример использования открытых и закрытых членов класса.

Листинг 4.4. Закрытые и открытые члены класса

```
class MyClass{
// Закрытые поля:
private int a;
private int b;
// Закрытый метод:
private void showab(){
// Обращение к закрытым полям в классе:
System.out.println("a="+a);
```

```
System.out.println("b="+b);}
// Открытый метод:
public void setab(int x,int y){
// Обращение к закрытым полям в классе:
a=x;
b=y;
System.out.println("Присвоены значения полям!");}
// Открытый метод:
void getab(){
System.out.println("Проверка значений полей:");
// Обращение к закрытому методу в классе:
showab();}
}
class PrivateDemo{
public static void main(String[] args){
// Создание объекта:
MyClass obj=new MyClass();
// Вызов открытых методов:
obj.setab(3,5);
obj.getab();
}}
```

В классе `MyClass` объявлены два закрытых целочисленных поля `a` и `b`. Поскольку поля закрытые, присвоить им значения вне класса простым обращением к полям невозможно. Для инициализации полей в классе объявляется метод `setab()`. У метода два целочисленных аргумента, значения которых присваиваются полям `a` и `b`. Обращаем внимание, что метод `setab()` имеет доступ ко всем членам класса, в том числе и к закрытым полям, поскольку метод описан в том же классе, что и поля. Метод `setab()` описан с использованием идентификатора уровня доступа `public`. Хотя в методе `getab()` идентификатор доступа не указан, метод также является открытым (по умолчанию). В этом открытом методе выполняется обращение к закрытому методу `showab()`, который обеспечивает вывод значений закрытых полей `a` и `b`.

В главном методе программы класса `PrivateDemo` сначала создается объект `obj` класса `MyClass`. Командой `obj.setab(3,5)` закрытым полям объекта `obj` присваиваются значения 3 и 5 соответственно. Командой `obj.getab()` значения полей выводятся на консоль. В результате выполнения программы получаем:

```
Присвоены значения полям!
Проверка значений полей:
a=3
b=5
```

Еще раз хочется отметить, что в методе `main()` нельзя, например, воспользоваться инструкцией вида `obj.a`, `obj.b` или `obj.showab()`, поскольку соответствующие члены класса `MyClass` являются закрытыми (`private`). В программе использован следующий подход: доступ к закрытым членам осуществляется через открытые

методы. На первый взгляд такой способ реализации класса может показаться нелогичным и неудобным, но это не так. Представим ситуацию, когда необходимо ограничить и четко регламентировать операции, допустимые с полями класса. Самый надежный способ для этого — сделать поля закрытыми, а для допустимых операций над полями предусмотреть открытые методы. Аналогия такая, как если бы мы имели дело с черным ящиком (класс). Внутреннее содержимое ящика — закрытые члены класса. Открытые методы — рычажки, которые позволяют запускать внутренние механизмы. Что не предусмотрено конструкцией, выполнено быть не может. Это позволит сделать код безопасным, а работу с объектами класса удобной. Данная методика широко используется на практике. Кроме того, нередко закрытыми членами класса делают вспомогательные методы, которые применяются только в пределах программного кода класса на промежуточных этапах вычислений.

Ключевое слово this

Словами вера лишь жива!
Как можно отрицать слова?!

И. Гёте. Фауст

Ключевое слово `this` является стандартной ссылкой на объект, из которого вызывается метод. При этом следует учесть, что в Java ссылка на объект фактически является именем этого объекта. Хотя наличие такой ссылки может на первый взгляд показаться излишним, она достаточно часто используется на практике. Условно можно выделить два типа ситуаций, когда может потребоваться ссылка `this`: если она реально необходима и если благодаря ей улучшается читабельность программного кода. Примеры первого типа приводятся в последующих главах книги — для их понимания необходимо сначала познакомиться с особенностями выполнения ссылок на объекты, способами передачи аргументов методам и механизмом возвращения методом объекта в качестве результата. Здесь же мы приведем примеры использования ключевого слова `this` в «косметических» целях.

В рассматривавшихся ранее примерах ссылки на члены класса в теле методов этого же класса выполнялись простым указанием имени соответствующего члена. Если для класса создается объект и из этого объекта вызывается метод, то обращение к члену класса в программном коде метода означает обращение к соответствующему члену объекта. В то же время обращение к полям и методам объекта выполняется в «точечном» синтаксисе, то есть указывается имя объекта, точка и затем имя поля или метода. Для упомянутой ситуации это означает, что в методе объекта выполняется обращение к члену того же объекта. Этот замечательный факт можно отразить в программном коде метода в явном виде с помощью ссылки `this`. Пример приведен в листинге 4.5.

Листинг 4.5. Использование ссылки this

```
class MyClass{
// Поля класса:
double Re,Im;
void set(double Re,double Im){
// Использование ссылки this:
this.Re=Re;
this.Im=Im;}
void get(){
// Инструкция перехода на новую строку \n:
System.out.println("Значения полей:\nRe="+this.Re+" и Im="+this.Im);}
}
class ThisDemo{
public static void main(String[] args){
MyClass obj=new MyClass();
obj.set(1,5);
obj.get();}
}
```

В данном случае класс `MyClass` содержит всего два поля `Re` и `Im` типа `double`. Кстати, обращаем внимание, что оба поля объявлены одной инструкцией, в которой указан тип полей, а сами поля перечислены через запятую. Данный класс является очень слабой аналогией реализации комплексных чисел, у которых есть действительная (поле `Re`) и мнимая (поле `Im`) части.

Кроме двух полей, у класса имеется два метода: метод `set()` для присваивания значений полям и метод `get()` для отображения значений полей. У метода `set()` два аргумента типа `double`, причем их названия совпадают с названиями полей класса. Разумеется, в таком экстремизме необходимости нет, вполне можно было предложить иные названия для аргументов, но мы не ищем простых путей, поэтому сложившаяся ситуация далеко не однозначна. Если в теле метода `set()` использовать обращение `Re` или `Im`, то это будет ссылка на аргумент метода, а не на поле класса, поскольку в случае совпадения имен переменных приоритет имеет локальная переменная (то есть объявленная в блоке, в котором находится инструкция вызова переменных с совпадающими именами). Из ситуации выходим, воспользовавшись ссылкой `this`. Так, инструкции `this.Re` и `this.Im` означают поле `Re` и поле `Im` соответственно объекта `this`, то есть того объекта, из которого вызывается метод `set()`. При этом инструкции `Re` и `Im` являются обращениями к аргументам метода. Поэтому, например, команду `this.Re=Re` следует понимать так: полю `Re` объекта `this` (объекта, из которого вызывается метод `set()`) присвоить значение аргумента `Re`, то есть первого аргумента, переданного методу `set()` при вызове. Хотя описанная ситуация вполне оправдывает использование ссылки `this`, все же такой прием считается несколько искусственным, поскольку в запасе всегда остается возможность просто поменять названия аргументов.

Совсем неоправданным представляется использование ссылки this в программном коде метода get(), который выводит сообщение со значениями полей объекта. В данном случае инструкции вида this.Re и this.Im можно заменить простыми обращениями Re и Im соответственно — функциональность кода не изменится. У метода аргументов нет вообще, поэтому никаких неоднозначностей не возникает. По большому счету, обращение к полю класса (или методу) по имени является упрощенной формой ссылки this (без применения самого ключевого слова this). Этой особенностью синтаксиса языка Java мы пользовались ранее и будем пользоваться в дальнейшем. Тем не менее в некоторых случаях указание ссылки this даже в «косметических» целях представляется оправданным.

Обращаем внимание на инструкцию перехода на новую строку \n в текстовом аргументе метода println() в теле метода get(). Эта инструкция включена непосредственно в текст и ее наличие приводит к тому, что в месте размещения инструкции при выводе текста выполняется переход на новую строку. Поэтому в результате выполнения программы получим сообщение из двух строк:

```
Значения полей:
Re=1.0 и Im=5.0
```

Как уже отмечалось, это далеко не единственный способ использования инструкции this. Далее в книге есть и другие примеры.

Внутренние классы

А если надо будет, снова пойдем кривым путем!

Из к/ф «Айболит-66»

Внутренний класс — это класс, объявленный внутри другого класса. Эту ситуацию не следует путать с использованием в качестве поля класса объекта другого класса. Здесь речь идет о том, что в рамках кода тела класса содержится описание другого класса, который и называется внутренним. Класс, в котором объявлен внутренний класс, называется внешним. В принципе, внутренний класс может быть статическим, но такие классы используются на практике крайне редко, поэтому рассматривать мы их не будем, а ограничимся только нестатическими внутренними классами.

Внутренний класс имеет несколько особенностей. Во-первых, члены внутреннего класса доступны только в пределах внутреннего класса и недоступны во внешнем классе (даже если они открытые). Во-вторых, во внутреннем классе можно обращаться к членам внешнего класса напрямую. Наконец, объявлять внутренние классы можно в любом блоке внешнего класса. Пример использования внутреннего класса приведен в листинге 4.6.

Листинг 4.6. Использование внутреннего класса

```
class MyOuter{
// Поле внешнего класса:
int number=123;
// Метод внешнего класса:
void show(){
// Создание объекта внутреннего класса:
MyInner InnerObj=new MyInner();
// Вызов метода объекта внутреннего класса:
InnerObj.display();}
// Внутренний класс:
class MyInner{
// Метод внутреннего класса:
void display(){
System.out.println("Поле number="+number);}
}
}
class InnerDemo{
public static void main(String  args[]){
// Создание объекта внешнего класса:
MyOuter OuterObj=new MyOuter();
// Вызов метода объекта внешнего класса:
OuterObj.show();}
}
```

В программе описаны три класса: внешний класс `MyOuter`, описанный в нем внутренний класс `MyInner`, а также класс `InnerDemo`. В классе `InnerDemo` описан метод `main()`, в котором создается объект внешнего класса `MyOuter` и вызывается метод этого класса `show()`.

Структура программы следующая: во внешнем классе `MyOuter` объявляется поле `number`, метод `show()` и описывается внутренний класс `MyInner`. У внутреннего класса есть метод `display()`, который вызывается из метода внешнего класса `show()`. Для вызова метода `display()` в методе `show()` создается объект внутреннего класса `InnerObj`. Причина в том, что вызывать метод `display()` напрямую нельзя — члены внутреннего класса во внешнем классе недоступны.

В методе `display()` выводится сообщение со значением поля внешнего класса `number`. Поскольку во внутреннем классе допускается непосредственное обращение к членам внешнего класса, обращение к полю `number` выполняется простым указанием его имени.

В результате выполнения программы получаем сообщение:

```
Поле number=123
```

Отметим, что в главном методе программы можно создать объект внешнего класса, но нельзя создать объект внутреннего класса — за пределами внешнего класса внутренний класс недоступен.

Анонимные объекты

Мы с тобой знакомы, незнакомка!

И. Николаев. Песня «Незнакомка»

Как уже отмечалось, при создании объектов с помощью оператора `new` возвращается ссылка на вновь созданный объект. Прелесть ситуации состоит в том, что эту ссылку не обязательно присваивать в качестве значения переменной. В таких случаях создается анонимный объект. Другими словами, объект есть, а переменной, которая бы содержала ссылку на этот объект, нет. С практической точки зрения такая возможность представляется сомнительной, но это только на первый взгляд. На самом деле анонимные объекты требуются довольно часто — обычно в тех ситуациях, когда единожды используется единственный объект класса. Достаточно простой пример применения анонимного объекта приведен в листинге 4.7.

Листинг 4.7. Анонимный объект

```
class MyClass{
void show(String msg){
System.out.println(msg);}
}
class NamelessDemo{
public static void main(String  args[]){
// Использование анонимного объекта:
new MyClass().show("Этот объект не имеет имени");}
}
```

В классе `MyClass` описан всего один метод `show()` с текстовым аргументом. Текстовый аргумент — это объект встроенного Java-класса `String`. Действие метода состоит в том, что на экран выводится текст, переданный аргументом метода.

В методе `main()` класса `NamelessDemo` всего одна команда, которая и демонстрирует применение анонимного объекта:

```
new MyClass().show("Этот объект не имеет имени")
```

Эту команду можно условно разбить на две части. Инструкцией `new MyClass()` создается новый объект класса `MyClass`, а сама инструкция в качестве значения возвращает ссылку на созданный объект. Поскольку ссылка никакой переменной в качестве значения не присваивается, созданный объект является анонимным. Однако это все равно объект класса `MyClass`, поэтому у него есть метод `show()`. Именно этот метод вызывается с аргументом `"Этот объект не имеет имени"`. Для этого после инструкции `new MyClass()` ставится точка и имя метода с нужным аргументом.

Это далеко не единственный способ применения анонимных объектов. В следующей главе приводятся примеры использования анонимных объектов при работе с конструкторами.

Примеры программ

Далее рассматриваются некоторые программы, в которых кроме главного класса программы (класса, содержащего метод `main()`) описываются и используются другие классы.

Схема Бернулли

Схемой Бернулли называется серия независимых испытаний, в каждом из которых может быть только один из двух случайных результатов — их принято называть *успехом* и *неудачей*. Есть два важных параметра, которые определяют все прочие свойства серии опытов: это вероятность успеха в одном опыте p и количество опытов в серии n. Величина $q = 1 - p$ называется вероятностью неудачи в одном опыте. Достаточно часто на практике используется случайная величина (назовем ее ξ), которая определяется как число успехов в схеме Бернулли. Математическое ожидание этой случайной величины $M\xi = np$, а дисперсия равна $D\xi = npq$. Среднее значение для количества успехов в схеме Бернулли является оценкой математического ожидания, поэтому для схемы с большим количеством испытаний с высокой вероятностью количество успехов в схеме Бернулли близко к математическому ожиданию. Корень квадратный из дисперсии определяет характерную область разброса количества успехов по отношению к математическому ожиданию.

В листинге 4.8 приведен код программы, в которой для реализации схемы Бернулли создается специальный класс.

Листинг 4.8. Схема Бернулли

```
class Bernoulli{
// Количество опытов (испытаний) в схеме:
private int n;
// Вероятность успеха:
private double p;
// Результат испытаний:
private boolean[] test;
// Метод для определения параметров схемы:
public void setAll(int n,double p){
if(n>=0) this.n=n;
else n=0;
if(p>=0&&p<=1) this.p=p;
else this.p=0;
test=new boolean[n];
for(int i=0;i<n;i++){
if(Math.random()<=p) test[i]=true;
else test[i]=false;}
}
// Подсчет количества успехов:
private int getVal(){
```

продолжение

Листинг 4.8 *(продолжение)*

```
int count,i;
for(i=0,count=0;i<n;i++) if(test[i]) count++;
return count;}
// Отображение основных характеристик:
public void show(){
System.out.println("СТАТИСТИКА ДЛЯ СХЕМЫ БЕРНУЛЛИ");
System.out.println("Испытаний: "+n);
System.out.println("Вероятность успеха: "+p);
System.out.println("Успехов: "+getVal());
System.out.println("Неудач: "+(n-getVal()));
System.out.println("Мат. ожидание: "+n*p);
System.out.println("Станд. отклонение: "+Math.sqrt(n*p*(1-p)));}
}
class BernoulliTest{
public static void main(String args[]){
// Создание объекта:
Bernoulli obj=new Bernoulli();
// Определение количества испытаний и вероятности успеха:
obj.setAll(10000,0.36);
// Отображение результата:
obj.show();
}}
```

В классе `Bernoulli` объявляется несколько полей и методов. Закрытое целочисленное поле `n` предназначено для записи количества испытаний в схеме Бернулли. Закрытое поле `p` типа `double` содержит значение вероятности успеха в одном испытании. Поля специально объявлены как закрытые, поскольку их изменение влечет за собой изменение всей статистики, связанной со схемой Бернулли. Чтобы нельзя было изменять значения полей независимо от других параметров, эти поля и были «закрыты». Это же замечание относится к закрытой переменной массива `test`. Переменная является ссылкой на массив с элементами типа `boolean` (значение `true` соответствует успеху в соответствующем опыте, а неудаче соответствует значение `false`). Данный массив представляет собой результат серии испытаний. Его размер определяется значением поля `n`. Заполнение осуществляется с учетом значения поля `p`. Поэтому при изменении хотя бы одного из этих полей должен изменяться и массив, на который ссылается переменная массива `test`: при изменении поля `p` меняются значения элементов массива `test`, а при изменении поля `n` меняется и размер самого массива. Вся эта схема реализована через метод `setAll()`, у которого два аргумента: первый целочисленный аргумент определяет значение поля `n`, а второй аргумент типа `double` определяет значение поля `p`. При присваивании значений полям проверяется условие, попадают ли переданные методу аргументы в диапазон допустимых значений. Для количества испытаний значение должно быть неотрицательным, а для вероятности успеха в одном опыте значение должно быть неотрицательным и не превышать единицу. Если данные критерии не соблюдаются, соответствующему полю присваивается нулевое значение.

После того как значения полям n и p присвоены, командой test=new boolean[n] создается массив нужного размера, и ссылка на него присваивается в качестве значения полю test. Поскольку в Java все массивы динамические, подход, когда поле класса ссылается то на один, то на другой массив, вполне приемлем с позиций как синтаксиса, так и общей идеологии Java. Заполнение массива реализовано посредством инструкции цикла. При заполнении элементов массива использована функция генерирования случайных чисел Math.random(), которая возвращает псевдослучайное число в диапазоне от 0 до 1. Правило заполнения элементов массива следующее: если сгенерированное число не превышает значения поля p, элементу присваивается значение true, в противном случае — значение false. Поскольку генерируемые функцией Math.random() значения равномерно распределены на интервале от 0 до 1, элемент массива test принимает значение true с вероятностью p и значение false с вероятностью 1-p, чего мы и добивались.

Возвращаемым значением метода getVal() является целое число — количество успехов в схеме Бернулли. Подсчет выполняется по элементам массива test. Результат записывается в локальную переменную count. Перебираются все элементы массива test, и если значение элемента равно true, переменная count увеличивается на единицу. После завершения цикла значение переменной count возвращается в качестве результата.

Метод getVal() закрытый и вызывается в открытом методе show(). Методом show() отображается практически вся полезная информация относительно схемы Бернулли. В частности, отображается количество испытаний, вероятность успеха в одном испытании, подсчитывается и отображается количество успехов в серии, вычисляется количество неудач, а также вычисляются и выводятся математическое ожидание и стандартное отклонение.

В главном методе программы командой Bernoulli obj=new Bernoulli() создается объект obj класса Bernoulli. Командой obj.setAll(10000,0.36) заполняются поля объекта, а командой obj.show() выполняется вывод результата. В результате мы получим нечто наподобие следующего:

```
СТАТИСТИКА ДЛЯ СХЕМЫ БЕРНУЛЛИ
Испытаний: 10000
Вероятность успеха: 0.36
Успехов: 3667
Неудач: 6333
Мат. ожидание: 3600.0
Станд. отклонение: 48.0
```

Обращаем внимание, что единственный способ получить доступ к полям класса Bernoulli для изменения их значений состоит в вызове метода setAll(). Такой подход делает невозможным «несанкционированное» изменение полей.

Математические функции

Хотя в Java есть достаточно неплохая библиотека математических функций, поэтому можно создать класс, в котором описать недостающие функции или

переопределить уже существующие. Пример такой программы приведен в листинге 4.9.

Листинг 4.9. Математические функции

```
// Класс с математическими функциями:
class MyMath{
// Интервал разложения в ряд Фурье:
static double L=Math.PI;
// Экспонента:
static double Exp(double x,int N){
int i;
double s=0,q=1;
for(i=0;i<N;i++){
s+=q;
q*=x/(i+1);}
return s+q;}
// Синус:
static double Sin(double x,int N){
int i;
double s=0,q=x;
for(i=0;i<N;i++){
s+=q;
q*=(-1)*x*x/(2*i+2)/(2*i+3);}
return s+q;}
// Косинус:
static double Cos(double x,int N){
int i;
double s=0,q=1;
for(i=0;i<N;i++){
s+=q;
q*=(-1)*x*x/(2*i+1)/(2*i+2);}
return s+q;}
// Функция Бесселя:
static double BesselJ(double x,int N){
int i;
double s=0,q=1;
for(i=0;i<N;i++){
s+=q;
q*=(-1)*x*x/4/(i+1)/(i+1);}
return s+q;}
// Ряд Фурье по синусам:
static double FourSin(double x,double[] a){
int i,N=a.length;
double s=0;
for(i=0;i<N;i++){
s+=a[i]*Math.sin(Math.PI*x*(i+1)/L);}
```

```
return s;}
// Ряд Фурье по косинусам:
static double FourCos(double x,double[] a){
int i,N=a.length;
double s=0;
for(i=0;i<N;i++){
s+=a[i]*Math.cos(Math.PI*x*i/L);}
return s;}
}
class MathDemo{
public static void main(String args[]){
System.out.println("Примеры вызова функций:");
// Вычисление экспоненты:
System.out.println("exp(1)="+MyMath.Exp(1,30));
// Вычисление синуса:
System.out.println("sin(pi)="+MyMath.Sin(Math.PI,100));
// Вычисление косинуса:
System.out.println("cos(pi/2)="+MyMath.Cos(Math.PI/2,100));
// Вычисление функции Бесселя:
System.out.println("J0(mu1)="+MyMath.BesselJ(2.404825558,100));
// Заполнение массивов коэффициентов рядов Фурье для функции y(x)=x:
int m=1000;
double[] a=new double[m];
double[] b=new double[m+1];
b[0]=MyMath.L/2;
for(int i=1;i<=m;i++){
a[i-1]=(2*(i%2)-1)*2*MyMath.L/Math.PI/i;
b[i]=-4*(i%2)*MyMath.L/Math.pow(Math.PI*i,2);}
// Вычисление функции y(x)=x через синус-ряд Фурье:
System.out.println("2.0->"+MyMath.FourSin(2.0,a));
// Вычисление функции y(x)=x через косинус-ряд Фурье:
System.out.println("2.0->"+MyMath.FourCos(2.0,b));
}}
```

В программе описывается класс `MyMath`, в котором объявлено несколько статических функций. В частности, это функции вычисления экспоненты, косинуса и синуса, а также функции Бесселя нулевого индекса. Во всех перечисленных случаях для вычисления значений функций используется ряд Тейлора. Каждая функция имеет по два аргумента: первый аргумент типа `double` определяет непосредственно аргумент математической функции, а второй целочисленный аргумент определяет количество слагаемых в ряде Тейлора, на основании которых вычисляется функция. Для экспоненты использован ряд:

$$\exp(x) \approx \sum_{k=0}^{N} \frac{x^k}{k!}.$$

Синус и косинус вычисляются соответственно по формулам:

$$\sin(x) \approx \sum_{k=0}^{N} \frac{(-1)^k x^{2k+1}}{(2k+1)!},$$

$$\sin(x) \approx \sum_{k=0}^{N} \frac{(-1)^k x^{2k}}{(2k)!}.$$

Для функции Бесселя нулевого индекса использован ряд:

$$J_0(x) \approx \sum_{k=0}^{N} \frac{(-1)^k \left(\frac{x}{2}\right)^{2k}}{(k!)^2}.$$

Кроме этих функций, в классе `MyMath` определены функции для вычисления рядов Фурье по базовым функциям $\sin\left(\frac{\pi n x}{L}\right)$ (функция `FourSin()`) и $\cos\left(\frac{\pi n x}{L}\right)$ (функция `FourCos()`). В частности, у функции `FourSin()` два аргумента: первый — это переменная `x` типа `double`, второй — массив `a` с элементами типа `double`. В качестве результата функцией возвращается сумма вида:

$$\sum_{n=1}^{N} a_{n-1} \sin\left(\frac{\pi n x}{L}\right).$$

Здесь через *an* обозначены элементы массива `a`, верхняя граница суммы *N* определяется по количеству элементов этого массива (команда `N=a.length`), а сама сумма представляет собой разложение в ряд (точнее, ограниченное количество слагаемых ряда) на интервале от 0 до `L` некоторой функции с коэффициентами разложения, записанными в массив `a`. Параметр `L` объявлен в классе `MyMath` как статическое поле со значением `Math.PI`.

Напомним, что разложением функции $y(x)$ в ряд Фурье по синусам на интервале от 0 до L называется бесконечная сумма:

$$y(x) = \sum_{n=1}^{\infty} y_n \sin\left(\frac{\pi n x}{L}\right).$$

Здесь коэффициенты разложения рассчитываются так:

$$y_n = \frac{2}{L} \int_0^L y(x) \sin\left(\frac{\pi n x}{L}\right) dx.$$

Этот ряд дает значение функции $y(x)$ при $0 < x < L$. Аналогично, рядом Фурье по косинусам для функции $y(x)$ на интервале от 0 до L называется бесконечная сумма:

$$y(x) = y_0 + \sum_{n=1}^{\infty} y_n \cos\left(\frac{\pi n x}{L}\right).$$

Коэффициенты разложения здесь:

$$y_n = \frac{2}{L}\int_0^L y(x)\cos\left(\frac{\pi n x}{L}\right)dx \text{ для } n > 0 \text{ и } y_0 = \frac{1}{L}\int_0^L y(x)dx \text{ для } n = 0.$$

Этот ряд также дает значение функции $y(x)$ при $0 < x < L$. Способ представления функции в виде ряда Фурье часто используется в математической физике, например при решении задач теплопроводности. С практической точки зрения задача сводится к вычислению коэффициентов разложения. Если коэффициенты разложения известны, то вычислив ряд по базисным функциям, получаем значение для функции (приближенное, поскольку ряд вычисляется по конечному количеству слагаемых).

В главном методе программы в классе `MathDemo` проверяется корректность вычисления описанных в классе `MyMath` функций. В частности, получаем следующее:

```
Примеры вызова функций:
exp(1)=2.7182818284590455
sin(pi)=2.4790606536130346E-16
cos(pi/2)=4.590388303752165E-17
J0(mu1)=-1.5793881580131606E-10
2.0->1.9996438367829905
2.0->1.999999349096476
```

Экспонента вычисляется с единичным первым аргументом, что позволяет получить оценку для постоянной Эйлера $e \approx 2,7182818284590452$ (вычисленное программой значение от приведенного отличается лишь в последнем знаке).

Для вычисления синуса в качестве первого аргумента указано значение `Math.PI`, а при вычислении косинуса первым аргументом соответствующей функции — значение `Math.PI/2`. Поэтому при вычислении как синуса, так и косинуса должно возвращаться нулевое значение. Точный результат обеспечивается в данном случае с достаточно неплохой точностью (полтора десятка знаков после запятой).

Для вычисления функции Бесселя в качестве первого аргумента указано значение нуля функции Бесселя $\mu_1 \approx 2.404825558$. Напомним, что нулями μ_n ($n = 1, 2, \ldots$) функции Бесселя нулевого индекса $J_0(x)$ называются неотрицательные решения уравнения $J_0(\mu_n) = 0$. Поэтому для указанного аргумента значение функции должно быть нулевым (в пределах точности вычислений) — что мы и наблюдаем по результатам выполнения программы.

Кроме этого, функции `FourSin()` и `FourCos()` служат для вычисления значения функции $y(x) = x$ в точке $x = 2$. Для этого предварительно формируются массивы `a` и `b` с коэффициентами разложения функции $y(x) = x$ в ряд по синусам и косинусам соответственно. При этом используются известные аналитические выражения для коэффициентов разложения этой функции в ряд Фурье. Так, для

синус-разложения коэффициенты разложения $y_n = (-1)^{n+1}\frac{2L}{\pi n}$, а для косинус-разложения коэффициенты $y_n = \frac{2L((-1)^n - 1)}{\pi^2 n^2}$ при $n > 0$ и $y_0 = \frac{L}{2}$.

Для заполнения массивов `a` и `b` соответствующими значениями в главном методе программы используется цикл. После заполнения массивов командами `MyMath.FourSin(2.0,a)` и `MyMath.FourCos(2.0,b)` вычисляются два различных ряда Фурье для функции $y(x) = x$ при значении $x = 2$, то есть в обоих случаях точным результатом является значение 2. Желающие могут сопоставить точность вычислений и количество слагаемых (размеры массивов `a` и `b`), оставленных в рядах Фурье для вычисления этого результата.

Динамический список из объектов

Следующий пример связан с созданием динамического списка из объектов. Каждый объект в этом списке, кроме числового поля, содержит ссылку на следующий объект, а самый последний объект ссылается на самый первый. При этом объекты имеют метод, возвращающий в качестве значения поле объекта, отстоящего от текущего объекта, из которого вызывается метод, на определенное количество позиций. Программный код приведен в листинге 4.10.

Листинг 4.10. Динамический список объектов

```
class MyClass{
// Поле для нумерации объектов:
int number=0;
// Ссылка на следующий объект;
MyClass next=this;
void create(int n){
int i;
MyClass objA=this;
MyClass objB;
// Создание списка:
for(i=1;i<=n;i++){
    objB=new MyClass();
    objA.next=objB;
    objB.number=objA.number+1;
    objA=objB;}
// Поледний объект списка ссылается на начальный:
objA.next=this;
}
// Номер объекта в списке:
int getNumber(int k){
int i;
MyClass obj=this;
for(i=1;i<=k;i++) obj=obj.next;
return obj.number;}
}
```

```
class ObjList{
public static void main(String[] args){
// Исходный объект:
MyClass obj=new MyClass();
// Создание списка из 4-х объектов (начальный + еще 3 объекта):
obj.create(3);
// Проверка содержимого списка:
System.out.println("Значение поля number объектов:");
System.out.println("2-й после начального -> "+obj.getNumber(2));
System.out.println("4-й после начального -> "+obj.getNumber(4));
System.out.println("2-й после 1-го -> "+obj.next.getNumber(2));
}}
```

В программе создается класс `MyClass`, у которого всего два поля: целочисленное поле `number` с нулевым значением по умолчанию и объектная переменная `next` класса `MyClass`. В эту переменную записывается ссылка на следующий в списке объект. По умолчанию значение переменной присваивается ссылке `this`, что означает ссылку на тот же объект, полем которого является переменная.

Для создания списка объектов предусмотрен метод `create()`. Метод не возвращает результата и в качестве аргумента ему передается целое число, которое определяет количество объектов, добавляемых в список. В методе объявляются локальная целочисленная индексная переменная `i`, локальная объектная переменная `objA` (текущий объект списка) класса `MyClass` с начальным значением `this` — ссылкой на объект, из которого вызывается метод `create()`, а также объектная переменная `objB` (следующий элемент списка) того же класса `MyClass`. Затем запускается цикл, в котором индексная переменная получает значения от 1 до `n` (аргумент метода `create()`) с единичным шагом дискретности. Командой `objB=new MyClass()` в цикле создается новый объект класса `MyClass`, и ссылка на этот объект присваивается в качестве значения переменной `objB`. Командой `objA.next=objB` в поле `next` объекта `objA` записывается ссылка на объект `objB`. То есть в поле `next` текущего объекта списка записывается ссылка на следующий элемент списка. Далее командой `objB.number=objA.number+1` полю `number` следующего элемента списка присваивается значение, на единицу большее значения поля `number` текущего элемента списка. Наконец, командой `objA=objB` переменной `objA` присваивается ссылка на следующий элемент списка. На очередной итерации новое значение получит и переменная `objB`. После завершения цикла переменные `objA` и `objB` будут ссылаться на последний объект в списке. Полю `number` этого объекта значение уже присвоено (при выполнении инструкции цикла). Осталось только присвоить значение полю `next` этого объекта (по умолчанию в этом поле содержится ссылка на объект-владелец поля). Новое значение полю присваивается командой `objA.next=this`. В данном случае `this` — это ссылка на объект, из которого вызывался метод `create()`, то есть ссылка на начальный объект списка. После этого список будет создан.

Метод `getNumber()` возвращает в качестве результата целочисленное значение поля `number` объекта, который расположен в списке на указанное аргументом

число позиций. Поскольку объекты в списке ссылаются друг на друга циклически (последний объект ссылается на первый), аргумент метода `getNumber()` может быть больше, чем количество объектов в списке. Алгоритм выполнения метода остаточно прост: в методе запускается цикл, в котором командой `obj=obj.next` в локальную объектную переменную в качестве значения записывается ссылка на следующий элемент списка (напомним, эта ссылка хранится в поле-переменной `next`). После завершения цикла переменная `obj` ссылается на нужный объект. В качестве результата возвращается поле `number` этого объекта.

В главном методе программы в классе `ObjList` командой `MyClass obj=new MyClass()` создается базовый начальный объект. Затем командой `obj.create(3)` создается список объектов (всего четыре объекта — один начальный и еще три к нему добавляются). Поле `number` начального элемента имеет по умолчанию значение 0, а у следующих в списке объектов значения полей `number` равны 1, 2 и 3. После этого несколькими командами выполняется проверка свойств созданной структуры объектов. В результате выполнения программы получаем следующее:

```
Значение поля number объектов:
2-й после начального -> 2
4-й после начального -> 0
2-й после 1-го -> 3
```

В частности, командой `obj.getNumber(2)` возвращается значение поля `number` объекта, смещенного от начального объекта на две позиции, то есть значение 2. Командой `obj.getNumber(4)` возвращается значение поля `number` объекта, отстоящего от начального на 4 позиции. На 3 позиции от начального размещен последний объект в списке. Этот объект ссылается на начальный объект. Поэтому в результате выполнения команды `obj.getNumber(4)` возвращается значение поля `number` начального объекта, то есть значение 0. Наконец, командой `obj.next.getNumber(2)` возвращается значение поля `number` объекта, смещенного на две позиции от объекта, ссылка на который записана в поле `next` объекта `obj` (начальный объект). Это третий объект после начального. Поэтому результатом команды является значение 3.

Работа с матрицами

В листинге 4.11 приведен простой пример программы, в которой для работы с квадратными матрицами создается специальный класс. В этом классе предусмотрены методы для выполнения таких операций, как вычисление детерминанта (определителя), транспонирования матрицы, вычисление следа (шпура) матрицы, заполнение матрицы числами в различных режимах, вывод значений матрицы на экран.

Листинг 4.11. Работа с квадратными матрицами

```
class SMatrix{
// Размер матрицы:
private int n;
```

```
// Объектная переменная:
private int[][] Matrix;
// Определение размера и создание матрицы:
void setBase(int n){
this.n=n;
Matrix=new int[n][n];}
// Заполнение случайными числами:
void setRND(){
int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
Matrix[i][j]=(int)(Math.random()*10);
}
// Заполнение одинаковыми числами:
void setVal(int a){
int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
Matrix[i][j]=a;
}
// Заполнение последовательностью цифр:
void setNums(){
int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
Matrix[i][j]=(i*n+j)%9+1;
}
// Единичная матрица:
void setE(){
int i;
setVal(0);
for(i=0;i<n;i++)
Matrix[i][i]=1;
}
// Отображение матрицы:
void show(){
int i,j;
for(i=0;i<n;i++){
   for(j=0;j<n;j++){
   System.out.print(Matrix[i][j]+(j==n-1?"\n":" "));}
}
}
// След матрицы:
int Sp(){
int i,s=0;
for(i=0;i<n;i++) s+=Matrix[i][i];
```

продолжение

Листинг 4.11 *(продолжение)*

```
return s;}
// Определитель матрицы:
int det(){
int D=0;
switch(n){
// Матрица размерами 1 на 1:
    case 1:
        D=Matrix[0][0];
        break;
// Матрица размерами 2 на 2:
    case 2:
        D=Matrix[0][0]*Matrix[1][1]-Matrix[0][1]*Matrix[1][0];
        break;
    case 3:
// Матрица размерами 3 на 3:
        int i,j,A,B;
        for(j=0;j<n;j++){
        A=1;
        B=1;
        for(i=0;i<n;i++){
        A*=Matrix[i][(j+i)%n];
        B*=Matrix[n-i-1][(j+i)%n];}
        D+=A-B;}
        break;
// Прочие случаи:
    default:
        int k,sign=1;
        SMatrix m;
        for(k=0;k<n;k++){
        m=new SMatrix();
        m.setBase(n-1);
        for(i=1;i<n;i++){
        for(j=0;j<k;j++) m.Matrix[i-1][j]=Matrix[i][j];
        for(j=k+1;j<n;j++) m.Matrix[i-1][j-1]=Matrix[i][j];
        }
        D+=sign*Matrix[0][k]*m.det();
        sign*=(-1);
        }
}
return D;}
// Транспонирование матрицы:
void trans(){
int i,j,s;
for(i=0;i<n;i++)
for(j=i+1;j<n;j++){
s=Matrix[i][j];
```

```
Matrix[i][j]=Matrix[j][i];
Matrix[j][i]=s;}
}
}
class MatrixDemo{
public static void main(String[] args){
// Создание объекта:
SMatrix obj=new SMatrix();
// Определение размера матрицы и ее создание:
obj.setBase(3);
// Заполнение случайными числами:
obj.setRND();
// Единичная матрица:
//obj.setE();
// Заполнение единицами:
//obj.setVal(1);
// Заполнение последовательностью цифр:
//obj.setNums();
System.out.println("Исходная матрица:");
obj.show();
System.out.println("После транспонирования:");
// Транспонирование матрицы:
obj.trans();
obj.show();
// Вычисление следа матрицы:
System.out.println("След матрицы: "+obj.Sp());
// Вычисление определителя матрицы:
System.out.println("Определитель матрицы: "+obj.det());
}}
```

Основу программы составляет класс `SMatrix`. Класс каптирует в себе квадратную матрицу, ссылка на которую (переменная массива `Matrix`) является закрытым полем класса. Для удобства полагаем, что элементы матрицы — целые числа. Еще одно закрытое целочисленное поле `n` определяет размер квадратной матрицы и, соответственно, размер двухмерного массива, через который эта матрица реализована. Значения поля `n` и размер массива, на который ссылается переменная массива `Matrix`, должны, очевидно, изменяться синхронно. Поэтому данные поля закрыты.

Несколько открытых методов класса позволяют заполнять матрицу числовыми значениями. В частности, метод `setRND()` предназначен для заполнения матрицы случайными целыми числами. Для генерирования случайного целого числа предназначена функция `Math.random()`. Функцией в качестве результата возвращается псевдослучайное действительное неотрицательное число, не превышающее единицу. Для получения на его основе целого числа служит команда `(int)(Math.random()*10)` — полученное в результате вызова функции `random()` случайное

число умножается на 10, после чего дробная часть отбрасывается (благодаря команде явного приведения типов).

Метод `setNums()` предназначен для заполнения элементов матрицы последовательностью цифр, начиная с 1 и до 9, затем снова 1 и т. д. Элементы массива `Matrix` заполняются в рамках двойного вложенного цикла командой `Matrix[i][j]=(i*n+j)%9+1`.

Методу `setVal()` передается целочисленный аргумент. Все элементы массива `Matrix` получают в качестве значения аргумент метода. Вызов этого метода с нулевым аргументом происходит в теле метода `setE()`. Данным методом выполняется заполнение массива `Matrix` по принципу единичной матрицы: элементы с одинаковыми индексами имеют единичное значение, все прочие — нулевое. Заполнение выполняется так: сначала все элементы получают нулевое значение (для этого вызывается метод `setVal()` с нулевым аргументом), затем отдельно заполняются единичные элементы. Отметим, что здесь и далее заполнение массивов в явном виде нулевыми значениями скорее дань традиции, поскольку в Java это делается автоматически при создании массива.

Определение размера матрицы (и массива, ссылка на который записывается в поле `Matrix`) выполняется в методе `setBase()`. Метод имеет целочисленный аргумент, который и определяет размер массива `Matrix` по каждому из двух индексов. Поскольку формальное название аргумента метода совпадает с полем класса `n`, присваивание значения полю выполняется командой `this.n=n` с явной ссылкой на объект класса `this`. Командой `Matrix=new int[n][n]` создается двухмерный целочисленный массив, а ссылка на этот массив в качестве значения присваивается переменной массива `Matrix`. Таким образом, только после вызова метода `setBase()` поле `Matrix` объекта оказывается связанным с реальным двухмерным массивом.

Метод `show()` не имеет аргументов, не возвращает результата и предназначен для поэлементного вывода массива `Matrix` на экран. Для этого используются вложенные циклы. Для вывода элементов в команде `System.out.print(Matrix[i][j]+(j==n-1?"\n":" "))` в аргументе метода `print()` применен тернарный оператор, который возвращает инструкцию перехода на новую строку для элементов со вторым индексом, равным `n-1`, и пробел во всех прочих случаях.

Метод `Sp()` в качестве результата возвращает целое число, равное следу матрицы. След матрицы определяется как сумма всех диагональных элементов (элементов с одинаковыми индексами).

Метод `trans()` не имеет аргументов и не возвращает результат. Он используется для транспонирования матрицы, записанной в массив `Matrix`. При выполнении метода перебираются элементы массива `Matrix`, размещенные над главной диагональю (второй индекс у этих элементов массива больше первого индекса). При этом выполняется взаимный обмен значений элементов, расположенных симметрично относительно главной диагонали (у этих элементов индексы отличаются порядком следования).

Самый значительный и по объему кода и по сложности метод `det()` служит для вычисления определителя матрицы, записанной в массив `Matrix`. Основу метода составляет инструкция выбора `switch()` — в ней проверяется значение поля `n` объекта, из которого вызывается метод. Выделяются четыре варианта: когда поле `n` равно 1, 2, 3 и прочие случаи. Результат метода записывается в переменную `D` — именно эта переменная в конце метода возвращается как результат. Если поле `n` равно 1, мы имеем дело, фактически, со скаляром. В этом случае под определителем принято подразумевать сам скаляр. Поэтому для первого случая в инструкции выбора переменной `D` присваивается значение `Matrix[0][0]` — единственный элемент массива `Matrix`.

Если размер матрицы по каждому из индексов равен 2, то определитель вычисляется командой `D=Matrix[0][0]*Matrix[1][1]-Matrix[0][1]*Matrix[1][0]`. Здесь уместно напомнить, что для матрицы A размерами 2 на 2 и элементами a_{ij} (индексы $i, j = 1,2$) определитель вычисляется как $\det(A) = a_{11}a_{22} - a_{12}a_{21}$, то есть как разность произведений диагональных и недиагональных элементов. Для матрицы размерами 3 на 3 можно воспользоваться аналогичной формулой, причем при добавлении числа к индексу применяется правило циклической перестановки: если индекс выходит за допустимые границы, автоматически начинается отсчет с начала. Другими словами, индекс 4 означает индекс 1, индекс 5 означает индекс 2 и т. д.

Именно такое соотношение для определителя квадратной матрицы размерами 3 на 3 использовано в инструкции `switch()` для случая, когда поле `n` равно 3. В этом случае запускается цикл по индексной переменной `j`. В рамках этого цикла целочисленным переменным `A` и `B` присваивается единичное значение, а затем в эти переменные с помощью еще одного (вложенного) цикла (по индексной переменной `i`) записываются произведения троек элементов: в переменную `A` то, которое прибавляется, а в переменную `B` — то, которое отнимается при вычислении определителя. Первое произведение формируется по следующему принципу. При фиксированном втором индексе выбирается элемент в первой строке. Он умножается на элемент во второй строке, смещенный по второму индексу на одну позицию вправо (с учетом циклической перестановки). Результат умножается на элемент в третьей строке, смещенной на две позиции вправо по отношению к элементу в первой строке. Второе произведение формируется по тому же принципу, но только первый элемент выбирается в последней строке, второй — в предпоследней (второй), третий — в первой строке. Вся эта нехитрая схема реализована во внутреннем цикле с помощью команд `A*=Matrix[i][(j+i)%n]` и `B*=Matrix[n-i-1][(j+i)%n]`. Значение переменной `D` меняется с помощью команды `D+=A-B` уже после того, как вычислены значения переменных `A` и `B`.

Самая сложная ситуация — когда размер матрицы превышает значение 3. В этом случае определитель вычисляется по правилу Лапласа. В частности, если мы имеем дело с квадратной матрицей A размера n с элементами $i, j = 1,2,...,n$,

то выражение для вычисления определителя матрицы можно записать в следующем виде:

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(M^{(ij)}).$$

Здесь матрица $M^{(ij)}$ получается из матрицы A вычеркиванием i-й строки и j-го столбца. Матрицы $M^{(ij)}$ являются квадратными матрицами размера $n-1$. Таким образом, задача вычисления определителя матрицы размера n сводится к вычислению n определителей матриц размера $n-1$. Эту процедуру можно продолжить и по той же схеме записать выражения для определителей матриц $M^{(ij)}$ через определители матриц размера $n-2$ и т. д. — пока все не сведется к вычислению определителя квадратной матрицы размера 3.

Данная схема реализована в блоке `default` инструкции `switch()`. Разложение всегда выполняется по первой строке, то есть в приведенной формуле значение индекса i зафиксировано на значении 1. Для массива `Matrix` это первый нулевой индекс.

В блоке `default` объявляются целочисленная индексная переменная `k` и переменная `sign` с начальным значением 1. Переменная `sign` служит для запоминания знака (множитель 1 или -1), с которым соответствующее слагаемое входит в выражение для определителя (он поочередно меняется). Затем объявляется объектная переменная `m` класса `SMatrix`. Эта переменная используется при вычислении определителей матриц меньшего ранга (размера), чем исходная. Основу блока `default` составляет цикл. Индексная переменная `k` получает значения от 0 до `n-1`. В цикле командой `m=new SMatrix()` создается новый объект класса `SMatrix`, и ссылка на этот объект присваивается в качестве значения переменной `m`. Командой `m.setBase(n-1)` ранг соответствующей матрицы устанавливается на единицу меньшим, чем ранг матрицы объекта, из которого вызывается метод. Затем запускается двойной вложенный цикл для заполнения матрицы вновь созданного объекта. Элементы матрицы объекта `m` (то есть массив `m.Matrix`) получаются из матрицы `Matrix`, если «вычеркнуть» строку с нулевым индексом и столбец с индексом `k` (значение индексной переменной внешнего цикла).

Индексная переменная `i` первого цикла получает значения от 1 до `n-1` и «перебирает» строки массива `Matrix` (то есть массива `this.Matrix`), используемые при заполнении массива `m.Matrix`. При этом строки массивов смещены на одну позицию: строка с индексом 1 массива `Matrix` служит для заполнения строки с индексом 0 массива `m.Matrix`, на основе строки с индексом 2 массива `Matrix` заполняется строка с индексом 1 массива `m.Matrix` и т. д. Что касается столбцов, то до `k`-го имеет место однозначное соответствие: `j`-му столбцу массива `Matrix` соответствует `j`-й столбец массива `m.Matrix`. Далее, `k`-й столбец массива `Matrix` в расчет не принимается, а начиная с (`k+1`)-го столбца `j`-му столбцу массива `Matrix` соответствует (`j-1`)-й столбец массива

m.Matrix. Поэтому перебор столбцов (при фиксированном индексе строки i) осуществляется в два этапа: переменная j сначала получает значения от 0 до k-1, а затем от k+1 до n-1.

После заполнения массива m.Matrix командой D+=sign*Matrix[0][k]*m.det() изменяется значение переменной D. Обращаем внимание, что эта команда содержит вызов метода det(), но уже из объекта m, а не из текущего объекта. Другими словами, в определении метода det() происходит вызов этого же метода, но из другого объекта. Таким образом, имеет место рекурсия. Кроме того, командой sign*=(-1) меняется значение используемой в качестве знакового множителя переменной sign.

В главном методе программы в классе MatrixDemo командой SMatrix obj=new SMatrix() создается объект класса SMatrix. Размер поля-массива этого объекта задается командой obj.setBase(3) (поле-матрица ранга 3). Затем поле-массив объекта obj заполняется случайными числами, для чего используется команда obj.setRND(). Далее приведено несколько закомментированных команд, которые определяют иной способ заполнения поля-массива объекта obj — желающие могут поэкспериментировать, убрав в нужном месте символы комментария.

С помощью команды obj.show() исходная матрица выводится на экран. Транспонирование матрицы выполняется с помощью команды obj.trans(), после чего командой obj.show() уже транспонированная матрица выводится на экран. След матрицы вычисляется командой obj.Sp(), а ее определитель — командой obj.det(). Результат выводится на экран. В итоге получаем следующее:

```
Исходная матрица:
3 0 5
8 3 3
9 8 4
После транспонирования:
3 8 9
0 3 8
5 3 4
След матрицы: 10
Определитель матрицы: 149
```

Если установить размер матрицы командой obj.setBase(5), а заполнить матрицу методом setNums(), результат будет таким:

```
Исходная матрица:
1 2 3 4 5
6 7 8 9 1
2 3 4 5 6
7 8 9 1 2
3 4 5 6 7
```

```
После транспонирования:
1 6 2 7 3
2 7 3 8 4
3 8 4 9 5
4 9 5 1 6
5 1 6 2 7
След матрицы: 20
Определитель матрицы: 0
```

Другие варианты читатель может исследовать самостоятельно.

Траектория полета тела

Рассмотрим задачу о вычислении координат тела, брошенного под углом к горизонту при условии, что кроме силы гравитации на тело действует сила сопротивления воздуха. Полагаем, что эта сила пропорциональна скорости и направлена в противоположную сторону к направлению движения тела (направление движения совпадает с направлением скорости). Если через x и y обозначить координаты тела (они являются функцией времени t), то движение тела описывается системой дифференциальных уравнений:

$$\frac{d^2x}{dt^2} = F_x/m;$$

$$\frac{d^2y}{dt^2} = F_y/m.$$

Здесь m — масса тела, а F_x и F_y — соответственно проекции на горизонтальную и вертикальную координатные оси действующей на тело силы. В силу сделанных предположений $F_x = -\gamma V_x$ и $F_y = -mg - \gamma V_y$, где через g обозначено ускорение свободного падения, γ — коэффициент сопротивления, а $V_x = \frac{dx}{dt}$ и $V_y = \frac{dy}{dt}$ являются проекциями скорости на координатные оси.

Для решения задачи в числовом виде воспользуемся следующей итерационной схемой. Предположим, в какой-то момент времени тело имеет координаты x и y, а также скорость, проекции которой на координатные оси составляют V_x и V_y. Чтобы рассчитать положение и скорость тела, которые у него будут через время dt (этот параметр должен быть достаточно малым, чем меньше — тем лучше), к текущей координате x добавляем величину $V_x dt$, а к текущей координате y — величину $V_y dt$. Поправка к новым значениями для компонент скорости равняется $F_x dt/m$ и $F_y dt/m$ соответственно для V_x и V_y. Последовательно выполняя такие итерации нужное количество раз (это количество определяется как целая часть от выражения t/dt, но обычно параметр dt задается на основе времени t и количества итераций n как $dt = t/n$), получим координаты тела и его скорость в момент времени t.

Данная итерационная процедура реализована в программе, представленной в листинге 4.12.

Листинг 4.12. Полет тела

```
class BodyFlight{
// Ускорение свободного падения:
private final static double g=9.8;
// Коэффициент для силы сопротивления:
private final static double gamma=0.005;
// Количество итераций:
private final static int n=1000;
// Момент времени:
private double t;
// Тело:
private Body body;
// Расчет траектории тела:
private void calculate(){
// Координаты, скорость и приращения:
double x,y,dx,dy,Vx,Vy,dVx,dVy,dt;
// Индексная переменная:
int i;
// Сила:
Force F=new Force();
// Начальные координаты и скорость:
x=body.x;
y=body.y;
Vx=body.Vx;
Vy=body.Vy;
// Шаг дискретности по времени:
dt=t/n;
// Вычисление координат и скорости:
for(i=1;i<=n;i++){
dx=Vx*dt;
dy=Vy*dt;
dVx=F.x(body)*dt/body.m;
dVy=F.y(body)*dt/body.m;
x+=dx;
y+=dy;
Vx+=dVx;
Vy+=dVy;
body.set(x,y,Vx,Vy);}
}
// Конструктор класса:
BodyFlight(double[] params){
// Масса (перевод из граммов в килограммы):
double m=params[0]/1000;
// Координаты:
double x=params[1];
double y=params[2];
```

продолжение

Листинг 4.12 *(продолжение)*

```
// Угол к горизонту:
double alpha=Math.toRadians(params[4]);
// Компоненты скорости:
double Vx=params[3]*Math.cos(alpha);
double Vy=params[3]*Math.sin(alpha);
// Объект для "тела":
body=new Body(m,x,y,Vx,Vy);
// Время:
this.t=params[5];
// Расчет положения тела в момент времени t:
calculate();
// Отображение результата расчетов:
body.show();
}
//Внутренний класс для "тела":
class Body{
// Масса:
double m;
// Координаты:
double x;
double y;
// Компоненты скорости:
double Vx;
double Vy;
// Модуль скорости:
double V(){
return Math.sqrt(Vx*Vx+Vy*Vy);}
// Метод для вычисления угла к горизонту:
double phi(){
return Math.atan2(Vy,Vx);}
// Конструктор внутреннего класса:
Body(double m,double x,double y,double Vx,double Vy){
this.m=m;
set(x,y,Vx,Vy);
}
// Метод для присваивания значений полям
// (координаты и компоненты скорости):
void set(double x,double y,double Vx,double Vy){
this.x=x;
this.y=y;
this.Vx=Vx;
this.Vy=Vy;}
// Метод для отображения параметров объекта:
void show(){
```

```
double alpha=Math.round(Math.toDegrees(phi())*100)/100.0;
System.out.println("В момент времени t="+t+" секунд положение тела следующее.");
System.out.println("Высота: "+Math.round(y*100)/100.0+" метров над горизонтом.");
System.out.println("Расстояние от точки броска: "+Math.round(x*100)/100.0+"
метров.");
System.out.println("Скорость: "+Math.round(V()*100)/100.0+" метров в секунду.");
System.out.println("Угол к горизонту: "+alpha+" градусов.");}
}
// Внутренний класс для "силы":
class Force{
// Проекция на горизонтальную ось:
double x(Body obj){
return -gamma*obj.Vx;}
// Проекция на вертикальную ось:
double y(Body obj){
return -g*obj.m-gamma*obj.Vy;}
}}
class BodyFlightDemo{
public static void main(String[] args){
// Параметры (масса (в граммах), начальные координаты (в метрах),
// скорость (в м/с)), угол (в градусах) и время (в секундах)):
double[] params={100,0,0,150,30,5};
// Анонимный объект:
new BodyFlight(params);
}}
```

Для решения программными методами поставленной задачи создается класс `BodyFlight`, в котором объявляются еще два внутренних класса: класс `Body` для реализации объекта «тела» и класс `Force` для реализации объекта «силы».

Внутренний класс `Body` имеет пять полей типа `double`: масса тела `m`, координаты `x` и `y`, а также проекции скорости на координатные оси `Vx` и `Vy`. Методом класса `V()` в качестве значения возвращается модуль скорости тела (определяется как $V = \sqrt{V_x^2 + V_y^2}$). Методом `phi()` возвращается угол к горизонту, под которым направлена траектория тела. Для вычисления результата предназначен встроенный метод `atan2()`, который возвращает угол (в радианах) точки, координаты которой заданы аргументами метода (первый аргумент — координата по вертикали, второй аргумент — координата по горизонтали). Метод `set()` предназначен для того, чтобы задавать значения координат и проекций скорости для тела (соответствующие значения указываются аргументами метода). Этот метод, кроме прочего, вызывается в конструкторе класса. У конструктора пять аргументов — при создании объекта для каждого из полей нужно указать значение.

Метод `show()` внутреннего класса `Body` предназначен для отображения таких параметров, как координаты тела, его скорость и направление. Выводимые

значения имеют до двух цифр после запятой, для чего соответствующим образом округляются. Угол к тому же переводится в градусы.

Внутренний класс `Force` достаточно прост и содержит всего два метода `x()` и `y()` для вычисления проекций действующей на тело силы. Объект, описывающий это тело, передается аргументом методам.

Кроме внутренних классов, в классе `BodyFlight` описываются закрытые поля: поле `g` (ускорение свободного падения), поле `gamma` (коэффициент сопротивления), поле `n` (количество итераций) описаны как `final` (значение полей изменять нельзя) и `static` (статические). Закрытое поле `t` предназначено для записи момента времени, для которого вычисляется положение тела. Закрытое поле `body` — это объектная переменная класса `Body`. Переменная предназначена для записи в нее ссылки на объект, положение которого вычисляется в главном методе программы. Эта переменная используется в методе `calculate()` — закрытом методе, который вызывается в конструкторе класса `BodyFlight` и предназначен для непосредственного вычисления координат и компонентов скорости тела. В методе реализована описанная ранее схема вычислений. В нем, кроме прочего, создается объект внутреннего класса `Force`, который необходим для вычисления изменения скорости тела.

Конструктору класса `BodyFlight` в качестве аргумента передается массив из пяти элементов. Эти элементы последовательно определяют следующие параметры: масса тела (в граммах), координаты (в метрах), модуль скорости (в метрах в секунду), угол, под которым направлена скорость к горизонту (в градусах), момент времени (в секундах), для которого вычисляется положение тела. Отметим, что значение для массы из граммов должно быть переведено в килограммы, для чего делится на 1000. Градусы при расчетах переводятся в радианы с помощью функции `Math.toRadians()`, а для обратного преобразования (радиан в градусы) служит функция `Math.toDegrees()`. Компоненты скорости тела по модулю скорости V и углу (к горизонту) α определяются соответственно как $V_x = V\cos(\alpha)$ и $V_y = V\sin(\alpha)$.

В результате выполнения программы получаем:

```
В момент времени t=5.0 секунд положение тела следующее.
Высота: 219.03 метра над горизонтом.
Расстояние от точки броска: 574.76 метра.
Скорость: 102.28 метра в секунду.
Угол к горизонту: 8.46 градусов.
```

Обращаем внимание, что в главном методе программы, кроме массива `params` с исходными расчетными значениями, создается анонимный объект класса `BodyFlight`. При этом все необходимые вычисления, включая вывод результатов на экран, выполняются автоматически при вызове конструктора.

Резюме

1. Любой объектно-ориентированный язык базируется на трех «китах»: инкапсуляции, полиморфизме и наследовании. Под инкапсуляцией подразумевают механизм, который позволяет объединить в одно целое данные и код для их обработки. Базовой единицей инкапсуляции является класс. Конкретной реализацией класса (экземпляром класса) является объект. Полиморфизм — это механизм, который позволяет использовать единый интерфейс для выполнения однотипных действий. Реализуется полиморфизм, кроме прочего, через перегрузку и переопределение методов. Наследование позволит одним объектам получать (наследовать) свойства других объектов.
2. Описание класса начинается с ключевого слова `class`, после чего следуют имя класса и в фигурных скобках описание класса. В состав класса могут входить поля (переменные) и методы (функции), которые называются членами класса. При описании поля указываются его тип и имя. При описании метода указываются тип результата, имя метода, в круглых скобках список аргументов и в фигурных скобках тело метода.
3. При описании членов класса могут использоваться дополнительные идентификаторы, определяющие доступность членов и их тип. В частности, член класса может быть статическим. В этом случае у всех объектов этого класса соответствующее поле или метод един для всех объектов: и существующих, и тех, что будут созданы. Статический член описывается в классе с идентификатором `static`. Могут также использоваться идентификаторы, определяющие доступность членов класса: `public` (открытые члены), `private` (закрытые члены) и `protected` (защищенные члены). По умолчанию члены класса считаются открытыми — они доступны не только в самом классе, но и вне его.
4. Программа в Java может состоять (и обычно состоит) из нескольких классов. Один из них содержит метод `main()`. При выполнении программы выполняется метод `main()`. Метод не возвращает результат и объявляется с идентификаторами `public` и `static`.
5. Создание объектов осуществляется с помощью оператора `new`. После оператора указывается имя класса, на основе которого создается объект, с круглыми скобками (в рассмотренных примерах пустыми). Так создается объект (для него выделяется место в памяти), а в качестве результата возвращается ссылка на созданный объект. Обычно эта ссылка присваивается в качестве значения объектной переменной, которую отождествляют с объектом. Для объявления объектной переменной указывают имя соответствующего класса и имя этой переменной. На практике команды объявления объектной переменной и создания объекта (с присваиванием переменной ссылки на объект) объединяют в одну команду. Если ссылка на созданный объект не присваивается никакой объектной переменной, говорят об анонимном объекте.
6. В Java один класс может объявляться внутри другого класса. В этом случае говорят о внутреннем классе. Особенность внутреннего класса состоит в том, что он имеет доступ к полям и методам содержащего его класса (класса-

контейнера). Напротив, члены внутреннего класса во внешнем классе недоступны.

7. Ключевое слово `this` является ссылкой на объект, из которого вызывается метод. Это ключевое слово используется при описании методов касса.
8. Доступ к членам класса осуществляется с использованием «точечного» синтаксиса: после имени объекта указывается, через точку, имя поля или метода этого объекта. Вне класса доступны только открытые члены. Доступ к статическим членам, кроме обычного способа, может осуществляться через имя класса — оно указывается вместо имени объекта.

Глава 5. Методы и конструкторы

— Героическая у вас работа! С вами будет спокойнее.
— За беспокойство не беспокойтесь!

Из к/ф «Полосатый рейс»

В предыдущей главе уже упоминалось такое понятие, как конструктор. Конструкторы существенно упрощают процесс создания объектов и повышают эффективность программного кода. В этой главе описываются способы создания конструкторов и способы работы с ними, а также перегрузка методов. Что касается перегрузки операторов, то она является важным механизмом реализации полиморфизма и, кроме того, имеет прямое отношение к конструкторам, поскольку конструкторы также можно перегружать, причем обычно так и поступают. Поэтому сначала мы рассмотрим тему перегрузки методов.

Перегрузка методов

А это как бы премия оптовому покупателю от фирмы.

Из к/ф «Полосатый рейс»

Необходимость в перегрузке методов поясним на простом примере. Допустим, имеется класс с двумя числовыми полями и методом, с помощью которого задаются значения этих полей. Метод имеет два аргумента — по одному для каждого поля. Мы хотим, чтобы в случае если полям присваиваются одинаковые значения, можно было вызывать метод с одним аргументом. Если бы не было возможности перегрузить метод, пришлось бы описывать новый метод. Это неудобно, поскольку для одного и того же, по сути, действия пришлось бы использовать два разных метода. Намного разумнее и удобнее было бы вызывать один и тот же метод, но с разным количеством аргументов в зависимости от ситуации. Благодаря перегрузке методов такая возможность существует.

При перегрузке методов создается несколько методов с одинаковыми именами, но разными сигнатурами. Напомним, что сигнатура метода состоит из типа результата, возвращаемого методом, имени метода и списка аргументов. Поскольку имя общее, разные варианты метода могут отличаться типом возвращаемого

результата и (или) списком аргументов. Технически речь идет о разных методах, но поскольку все они имеют одинаковые названия, обычно говорят об одном методе.

Что касается функциональности различных вариантов перегруженного метода, то формальных ограничений нет, однако общепринятым является принцип, согласно которому перегруженные версии метода должны реализовывать один общий алгоритм (поэтому обычно варианты перегруженного метода отличаются списком аргументов). В этом смысле неприемлема ситуация, когда одна версия метода, например, выводит сообщение со значением аргумента, а другая выполняет поиск наибольшего значения среди переданных методу аргументов. Хотя с технической точки зрения это возможно.

В листинге 5.1 представлен пример программы с перегруженным методом.

Листинг 5.1. Перегрузка метода

```
// Класс с перегруженным методом:
class OverloadDemo{
// Вариант метода без аргументов:
void test(){
System.out.println("Аргументы отсутствуют!");
}
// Вариант метода с одним целым аргументом:
void test(int a){
System.out.println("аргумент типа int: "+a);
}
// Вариант метода с аргументом типа double и результатом типа double:
double test(double a){
System.out.println("аргумент типа double: "+a);
return a;
}}
// Класс с методом main():
class Overload{
public static void main(String args[]){
OverloadDemo obj=new OverloadDemo();
double result;
// Вызов перегруженного метода:
obj.test();
obj.test(10);
result=obj.test(12.5);
System.out.println("Результат: "+result);
}}
```

В программе описывается класс `OverloadDemo`, который содержит четыре варианта метода с названием `test()`. Первый вариант метода не имеет аргументов, и в результате его вызова выводится сообщение `Аргументы отсутствуют!`. Данный вариант метода результат не возвращает. Второй вариант метода имеет аргумент типа `int` и также не возвращает результат. После вызова этого варианта метода

появляется сообщение `аргумент типа int:` со значением переданного методу аргумента. Наконец, еще один вариант метода имеет аргумент типа `double` и в качестве результата возвращает значение типа `double`. После вызова метода выводится сообщение `аргумент типа double:` со значением аргумента, и это значение возвращается в качестве результата.

Класс `Overload` содержит метод `main()`, в котором вызываются разные варианты перегруженного метода `test()`. В результате выполнения программы получаем:

```
Аргументы отсутствуют!
аргумент типа int: 10
аргумент типа double: 12.5
Результат: 12.5
```

Обращаем внимание читателя, что вызов нужного варианта метода осуществляется на основе способа этого вызова, то есть в зависимости от указанного количества аргумента и их типа. Причем при выборе «правильного» варианта перегруженного метода может возникнуть неоднозначная ситуация, особенно с учетом автоматического приведения типов. Пример такой ситуации приведен в листинге 5.2.

Листинг 5.2. Перегрузка метода и приведение типов

```
class OverloadDemo2{
// Метод без аргументов:
void test(){
System.out.println("Аргументы отсутствуют!");}
// Метод с двумя аргументами:
void test(int a,int b){
System.out.println("аргументы типа int: "+a+" и "+b);}
// Метод с одним аргументом типа double:
void test(double a){
System.out.println("аргумент типа double: "+a);}
}
class Overload2{
public static void main(String args[]){
OverloadDemo2 obj=new OverloadDemo2();
int i=88;
obj.test();
obj.test(10,20);
obj.test(i);  // приведение типа!
obj.test(12.5);
}}
```

В классе `OverloadDemo2` объявлены три варианта метода `test()`: без аргументов, с двумя аргументами типа `int` и с одним аргументом типа `double`. Во всех трех случаях метод не возвращает результат. В методе `main()` класса `Overload2` метод вызывается: без аргументов, с двумя аргументами типа `int`, с одним аргументом

типа int и с одним аргументом типа double. Несложно заметить, что вариант метода с одним аргументом типа int в классе OverloadDemo2 не предусмотрен. Тем не менее программа работает корректно:

```
Аргументы отсутствуют!
аргументы типа int: 10 и 20
аргумент типа double: 88.0
аргумент типа double: 12.5
```

Причем во втором случае целочисленное значение аргумента выводится в формате числа с плавающей точкой. Причина в том, что поскольку для метода test() вариант с одним целочисленным аргументом не предусмотрен, имеет место автоматическое приведение типа, то есть тип аргумента int расширяется до типа double, после чего вызывается вариант метода с одним аргументом типа double. Если в классе OverloadDemo2 описать вариант метода test() с одним аргументом типа int, то будет вызываться этот вариант метода.

Конструкторы

Когда садовник садит деревцо,
Плод наперед известен садоводу.

И. Гёте. Фауст

Конструктор — это метод, который вызывается автоматически при создании объекта. Кроме того, что конструктор вызывается автоматически, от обычного метода, объявленного в классе, конструктор отличается тем, что:

- имя конструктора совпадает с именем класса;
- конструктор не возвращает результат, поэтому идентификатор типа для конструктора не указывается;
- как и обычный метод, конструктор может иметь аргументы и его можно перегружать.

До этого мы использовали классы, в которых конструкторов не было. В таких случаях, то есть когда конструктор класса явно не описан, применяется конструктор по умолчанию. Конструктор по умолчанию не имеет аргументов, и именно этот конструктор вызывался в рассмотренных ранее примерах при создании объектов класса. После описания в классе хотя бы одного конструктора конструктор по умолчанию становится недоступным.

В листинге 5.3 приведен пример программного кода с классом, в котором описан конструктор.

Листинг 5.3. Класс с конструктором

```
class MyClass{
// Поля класса:
double Re,Im;
```

```
// Метод для отображения значений полей:
void show(){
System.out.println("Поля объекта:\nRe="+Re+" и Im="+Im);}
// Конструктор:
MyClass(){
// Вывод сообщения:
System.out.println("Создание нового объекта!");
// Инициализация полей:
Re=0;
Im=0;
// Отображение значения полей:
show();}
}
class ConstrDemo{
public static void main(String[] args){
// Создание объекта:
MyClass obj=new MyClass();
}
}
```

Класс `MyClass` имеет два поля `Re` и `Im` типа `double`, метод `show()` для отображения значений полей и конструктор. Хочется верить, что если не считать конструктора, оставшийся программный код комментариев не требует. Код конструктора:

```
MyClass(){
System.out.println("Создание нового объекта!");
Re=0;
Im=0;
show();}
```

Имя конструктора `MyClass()` совпадает с именем класса `MyClass`. Тип результата для конструктора не указывается. Аргументы конструктору также не передаются — после имени конструктора идут пустые круглые скобки. Программный код в фигурных скобках — это тело конструктора.

Первой командой в теле конструктора следует инструкция вывода сообщения о создании нового объекта. Далее полям `Re` и `Im` присваиваются нулевые значения. Наконец, вызывается метод `show()` класса, отображающий значения полей.

В результате выполнения программы получаем следующее:

```
Создание нового объекта!
Поля объекта:
Re=0.0 и Im=0.0
```

Обращаем внимание, что в методе `main()` всего одна команда создания объекта класса `MyClass` — это команда `MyClass obj=new MyClass()`. Другими словами, приведенные сообщения появляются в результате создания объекта как следствие автоматического вызова описанного в классе конструктора.

Как и обычному методу, конструктору можно передавать аргументы. Передаются и используются они по той же схеме, что и для прочих методов класса. Однако теперь при создании объекта необходимо передать аргументы для конструктора. Аргументы передаются в круглых скобках после имени класса в команде создания объекта. Более того, конструкторы можно перегружать. Это означает, что в классе может быть одновременно несколько конструкторов. Усовершенствованный пример предыдущей программы с добавленным конструктором с двумя и одним аргументами приведен в листинге 5.4.

Листинг 5.4. Перегруженный конструктор

```
class MyClass{
// Поля класса:
double Re,Im;
// Метод для отображения значений полей:
void show(){
System.out.println("Поля объекта:\nRe="+Re+" и Im="+Im);}
// Конструктор без аргументов:
MyClass(){
// Вывод сообщения:
System.out.println("Создание нового объекта!");
// Инициализация полей:
Re=0;
Im=0;
// Отображение значения полей:
show();}
// Конструктор с одним аргументом:
MyClass(double x){
// Вывод сообщения:
System.out.println("Создание нового объекта!");
// Инициализация полей:
Re=x;
Im=x;
// Отображение значения полей:
show();}
// Конструктор с двумя аргументами:
MyClass(double x,double y){
// Вывод сообщения:
System.out.println("Создание нового объекта!");
// Инициализация полей:
Re=x;
Im=y;
// Отображение значения полей:
show();}
}
class ConstrDemo2{
public static void main(String[] args){
```

```
// Конструктор без аргументов:
MyClass obj1=new MyClass();
// Конструктор с одним аргументом:
MyClass obj2=new MyClass(10);
// Конструктор с двумя аргументами:
MyClass obj3=new MyClass(100,200);
}
}
```

По сравнению с предыдущим случаем, добавлен код конструктора с одним аргументом, в котором обоим полям присваиваются одинаковые значения. Вариант конструктора с двумя аргументами служит для присваивания разных значений полям создаваемого объекта.

Основной метод содержит три команды создания объектов. При создании объекта командой `MyClass obj1=new MyClass()` вызывается конструктор без аргументов. При создании объекта командой `MyClass obj2=new MyClass(10)` вызывается конструктор с одним аргументом. Конструктор с двумя аргументами вызывается при использовании для создания объекта команды `MyClass obj3=new MyClass(100,200)`. Результат выполнения программы следующий:

```
Создание нового объекта!
Поля объекта:
Re=0.0 и Im=0.0
Создание нового объекта!
Поля объекта:
Re=10.0 и Im=10.0
Создание нового объекта!
Поля объекта:
Re=100.0 и Im=200.0
```

Стоит обратить внимание на следующее важное обстоятельство (которое уже упоминалось ранее). Если в классе не определен конструктор без аргументов (но определен хотя бы один конструктор), рассчитывать на конструктор по умолчанию (конструктор без аргументов) нельзя — необходимо передавать аргументы в соответствии с описанным вариантом конструктора. К конструкторам мы еще вернемся, но предварительно рассмотрим способы возвращения объектов в качестве результатов и особенности передачи аргументов методам в Java.

Объект как аргумент и результат метода

Теория, мой друг, суха,
Но зеленеет жизни древо!

И. Гёте. Фауст

При передаче объекта методу как аргумента в качестве типа переменной-аргумента указывается класс объекта (если точнее, тип объектной переменной, то есть переменной-ссылки на объект).

При возвращении объекта как результата метода в качестве типа результата указывается имя класса объекта. Если результатом метода является объект, то в теле метода должна быть инструкция return с указанным после нее объектом соответствующего класса. Обычно предварительно создается локальный объект, который и возвращается в качестве результата. В листинге 5.5 приведен пример программного кода, в котором имеется конструктор копирования (создания объекта на основе уже существующего объекта) и метод, возвращающий в качестве результата объект.

Листинг 5.5. Объект как аргумент и результат метода

```
class MyObjs{
// Поля класса:
double Re,Im;
// Присваивание значений полям:
void set(double Re,double Im){
this.Re=Re;
this.Im=Im;
show();}
// Отображение значений полей:
void show(){
System.out.println("Re="+Re+" и "+"Im="+Im);}
// Конструктор без аргументов:
MyObjs(){
set(0,0);}
// Конструктор с одним аргументом:
MyObjs(double x){
set(x,x);}
// Конструктор с двумя аргументами:
MyObjs(double x,double y){
set(x,y);}
// Конструктор копирования:
MyObjs(MyObjs obj){
set(obj.Re,obj.Im);}
// Аргумент и результат - объекты:
MyObjs getSum(MyObjs obj){
// Создание локального объекта:
MyObjs tmp=new MyObjs();
// Определение параметров локального объекта:
tmp.Re=Re+obj.Re;
tmp.Im=Im+obj.Im;
// Возвращение результата методом:
return tmp;}
// "Прибавление" объекта к объекту:
void add(MyObjs obj){
Re+=obj.Re;
```

```
Im+=obj.Im;}
}
class ObjsDemo{
public static void main(String[] args){
// Создание объектов:
MyObjs a=new MyObjs(1);
MyObjs b=new MyObjs(-3,5);
MyObjs c=new MyObjs(b);
// Вычисление "суммы" объектов:
c=a.getSum(b);
// Проверка результата:
c.show();
// Изменение объекта:
a.add(c);
// Проверка результата:
a.show();
}
}
```

Класс `MyObjs` имеет два поля `Re` и `Im`, метод `set()` с двумя аргументами для присваивания значений полям, метод `show()` для отображения значений полей объектов, а также метод `getSum()`, с помощью которого вычисляется «сумма» двух объектов. В результате выполнения метода создается объект, причем значения его полей равны сумме соответствующих полей объекта, из которого вызывается метод, и объекта, переданного методу в качестве аргумента. В отличие от этого метода, методом `add()` изменяется объект, из которого вызывается метод, а результат методом `add()` не возвращается. Действие метода `add()` состоит в том, что к полям объекта, из которого вызывается метод, прибавляются значения соответствующих полей объекта-аргумента метода.

В методе `getSum()` командой `MyObjs tmp=new MyObjs()` создается локальный (доступный только в пределах метода) объект `tmp` класса `MyObjs`. При создании объекта использован конструктор без аргументов, поэтому при создании объект `tmp` получает нулевые значения для полей, хотя это и не принципиально. Командами `tmp.Re=Re+obj.Re` и `tmp.Im=Im+obj.Im` полям `tmp.Re` и `tmp.Im` созданного объекта присваиваются нужные значения — суммы соответствующих полей объекта, из которого вызывается метод (поля `Re` и `Im`), и полей объекта `obj`, переданного аргументом методу (поля `obj.Re` и `obj.Im`). После того как для локального объекта `tmp` заданы все свойства, этот объект командой `return tmp` возвращается в качестве результата.

Для понимания принципов работы метода следует учесть особенности возвращения объекта в качестве результата. Так, если методом возвращается объект, при выполнении метода выделяется место в памяти для записи результата. Этот процесс (выделение места в памяти) не следует путать с созданием объекта. После того как в методе выполнена инструкция возврата значения, локальный объект, используемый как результат метода, копируется в место, выделенное

в памяти для результата. Ссылка на эту область памяти фактически и является тем значением, которое присваивается переменной, указанной слева от оператора присваивания в команде вызова метода (как, например, в команде `c=a.getSum(b)` метода `main()`).

В методе `add()` командами `Re+=obj.Re` и `Im+=obj.Im` изменяются поля `Re` и `Im` того объекта, из которого вызывается метод. Поэтому хотя метод результата не возвращает, после его выполнения объект, из которого вызывается метод, изменяется.

Кроме перечисленных методов в классе описаны четыре конструктора, позволяющие создавать объекты без передачи аргумента, с передачей одного аргумента, с передачей двух аргументов, и конструктор копирования. Имеет смысл остановиться на последнем случае.

Аргументом конструктора копирования указывается объект того же класса. В результате создается копия этого объекта. Что касается технической реализации всех конструкторов, то применен следующий прием. В классе, как отмечалось, описан метод `set()`, которому передаются два аргумента и который позволяет присвоить полям соответствующие значения. В конструкторах этот метод вызывается с аргументами в зависимости от того, какие аргументы передаются конструктору. Так, если конструктору аргументы не передаются, метод `set()` вызывается с нулевыми аргументами. Для конструктора с одним аргументом метод `set()` вызывается с двумя одинаковыми аргументами. Если конструктору переданы два аргумента, эти же аргументы передаются методу `set()`. В конструкторе копирования аргументами метода `set()` указываются соответствующие поля объекта-аргумента конструктора. Такой подход нередко облегчает процесс создания эффективного программного кода. Например, в случаях когда каждый конструктор должен содержать стандартный блок команд, эти команды могут быть вынесены в отдельный метод, который затем вызывается в конструкторе.

В результате выполнения программы мы получаем последовательность сообщений:

```
Re=1.0 и Im=1.0
Re=-3.0 и Im=5.0
Re=-3.0 и Im=5.0
Re=0.0 и Im=0.0
Re=-2.0 и Im=6.0
Re=-1.0 и Im=7.0
```

Каждый раз при создании объекта выводится сообщение со значениями полей созданного объекта. Думается, причина появления каждого сообщения понятна, исходя из структуры программы. Интерес может представлять разве что четвертая строка, в которой выводится сообщение с нулевыми значениями полей. Она появляется вследствие создания локального объекта `tmp` в методе `getSum()`.

Рассмотрим еще один пример, иллюстрирующий способ использования анонимных объектов в качестве аргументов конструктора. Код соответствующей программы приведен в листинге 5.6.

Листинг 5.6. Анонимный объект как аргумент конструктора

```
class MyClass{
int n,m;
// Конструктор с двумя аргументами:
MyClass(int a,int b){
n=a;
m=b;
System.out.println("Первый конструктор!");}
// Конструктор создания "копии":
MyClass(MyClass obj){
n=obj.n+1;
m=obj.m-1;
System.out.println("Второй конструктор!");}
// Метод для отображения значения полей:
void show(){
System.out.println("Значения полей: "+n+" и "+m);}
}
class Nameless2{
public static void main(String args[]){
// Аргумент конструктора - анонимный объект:
MyClass obj=new MyClass(new MyClass(10,100));
// Проверка результата:
obj.show();}
}
```

В классе `MyClass` объявлено два целочисленных поля `a` и `b`, метод `show()` для отображения значения полей и два варианта конструктора. Первый конструктор принимает два аргумента и присваивает их в качестве значения полям `a` и `b`, выводя при этом сообщение, что используется первый конструктор. Второй конструктор — это конструктор копирования, позволяющий на основе существующего объекта создать новый. Значение первого поля нового объекта на единицу больше значения первого поля исходного объекта, значение второго — на единицу меньше значения второго поля исходного объекта.

В главном методе программы командой `MyClass obj=new MyClass(new MyClass(10,100))` создается объект `obj` класса `MyClass`. Создается он на основе анонимного объекта, который, в свою очередь, создается командой `new MyClass(10,100)`. Анонимный объект имеет значения полей `a` и `b` равные 10 и 100 соответственно. Инструкция создания анонимного объекта указана аргументом конструктора при создании объекта `obj`. В результате значения полей `a` и `b` этого объекта становятся равны 11 и 99. Итог выполнения программы:

```
Первый конструктор!
Второй конструктор!
Значения полей: 11 и 99
```

Сообщения появляются в соответствии с порядком вызова конструкторов: первым вызывается конструктор с двумя аргументами (создание анонимного объекта),

затем — конструктор копирования (создание объекта obj). Последняя строка является результатом выполнения команды obj.show(), которой проверяются значения полей объекта obj.

Способы передачи аргументов

Я с Вами относительно этого важного пункта не согласен
и могу Вам запятую поставить.

А. Чехов. Письмо к ученому соседу

В Java существует два способа передачи аргументов методам: по значению и по ссылке. При передаче аргумента по значению в метод передается копия этого аргумента. Другими словами, если аргумент передается по значению, то перед передачей аргумента методу сначала создается безымянная копия переменной, передаваемой аргументом, и именно эта безымянная копия используется при вычислениях в рамках выполнения метода. После того как метод завершит свою работу, эта безымянная переменная автоматически уничтожается.

Если аргумент передается по ссылке, все операции в теле метода выполняются непосредственно с аргументом.

На практике разница между способами передачи аргументов проявляется в том случае, если в методе предпринимается попытка изменить аргументы метода.

В Java переменные базовых (простых) типов передаются по значению, а объекты — по ссылке. Хотя на первый взгляд это деление может показаться несколько искусственным, оно представляется практически необходимым, если принять во внимание способ создания объектов и реальную связь объекта и объектной переменной. Ранее этот механизм уже обсуждался, но нелишне будет напомнить еще раз.

Итак, при создании объектной переменной (это переменная типа «класс») объект не создается. Для этой переменной выделяется место в памяти, и в эту область памяти может быть записана ссылка на объект, не больше. Для создания объекта используется оператор new с вызовом конструктора для создания объекта. В этом случае в памяти выделяется место под объект, если конструктором предусмотрено, то заполняются поля этого объекта и выполняются нужные действия. Результатом, который может быть присвоен объектной переменной, является ссылка на созданный объект. Стандартный процесс создания объекта иллюстрирует рис. 5.1.

Когда объект передается методу в качестве аргумента, аргументом на самом деле указывается имя объекта, то есть ссылка на этот объект. В принципе, ссылка передается по значению, то есть создается анонимная копия объектной переменной. Это означает, что переменная имеет то же значение ссылки. Другими словами, анонимная копия ссылается на тот же объект, что и оригинальный аргумент-имя объекта. Операции над объектной переменной затрагивают

объект, на который она ссылается. Поэтому изменения, применяемые к объекту, на самом деле изменяют сам объект.

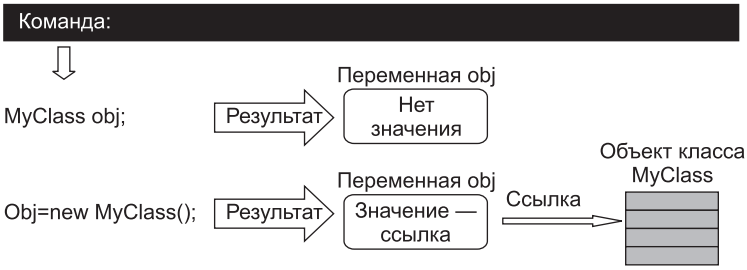

Рис. 5.1. Стандартный процесс создания объекта

В известном смысле объектная переменная в Java напоминает указатель на объект в C++ с той лишь разницей, что в Java программист не имеет прямого доступа к реальному адресу объекта. Кроме того, в C++ существует возможность в явном виде задать механизм передачи аргументов.

В листинге 5.7 приведен пример программы, в которой иллюстрируется разница в способах передачи аргументов методам.

Листинг 5.7. Способы передачи аргументов методам

```
class Base{
int a,b;
void show(){
System.out.println(a+":"+b);}
}
class Test{
void f(int x,int y){
x*=2;
y/=2;
System.out.println(x+":"+y);}
void f(Base obj){
obj.a*=2;
obj.b/=2;
System.out.println(obj.a+":"+obj.b);}
}
class TestDemo{
public static void main(String args[]){
Base obj=new Base();
Test tstFunc=new Test();
obj.a=10;
obj.b=200;
tstFunc.f(obj.a,obj.b);
obj.show();
```

продолжение

Листинг 5.7 *(продолжение)*

```
tstFunc.f(obj);
obj.show();}
}
```

В классе Base имеются два целочисленных поля a и b, а также метод show(), предназначенный для отображения значений этих полей.

В классе Test описан перегруженный метод f(). У метода два варианта: с двумя аргументами типа int и с одним аргументом-объектом класса Base. В первом случае при выполнении метода f() его первый аргумент умножается на два, а второй делится на два. Затем выводится сообщение с парой значений первого и второго аргументов (после внесения в них изменений). Во втором варианте метода та же процедура проделывается с полями объекта, указанного аргументом метода: первое поле объекта умножается на два, а второе делится на два, и новые значения полей выводятся в сообщении.

В главном методе программы в классе TestDemo создаются два объекта: объект obj класса Base и объект tstFunc класса Test. Командами obj.a=10 и obj.b=200 полям объекта obj присваиваются значения, после чего командой tstFunc.f(obj.a,obj.b) вызывается вариант метода f(), аргументами которого указаны поля объекта obj. В результате, как и следовало ожидать, появляется сообщение 20:100. Однако если проверить значения полей объекта obj с помощью команды obj.show(), то мы получим сообщение 10:200, означающее, что значения полей не изменились. Причина в том, что хотя объект передается аргументом по ссылке, целочисленные поля объекта — это переменные базового типа, поэтому передаются по значению. По этой причине при вызове метода f() изменялись копии этих полей, а сами поля не изменились.

После выполнения команды tstFunc.f(obj) мы снова получаем сообщение 20:100. Такое же сообщение мы получаем при выполнении команды obj.show(). Следовательно, поля объекта изменились. В данном случае это вполне ожидаемо: аргументом методу передавался объект, а объекты передаются по ссылке, поэтому все вносимые изменения сказываются на самом объекте.

Примеры программ

Далее рассматриваются примеры, в которых применены некоторые полезные при составлении эффективных программных кодов подходы, связанные с использованием конструкторов и реализацией различных методов.

Интерполяционный полином

Важной задачей прикладного числового анализа является проблема интерполяции и аппроксимации зависимостей, заданных в табулированном виде, то есть в виде массивов узловых точек и значений некоторой, обычно неизвестной функции в этих точках. Рассмотрим иллюстративный пример, позволяющий получить некоторое представление о способах решения этой задачи методами объектно-ориентированного программирования.

В программе создается класс, содержащий, кроме прочего, двухмерный массив со значениями узловых точек аргумента и значениями функции в этих узловых точках. В классе описывается несколько методов, позволяющих вычислять на основе упомянутых данных интерполяционный полином, а также строить простую регрессионную модель (листинг 5.8).

Листинг 5.8. Интерполяция и аппроксимация

```
class InterpAndApprox{
// Размер массива:
private int n;
// Параметры регрессионной модели:
private double a,b;
// Массив данных:
double[][] data;
// Базовые функции в схеме Лагранжа:
private double psi(int k,double x){
int i;
double s=1;
for(i=0;i<k;i++)
s*=(x-data[0][i])/(data[0][k]-data[0][i]);
for(i=k+1;i<=n;i++)
s*=(x-data[0][i])/(data[0][k]-data[0][i]);
return s;}
// Вычисление параметров регрессионной модели:
private void setab(){
double Sxy=0,Sx=0,Sy=0,Sxx=0;
for(int i=0;i<=n;i++){
Sx+=data[0][i];
Sy+=data[1][i];
Sxx+=data[0][i]*data[0][i];
Sxy+=data[0][i]*data[1][i];}
a=((n+1)*Sxy-Sx*Sy)/((n+1)*Sxx-Sx*Sx);
b=Sy/(n+1)-a/(n+1)*Sx;
}
// Регрессионная функция:
double approx(double x){
return a*x+b;}
// Интерполяционный полином:
double interp(double x){
double s=0;
for(int i=0;i<=n;i++)
s+=data[1][i]*psi(i,x);
return s;}
// Конструктор класса:
InterpAndApprox(int n){
```

продолжение

Листинг 5.8 *(продолжение)*

```
this.n=n;
data=new double[2][n+1];
for(int i=0;i<=n;i++){
data[0][i]=Math.PI*i/n/2;
data[1][i]=Math.sin(data[0][i]);}
setab();
}}
class InterpAndApproxDemo{
public static void main(String[] args){
double x;
int i,n=4,N=2*n;
InterpAndApprox obj=new InterpAndApprox(n);
System.out.printf("%60s","Таблица значений:\n");
System.out.printf("%25s","Аргумент x");
System.out.printf("%25s","Функция y=sin(x)");
System.out.printf("%25s","Интерп. полином L(x)");
System.out.printf("%25s","Регр. функция f(x)\n");
for(i=0;i<=N;i++){
x=i*Math.PI/N/2;
System.out.printf("%25s",x);
System.out.printf("%25s",Math.sin(x));
System.out.printf("%25s",obj.interp(x));
System.out.printf("%25s",obj.approx(x)+"\n");}
}}
```

Основу программы составляет класс `InterpAndApprox`, который содержит двухмерный массив с исходными данными для интерполяции и аппроксимации, методы для выполнения соответствующих оценок, а также ряд вспомогательных полей и методов. Узловые точки и значения табулируемой в этих точках функции при создании объекта класса `InterpAndApprox` заносятся в двухмерный массив, ссылка на который `data` является полем класса. По первому индексу массив имеет размер 2 (индекс 0 соответствует узловым точкам, индекс 1 — значениям табулируемой функции в этих точках), а по второму индексу количество элементов определяется закрытым полем `n` (массив содержит `n+1` элемент). Второй индекс определяет номер узловой точки. Конструктору класса `InterpAndApprox` в качестве аргумента передается значение поля `n`. После того как полю `n` присвоено значение, командой `data=new double[2][n+1]` создается массив, и ссылка на него присваивается в качестве значения переменной массива `data`. После этого начинается заполнение массива. В инструкции цикла, индексная переменная `i` которого меняется от 0 до `n+1` включительно, командами `data[0][i]=Math.PI*i/n/2` и `data[1][i]=Math.sin(data[0][i])` присваиваются значения массива исходных данных. Несложно догадаться, что в данном случае табулируется функция $y(x) = \sin(x)$ на интервале значений аргумента $0 \le x \le \pi/2$.

После заполнения массива вызывается метод `setab()`. Это — закрытый метод, позволяющий на основе данных массива `data` вычислить параметры линейной

регрессионной модели. Эти параметры записываются в закрытые поля a и b. Для того чтобы разобраться в методике расчета этих параметров, кратко остановимся на способах построения линейной регрессионной модели.

Предположим, необходимо выполнить аппроксимацию по наборам данных $\{x_k\}$ и $\{y_k\}$ (индекс $k = 0,1,2,...,n$), которые представляют собой значения аргумента x в узловых точках и некоторой функции $y(x)$ в этих же точках (то есть по определению $y(x_k) = y_k$) соответственно. Задача состоит в том, чтобы найти такие параметры a и b, чтобы функция $f(x) = ax + b$ (регрессионная функция) «наилучшим образом» описывала эту зависимость. Для определения «наилучшего образа» обычно используют метод наименьших квадратов, согласно которому указанные параметры вычисляются так, чтобы сумма квадратов отклонений табличных значений функции y_k и регрессионной функции $f(x_k)$ в узловых точках был минимальной. Другими словами, необходимо найти такие значения параметров a и b, чтобы была минимальной сумма:

$$\sum_{k=0}^{n}(y_k - f(x_k))^2 = \sum_{k=0}^{n}(y_k - ax_k - b)^2.$$

Легко показать, что в этом случае параметры a и b вычисляются следующим образом:

$$a = \frac{(n+1)\sum_{k=0}^{n} x_k y_k - \sum_{k=0}^{n} x_k \sum_{k=0}^{n} y_k}{(n+1)\sum_{k=0}^{n} x_k^2 - \left(\sum_{k=0}^{n} x_k\right)^2},$$

$$b = \frac{1}{n+1}\sum_{k=0}^{n} y_k - \frac{a}{n+1}\sum_{k=0}^{n} x_k.$$

В методе setab() в инструкции цикла вычисляются суммы $\sum_{k=0}^{n} x_k$ (переменная Sx), $\sum_{k=0}^{n} y_k$ (переменная Sy), $\sum_{k=0}^{n} x_k y_k$ (переменная Sxy) и $\sum_{k=0}^{n} x_k^2$ (переменная Sxx), а затем на основе вычисленных значений определяются параметры регрессионной модели. Метод approx(), предназначенный для вычисления значения регрессионной функции, достаточно прост и особых комментариев не требует. Несколько сложнее вычисляется интерполяционный полином в методе interp(). При вычислении значения интерполяционного полинома вызывается закрытый метод psi(). Это метод, предназначенный для расчета базисных функций интерполяционного полинома в схеме Лагранжа. Кратко опишем основные моменты использованного подхода.

Для создания интерполяционного полинома на основе наборов данных $\{x_k\}$ и $\{y_k\}$ (индекс $k = 0,1,2,...,n$) необходимо вычислить параметры (коэффициенты) полинома $L(x)$ степени n по аргументу x такого, чтобы в узловых точках значения полинома совпадали со значениями табулированной функции, то есть должно выполняться условие $L(x_k) = y_k$ для всех $k = 0,1,2,...,n$. При создании интерполяционного полинома по схеме Лагранжа соответствующее полиномиальное выражение ищется в виде:

$$L(x) = \sum_{k=0}^{n} y_k \psi_k(x).$$

Базисные функции здесь:

$$\psi_k(x) = \prod_{\substack{m=0,\\ m\neq k}}^{n} \frac{x - x_k}{x_m - x_k}.$$

Особенность базисной функции состоит в том, что $\psi_k(x_m) = \delta_{km}$, где символ Кронекера δ_{km} равен нулю для разных индексов и единице для одинаковых.

Именно схема Лагранжа реализована при вычислении интерполяционного полинома. Значение базисной функции вычисляется методом `psi()`, а непосредственно значение полинома — методом `interp()`.

Что касается главного метода программы, то в нем создается объект класса `InterpAndApprox`, а затем на основе этого объекта для нескольких значений аргумента выводятся на экран: непосредственно аргумент, значение исходной функции в соответствующей точке, значения в этой точке интерполяционного полинома и регрессионной функции. Данные выводятся в виде импровизированной таблицы. Стоит обратить внимание, что для вывода данных использован метод `printf()`, позволяющий задавать способ форматирования. В частности, первый аргумент этого метода является текстовой строкой. В частности, инструкция `"%25s"` означает, что для вывода данных (в данном случае текстового представления числа) используется 25 позиций. Результат выполнения программы будет иметь следующий вид:

```
                    Таблица значений:
    Аргумент x  Функция y=sin(x)  Интерп. полином L(x)  Регр. функция f(x)
           0.0               0.0                   0.0     0.0944947291833
0.1963495408493  0.1950903220161       0.1948971309247     0.2215545341906
0.3926990816987  0.3826834323651       0.3826834323650     0.3486143391979
0.5890486225481  0.5555702330196       0.5556486374881     0.4756741442052
0.7853981633974  0.7071067811865       0.7071067811865     0.6027339492125
0.9817477042468  0.8314696123025       0.8313962000794     0.7297937542198
1.1780972450961  0.9238795325112       0.9238795325112     0.8568535592272
1.3744467859455  0.9807852804032       0.9809437185526     0.9839133642345
1.5707963267948             1.0                   1.0     1.1109731692418
```

Несмотря на довольно неплохое совпадение значения для синуса, рассчитанного на основе встроенной функции и интерполяционного полинома, особо обольщаться не стоит — причина в том, что на выбранном интервале исходная табулированная функция достаточно гладкая, поэтому и интерполяция дает неплохие результаты. Гораздо хуже обстоят дела с линейной регрессией — различие в результатах значительное. Причина в том, что в данном случае линейная регрессия не является оптимальной для аппроксимации исходной функциональной зависимости.

Геометрические фигуры

В листинге 5.9 приведен код программы, в которой реализован класс для работы с некоторыми графическими объектами на плоскости. В частности, предлагается класс для обработки таких геометрических фигур, как правильные многоугольники. Класс содержит методы для их создания (определение координат вершин), а также для вычисления их периметра и площади.

В классе `Figures` целочисленное закрытое поле `n` предназначено для записи количества вершин. Многоугольник определяется, фактически, набором точек на плоскости. Для реализации объекта «точка» в классе `Figures` описывается внутренний класс `Point`. У этого внутреннего класса есть символьное (тип `char`) поле `name` для записи названия точки (латинская буква). Поля `x` и `y` типа `double` предназначены для записи и хранения координат точки.

Конструктору внутреннего класса в качестве аргументов передаются символ (буква) названия точки, а также две ее координаты. Метод `dist()` в качестве результата возвращает расстояние от начала координат до точки, реализованной через объект вызова. Расстояние вычисляется как корень квадратный из суммы квадратов координат точки. Наконец, методом `show()` на экран выводится название точки с ее координатами в круглых скобках, разделенные запятой. При выводе координат точки после запятой оставляется не более двух цифр. Для округления используется метод `Math.round()`.

Идея, положенная в основу алгоритма выполнения программы, следующая. На основе начальной точки создается правильный многоугольник с указанным количеством вершин. Конструктор класса имеет три аргумента: количество вершин многоугольника и координаты первой точки. Прочие точки находятся на таком же расстоянии от начала координат, что и первая точка. Каждая следующая получается смещением точки против часовой стрелки на один и тот же угол. Каждая точка — объект класса `Point`. Ссылки на эти объекты записываются в закрытое поле `points` класса `Figures`. Поле `points` объявляется как переменная массива, элементами которого являются переменные-ссылки на объекты класса `Point` (соответствующая инструкция имеет вид `Point[] points`). Размер массива определяется значением поля `n`. Поля `n` и `points` объявлены как закрытые для предотвращения их несанкционированного или несинхронного изменения.

В классе также описан метод `perimeter()` для вычисления периметра и метод `square()` для вычисления площади многоугольника. Метод `dist()`, описанный

в классе Figures, в качестве аргументов принимает два объекта класса Point, а результатом метода является расстояние между соответствующими точками (корень квадратный из суммы квадратов разностей соответствующих координат точек).

Все основные вычисления происходят при вызове конструктора. В первую очередь там создается начальная точка (объект p класса Point). Для этого используется команда Point p=new Point('A',x,y). Первая точка имеет имя A, а ее координаты определяются вторым и третьим аргументами конструктора класса Figures. Командой this.n=n полю n класса присваивается значение, переданное первым аргументом конструктору. После этого командой points=new Point[n] создается массив для записи ссылок на объекты точек. Угол phi0 на начальную точку и расстояние r до нее вычисляются соответственно командами phi0=Math.atan2(y,x) и r=p.dist(). Напомним, что инструкцией Math.atan2(y,x) возвращается угол на точку с координатами x и y. Угол поворота определяется как phi=2*Math.PI/n. Далее для расчета следующих точек запускается цикл. Одновременно с расчетом имен и координат этих точек методом show() осуществляется вывод этих данных на экран. Каждая новая вершина (ссылка на соответствующий объект) записывается в качестве значения переменной p. Предварительно старая ссылка заносится в поле-массив points. При вычислении имени новой точки к имени старой точки (ссылка p.name) прибавляется единица. Результатом является код следующего после p.name символа. Это значение преобразуется через механизм явного приведения типов в значение типа char. В результате получаем букву, следующую после буквы p.name. При вычислении координат вершины использовано то свойство, что точка, находящаяся на расстоянии r от начала координат в направлении угла α, имеет координаты $x = r\cos(\alpha)$ и $y = r\sin(\alpha)$. В то же время для k-й точки (по порядку, а не по индексу массива) угол определяется как $\alpha = \varphi_0 + (k-1)\varphi$, где φ_0 — угол направления на начальную точку, а $\varphi = \frac{2\pi}{n}$ — угол поворота. Именно эти соотношения использованы при вычислении вершин многоугольника.

После вычисления вершин многоугольника вычисляется и выводится на экран значение для периметра и площади многоугольника.

Листинг 5.9. Геометрические фигуры

```
class Figures{
// Количество вершин:
private int n;
// Точки (вершины):
private Point[] points;
// Конструктор класса:
Figures(int n,double x,double y){
// Угол на начальную точку, угол приращения
// и расстояние до начальной точки:
double phi0,phi,r;
```

```
// Начальная точка:
Point p=new Point('A',x,y);
// Индексная переменная:
int i;
// Значение для количества вершин:
this.n=n;
// Массив переменных-ссылок на объекты "точек":
points=new Point[n];
// Угол на начальную точку - вычисление:
phi0=Math.atan2(y,x);
// Угол приращения - вычисление:
phi=2*Math.PI/n;
// Расстояние до начальной точки от начала координат:
r=p.dist();
System.out.print("Правильный "+n+"-угольник с вершинами в точках ");
// Заполнение массива "точек" и вывод результата на экран:
for(i=0;i<n-1;i++){
p.show();
System.out.print(i==n-2?" и ":", ");
points[i]=p;
// "Вычисление" вершин:
p=new Point((char)(p.name+1),r*Math.cos(phi0+(i+1)*phi),r*Math.
sin(phi0+(i+1)*phi));
}
// "Последняя" вершина:
points[n-1]=p;
p.show();
System.out.println(".");
// Периметр фигуры:
System.out.println("Периметр:\t"+perimeter()+".");
// Площадь фигуры:
System.out.println("Площадь:\t"+square()+".");
}
// Расстояние между точками:
double dist(Point A,Point B){
return Math.sqrt((A.x-B.x)*(A.x-B.x)+(A.y-B.y)*(A.y-B.y));}
// Метод для вычисления периметра:
double perimeter(){
double P=0;
int i;
for(i=0;i<n-1;i++)
P+=dist(points[i],points[i+1]);
P+=dist(points[n-1],points[0]);
return P;
}
```

продолжение

Листинг 5.9 *(продолжение)*

```
// Метод для вычисления площади:
double square(){
double r=points[0].dist();
double phi=2*Math.PI/n;
double s=r*r*Math.sin(phi)/2;
return s*n;
}
// Внутренний класс для "точек":
    class Point{
    // Название вершины:
    char name;
    // Координаты вершины:
    double x,y;
    // Конструктор внутреннего класса:
    Point(char name,double x,double y){
    // Название точки:
    this.name=name;
    // Координаты точки:
    this.x=x;
    this.y=y;}
    // Метод для вычисления расстояния от начала координат до точки:
    double dist(){
    return Math.sqrt(x*x+y*y);}
    // Метод для отображения названия точки и ее координат:
    void show(){
    System.out.print(name+"("+Math.round(x*100)/100.0+","+Math.
round(y*100)/100.0+")");}
    }
}
class FiguresDemo{
public static void main(String[] args){
// Создание квадрата:
new Figures(4,1,1);}
}
```

Таким образом, для создания и вычисления объекта «многоугольник» достаточно вызвать конструктор класса `Figures`, для чего в главном методе программы `main()` в классе `FiguresDemo` создается анонимный объект этого класса. Результат выполнения программы будет иметь вид:

```
Правильный 4-угольник с вершинами в точках A(1.0,1.0), B(-1.0,1.0), C(-1.0,-1.0)
и D(1.0,-1.0).
Периметр:    8.0.
Площадь: 4.000000000000001.
```

Обращаем внимание, что для правильного многоугольника периметр и площадь можно вычислить на основе информации о количестве вершин и расстоянии от начала координат до первой точки. Здесь вычисления в иллюстративных целях

проводились, что называется, «в лоб». Например, для расчета периметра вычислялись расстояния между соседними точками, причем необходимо учесть, что для последней точки «соседней» является первая точка. При расчете площади вычислялась площадь одного сегмента (треугольник с вершиной в начале координат и основанием — отрезком, соединяющим соседние вершины многоугольника), а затем полученное значение умножалось на количество таких сегментов (равное количеству вершин).

Матричная экспонента

В следующем примере представлена программа, с помощью которой вычисляется матричная экспонента. Известно, что экспоненциальная функция от действительного аргумента x вычисляется в виде ряда:

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Если речь идет о матричной экспоненте, то ее можно вычислять по тому же принципу. В частности, если A — некоторая квадратная матрица, то по определению матричной экспонентой от этой матрицы называется матрица $\exp(A)$, которая вычисляется так:

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!}.$$

Здесь опять же по определению полагают, что матрица A в нулевой степени равняется единичной матрице E (по диагонали единицы, все остальные элементы — нули), то есть $A^0 = E$.

Разумеется, на практике при вычислении бесконечного ряда, в том числе и для матричной экспоненты, ограничиваются конечным числом слагаемых. Что касается непосредственно ряда, то для его вычисления с матрицами нужно уметь проделывать как минимум три операции: складывать матрицы, умножать матрицу на матрицу и умножать матрицу на действительное число. Именно такие действия (методы для их реализации) нужно предусмотреть в программе.

Для реализации проекта по вычислению матричной экспоненты создается класс `MatrixExp` (листинг 5.10). Полем этого класса является ссылка `matrix` на двухмерный массив, через который и мы реализуем квадратные матрицы. Таким образом, матрица упакована в объект `MatrixExp`. Операции с матрицами, в том числе и такие, как вычисление экспоненты, реализуются через методы класса.

Листинг 5.10. Матричная экспонента

```
class MatrixExp{
// Количество слагаемых в ряде для экспоненты:
private final static int N=100;
// Размер матрицы:
private int n;
```

продолжение

Листинг 5.10 *(продолжение)*

```
// Ссылка на матрицу:
private double[][] matrix;
// Конструктор (размер матрицы и диапазон случайных значений):
MatrixExp(int n,double Xmin,double Xmax){
double x=Math.abs(Xmax-Xmin);
int i,j;
this.n=n;
matrix=new double[n][n];
for(i=0;i<n;i++){
for(j=0;j<n;j++){
matrix[i][j]=x*Math.random()+Xmin;}
}}
// Конструктор (на основе существующей матрицы):
MatrixExp(double[][] matrix){
this.n=matrix[0].length;
this.matrix=new double[n][n];
int i,j;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
this.matrix[i][j]=matrix[i][j];}}
}
// Конструктор (единичная матрица заданного ранга):
MatrixExp(int n){
this.n=n;
matrix=new double[n][n];
int i,j;
for(i=0;i<n;i++){
for(j=0;j<i;j++){
matrix[i][j]=0;}
matrix[i][i]=1;
for(j=i+1;j<n;j++){
matrix[i][j]=0;}}
}
// Конструктор (заполнение одним числом):
MatrixExp(int n,double a){
this.n=n;
matrix=new double[n][n];
int i,j;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
matrix[i][j]=a;}
}}
// Метод для отображения матрицы:
void show(){
int i,j;
for(i=0;i<n;i++){
```

```
for(j=0;j<n-1;j++){
System.out.print(Math.round(1000*matrix[i][j])/1000.0+"\t");}
System.out.print(Math.round(1000*matrix[i][n-1])/1000.0+"\n");
}}
// Метод для вычисления суммы матриц:
MatrixExp sum(MatrixExp B){
MatrixExp t=new MatrixExp(n,0);
int i,j;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
t.matrix[i][j]=matrix[i][j]+B.matrix[i][j];}}
return t;}
// Метод для вычисления произведения матрицы на число:
MatrixExp prod(double x){
MatrixExp t=new MatrixExp(matrix);
int i,j;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
t.matrix[i][j]*=x;}}
return t;}
// Метод для вычисления произведения матриц:
MatrixExp prod(MatrixExp B){
MatrixExp t=new MatrixExp(n,0);
int i,j,k;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
for(k=0;k<n;k++){
t.matrix[i][j]+=matrix[i][k]*B.matrix[k][j];}
}}
return t;}
// Метод для вычисления матричной экспоненты:
MatrixExp mExp(){
MatrixExp t,q;
// Начальное значение - единичная матрица:
t=new MatrixExp(n);
// Начальная добавка:
q=new MatrixExp(matrix);
int i;
// Вычисление ряда для экспоненты:
for(i=1;i<=N;i++){
t=t.sum(q);
q=q.prod(this).prod(1.0/(i+1));}
return t;}
}
class MatrixExpDemo{
public static void main(String[] args){
```

продолжение

Листинг 5.10 *(продолжение)*

```
// Исходная матрица (поле объекта):
MatrixExp A=new MatrixExp(3,-1,1);
System.out.println("Матрица A:");
// Отображение исходной матрицы:
A.show();
System.out.println("Матрица exp(A):");
// Вычисление матричной экспоненты и отображение результата:
A.mExp().show();
}}
```

В частности, класс имеет метод `show()` для отображения матрицы `matrix`, метод `sum()` для вычисления суммы матриц, перегруженный метод `prod()` для вычисления произведения матриц и умножения матрицы на число, а также метод `mExp()` для вычисления матричной экспоненты. Кроме того, в классе имеется несколько конструкторов для создания объекта с полем:

- единичной матрицей;
- матрицей, заполненной случайными числами;
- матрицей на основе уже существующей матрицы;
- матрицей, заполненной одним и тем же числом.

Остановимся подробнее на программном коде класса `MatrixExp`. Кроме упоминавшегося уже поля `matrix`, класс имеет еще два закрытых поля: статическое целочисленное неизменное (`final`) поле `N` определяет количество слагаемых при вычислении ряда для экспоненты, а целочисленное поле `n` — размер поля-матрицы класса.

Конструктор класса, создающий массив, на который ссылается поле `matrix`, заполняется случайными числами. Конструктор имеет три аргумента: целочисленный размер матрицы и границы диапазона, в котором генерируются случайные числа для заполнения поля-матрицы объекта. При создании объекта на основе уже существующей матрицы ссылка на соответствующий двухмерный массив передается конструктору. На основе этой ссылки вычисляется размер массива (по первой строке, при этом априори предполагается, что массив по каждому из индексов имеет одинаковый размер). Затем после создания объекта-массива выполняется поэлементное копирование. При создании объекта с единичной матрицей аргументом конструктору передается целое число, которое определяет размер этой матрицы. Наконец, для заполнения всех элементов матрицы одинаковыми числами кроме размера матрицы конструктору нужно передать и число для заполнения.

Методом `show()` осуществляется вывод значений поля-матрицы объекта на экран. У метода нет аргументов и он не возвращает результата. Элементы матрицы `matrix` выводятся построчно, причем предварительно соответствующее значение округляется и на экран выводится не больше трех цифр после десятичной точки. Обращаем внимание, что округляется отображаемое значение. Соответствующий элемент матрицы остается неизменным.

Метод sum() предназначен для вычисления суммы матриц. Аргументом ему передается объект класса MatrixExp. Его поле-матрица matrix складывается с полем-матрицей matrix объекта, из которого вызывается метод. В теле метода создается локальный объект класса MatrixExp. Поле matrix этого локального объекта вычисляется как поэлементная сумма, для чего используется вложенный цикл. После этого локальный объект возвращается как результат метода.

Методу prod() в качестве аргумента можно передать как число, так и объект класса MatrixExp. В первом случае на основе объекта вызова создается локальный объект, все элементы поля-матрицы matrix которого затем умножаются на число, переданное аргументом методу. После этого объект возвращается как результат метода.

При передаче методу prod() в качестве аргумента объекта класса MatrixExp вычисляется произведение матриц. Первая матрица произведения — поле matrix объекта, из которого вызывается метод, вторая матрица — поле matrix объекта, указанного аргументом метода. Произведение вычисляется по правилу вычисления произведения для матриц (сумма произведений соответственных элементов строк первой и столбцов второй матрицы). Результатом метода является объект класса MatrixExp с полем matrix, в которое записан результат произведения матриц. При создании локального объекта класса MatrixExp, который затем возвращается в качестве результата (после выполнения всех необходимых вычислений), использован конструктор заполнения элементов поля-матрицы создаваемого объекта одинаковыми числами (в данном случае нулями).

Вычисление значений элементов матрицы-результата осуществляется с помощью вложенного тройного цикла. Две внешние индексные переменные нумеруют элементы матрицы-результата, а еще одна внутренняя переменная служит для вычисления суммы по элементам строк и столбцов матриц-операндов.

При вычислении матричной экспоненты в методе mExp() объявляются две объектные переменные t и q класса MatrixExp. Значения этим переменным присваиваются командами t=new MatrixExp(n) и q=new MatrixExp(matrix) соответственно. В первом случае создается единичная матрица, и ссылка на нее присваивается в качестве значения объектной переменной t. Это — начальное значение для вычисления матричной экспоненты. Второй командой создается копия исходной матрицы (поле matrix), и ссылка на результат записывается в переменную q. Это — «добавка», то есть следующее после текущего слагаемое в ряде для матричной экспоненты. На первом итерационном шаге эта добавка (ее поле-массив matrix) должна совпадать с исходной матрицей. В общем случае, на k-м итерационном шаге добавка q_k определяется как $q_k = \frac{A^k}{k!}$, где через A обозначена исходная матрица. Принимая во внимание, что $q_{k+1}/q_k = \frac{A}{k+1}$, легко приходим к выводу, что на каждом итерационном шаге для вычисления добавки следующего итерационного шага текущее значение-массив объекта, на который ссылается переменная q, нужно умножить (по правилу умножения матриц) на

исходную матрицу, и результат умножить на число $\frac{1}{k+1}$, где через k обозначено текущее значение индексной переменной. На следующем итерационном шаге значение-массив объекта, на который ссылается переменная q, прибавляется (по правилу сложения матриц) к массиву объекта, на который ссылается переменная t, и ссылка на полученный в результате таких действий объект присваивается в качестве нового значения переменной t. Последняя операция реализуется с помощью команды `t=t.sum(q)` (первая команда в цикле в методе `mExp()`). Что касается процедуры изменения поля-массива объекта, на который ссылается переменная q, то для этих целей предназначена команда `q=q.prod(this).prod(1.0/(i+1))`. Интерес представляет правая часть этого выражения. Она формально состоит из двух частей. Инструкцией `q.prod(this)` вычисляется произведение матриц (той, что записана в объекте q, и той, что записана в объекте, из которого вызывается метод `mExp()`), и в качестве результата возвращается объект с соответствующим полем-массивом. Ссылка на этот объект никуда не записывается, поэтому объект анонимный. Из этого анонимного объекта вызывается метод `prod()` с числовым аргументом (инструкция `q.prod(this).prod(1.0/(i+1))`). В результате на основе анонимного объекта создается еще один объект, элементы матрицы которого получаются умножением соответствующих элементов матрицы анонимного объекта на число, указанное аргументом метода `prod()`. Ссылка на результат присваивается в качестве значения объектной переменной q. Обращаем внимание, что при передаче числового аргумента методу `prod()` единица в числителе вводилась с десятичной точкой дабы избежать целочисленного деления.

После выполнения всех необходимых итераций в качестве значения метода `mExp()` возвращается объект, на который ссылается переменная t.

В главном методе программы в классе `MatrixExpDemo` командой `MatrixExp A=new MatrixExp(3,-1,1)` создается новый объект класса `MatrixExp` с полем-матрицей ранга 3, заполненной случайными числами в диапазоне от −1 до 1. Ссылка на объект записывается в объектную переменную `A`. Выводится матрица на экран с помощью команды `A.show()`. Командой `A.mExp().show()` вычисляется матричная экспонента, и результат выводится на экран. В данном случае также используется анонимный объект — результат выполнения инструкции `A.mExp()`. Поскольку метод `mExp()` возвращает результат, являющийся объектом класса `MatrixExp` с вычисленной матричной экспонентой в качестве поля-массива, то из этого объекта (имеется виду объект `A.mExp()`) можно вызвать метод `show()`, что и происходит. Результат выполнения программы может иметь следующий вид:

```
Матрица A:
0.844    0.797    0.095
0.891    -0.407   -0.528
-0.549   0.168    0.63
Матрица exp(A):
2.931    1.18-0.111
1.532    1.068    -0.621
-1.131   -0.121   1.825
```

В том, что программа работает корректно, желающие могут убедиться, выбрав в качестве начальной матрицы A единичную матрицу. В результате получим матрицу, на главной диагонали которой размещены константы Эйлера (значение 2,718). Отметим, что по описанному принципу можно вычислять не только матричные экспоненты, но и матричные синусы, косинусы и т. д.

Операции с векторами

Следующая достаточно простая программа служит иллюстрацией к созданию класса для реализации векторов в трехмерном пространстве и выполнения основных операций с ними. Код программы приведен в листинге 5.11.

Листинг 5.11. Операции с векторами

```
class Vector{
// Поле - ссылка на вектор (массив):
private double[] vect=new double[3];
// Метод для определения компонентов вектора:
void set(double x,double y,double z){
vect[0]=x;
vect[1]=y;
vect[2]=z;}
// Метод для определения компонентов вектора:
void set(double[] params){
for(int i=0;i<3;i++)
vect[i]=params[i];}
void set(){
set(0,0,0);}
// Метод для отображения компонентов вектора:
void show(){
double[] x=new double[3];
for(int i=0;i<3;i++)
x[i]=Math.round(vect[i]*100)/100.0;
System.out.print("<"+x[0]+";"+x[1]+";"+x[2]+">");}
// Метод для отображения компонентов вектора:
void show(char s){
show();
System.out.print(s);}
// Метод для вычисления суммы векторов:
Vector plus(Vector b){
Vector t=new Vector();
for(int i=0;i<3;i++)
t.vect[i]=vect[i]+b.vect[i];
return t;}
// Метод для вычисления разности векторов:
Vector minus(Vector b){
Vector t=new Vector();
```

продолжение

Листинг 5.11 *(продолжение)*

```
for(int i=0;i<3;i++)
t.vect[i]=vect[i]-b.vect[i];
return t;}
// Метод для вычисления произведения вектора на число:
Vector prod(double x){
Vector t=new Vector();
for(int i=0;i<3;i++)
t.vect[i]=vect[i]*x;
return t;}
// Метод для вычисления скалярного произведения векторов:
double prod(Vector b){
double x=0;
for(int i=0;i<3;i++)
x+=vect[i]*b.vect[i];
return x;}
// Метод для вычисления векторного произведения векторов:
Vector vprod(Vector b){
Vector t=new Vector();
for(int i=0;i<3;i++)
t.vect[i]=vect[(i+1)%3]*b.vect[(i+2)%3]-vect[(i+2)%3]*b.vect[(i+1)%3];
return t;}
// Метод для вычисления смешанного произведения векторов:
double mprod(Vector b,Vector c){
return vprod(b).prod(c);}
// Метод для деления вектора на число:
Vector div(double x){
Vector t=new Vector();
for(int i=0;i<3;i++)
t.vect[i]=vect[i]/x;
return t;}
// Метод для вычисления модуля вектора:
double module(){
return Math.sqrt(prod(this));}
// Метод для вычисления угла между векторами (в радианах):
double ang(Vector b){
double z;
z=prod(b)/module()/b.module();
return Math.acos(z);}
// Метод для вычисления угла между векторами (в градусах):
double angDeg(Vector b){
return Math.toDegrees(ang(b));}
// Метод для вычисления площади параллелограмма:
double square(Vector b){
Vector t;
t=vprod(b);
return t.module();}
```

```
// Конструктор класса:
Vector(double[] params){
set(params);}
// Конструктор класса:
Vector(double x,double y,double z){
set(x,y,z);}
// Конструктор класса:
Vector(){
set();}
}
class VectorDemo{
public static void main(String[] args){
Vector a=new Vector(1,0,0);
Vector b=new Vector(new double[]{0,1,0});
Vector c;
System.out.println("Векторное произведение:");
(c=a.vprod(b)).show('\n');
System.out.println("Смешанное произведение: "+a.mprod(b,c));
System.out.println("Линейная комбинация векторов:");
a.prod(3).plus(b.div(2)).minus(c).show('\n');
a.set(4,0,-3);
b.set(0,10,0);
System.out.println("Угол между векторами (в градусах): "+a.angDeg(b));
System.out.println("Площадь параллелограмма: "+a.square(b));}
}
```

Для работы с векторами предлагается класс `Vector`. У класса есть закрытое полессылка на массив из трех элементов `vect`. Поле объявлено как закрытое, хотя настоятельной необходимости в данном случае в этом нет. Перегруженный метод `set()` предназначен для присваивания значения элементам массива `vect`. Методу в качестве аргумента могут передаваться три числа типа `double` (значения трех элементов массива `vect`) и один аргумент — ссылка на массив из трех элементов, или не передаваться вовсе (в этом случае создается нуль-вектор). В свою очередь, версия метода `set()` без аргументов реализована на основе вызова версии метода `set()` с тремя нулевыми аргументами.

В соответствии с вариантами вызова метода `set()` в классе создаются и варианты перегружаемого конструктора. Конструктору могут передаваться те же аргументы, что и методу `set()`, причем реализованы разные версии конструктора на основе вызова соответствующих версий метода `set()`.

Методу `show()`, предназначенному для вывода компонентов вектора на экран, можно передать символьный аргумент (тип `char`), или же метод вызывается без аргументов. Во втором случае выполняется вывод в треугольных скобках значений элементов вектора (с точностью до двух цифр после запятой) без перехода к новой строке. Если понадобится, чтобы после вывода значений компонентов вектора следующее сообщение выводилось в новой строке, можно вызвать метод `show()` с аргументом — символом перехода к новой строке `'\n'`.

Ряд методов предназначен для выполнения базовых операций с векторами. Так, с помощью метода `plus()` вычисляется сумма двух векторов. В результате создается новый объект класса `Vector`, элементы поля `vect` которого равны сумме соответственных элементов полей `vect` исходных объектов. Аналогично вычисляется разность двух векторов методом `minus()`, только в этом случае вычисляется разность компонентов.

Если в качестве аргумента методу `prod()` передается действительное число, вычисляется произведение вектора на скаляр — каждый элемент умножается на число, переданное методу в качестве аргумента. Если же аргумент — объект класса `Vector`, методом `prod()` вычисляется скалярное произведение векторов (сумма произведений соответственных компонентов полей `vect` объектов). Для деления вектора на число предназначен метод `div()`. В этом случае каждый элемент массива `vect` делится на число — аргумент метода.

При вычислении векторного произведения использовано то свойство, что если вектор c равен векторному произведению векторов a и b (то есть $c = a \times b$), то компоненты вектора c вычисляются на основе компонентов векторов a и b согласно правилу $c_k = a_{k+1}b_{k+2} - a_{k+2}b_{k+1}$, причем индексирование выполняется с учетом циклической перестановки индексов — следующим после последнего индекса является первый и т. д. Именно для реализации правила циклической перестановки индексов при индексации элементов в методе `vprod()` используется оператор `%` вычисления остатка от целочисленного деления.

Смешанное произведение векторов вычисляется методом `mprod()`. Результатом является скаляр (значение типа `double`). Смешанное произведение — это векторное произведение векторов a и b, скалярно умноженное на вектор c. Векторы b и c в виде объектов класса `Vector` передаются методу `mprod()`. При вычислении смешанного произведения использован метод вычисления векторного произведения `vprod()`. При этом результат вычисляется инструкцией `vprod(b).prod(c)`, где `b` и `c` — аргументы метода. Здесь использовано то свойство, что результатом инструкции `vprod(b)` является объект класса `Vector`, соответствующий векторному произведению поля `vect` объекта, из которого вызывается метод `mprod()`, и поля `vect` объекта `b`. Для вычисления скалярного произведения из объекта `vprod(b)` вызывается метод `prod(c)` с аргументом `c` (второй аргумент метода `mprod()`).

Методом `module()` в качестве результата возвращается модуль вектора — корень квадратный из скалярного произведения вектора на самого себя. При ссылке на объект вызова использовано ключевое слово `this`.

Методами `ang()` и `angDeg()` возвращается в качестве значения угол между векторами (в радианах и градусах соответственно). Использовано свойство, что косинус угла между векторами равен отношению скалярного произведения векторов к произведению их модулей. Наконец, методом `square()` возвращается модуль векторного произведения двух векторов. Это число, которое равняется площади параллелограмма, образованного двумя данными векторами.

В главном методе программы в методе VectorDemo проверяется функциональность созданного программного кода. Результат выполнения программы имеет следующий вид:

```
Векторное произведение:
<0.0;0.0;1.0>
Смешанное произведение: 1.0
Линейная комбинация векторов:
<3.0;0.5;-1.0>
Угол между векторами (в градусах): 90.0
Площадь параллелограмма: 50.0
```

Стоит обратить внимание на следующие команды в главном методе программы:

```
(c=a.vprod(b)).show('\n');
a.prod(3).plus(b.div(2)).minus(c).show('\n');
```

В первом случае объекту c в качестве значения присваивается результат операции a.vprod(b), и поскольку оператор присваивания возвращает значение, для объекта-результата вызывается метод show() (с символьным аргументом). Второй командой для векторов, представленных объектами a, b и c, вычисляется линейная комбинация $3a + b/2 - c$.

Операции с полиномами

В некотором смысле перекликается с рассмотренным примером следующая программа. В ней для работы с выражениями полиномиального типа создается класс Polynom, в котором описаны методы сложения, вычитания, умножения полиномов, умножения и деления полинома на число и вычисления производной для полинома. Рассмотрим программный код, представленный в листинге 5.12.

Листинг 5.12. Операции с полиномами

```
class Polynom{
// Полином степени n-1:
private int n;
// Коэффициенты полинома:
private double[] a;
// Определение коэффициентов полинома на основе массива:
void set(double[] a){
this.n=a.length;
this.a=new double[n];
int i;
for(i=0;i<n;i++)
this.a[i]=a[i];
}
// Определение коэффициентов полинома
// (аргументы - размер массива и число для заполнения):
void set(int n,double z){
```

продолжение

Листинг 5.12 *(продолжение)*

```
this.n=n;
this.a=new double[n];
int i;
for(i=0;i<n;i++)
this.a[i]=z;
}
// Определение коэффициентов полинома
// (аргумент - размер массива, массив заполняется нулями):
void set(int n){
set(n,0);}
// Вычисление значения полинома в точке:
double value(double x){
double z=0,q=1;
for(int i=0;i<n;i++){
z+=a[i]*q;
q*=x;}
return z;}
// Отображение коэффициентов полинома:
void show(){
int i;
System.out.print("Степень x:\t");
for(i=0;i<n-1;i++){
System.out.print("  "+i+"\t");}
System.out.println("  "+(n-1));
System.out.print("Коэффициент:\t");
for(i=0;i<n-1;i++){
System.out.print(a[i]+"\t");}
System.out.println(a[n-1]);
}
// Отображение значения полинома в точке:
void show(double x){
System.out.println("Значение аргумента   x="+x);
System.out.println("Значение полинома P(x)="+value(x));
}
// Производная от полинома:
Polynom diff(){
Polynom t=new Polynom(n-1);
for(int i=0;i<n-1;i++)
t.a[i]=a[i+1]*(i+1);
return t;}
// Производная от полинома порядка k:
Polynom diff(int k){
if(k>=n) return new Polynom(1);
if(k>0) return diff().diff(k-1);
else return new Polynom(a);
}
```

```
// Сумма полиномов:
Polynom plus(Polynom Q){
Polynom t;
int i;
if(n>=Q.n){
t=new Polynom(a);
for(i=0;i<Q.n;i++)
t.a[i]+=Q.a[i];}
else{
t=new Polynom(Q.a);
for(i=0;i<n;i++)
t.a[i]+=a[i];
}
return t;}
// Разность полиномов:
Polynom minus(Polynom Q){
return plus(Q.prod(-1));}
Polynom div(double z){
return prod(1/z);}
// Произведение полинома на число:
Polynom prod(double z){
Polynom t=new Polynom(a);
for(int i=0;i<n;i++)
a[i]*=z;
return t;}
// Произведение полинома на полином:
Polynom prod(Polynom Q){
int N=n+Q.n-1;
Polynom t=new Polynom(N);
for(int i=0;i<n;i++){
for(int j=0;j<Q.n;j++){
t.a[i+j]+=a[i]*Q.a[j];}
}
return t;}
// Конструкторы класса:
Polynom(double[] a){
set(a);
}
Polynom(int n,double z){
set(n,z);}
Polynom(int n){
set(n);}
}
class PolynomDemo{
public static void main(String[] args){
```

продолжение

Листинг 5.12 *(продолжение)*

```
// Коэффициенты для полинома:
double[] coefs=new double[]{3,-2,-1,0,1};
// Создание полинома:
Polynom P=new Polynom(coefs);
System.out.println("\tКоэффициенты исходного полинома:");
// Коэффициенты полинома:
P.show();
System.out.println("\tЗначение полинома в точке:");
// Значение полинома для единичного аргумента:
P.show(1);
System.out.println("\tВторая производная:");
// Вычисление второй производной для полинома:
Polynom Q=P.diff(2);
// Результат вычисления производной (коэффициенты):
Q.show();
System.out.println("\tСумма полиномов:");
// Сумма полиномов (результат):
Q.plus(P).show();
System.out.println("\tПроизведение полиномов:");
// Произведение полиномов (результат):
Q.prod(P).show();
}}
```

В классе `Polynom` закрытое целочисленное поле `n` определяет степень полинома (степень полинома равняется `n-1`, и такой полином определяется набором из `n` коэффициентов), а закрытое поле-массив `a` (ссылка на массив типа `double`) предназначено для записи коэффициентов полинома.

Для заполнения поля `a` предназначен перегружаемый метод `set()`. В методе предусмотрена возможность передавать для заполнения коэффициентов полинома ссылки на массив, заполнять массив одинаковыми числами, передав аргументом методу размер массива и число для заполнения, а также описан частный случай, когда аргументом методу передается только размер массива (при этом массив заполняется нулями). Реализация последней версии метода `set()` базируется на вызове варианта этого же метода с первым аргументом — размером массива, и нулевым вторым аргументом.

Метод `value()` имеет аргумент типа `double` и в качестве результата возвращает значение полинома в точке, то есть для переменной, переданной аргументом метода.

Перегружаемый метод `show()` имеет две версии: без аргументов и с одним аргументом типа `double`. В первом случае на экран вместе с дополнительной информацией выводятся значения коэффициентов полинома. Если методу `show()` передается аргумент (значение типа `double`), то на экран выводится сообщение о значении полинома в этой точке.

В классе `Polynom` предусмотрена возможность вычислять производные, причем произвольного порядка. Результатом вычисления производной от полинома также

является полином. Задача по вычислению производной от полинома сводится, по большому счету, к расчету на основе коэффициентов исходного полинома коэффициентов полинома-производной. Поскольку для степенной функции $y = x^n$ производная $\frac{dy}{dx}$ определяется как $\frac{dy}{dx} = nx^{n-1}$, а производная суммы функций равняется сумме производных, то производной для полинома $P(x) = \sum_{k=0}^{n} a_k x^k$ является функция-полином:

$$P'(x) = \sum_{k=1}^{n} k a_k x^{k-1} = \sum_{k=0}^{n-1} (k+1) a_{k+1} x^k \equiv \sum_{k=0}^{n-1} b_k x^k.$$

Коэффициенты полинома-производной b_k связаны с коэффициентами a_k исходного полинома соотношением $b_k = (k+1)a_{k+1}$ для $k = 0, 1, ..., n-1$, а старший коэффициент $b_n = 0$. Это означает, фактически, что производная является полиномом степени, на единицу меньшей, чем исходный полином. Данное обстоятельство, а также соотношения между коэффициентами исходного полинома и полинома-производной нашли отображение в коде метода `diff()`, которым в качестве результата возвращается объект класса `Polynom`. Это первая производная для полинома, реализованного через объект, из которого вызывается метод. В теле метода командой `Polynom t=new Polynom(n-1)` создается объект `t`, соответствующий полиному степени, на единицу меньшей, чем исходный. При создании объекта использован конструктор с одним аргументом (размер массива), поэтому коэффициенты поля `a` этого объекта заполняются нулями (хотя в данном случае это не принципиально). В цикле с индексной переменной `i` командой `t.a[i]=a[i+1]*(i+1)` выполняется последовательное заполнение коэффициентов полинома-производной. После этого объект `t` возвращается как результат метода.

Метод `diff()` перегружается. Если этому методу передать целочисленный аргумент, в результате возвращается производная соответствующего порядка. При этом если аргумент `k` (порядок производной) превышает степень полинома (равняется `n-1`), командой `if(k>=n) return new Polynom(1)` создается анонимный объект, соответствующий полиному нулевой степени (то есть это число с единственным нулевым коэффициентом), и этот полином возвращается в качестве результата. Здесь принято во внимание, что если порядок производной превышает показатель степенной функции, производная тождественно равна нулю. В противном случае при положительном (больше нуля) значении порядка производной в качестве результата возвращается объект `diff().diff(k-1)`. Иначе возвращается анонимный объект, который создается командой `new Polynom(a)`. Это — копия исходного полинома. Этим реализовано формальное правило, гласящее, что производная нулевого порядка по определению совпадает с исходной функцией.

Сделаем несколько замечаний относительно команды `diff().diff(k-1)`. В ней реализован рекурсивный вызов метода `diff()` одновременно с вызовом версии этого метода без аргумента. В частности, команда `diff().diff(k-1)` реализует

правило, согласно которому производная порядка k — это производная порядка $k-1$ от первой производной. Первая производная вычисляется инструкцией. Результатом инструкции является объект (в данном случае анонимный), который соответствует полиному-производной. Из этого анонимного объекта инструкцией `diff(k-1)` снова вызывается метод для вычисления производной, но уже меньшего порядка.

Методом `plus()` в качестве результата вычисляется объект класса `Polynom`, соответствующий сумме двух полиномов — один реализован через объект вызова, а объект для второго полинома передается аргументом методу. Общий принцип вычисления суммы полиномов состоит в том, что нужно сложить коэффициенты, соответствующие одинаковым показателям степени переменной полинома. Главная проблема в данном случае связана с тем, что складываемые полиномы могут иметь разную степень и, как результат, разные размеры полей-массивов `a` соответствующих объектов. Поэтому на начальном этапе проверяется, какой из объектов имеет больший размер поля-массива. На основе этого объекта создается его локальная копия. Затем перебираются элементы второго объекта (того, где поле-массив имеет меньший размер), и эти элементы прибавляются к соответствующим элементам локального объекта. После этого локальный объект возвращается в качестве результата метода.

Разность полиномов вычисляется методом `minus()`. С помощью метода `prod()` полином, соответствующий объекту-аргументу метода `minus()`, умножается на −1, после чего с помощью метода `plus()` полученный в результате полином прибавляется к исходному, реализованному через объект вызова метода `minus()`. Вся эта процедура по вычислению объекта-результата разности двух полиномов реализована инструкцией `plus(Q.prod(-1))` где `Q` — объект класса `Polynom`, переданный аргументом методу `minus()`. Что касается метода `prod()`, то он позволяет умножать полином (объект класса `Polynom`) на число и на другой полином. При умножении полинома на число аргументом методу `prod()` передается значение типа `double`. На это число умножается каждый коэффициент полинома. Если же аргументом методу `prod()` передать объект класса `Polynom`, вычисляется произведение полиномов (один реализован через объект вызова, второй — через объект-аргумент метода). Результатом также является объект класса `Polynom`. При вычислении его параметров (размера поля-массива и значения элементов этого массива) приняты во внимание следующие обстоятельства. Во-первых, если умножаются полиномы степени n и m, результатом будет полином степени $n+m$. Поскольку степень полинома на единицу меньше количества коэффициентов, которыми однозначно определяется полином, то размер поля-массива объекта-результата на единицу меньше, чем сумма размеров исходных объектов. Это правило реализовано в команде `int N=n+Q.n-1`, где `Q` является объектом класса `Polynom`, который передается методу `prod()`. Переменная `N` таким образом определяет размер поля-массива объекта-результата метода `prod()`. Локальный объект, возвращаемый в последующем в качестве результата, создается командой `Polynom t=new Polynom(N)`. Начальные значения элементов поля-массива локального объекта `t` равны нулю. Дальше запускается

двойной цикл, индексная переменная в одном цикле перебирает элементы первого поля-массива первого объекта, а вторая — второго. Изменение значений поля-массива локального объекта выполняется командой `t.a[i+j]+=a[i]*Q.a[j]`. В данном случае принято во внимание, что индекс элемента массива `a` совпадает со степенью переменной полинома в соответствующем слагаемом. Умножение коэффициентов, соответствующих индексам `i` и `j` в полиноме-результате, соответствует степени `i+j` переменной, то есть коэффициенту с индексом `i+j`. В частности, если

$$P(x)=\sum_{i=0}^{n}a_i x^i \quad \text{и} \quad Q(x)=\sum_{j=0}^{m}b_j x^j,$$

то:

$$R(x)\equiv P(x)Q(x)=\sum_{i=0}^{n}a_i x^i\sum_{j=0}^{m}b_j x^j=\sum_{\substack{0\le i\le n,\\ 0\le j\le m}}a_i b_j x^{i+j}.$$

Другими словами, чтобы рассчитать коэффициент для полинома-результата с индексом k, необходимо найти сумму $\sum_{i+j=k}a_i b_j$ произведений коэффициентов исходных полиномов, сумма индексов которых равняется k. Именно такой подход и был реализован в программе.

Конструкторы класса `Polynom` реализованы на основе различных версий метода `set()`, которые уже описывались и, думается, особых комментариев не требуют.

Что касается главного метода программы в классе `PolynomDemo`, то там проверяется работа некоторых методов, описанных в классе `Polynom`. Обращаем внимание на способ вызова методов из анонимных объектов. Например, командой `Polynom Q=P.diff(2)` на основе определенного ранее объекта `P` вычисляется объект для второй производной от исходного полинома, и ссылка на этот объект записывается в объектную переменную `Q`. Аналогично, командой `Q.plus(P).show()` вычисляется анонимный объект для суммы полиномов (инструкция `Q.plus(P)`), и из этого объекта вызывается метод `show()`. Результат выполнения программы имеет следующий вид:

```
        Коэффициенты исходного полинома:
Степень x:         0        1        2        3        4
Коэффициент:     3.0     -2.0     -1.0      0.0      1.0
        Значение полинома в точке:
Значение аргумента   x=1.0
Значение полинома P(x)=1.0
        Вторая производная:
Степень x:         0        1        2
Коэффициент:     -2.0      0.0     12.0
        Сумма полиномов:
Степень x:         0        1        2        3        4
Коэффициент:      1.0     -2.0     11.0      0.0      1.0
```

```
        Произведение полиномов:
Степень x:        0       1       2       3       4       5       6
Коэффициент:   -6.0     4.0    38.0   -24.0   -14.0     0.0    12.0
```

Желающие могут проверить, что все коэффициенты и значения вычислены корректно. Отметим также, что вывод результатов в данном случае реализован с помощью специального метода `show()` класса `Polynom`. На практике существует более простой, надежный и эффективный способ обеспечить приемлемый способ вывода данных объекта. Состоит он в переопределении метода `toString()`. Подробнее речь о работе с текстом и, в частности, о работе с методом `toString()` идет в главе 8.

Бинарное дерево

Рассмотрим программу, в которой на основе конструкторов класса создается бинарное дерево объектов — каждый объект имеет по две ссылки на объекты того же класса. Пример учебный, поэтому ситуация упрощена до предела. Каждый объект, кроме прочего, имеет три поля. Символьное (типа `char`) поле `Level` определяет уровень объекта: например, в вершине иерархии находится объект уровня `A`, который ссылается на два объекта уровня `B`, которые, в свою очередь, ссылаются в общей сложности на четыре объекта уровня `C` и т. д. Объекты нумеруются, для чего используется целочисленное поле `Number`. Нумерация выполняется в пределах одного уровня. Например, на верхнем уровне `A` всего один объект с номером 1. На втором уровне `B` два объекта с номерами 1 и 2. На третьем уровне `C` четыре объекта с номерами от 1 до 4 включительно и т. д. Описанная структура объектов представлена на рис. 5.2.

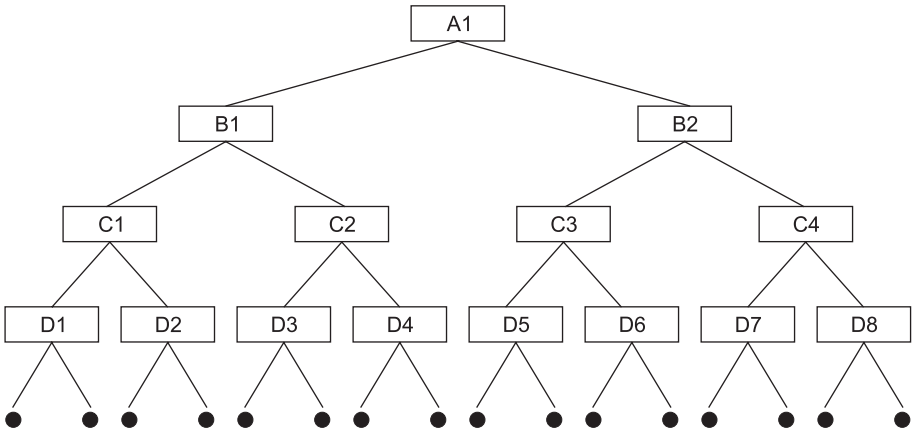

Рис. 5.2. Структура бинарного дерева объектов

Кроме метки уровня и номера объекта на уровне, каждый объект имеет еще и свой «идентификационный код». Этот код генерируется случайным образом при создании объекта и состоит по умолчанию из восьми цифр (количество цифр в коде определяется закрытым статическим целочисленным полем `IDnum`).

Что касается самого кода, то он записывается в целочисленный массив, на который ссылается переменная массива ID — закрытое поле класса ObjectTree (листинг 5.13). Каждая цифра кода записывается отдельным элементом соответствующего массива. Для генерирования (создания) кода объекта используется метод getID(). Для этого в теле метода командой ID[i]=(int)(Math.random()*10) в рамках цикла генерируются случайные целые числа (инструкцией (int)(Math.random()*10)) и записываются в качестве значений элементов массива ID.

Для вывода кода объекта используется закрытый метод showID(). Методом последовательно выводятся на экран элементы массива ID, при этом в качестве разделителя используется вертикальная черта. Сам метод вызывается в методе show(), который, в свою очередь, предназначен для отображения параметров объекта: уровня объекта в структуре, порядкового номера объекта на уровне и идентификационного кода объекта.

Открытые поля FirstRef и SecondRef являются объектными переменными класса ObjectTree и предназначены для записи ссылок на объекты следующего уровня. Присваивание значений этим переменным выполняется при вызове конструктора класса. Обратимся к листингу 5.13.

Листинг 5.13. Бинарное дерево объектов

```
class ObjectTree{
// Количество цифр в ID-коде объекта:
private static int IDnum=8;
// Уровень объекта (буква):
private char Level;
// Номер объекта на уровне:
private int Number;
// Код объекта (массив цифр):
private int[] ID;
// Ссылка на первый объект:
ObjectTree FirstRef;
// Ссылка на второй объект:
ObjectTree SecondRef;
// Метод для генерирования ID-кода объекта:
private void getID(){
ID=new int[IDnum];
for(int i=0;i<IDnum;i++)
ID[i]=(int)(Math.random()*10);
}
// Метод для отображения ID-кода объекта:
private void showID(){
for(int i=0;i<IDnum;i++)
System.out.print("|"+ID[i]);
System.out.print("|\n");
}
```

продолжение

Листинг 5.13 *(продолжение)*

```
// Метод для отображения параметров объекта:
void show(){
System.out.println("Уровень объекта: \t"+Level);
System.out.println("Номер на уровне: \t"+Number);
System.out.print("ID-код объекта: \t");
showID();
}
// Конструктор создания бинарного дерева:
ObjectTree(int k,char L,int n){
System.out.println("\tСоздан новый объект!");
Level=L;
Number=n;
getID();
show();
if(k==1){
FirstRef=null;
SecondRef=null;
}
else{
// Рекурсивный вызов конструктора:
FirstRef=new ObjectTree(k-1,(char)((int)L+1),2*n-1);
SecondRef=new ObjectTree(k-1,(char)((int)L+1),2*n);}
}}
class ObjectTreeDemo{
public static void main(String[] args){
// Дерево объектов:
ObjectTree tree=new ObjectTree(4,'A',1);
System.out.println("\tПроверка структуры дерева объектов!");
// Проверка структуры дерева объектов:
tree.FirstRef.SecondRef.FirstRef.show();
}}
```

Конструктор класса принимает три аргумента: целочисленный аргумент определяет количество уровней в структуре, начиная с текущего объекта, а символьные аргументы определяют метку уровня и номер объекта на уровне. Детальнее остановимся на коде конструктора, поскольку именно при его вызове создается вся структура бинарного дерева.

При вызове конструктора выводится сообщение о создании объекта, после чего на основе значений аргументов конструктора присваиваются значения полям `Level` и `Number`. Затем с помощью метода `getID()` генерируется идентификационный код объекта и методом `show()` выводится информация о созданном объекте. Вторая часть кода конструктора реализована через условную инструкцию. В ней первый аргумент конструктора проверяется на предмет равенства единице. Если первый аргумент равен единице (это означает, что после текущего объекта других объектов нет), полям-ссылкам `FirstRef` и `SecondRef` в качестве значений присваиваются нулевые ссылки (значение `null`), означающие, что

текущий объект не имеет ссылок на другие объекты. В противном случае, то есть если аргумент конструктора отличен от единицы, следующими командами создаются два новых объекта, и ссылки на них в качестве значений присваиваются полям FirstRef и SecondRef текущего объекта:

```
FirstRef=new ObjectTree(k-1,(char)((int)L+1),2*n-1);
SecondRef=new ObjectTree(k-1,(char)((int)L+1),2*n);
```

При этом конструкторам при создании новых объектов передается на единицу уменьшенный первый аргумент. Уровень новых создаваемых объектов вычисляется на основе текущего значения L для уровня текущего объекта как (char)((int)L+1). Инструкцией (int)L вычисляется код символа L, а затем, после увеличения кода символа на единицу, выполняется явное преобразование в символьный вид (инструкцией (char)). Для первого из двух создаваемых объектов номер объекта вычисляется на основе номера текущего объекта n как 2*n-1. Второй объект получает номер 2*n. Принцип нумерации рассчитан так, что если в вершине иерархии объект имеет номер 1, то на всех прочих уровнях объекты нумеруются последовательностью натуральных чисел. Таким образом, код конструктора класса реализован по рекурсивному принципу: в конструкторе вызывается конструктор, но с другими аргументами. Чтобы создать бинарное дерево, вызывается конструктор, первым аргументом которому передается количество уровней в бинарном дереве, имя (буква) для первого объекта и номер объекта в вершине иерархии объектов дерева.

В главном методе программы командой ObjectTree tree=new ObjectTree(4,'A',1) создается дерево из четырех уровней, после чего выполняется проверка созданной структуры: через систему последовательных ссылок вызывается метод show() для отображения параметров одного из объектов в структуре дерева (командой tree.FirstRef.SecondRef.FirstRef.show()). Результат выполнения программы может иметь следующий вид:

```
        Создан новый объект!
Уровень объекта:   A
Номер на уровне:   1
ID-код объекта:    |3|5|1|1|1|1|5|0|
        Создан новый объект!
Уровень объекта:   B
Номер на уровне:   1
ID-код объекта:    |3|6|4|8|2|2|2|9|
        Создан новый объект!
Уровень объекта:   C
Номер на уровне:   1
ID-код объекта:    |7|6|7|8|9|1|5|7|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   1
ID-код объекта:    |5|6|6|1|6|6|5|4|
```

```
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   2
ID-код объекта:    |3|6|5|0|2|1|6|7|
        Создан новый объект!
Уровень объекта:   C
Номер на уровне:   2
ID-код объекта:    |7|6|0|9|6|1|0|2|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   3
ID-код объекта:    |9|2|6|2|5|5|9|9|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   4
ID-код объекта:    |8|7|7|0|1|2|1|4|
        Создан новый объект!
Уровень объекта:   B
Номер на уровне:   2
ID-код объекта:    |1|7|3|8|8|1|9|2|
        Создан новый объект!
Уровень объекта:   C
Номер на уровне:   3
ID-код объекта:    |9|3|2|4|7|9|4|7|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   5
ID-код объекта:    |7|9|6|4|9|4|4|4|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   6
ID-код объекта:    |0|9|5|4|5|5|7|4|
        Создан новый объект!
Уровень объекта:   C
Номер на уровне:   4
ID-код объекта:    |4|1|6|6|9|7|8|1|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   7
ID-код объекта:    |6|4|7|5|0|0|0|3|
        Создан новый объект!
Уровень объекта:   D
Номер на уровне:   8
ID-код объекта:    |3|9|9|6|7|3|6|3|
```

```
      Проверка структуры дерева объектов!
Уровень объекта:   D
Номер на уровне:   3
ID-код объекта:    |9|2|6|2|5|5|9|9|
```

Сначала сверху вниз (см. рис. 5.2) создаются объекты, имеющие номер 1. Затем создается объект последнего уровня с номером 2. Далее создается объект предпоследнего уровня с номером 2, после чего объект предпоследнего уровня с номером 3, объект с номером 4 и т. д. Командой `tree.FirstRef.SecondRef.FirstRef.show()` метод `show()` вызывается из объекта уровня `D` с номером 3. Параметры именно этого объекта отображаются в конце программы.

Резюме

1. В Java методы классов могут перегружаться. В этом случае создается несколько вариантов одного метода. Они все имеют одинаковое название, но отличаются сигнатурой — типом результата, именем и (или) списком аргументов. Какой именно вариант метода необходимо вызывать при выполнении программы, определяется в зависимости от типа и количества переданных методу аргументов.
2. Конструктор класса — это метод, который вызывается автоматически при создании объекта класса. Имя конструктора совпадает с именем класса. Конструктор не возвращает результат, и идентификатор типа результата для него не указывается. Конструктору можно передавать аргументы и конструктор можно перегружать. Аргументы, которые передаются конструктору при создании объекта, указывается в круглых скобках после имени класса в инструкции вида `new имя_класса(аргументы)`.
3. Объекты могут передаваться методам в качестве аргументов, а также возвращаться методами в качестве результата. В этом случае формально метод объявляется так же, как и для базовых типов, только в качестве идентификатора типа указывается имя соответствующего класса.
4. Существует два способа передачи аргументов методам: по значению и по ссылке. При передаче аргументов по значению создается копия переменной, указанной аргументом, и все вычисления в методе осуществляются с этой копией. При передаче аргумента по ссылке операции выполняются непосредственно с аргументом. В Java переменные базовых типов передаются по значению, а объекты — по ссылке.

Глава 6. Наследование и переопределение методов

Не забывайте, что всему нашему делу
положила начало мышь.

У. Дисней

Одним из фундаментальных механизмов, лежащих в основе любого объектно-ориентированного языка, в том числе Java, является наследование. Наследование позволяет одним объектам получать характеристики других объектов. Представим себе ситуацию, когда на основе уже существующего, проверенного и работающего кода нужно создать новую программу. Есть два пути решения этой задачи. Во-первых, можно скопировать уже существующий код в новый проект и внести необходимые изменения. Во-вторых, в новом проекте можно сделать ссылку на уже существующий код. Второй вариант во многих отношениях предпочтительнее, поскольку позволяет сэкономить время и усилия на создание нового кода и обеспечивает более высокую степень совместимости программ. Ведь если окажется, что базовый код необходимо доработать, то это достаточно сделать единожды: поскольку код инкапсулируется через ссылку, внесенные изменения вступят в силу автоматически во всех местах, где этот код используется. Именно по этому принципу реализован механизм наследования.

С практической точки зрения наследование позволяет одним объектам получать (наследовать) свойства других объектов. Реализуется наследование путем создания классов на основе уже существующих классов. При этом члены класса, на основе которого создается новый класс, с некоторыми оговорками, автоматически включаются в новый класс. Кроме того, в создаваемый класс можно добавлять новые члены. Согласно общепринятой терминологии, класс, на основе которого создается новый класс, называется суперклассом. Новый создаваемый на основе суперкласса класс называется подклассом.

Создание подкласса

Чтобы продать что-нибудь ненужное, надо сначала купить
что-нибудь ненужное. А у нас денег нет!

Из м/ф «Трое из Простоквашино»

Как отмечалось, подкласс создается на основе суперкласса. Создание подкласса практически не отличается от создания обычного класса, только при создании

подкласса необходимо указать суперкласс, на основе которого создается подкласс.

Для реализации наследования в описании подкласса после имени класса указывается ключевое слово `extends` и имя суперкласса. Во всем остальном описание подкласса не отличается от описания обычного класса (то есть класса, который создается «с нуля»). Синтаксис описания подкласса имеет вид:

```
class A extends B{
// код
}
```

В данном случае подкласс `A` создается на основе суперкласса `B`. В результате подкласс `A` получает (наследует) открытые и защищенные члены класса `B`.

Обращаем внимание читателя, что в языке Java, в отличие от языка C++, отсутствует множественное наследование, то есть подкласс в Java может создаваться на основе только одного суперкласса. При этом в Java, как и в C++, существует многоуровневое наследование: подкласс, в свою очередь, может быть суперклассом для другого класса. Благодаря многоуровневому наследованию можно создавать целые цепочки связанных механизмом наследования классов. В листинге 6.1 приведен пример создания подкласса.

Листинг 6.1. Создание подкласса

```
class A{  // Суперкласс
int i,j;
void showij(){
System.out.println("Поля i и j: "+i+" и "+j);}
}
class B extends A{ // Подкласс
int k;
void showk(){
System.out.println("Поле k: "+k);}
void sum(){
// Обращение к наследуемым полям:
System.out.println("Сумма i+j+k="+(i+j+k));}
}
class AB{
public static void main(String arg[]){
// Объект суперкласса:
A SuperObj=new A();
// Объект подкласса:
B SubObj=new B();
SuperObj.i=10;
SuperObj.j=20;
SuperObj.showij();
SubObj.i=7;
SubObj.j=8;
```

продолжение

Листинг 6.1 *(продолжение)*

```
SubObj.k=9;
SubObj.showij();
SubObj.showk();
SubObj.sum();}
}
```

В программе описан суперкласс A, в котором объявлены два целочисленных поля i и j, а также метод showij() для отображения значений этих полей. На основе класса A создается класс B (подкласс суперкласса A). Непосредственно в теле класса B описано целочисленное поле k, а также методы showk() и sum() для вывода значения поля k и вычисления суммы полей i, j и k. Обращаем внимание, что хотя поля i и j непосредственно в классе B не описаны, в классе B о них известно, поскольку они наследуются этим классом (то есть у класса B имеются целочисленные поля i и j) и к ним можно обращаться.

В методе main() класса AB создаются два объекта: объект SuperObj суперкласса A и объект SubObj подкласса B. Полям i и j объекта SuperObj присваиваются значения 10 и 20 соответственно, после чего с помощью метода showij() значения полей выводятся на экран.

Полям i, j и k объекта SubObj присваиваются целочисленные значения 7, 8 и 9. Методом showij() отображаются значения полей i и j, а значение поля k отображается с помощью метода showk(). Наконец, сумма полей вычисляется методом sum(). Результат выполнения программы следующий:

```
Поля i и j: 10 и 20
Поля i и j: 7 и 8
Поле k: 9
Сумма i+j+k=24
```

Другими словами, ситуация такая, как если бы поля i и j, а также метод showij() были описаны в классе B. Достигается такой эффект благодаря наследованию.

Доступ к элементам суперкласса

Избытком мысли поразить нельзя,
Так удивите недостатком связи.

И. Гёте. Фауст

Не все члены суперкласса наследуются в подклассе. Наследование не распространяется на закрытые члены суперкласса. Другими словами, в подклассе закрытые члены суперкласса недоступны. Напомним, что закрытые члены класса объявляются с ключевым словом private, а по умолчанию, если никакое ключевое слово не указано, члены класса считаются открытыми. Именно поэтому, несмотря на отсутствие ключевых слов, описывающих уровень доступа, в рассмотренном примере никаких проблем с наследованием не возникало.

Для иллюстрации того, что происходит при наследовании, когда суперкласс содержит закрытые члены, рассмотрим пример в листинге 6.2.

Листинг 6.2. Закрытые члены суперкласса

```
class MySuperClass{  // Суперкласс
// Закрытое поле:
private int a;
// Закрытый метод:
private void showa(){
System.out.println("Поле a: "+a);}
// Открытый метод:
void seta(int n){
a=n;
showa();}
}
class MySubClass extends MySuperClass{ // Подкласс
int b;
void setall(int i,int j){
seta(i);
b=j;
System.out.println("Поле b: "+b);}
}
class PrivateSuperDemo{
public static void main(String arg[]){
// Объект подкласса:
MySubClass obj=new MySubClass();
obj.setall(1,5);}
}
```

В результате выполнения этой программы получаем сообщения:

```
Поле a: 1
Поле b: 5
```

Рассмотрим подробнее программный код и особенности его выполнения. В первую очередь имеет смысл обратить внимание на суперкласс `MySuperClass`, в котором описывается закрытое (с идентификатором доступа `private`) целочисленное поле `a` и два метода. Закрытый метод `showa()` предназначен для отображения значения поля `a`. Открытый метод `seta()` позволяет присвоить значение закрытому полю `a` и вывести значение этого поля на экран — для этого в методе `seta()` вызывается метод `showa()`. Следовательно, при вызове открытого метода `seta()` выполняется обращение к закрытому полю `a`, причем как напрямую, так и через вызов закрытого метода `showa()`.

В подклассе `MySubClass` описывается открытое целочисленное поле `b` и открытый метод `setall()`. Кроме того, классом `MySubClass` из класса `MySuperClass` наследуется открытый метод `seta()`. Закрытое поле `a` и закрытый метод `showa()` классом `MySubClass` не наследуются.

Ситуация складывается интригующая. Объявленный непосредственно в классе `MySubClass` метод `setall()` вызывает, кроме прочего, наследуемый из класса `MySuperClass` метод `seta()`, который, в свою очередь, обращается к ненаследуемому полю `a` и ненаследуемому методу `showa()`. Может сложиться впечатление, что такой код некорректен, поскольку, например, при вызове метода `setall()` из объекта `obj` класса `MySubClass` делается попытка присвоить и считать значение для поля `a`, которого в объекте `obj` в принципе нет. Тем не менее код работает.

Все становится на свои места, если уточнить понятия «наследуется» и «не наследуется». Дело в том, что наследование членов суперкласса подразумевает, что эти поля доступны в подклассе. Другими словами, подкласс «знает» о существовании наследуемых членов, и к этим членам можно обращаться так, как если бы они были описаны в самом классе. Если же член классом не наследуется, то о таком члене класс ничего «не знает», и, соответственно, попытка обратиться к такому «неизвестному» для класса члену напрямую ведет к ошибке. Однако технически ненаследуемые члены в классе существуют, о чем свидетельствует хотя бы приведенный пример. Причина кроется в способе создания объектов подкласса. Дело в том, что при создании объекта подкласса сначала вызывается конструктор суперкласса, а затем непосредственно конструктор подкласса. Конструктором суперкласса выделяется в памяти место для всех членов объекта, в том числе и ненаследуемых. Подробнее об этом — в следующем разделе.

Конструкторы и наследование

В каком порядке и согласье
Идет в пространствах ход работ!

И. Гёте. Фауст

Если суперкласс и подкласс используют конструкторы по умолчанию (то есть ни в суперклассе, ни в подклассе конструкторы не описаны), то процесс создания объекта подкласса для программиста проходит обыденно — так же, как создание объекта обычного класса. Ситуация несколько меняется, если конструктору суперкласса необходимо передавать аргументы. Возникает проблема: поскольку при создании объекта подкласса сначала автоматически вызывается конструктор суперкласса, в этот конструктор как-то нужно передать аргументы, даже если непосредственно конструктор подкласса может без них обойтись. Все это накладывает некоторые ограничения на способ описания конструктора подкласса. Формально эти ограничения сводятся к тому, что в конструкторе подкласса необходимо предусмотреть передачу аргументов конструктору суперкласса (разумеется, если такая передача аргументов вообще требуется).

Технически решение проблемы сводится к тому, что в программный код конструктора подкласса добавляется инструкция вызова конструктора суперкласса с указанием аргументов, которые ему передаются. Для этого используется ключевое слово `super`, после которого в круглых скобках указываются аргументы,

передаваемые конструктору суперкласса. Инструкция вызова конструктора суперкласса указывается первой командой в теле конструктора подкласса. Таким образом, общий синтаксис объявления конструктора подкласса имеет следующий вид:

```
конструктор_подкласса(аргументы1){
super(аргументы2); // аргументы конструктора суперкласса
// тело конструктора подкласса
}
```

Если в теле конструктора подкласса инструкцию `super` не указать вовсе, в качестве конструктора суперкласса вызывается конструктор по умолчанию (конструктор без аргументов). Пример описания конструкторов при наследовании приведен в листинге 6.3.

Листинг 6.3. Конструкторы и наследование

```
// Суперкласс:
class MySuperClass{
int a;
void showa(){
System.out.println("Объект с полем a="+a);}
// Конструкторы суперкласса:
MySuperClass(){
a=0;
showa();}
MySuperClass(int i){
a=i;
showa();}
}
// Подкласс:
class MySubClass extends MySuperClass{
double x;
void showx(){
System.out.println("Объект с полем x="+x);}
// Конструкторы подкласса:
MySubClass(){
super(); // Вызов конструктора суперкласса
x=0;
showx();}
MySubClass(int i,double z){
super(i); // Вызов конструктора суперкласса
x=z;
showx();}
}
class SuperConstrDemo{
```

продолжение

Листинг 6.3 *(продолжение)*

```
public static void main(String[] args){
System.out.println("Первый объект:");
MySubClass obj1=new MySubClass();
System.out.println("Второй объект:");
MySubClass obj2=new MySubClass(5,3.2);}
}
```

В результате выполнения этой программы получаем последовательность сообщений:

```
Первый объект:
Объект с полем a=0
Объект с полем x=0.0
Второй объект:
Объект с полем a=5
Объект с полем x=3.2
```

Программа состоит из трех классов. В первом классе `MySuperClass` описано целочисленное поле `a`, метод `showa()` для отображения значения этого поля, а также два варианта конструкторов: без аргументов и с одним аргументом. В конструкторе без аргументов полю `a` присваивается нулевое значение. В конструкторе с аргументом полю присваивается значение аргумента. В обоих случаях с помощью метода `showa()` значение поля `a` выводится на экран.

На основе класса `MySuperClass` создается подкласс `MySubClass`. Непосредственно в классе описывается поле `x` типа `double` и метод `showx()` для отображения значения этого поля.

В подклассе определяются два конструктора: без аргументов и с двумя аргументами. В каждом из этих конструкторов с помощью инструкции `super` вызывается конструктор суперкласса. В конструкторе подкласса без аргументов командой `super()` вызывается конструктор суперкласса без аргументов. Если при создании объекта подкласса конструктору передаются два аргумента (типа `int` и типа `double`), то аргумент типа `int` передается аргументом конструктору суперкласса (командой `super(i)` в теле конструктора подкласса с двумя аргументами).

В главном методе программы создаются два объекта подкласса `MySubClass`. В первом случае вызывается конструктор без аргументов, во втором — конструктор с двумя аргументами.

Ссылка на элемент суперкласса

Мы вам ничего не позволим показывать.
Мы вам сами все покажем!

Из к/ф «Гараж»

При наследовании могут складываться достаточно неоднозначные ситуации. Один из примеров такой ситуации — совпадение названия наследуемого подклассом

поля с названием поля, описанного непосредственно в подклассе. С формальной точки зрения подобная ситуация выглядит так, как если бы у подкласса было два поля с одним и тем же именем: одно поле собственно подкласса и одно, полученное «по наследству». Технически так оно и есть. В этом случае естественным образом возникает вопрос о способе обращения к таким полям. По умолчанию если обращение выполняется в обычном формате, через указание имени поля, то используется то из двух полей, которое описано непосредственно в подклассе. Рассмотрим пример, представленный в листинге 6.4.

Листинг 6.4. Дублирование полей при наследовании

```
// Суперкласс:
class MyClassA{
// Поле:
int number;
// Конструктор суперкласса:
MyClassA(){
number=0;
System.out.println("Создан объект суперкласса с полем "+number);}
// Отображение значения поля:
void showA(){
System.out.println("Поле number: "+number);}
}
// Подкласс:
class MyClassB extends MyClassA{
// Поле с тем же именем:
int number;
// Конструктор подкласса:
MyClassB(){
super(); // Вызов конструктора суперкласса
number=100;
System.out.println("Создан объект подкласса с полем "+number);}
// Отображение значения поля:
void showB(){
System.out.println("Поле number: "+number);}
}
class TwoFieldsDemo{
public static void main(String[] args){
// Создание объекта подкласса:
MyClassB obj=new MyClassB();
// Изменение значения поля:
obj.number=50;
// Отображение значения поля:
obj.showA();
obj.showB();
}}
```

Результат выполнения программы имеет вид:

```
Создан объект суперкласса с полем 0
Создан объект подкласса с полем 100
Поле number: 0
Поле number: 50
```

В классе MyClassA объявлены числовое поле number, метод showA() для отображения значения этого поля и конструктор без аргументов, которым присваивается нулевое значение полю number и выводится сообщение о создании объекта суперкласса с указанием значения поля.

Подкласс MyClassB, создаваемый на основе суперкласса MyClassA, также содержит описание числового поля number. Описанный в классе метод showB() выводит на экран значение поля number, а конструктор без аргументов позволяет создать объект подкласса с полем number, инициализированным по умолчанию значением 100. Таким образом, в программном коде класса MyClassB складывается довольно интересная ситуация: класс имеет два поля number. Объявленное непосредственно в классе поле «перекрывает» наследуемое поле с таким же именем, поэтому как в методе showB(), так и в конструкторе подкласса инструкция number является обращением именно к полю, описанному в классе.

В главном методе main() в классе TwoFieldsDemo создается объект obj подкласса MyClassB. Результатом выполнения команды new MyClassB() являются сообщения:

```
Создан объект суперкласса с полем 0
Создан объект подкласса с полем 100
```

Первое сообщение появляется в результате вызова конструктора суперкласса в рамках вызова конструктора подкласса. Конструктор суперкласса «своему» полю number присваивает значение 0 и выводит сообщение о создании объекта суперкласса. Затем выполняются команды из тела конструктора подкласса. В результате другому полю number (описанному в подклассе) присваивается значение 100 и выводится сообщение о создании объекта подкласса. Таким образом, при создании поля number объекта obj получают значения 0 и 100.

В главном методе при обращении к полю number командой obj.number=50 изменяется значение того поля, которое описано в подклассе. Другими словами, поле number, имевшее значение 100, получает значение 50.

При выводе значения поля number командой obj.showA() выполняется обращение к полю, описанному в суперклассе: метод showA() обращается в своем программном коде к полю по имени и для него это то поле, которое описано в суперклассе — там же, где описан соответствующий метод. Командой obj.showB() выводится значение поля number, описанного в подклассе.

Чтобы различать одноименные поля, описанные и унаследованные, указывают инструкцию super, то есть ту же самую инструкцию, что и при вызове конструктора суперкласса. Только в этом случае синтаксис ее использования несколько иной.

Обращение к полю, наследованному из суперкласса (описанному в суперклассе), выполняется в формате super.имя_поля. Например, чтобы в методе showB()

из рассмотренного примера обратиться к полю number суперкласса, достаточно воспользоваться инструкцией super.number. В листинге 6.5 приведен измененный код предыдущего примера, в котором в подклассе выполняется обращение как к унаследованному, так и описанному непосредственно в подклассе полю number.

Листинг 6.5. Обращение к дублированным полям

```
// Суперкласс:
class MyClassA{
// Поле:
int number;
// Конструктор суперкласса:
MyClassA(int a){
number=a;
System.out.println("Создан объект суперкласса с полем "+number);}
// Отображение значения поля:
void showA(){
System.out.println("Поле number: "+number);}
}
// Подкласс:
class MyClassB extends MyClassA{
// Поле с тем же именем:
int number;
// Конструктор подкласса:
MyClassB(int a){
super(a-1); // Вызов конструктора суперкласса
number=a;   // Поле из подкласса
// Обращение к полю из суперкласса и подкласса:
System.out.println("Создан объект с полями: "+super.number+" и "+number);}
// Отображение значения поля:
void showB(){
// Обращение к полю из суперкласса и подкласса:
System.out.println("Поля объекта "+super.number+" и "+number);}
}
class TwoFieldsDemo2{
public static void main(String[] args){
// Создание объекта подкласса:
MyClassB obj=new MyClassB(5);
// Изменение значения поля:
obj.number=10;
// Отображение значений полей:
obj.showA();
obj.showB();
}}
```

В отличие от предыдущего случая, конструктору суперкласса передается аргумент, который присваивается в качестве значения полю number. Как и ранее, значение поля отображается с помощью метода суперкласса showA().

Конструктор подкласса также имеет аргумент. Значение аргумента присваивается полю number, определенному непосредственно в классе. Одноименное наследуемое поле получает значение, на единицу меньшее аргумента конструктора. Для этого вызывается конструктор суперкласса с соответствующим аргументом. При выполнении конструктора подкласса также выводится сообщение о значении двух полей, причем обращение к полю, определенному в подклассе, выполняется по имени number, а обращение к полю, определенному в суперклассе, через инструкцию super.number. Значения обоих полей можно вывести на экран с помощью метода showB().

Главный метод программы содержит команду создания объекта подкласса, команду изменения значения поля number, определенного в подклассе (инструкцией obj.number=10), а также команды вывода значений полей с помощью методов showA() и showB(). В результате выполнения этой программы получаем следующее:

```
Создан объект суперкласса с полем 4
Создан объект с полями: 4 и 5
Поле number: 4
Поля объекта 4 и 10
```

По тому же принципу, что и замещение полей с совпадающими именами, замещаются и методы с одинаковыми сигнатурами. Однако с методами ситуация обстоит несколько сложнее, поскольку существует такой механизм, как перегрузка методов. Кроме перегрузки важным понятием является переопределение методов.

Переопределение методов при наследовании

Удивляюсь вашей принципиальности.
То вы за правление, то против!

Из к/ф «Гараж»

Как уже отмечалось, если в подклассе описан метод с сигнатурой, совпадающей с сигнатурой метода, наследуемого из суперкласса, то метод подкласса замещает метод суперкласса. Другими словами, если вызывается соответствующий метод, то используется та его версия, которая описана непосредственно в подклассе. При этом старый метод из суперкласса становится доступным, если к нему обратиться в формате ссылки с использованием ключевого слова super.

Между переопределением и перегрузкой методов существует принципиальное различие. При перегрузке методы имеют одинаковые названия, но разные сигнатуры. При переопределении совпадают не только названия методов, но

и полностью сигнатуры (тип результата, имя и список аргументов). Переопределение реализуется при наследовании. Для перегрузки в наследовании необходимости нет. Если наследуется перегруженный метод, то переопределение выполняется для каждой его версии в отдельности, причем переопределяются только те версии перегруженного метода, которые описаны в подклассе. Если в подклассе какая-то версия перегруженного метода не описана, эта версия наследуется из суперкласса.

Может сложиться и более хитрая ситуация. Допустим, в суперклассе определен некий метод, а в подклассе определяется метод с таким же именем, но другой сигнатурой. В этом случае в подклассе будут доступны обе версии метода: и исходная версия, описанная в суперклассе, и версия метода, описанная в подклассе. То есть имеет место перегрузка метода, причем одна версия метода описана в суперклассе, а вторая — в подклассе.

В листинге 6.6 приведен пример программы с кодом переопределения метода.

Листинг 6.6. Переопределение метода

```
class ClassA{
static int count=0;
private int code;
int number;
ClassA(int n){
set(n);
count++;
code=count;
System.out.println("Объект №"+code+" создан!");}
void set(int n){
number=n;}
void show(){
System.out.println("Для объекта №"+code+":");
System.out.println("Поле number: "+number);}
}
class ClassB extends ClassA{
char symbol;
ClassB(int n,char s){
super(n);
symbol=s;}
void set(int n,char s){
number=n;
symbol=s;}
void show(){
super.show();
System.out.println("Поле symbol: "+symbol);}
}
class MyMethDemo{
```

продолжение

Листинг 6.6 *(продолжение)*

```
public static void main(String[] args){
ClassA objA=new ClassA(10);
ClassB objB=new ClassB(-20,'a');
objA.show();
objB.show();
objB.set(100);
objB.show();
objB.set(0,'z');
objB.show();}
}
```

В результате выполнения программы получаем последовательность сообщений:

```
Объект №1 создан!
Объект №2 создан!
Для объекта №1:
Поле number: 10
Для объекта №2:
Поле number: -20
Поле symbol: a
Для объекта №2:
Поле number: 100
Поле symbol: a
Для объекта №2:
Поле number: 0
Поле symbol: z
```

Разберем программный код и результат его выполнения. В программе описывается класс `ClassA` (суперкласс), на основе которого создается подкласс `ClassB`. Класс `ClassA` имеет целочисленное поле `number`, статическое целочисленное поле `count` (инициализированное нулевым значением) и закрытое целочисленное поле `code`. Кроме этого, в классе описан конструктор с одним аргументом (значением поля `number`), метод `set()` с одним аргументом для присваивания значения полю `number`, а также метод `show()` для отображения значения поля `number`.

Статическое поле `count` предназначено для учета количества созданных объектов. При создании очередного объекта класса значение этого счетчика увеличивается на единицу. Для этого в конструкторе класса `ClassA` размещена команда `count++`. Кроме этого в конструкторе с помощью метода `set()` присваивается значение полю `number` (в качестве аргумента методу передается аргумент конструктора), а командой `code=count` присваивается значение закрытому полю `code`. В поле `code` записывается порядковый номер, под которым создан соответствующий объект. Поле `count` для этой цели не подходит, поскольку оно статическое и изменяется каждый раз при создании очередного объекта. В поле `code` записывается значение поля `count` после создания объекта и впоследствии поле `code` этого объекта не меняется.

Поле code (после присваивания значения полю) служит в конструкторе для вывода сообщения о создании объекта с соответствующим номером. Номер объекта (поле code) используется также в методе show(), чтобы легче было проследить, для какого именно объекта выводится информация о значении поля number.

Подкласс ClassB создается на основе суперкласса ClassA. В подклассе ClassB наследуется статическое поле count и поле number. Закрытое поле code не наследуется. Кроме этих наследуемых полей, непосредственно в классе ClassB описано символьное поле symbol. Конструктор класса принимает два аргумента: первый типа int для поля number и второй типа char для поля symbol.

Код конструктора класса ClassB состоит всего из двух команд: команды вызова конструктора суперкласса super(n) и команды присваивания значения символьному полю symbol=s (n и s — аргументы конструктора). Со второй командой все просто и понятно. Интерес представляет команда вызова конструктора суперкласса. Во-первых, этим конструктором наследуемому полю number присваивается значение. Во-вторых, значение наследуемого статического поля count увеличивается на единицу. Это означает, что ведется общий учет всех объектов, как суперкласса, так и подкласса. В-третьих, хотя поле code не наследуется, под него выделяется место в памяти и туда заносится порядковый номер созданного объекта. На экран выводится сообщение о создании нового объекта, а номер объекта считывается из «несуществующего» поля code.

Метод show() в классе ClassB переопределяется. Сигнатура описанного в классе ClassB метода show() совпадает с сигнатурой метода show(), описанного в классе ClassA. Если в классе ClassA методом show() отображается информация о номере объекта и значении его поля number, то в классе ClassB метод show() выводит еще и значение поля symbol. При этом в переопределенном методе show() вызывается также прежняя (исходная) версия метода из класса ClassA. Для этого используется инструкция вида super.show(). Этот исходный вариант метода, кроме прочего, считывает из ненаследуемого (но реально существующего) поля code порядковый номер объекта и отображает его в выводимом на экран сообщении.

Метод set() в классе ClassB перегружается. Хотя в классе ClassA есть метод с таким же названием, сигнатуры методов в суперклассе и подклассе разные. В суперклассе у метода set() один числовой аргумент, а в подклассе у этого метода два аргумента: числовой и символьный. Поэтому в классе ClassB имеется два варианта метода set() — с одним и двумя аргументами. Первый наследуется из суперкласса ClassA, а второй определен непосредственно в подклассе ClassB.

В главном методе программы командами ClassA objA=new ClassA(10) и ClassB objB=new ClassB(-20,'a') создаются два объекта: объект objA суперкласса и объект objB подкласса. В результате выполнения этих команд на экране появляются сообщения Объект №1 создан! и Объект №2 создан! — сообщения выводятся конструкторами. Проверяются значения полей созданных объектов командами objA.show() и objB.show(). Поскольку метод show() перегружен, то в первом случае

вызывается метод show(), описанный в суперклассе ClassA, а во втором — метод show(), описанный в подклассе ClassB. Поэтому для объекта objA выводится значение одного (и единственного) поля, а для объекта objB — значения двух полей.

Командой objB.set(100) метод set() вызывается из объекта objB. Поскольку в данном случае методу передан всего один аргумент, вызывается версия метода, описанная в классе ClassA. В результате меняется значение поля number объекта objB, а поле symbol остается неизменным. Подтверждается данное утверждение после вызова метода objB.show() (см. приведенный ранее результат выполнения программы). Если же воспользоваться командой objB.set(0,'z'), будет вызван тот вариант метода set(), который описан в классе ClassB. Выполнение команды objB.show() показывает, что в результате изменились оба поля объекта objB.

Многоуровневое наследование

Бюрократия разрастается, чтобы поспеть за потребностями
разрастающейся бюрократии.

А. Азимов

Хотя множественное наследование (наследование сразу нескольких классов) в Java не допускается, с успехом может использоваться многоуровневое наследование. В этом случае подкласс становится суперклассом для другого подкласса. Пример такой ситуации приведен в листинге 6.7.

Листинг 6.7. Многоуровневое наследование

```
class A{
int a;
A(int i){
a=i;
System.out.println("Поле a: "+a);}
}
class B extends A{
int b;
B(int i,int j){
super(i);
b=j;
System.out.println("Поле b: "+b);}
}
class C extends B{
int c;
C(int i,int j,int k){
super(i,j);
c=k;
System.out.println("Поле c: "+c);}
}
```

```
class MultiCall{
public static void main(String args[]){
C obj=new C(1,2,3);}
}
```

Ситуация достаточно простая: класс A является суперклассом для подкласса B. Класс B, в свою очередь, является суперклассом для подкласса C. Таким образом, получается своеобразная иерархия классов. В классе A всего одно числовое поле a и конструктор с одним аргументом. Аргумент определяет значение поля создаваемого объекта. Кроме того, при этом выводится сообщение о значении поля объекта.

В классе B наследуется поле a из класса A и появляется еще одно поле b. Соответственно, конструктор имеет два аргумента. Первый передается конструктору суперкласса (класс A), а второй определяет значение нового поля b. Также выводится сообщение о значении этого поля, однако прежде сообщение о значении поля a выводится конструктором суперкласса.

Два поля a и b наследуются в классе C. Там же описано числовое поле c. Первые два аргумента конструктора передаются конструктору суперкласса (класса B), а третий присваивается в качестве значения полю c. В конструкторе класса C имеется также команда вывода на экран значения этого поля. Значения полей a и b выводятся при выполнении конструктора суперкласса.

В главном методе программы командой `C obj=new C(1,2,3)` создается объект класса C. В результате на экране появляются сообщения:

```
Поле a: 1
Поле b: 2
Поле c: 3
```

Путем многоуровневого наследования можно создавать достаточно сложные иерархические структуры классов. Особенно механизм многоуровневого наследования становится эффективным при одновременном использовании механизмов перегрузки и переопределения методов. Пример простой, но показательной программы приведен в листинге 6.8.

Листинг 6.8. Многоуровневое наследование, перегрузка и переопределение методов

```
class A{
void show(){
System.out.println("Метод класса A");}
}
class B extends A{
void show(String msg){
System.out.println(msg);}
}
class C extends B{
void show(){
```

продолжение

Листинг 6.8 *(продолжение)*

```
System.out.println("Метод класса C");}
}
class MultiOverride{
public static void main(String args[]){
A obj1=new A();
B obj2=new B();
C obj3=new C();
obj1.show();
obj2.show();
obj2.show("Класс B");
obj3.show();
obj3.show("Класс C");}
}
```

Как и в предыдущем примере, создается иерархическая цепочка из трех классов: в вершине находится суперкласс `A`, на основе которого создается подкласс `B`, в свою очередь являющийся суперклассом для подкласса `C`. При этом классами наследуется, перегружается или переопределяется описанный в классе `A` метод `show()`. Схему перегрузки и переопределения этого метода иллюстрирует рис. 6.1.

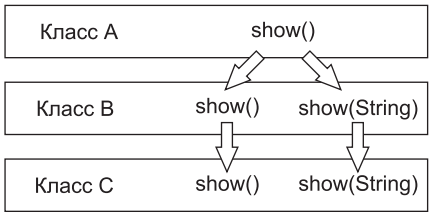

Рис. 6.1. Схема перегрузки и переопределения метода show() при многоуровневом наследовании

В частности, метод `show()` класса `A` не имеет аргументов и выводит сообщение `Метод класса A`. В классе `B` этот метод наследуется. Кроме того, в классе `B` метод `show()` перегружен с текстовым аргументом так, что он выводит сообщение, переданное в качестве его аргумента. Забегая наперед, отметим, что текстовый аргумент — это объект класса `String`. Текстовая строка, при передаче аргументом, заключается в двойные кавычки.

В классе `C` версия метода `show()` без аргумента переопределяется, а версия этого метода с текстовым аргументом наследуется из класса `B`.

В главном методе программы создаются три объекта — по объекту для каждого из классов. Затем из каждого объекта вызывается метод `show()` (с аргументами или без в зависимости от того, из какого объекта вызывается метод). В результате мы получаем следующее:

```
Метод класса A
Метод класса A
Класс B
```

```
Метод класса C
Класс C
```

Из объекта класса A вызывается версия метода без аргументов. Из объекта класса B метод вызывается без аргументов (версия метода из класса A) и с текстовым аргументом (версия метода, описанная в классе B). Вызываемая из объекта класса C версия метода без аргумента описана в классе C, а версия метода с текстовым аргументом наследуется из класса B.

Объектные переменные суперкласса и динамическое управление методами

Я бы не стал увязывать эти вопросы так перпендикулярно.

В. Черномырдин

В наследовании было бы мало пользы, если бы не одно важное и интересное свойство объектных переменных суперкласса. Они могут ссылаться на объекты подкласса!

Напомним, что объектная переменная — это переменная, значением которой является ссылка на объект соответствующего класса, то есть фактически та переменная, которую мы отождествляем с объектом. Объектная переменная объявляется так же, как обычная переменная базового типа, с той лишь разницей, что в качестве типа переменной указывается имя класса. Создается же объект с помощью оператора new и конструктора класса. Сказанное означает, что ссылку на объект подкласса (объект, созданный конструктором подкласса) можно присвоить в качестве значения объектной переменной суперкласса (в качестве типа переменной указав имя суперкласса).

Важное ограничение состоит в том, что через объектную переменную суперкласса можно ссылаться только на те члены подкласса, которые наследуются из суперкласса или переопределяются в подклассе. Пример приведен в листинге 6.9.

Листинг 6.9. Объектная переменная суперкласса ссылается на объект подкласса

```
class ClassA{
double Re;
void set(double x){
Re=x;}
void show(){
System.out.println("Класс A:");
System.out.println("Поле Re: "+Re);}
}
class ClassB extends ClassA{
double Im;
```

продолжение

Листинг 6.9 *(продолжение)*

```
void set(double x,double y){
Re=x;
Im=y;}
void show(){
System.out.println("Класс B:");
System.out.println("Поле Re: "+Re);
System.out.println("Поле Im: "+Im);}
}
class SuperRefs{
public static void main(String[] args){
ClassA objA;
ClassB objB=new ClassB();
objA=objB;
objB.set(1,5);
objB.show();
objA.set(-10);
objA.show();}
}
```

В данном случае описывается суперкласс `ClassA`, на основе которого создается подкласс `ClassB`. В суперклассе `ClassA` объявлено поле `double Re` и методы `set()` и `show()`. Метод `show()` не имеет аргументов и выводит сообщение с названием класса (буквы-идентификатора класса) и значением поля `Re`. Метод `set()` имеет один аргумент, который присваивается в качестве значения полю `Re`.

Поле `Re` наследуется в классе `ClassB`. В этом классе также описывается поле `double Im`. Метод `set()` перегружается так, чтобы иметь два аргумента — значения полей `Re` и `Im`. Перегружается и метод `show()`, чтобы выводить на экран значения двух полей.

В главном методе программы командой `ClassA objA` объявляется объектная переменная `objA` класса `ClassA`. Командой `ClassB objB=new ClassB()` создается объект класса `ClassB`, и ссылка на этот объект присваивается в качестве значения объектной переменной `objB` класса `ClassB`. Затем командой `objA=objB` ссылка на тот же объект присваивается в качестве значения объектной переменной `objA`. Таким образом, в результате и объектная переменная `objA`, и объектная переменная `objB` ссылаются на один и тот же объект. То есть переменных две, а объект один. Тем не менее ссылка на объект через переменную `objA` является «ограниченной» — через нее можно обращаться не ко всем членам объекта класса `ClassB`.

Командой `objB.set(1,5)` полям `Re` и `Im` объекта присваиваются значения 1 и 5 соответственно. Командой `objB.show()` значения полей объекта выводятся на экран. Для этого вызывается версия метода `show()`, описанная в классе `ClassB`. Командой `objA.set(-10)` меняется значение поля `Re`. Для этого вызывается версия метода `set()`, описанная в классе `ClassA` и наследуемая в классе `ClassB`. Вызвать через объектную переменную `objA` версию метода `set()` с двумя аргументами не получится — эта версия не описана в классе `ClassB`, поэтому через объектную переменную суперкласса версия метода недоступна. Однако командой `objA.`

show() можно вызвать переопределенный в классе ClassB метод show(). Результат выполнения программы следующий:

```
Класс B:
Поле Re: 1.0
Поле Im: 5.0
Класс B:
Поле Re: -10.0
Поле Im: 5.0
```

Отметим также, что в силу отмеченных особенностей ссылки на объект подкласса через объектную переменную суперкласса через переменную objA можно обратиться к полю Re объекта подкласса, но нельзя обратиться к полю Im.

Хотя описанная возможность ссылаться на объекты подклассов через объектные переменные суперклассов может показаться не очень полезной, она открывает ряд перспективных технологий, в том числе и динамическое управление методами.

Динамическое управление методами базируется на том, что выбор варианта перегруженного метода определяется не типом объектной ссылки, а типом объекта, причем на этапе не компиляции, а выполнения программы. С подобной ситуацией мы встречались в предыдущем примере, когда при ссылке на метод show() через объектную переменную objA суперкласса ClassA вызывалась переопределенная версия метода из подкласса ClassB, то есть версия, описанная в классе объекта, а не в классе объектной переменной. Рассмотрим еще один пример, представленный в листинге 6.10.

Листинг 6.10. Динамическое управление методами

```
class A{
void show(){
System.out.println("Класс A");}
}
class B extends A{
void show(){
System.out.println("Класс B");}
}
class C extends A{
void show(){
System.out.println("Класс C");}
}
class Dispatch{
public static void main(String args[]){
A a=new A();
B b=new B();
C c=new C();
A ref;
ref=a;
```

продолжение

Листинг 6.10 *(продолжение)*

```
ref.show();
ref=b;
ref.show();
ref=c;
ref.show();}
}
```

В программе описывается суперкласс A, на основе которого создаются два класса: B и C. На рис. 6.2 приведена общая иерархическая схема классов программы.

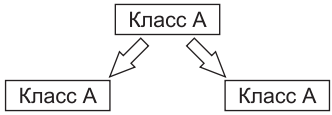

Рис. 6.2. Структура классов при наследовании

В классе A описан метод show(), действие которого сводится к выводу на экран сообщения Класс A. В каждом из классов B и C этот метод переопределяется. Версия метода show() из класса B выводит сообщение Класс B, а версия этого же метода из класса C — сообщение Класс C.

В главном методе программы создаются объекты a, b и c соответственно классов A, B и C, а также объявляется объектная переменная ref класса A. Далее этой объектной переменной последовательно в качестве значений присваиваются ссылки на объекты a, b и c (командами ref=a, ref=b и ref=c). Поскольку класс A является суперклассом и для класса B, и для класса C, данные операции возможны. Причем после каждого такого присваивания через объектную переменную ref командой ref.show() вызывается метод show(). Результат выполнения программы имеет вид:

```
Класс A
Класс B
Класс C
```

Мы видим, что хотя формально во всех трех случаях команда вызова метода show() одна и та же (команда ref.show()), результат разный в зависимости от того, на какой объект в данный момент ссылается объектная переменная ref.

Абстрактные классы

Курс у нас один — правильный.

В. Черномырдин

В Java существуют такие понятия, как абстрактный метод и абстрактный класс. Под абстрактным методом подразумевают метод, тело которого в классе не объявлено, а есть только сигнатура (тип результата, имя и список аргументов). Перед таким абстрактным методом указывается идентификатор abstract,

а заканчивается описание сигнатуры метода в классе традиционно — точкой с запятой.

Класс, который содержит хотя бы один абстрактный метод, называется абстрактным. Описание абстрактного класса начинается с ключевого слова `abstract`.

Абстрактный класс в силу очевидных причин не может использоваться для создания объектов. Поэтому абстрактные классы являются суперклассами для подклассов. При этом в подклассе абстрактные методы абстрактного суперкласса должны быть определены в явном виде (иначе подкласс тоже будет абстрактным). Пример использования абстрактного класса приведен в листинге 6.11.

Листинг 6.11. Абстрактный класс

```
// Абстрактный суперкласс:
abstract class A{
// Абстрактный метод:
abstract void callme();
// Неабстрактный метод:
void callmetoo(){
System.out.println("Второй метод");}
}
// Подкласс:
class B extends A{
// Определение наследуемого абстрактного метода:
void callme(){
System.out.println("Первый метод");}
}
class AbstDemo{
public static void main(String args[]){
// Объект подкласса:
B obj=new B();
obj.callme();
obj.callmetoo();}
}
```

Пример достаточно простой: описывается абстрактный суперкласс `A`, на основе которого затем создается подкласс `B`. Суперкласс `A` содержит абстрактный метод `call()` и обычный (неабстрактный) метод `callmetoo()`. Оба метод наследуются в классе `B`. Но поскольку метод `call()` абстрактный, то он описан в классе `B`.

Методом `call()` выводится сообщение `Первый метод`, а методом `callmetoo()` — сообщение `Второй метод`. В главном методе программы создается объект подкласса и последовательно вызываются оба метода. В результате получаем сообщения:

```
Первый метод
Второй метод
```

Что касается практического использования абстрактных классов, то обычно они бывают полезны при создании сложных иерархий классов. В этом случае

абстрактный класс, находящийся в вершине иерархии, служит своеобразным шаблоном, определяющим, что должно быть в подклассах. Конкретная же реализация методов выносится в подклассы. Такой подход, кроме прочего, нередко позволяет избежать ошибок, поскольку будь у суперкласса только неабстрактные наследуемые методы, было бы сложнее отслеживать процесс их переопределения в суперклассах. В то же время, если не определить в подклассе абстрактный метод, при компиляции появится ошибка.

Отметим еще одно немаловажное обстоятельство, которое касается наследования вообще. В некоторых случаях необходимо защитить метод от возможного переопределения в подклассе. Для этого при описании метода в его сигнатуре указывается ключевое слово `final`. Если это ключевое слово включить в сигнатуру класса, этот класс будет защищен от наследования — на его основе нельзя будет создать подкласс. Третий способ использования ключевого слова `final` касается описания полей (переменных). В этом случае оно означает запрет на изменение значения поля, то есть фактически означает определение константы.

Примеры программ

Рассмотрим некоторые примеры, в которых имеет место наследование классов и переопределение методов.

Комплексная экспонента

Далее в листинге 6.12 приведен код программы, в которой создается суперкласс для реализации комплексных чисел и выполнения базовых операций с ними: сложения комплексных чисел, умножения комплексных чисел и произведения комплексного и действительного чисел. На основе суперкласса создается подкласс, в котором описан метод для вычисления экспоненты от комплексного аргумента.

Листинг 6.12. Вычисление комплексной экспоненты

```
// Суперкласс:
class Compl{
// Действительная и мнимая части числа:
double Re,Im;
// Метод для вычисления суммы комплексных чисел:
Compl sum(Compl obj){
Compl tmp=new Compl();
tmp.Re=Re+obj.Re;
tmp.Im=Im+obj.Im;
return tmp;}
// Метод для вычисления произведения комплексных чисел:
Compl prod(Compl obj){
Compl tmp=new Compl();
```

```
tmp.Re=Re*obj.Re-Im*obj.Im;
tmp.Im=Im*obj.Re+Re*obj.Im;
return tmp;}
// Метод перегружен для вычисления произведения
// комплексного и действительного чисел:
Compl prod(double x){
Compl tmp=new Compl();
tmp.Re=Re*x;
tmp.Im=Im*x;
return tmp;}
// Метод для отображения полей объекта:
void show(){
System.out.println("Действительная часть Re="+Re);
System.out.println("Мнимая часть Im="+Im);}
// Конструктор без аргумента:
Compl(){
Re=0;
Im=0;}
// Конструктор с одним аргументом:
Compl(double x){
Re=x;
Im=0;}
// Конструктор с двумя аргументами:
Compl(double x,double y){
Re=x;
Im=y;}
// Конструктор создания копии:
Compl(Compl obj){
Re=obj.Re;
Im=obj.Im;}
}
// Подкласс:
class ComplNums extends Compl{
// Количество слагаемых ряда:
private int n;
// Метод для вычисления комплексной экспоненты:
ComplNums CExp(){
// Начальное значение - объект суперкласса:
Compl tmp=new Compl(1);
// Начальная добавка - объект суперкласса:
Compl q=new Compl(this);
// Индексная переменная:
int i;
// Вычисление ряда:
for(i=1;i<=n;i++){
tmp=tmp.sum(q);
```

продолжение

Листинг 6.12 *(продолжение)*

```
q=q.prod(this).prod(1.0/(i+1));}
// Результат - объект подкласса:
return new ComplNums(tmp);}
//Конструктор суперкласса без аргументов:
ComplNums(){
super();
n=100;}
// Конструктор суперкласса с одним аргументом:
ComplNums(double x){
super(x);
n=100;}
// Конструктор суперкласса с двумя аргументами:
ComplNums(double x,double y){
super(x,y);
n=100;}
// Конструктор суперкласса с тремя аргументами:
ComplNums(double x,double y,int m){
super(x,y);
n=m;}
// Конструктор создания объекта подкласса
// на основе объекта суперкласса:
ComplNums(Compl obj){
super(obj);
n=100;}
// Конструктор создания копии для суперкласса:
ComplNums(ComplNums obj){
super(obj);
n=obj.n;}
}
class ComplExtendsDemo{
public static void main(String[] args){
ComplNums z=new ComplNums(2,3);
// Вычисление комплексной экспоненты:
z.CExp().show();}
}
```

В суперклассе `Compl` описано два поля `Re` и `Im` — оба типа `double`. Кроме этого, класс имеет метод `sum()` для вычисления суммы двух комплексных чисел, реализованных в виде объектов класса `Compl`. В классе также есть перегруженный метод `prod()` для вычисления произведения двух комплексных чисел, а также комплексного числа на действительное число. Конструкторы класса `Compl` позволяют создавать объекты без передачи аргументов, а также с передачей одного и двух аргументов, кроме того, у класса имеется конструктор копирования. В последнем случае конструктору в качестве аргумента передается объект того же класса — на основе этого объекта создается копия.

На основе суперкласса `Compl` создается подкласс `ComplNums`. Кроме наследуемых из суперкласса полей и методов, в подклассе описывается закрытое целочисленное поле `n`, которое определяет количество слагаемых при вычислении ряда для экспоненты. Если через z обозначить комплексное число, которое передается аргументом экспоненте, то результат вычисляется в виде:

$$\exp(z) \approx \sum_{k=0}^{n} \frac{z^k}{k!} = 1 + z + \frac{z^2}{2!} + ... + \frac{z^n}{n!}.$$

В подклассе предусмотрены конструкторы создания объектов с передачей конструктору до трех аргументов. Также описан конструктор копирования — в этом случае объект подкласса создается на основе другого объекта подкласса. Кроме того, имеется конструктор создания объекта подкласса на основе объекта суперкласса.

Комплексная экспонента вычисляется методом `CExp()`. Аргументом экспоненты является комплексное число, реализованное через объект вызова. Результатом является объект подкласса. В самом методе командой `Compl tmp=new Compl(1)` создается локальный объект `tmp` суперкласса с начальным единичным значением. В этот объект будет записываться сумма комплексного ряда. Начальное значение для добавки при вычислении суммы определяется локальным объектом суперкласса `q`. Этот объект создается командой `Compl q=new Compl(this)`. Начальное значение добавки — это комплексный аргумент экспоненты. При создании объекта вызывается конструктор копирования, относящийся к суперклассу. При этом аргументом указана ссылка на объект вызова, то есть на объект подкласса. Однако благодаря тому, что объектная переменная суперкласса может ссылаться на объект подкласса, такая ситуация корректна.

Вычисление результата осуществляется в цикле. В теле цикла две команды. Первой командой `tmp=tmp.sum(q)` выполняется прибавление к текущему значению суммы очередной добавки. Второй командой `q=q.prod(this).prod(1.0/(i+1))` изменяется сама добавка (добавка умножается на аргумент экспоненты и затем делится на значение индексной переменной `i`, увеличенное на единицу). Обращаем внимание читателя на использование в данном случае ссылки `this`.

После завершения цикла командой `new ComplNums(tmp)` на основе локального объекта суперкласса создается анонимный объект подкласса, который и возвращается в качестве результата методом.

После выполнения программы получаем следующий результат:

```
Действительная часть Re=-7.315110094901102
Мнимая часть Im=1.042743656235904
```

Отметим, что как в суперклассе, так и в подклассе описана лишь незначительная часть методов, требующихся при работе с комплексными числами. На практике таких методов приходится описывать намного больше. Кроме того, не все конструкторы использованы при вычислении результата — комплексной экспоненты. Код для этих конструкторов приведен в качестве иллюстрации.

Произведение полиномов и ряд Тейлора

В следующей программе реализована процедура вычисления произведения полиномов и вычисления ряда Тейлора для произведения двух функций (ряды Тейлора для каждой из которых известны). При этом применяется механизм наследования. Программный код приведен в листинге 6.13. Сразу отметим, что структура программы и, в частности, организация классов далеко не оптимальны — пример иллюстративный и позволяет лучше понять некоторые особенности механизма наследования.

Листинг 6.13. Произведение полиномов и ряд Тейлора

```
// Суперкласс:
class PolyBase{
// Коэффициенты полинома:
double[] a;
// Метод для вычисления значения полинома в точке:
double value(double x){
double s=0,q=1;
for(int i=0;i<a.length;i++){
s+=a[i]*q;
q*=x;}
return s;}
// Степень полинома:
int power(){
for(int i=a.length-1;i>0;i--){
if(a[i]!=0) return i;}
return 0;}
// Отображение коэффициентов и степени полинома:
void show(){
System.out.println("Коэффициенты полинома:");
for(int i=0;i<a.length;i++)
System.out.print(a[i]+"  ");
System.out.print("\nСтепень полинома: ");
System.out.println(power()+".\n");}
}
// Подкласс:
class PolyDerive extends PolyBase{
// Метод для вычисления произведения полиномов:
PolyBase prod(PolyBase Q){
int i,j,n;
n=power()+Q.power()+1;
PolyBase tmp=new PolyBase();
tmp.a=new double[n];
for(i=0;i<=power();i++){
for(j=0;j<=Q.power();j++){
tmp.a[i+j]+=a[i]*Q.a[j];}
```

```
}
return tmp;}
// Отображение параметров полинома и значения в точке:
void show(double x){
System.out.println("Аргумент полинома: "+x);
System.out.println("Значение полинома: "+value(x));
show();}
PolyDerive(PolyBase obj){
a=new double[obj.a.length];
for(int i=0;i<a.length;i++)
a[i]=obj.a[i];}
}
// Подкласс для разложения в ряд Тейлора произведения:
class Taylor extends PolyDerive{
void show(){
System.out.println("Ряд Тейлора!");
super.show();
}
Taylor(PolyBase P,PolyBase Q){
super(P);
PolyBase tmp=prod(Q);
for(int i=0;i<a.length;i++)
a[i]=tmp.a[i];}
}
class PolyExtendsDemo{
public static void main(String[] args){
// Исходные полиномы:
PolyBase P=new PolyBase();
PolyBase Q=new PolyBase();
PolyBase R;
P.a=new double[]{1,-2,4,1,-3};
Q.a=new double[]{2,-1,3,0,4};
// Произведение полиномов:
R=new PolyDerive(P).prod(Q);
R.show();
new PolyDerive(P).show(-1);
// Ряд Тейлора:
new Taylor(P,Q).show();
}}
```

В суперклассе `PolyBase`, предназначенном для реализации полиномов, имеется поле `a` — переменная массива типа `double`. В этот массив будут заноситься коэффициенты полинома (напомним, что полиномом называется сумма степенных слагаемых вида $a_0 + a_1x + a_2x^2 + ... + a_nx^n$). Кроме того, в классе описан метод `value()` для вычисления значения полинома в точке (аргумент метода). Метод `power()` предназначен для вычисления степени полинома. Вообще степень

полинома определяется как наибольшая степень аргумента с ненулевым коэффициентом. Можно было бы считать, что степень полинома на единицу меньше размера массива коэффициентов, но в принципе старшие коэффициенты могут равняться нулю, поэтому формально степень полинома размером соответствующего массива не определяется. В методе `power()` коэффициенты полинома, начиная со старшего, проверяются на предмет отличия от нуля. В соответствии с этим определяется и степень полинома.

Метод `show()` предназначен для отображения коэффициентов полинома, записанных в поле-массив `a`. Этим же методом выводится степень полинома.

Подкласс `PolyDerive` создается на основе суперкласса `PolyBase`. В этом классе описан метод `prod()`, предназначенный для вычисления произведения полиномов. Обращаем внимание читателя, что и аргументом, и результатом этого метода является объект суперкласса `PolyBase`. При вычислении произведения принимается во внимание то обстоятельство, что степени исходных полиномов могут иметь нулевые старшие коэффициенты. Также в подклассе перегружен унаследованный из суперкласса метод `show()` таким образом, что методу передается аргумент, для которого вычисляется значение полинома в точке, и это значение, равно как и коэффициенты полинома и его степень, выводятся на консоль.

В подклассе также описан конструктор создания объекта на основе объекта суперкласса. Конструктор суперкласса в этом случае явно не вызывается — в суперклассе конструктор не описан. По умолчанию если вызова конструктора суперкласса в конструкторе подкласса не происходит, автоматически вызывается конструктор суперкласса по умолчанию. Что касается создаваемого объекта, то его поле `a` является копией соответствующего поля объекта, указанного аргументом конструктора.

На основе класса `PolyDerive` создается класс `Taylor`, предназначенный для разложения в ряд Тейлора произведения двух известных разложений. Формально эта процедура состоит в произведении двух полиномов. Особенность же связана с тем, что если нужно найти ряд Тейлора до степени n, то умножаются два полинома степени n, а в полиноме-рузультате (в общем случае это полином степени $2n$) необходимо оставить только слагаемые степени не выше n.

В классе `Taylor` переопределяется метод `show()` так, что при выводе информации о полиноме появляется дополнительное сообщение о том, что вычисляется ряд Тейлора. При переопределении метода `show()` вызывается версия этого метода из суперкласса, для чего используется инструкция `super.show()`. Это тот вариант метода `show()`, который унаследован классом `PolyDerive` из класса `PolyBase`.

Кроме этого метода в классе `Taylor` описан конструктор создания объекта класса на основе двух объектов класса `PolyBase`. Фактически речь идет о вычислении ряда Тейлора на основе двух полиномов. Другими словами, чтобы вычислить ряд Тейлора, достаточно создать объект класса `Taylor`, указав аргументами конструктора два исходных полинома.

В конструкторе вызывается конструктор суперкласса (для класса `Taylor` суперклассом является класс `PolyDerive`) с передачей в качестве аргумента первого

полинома (объекта класса PolyBase). Затем создается временный объект tmp — произведение двух полиномов. На основе полученного локального объекта tmp заполняются элементы массива a создаваемого объекта класса Taylor.

Отметим, что в данном случае неявно предполагается, что переданные конструктору класса Taylor полиномы имеют поля-массивы одинакового размера, а размер массива для объекта-результата определяется размером массива первого из двух передаваемых конструктору объектов.

В главном методе программы в классе PolyExtendsDemo проиллюстрирована функциональность рассмотренного кода. Результат выполнения программы имеет следующий вид:

```
Коэффициенты полинома:
2.0  -5.0  13.0  -8.0  9.0  -2.0  7.0  4.0  -12.0
Степень полинома: 8.

Аргумент полинома: -1.0
Значение полинома: 3.0
Коэффициенты полинома:
1.0  -2.0  4.0  1.0  -3.0
Степень полинома: 4.

Ряд Тейлора!
Коэффициенты полинома:
2.0  -5.0  13.0  -8.0  9.0
Степень полинома: 4.
```

В частности, объявляются три объектные переменные (P, Q и R) класса PolyBase. Для двух (P и Q) полям a присваиваются в явном виде значения, а в третью (переменную R) записывается ссылка на результат произведения двух полиномов. При этом инструкцией new PolyDerive(P) на основе первого объекта P класса PolyBase создается анонимный объект класса PolyDerive, из которого вызывается метод prod() для вычисления произведения полиномов. Аналогично проверяется функциональность перегруженного в подклассе PolyDerive метода show(). Для вычисления ряда Тейлора с одновременным выводом результата на консоль использована команда new Taylor(P,Q).show(). В данном случае также создается анонимный объект.

Резюме

1. В Java на основе одних классов можно создавать другие. Такой механизм называется наследованием. При наследовании поля и методы исходного класса, который называется суперклассом, наследуются классом, создаваемым на его основе. Этот второй класс называется подклассом.
2. При создании подкласса после его имени через ключевое слово extends указывается имя суперкласса, на основе которого создается подкласс. Наследование

или ненаследование членов суперкласса в подклассе, а также уровень доступа наследованных из суперкласса членов определяются уровнем доступа членов в суперклассе и взаимным размещением суперкласса и подкласса (с учетом наличия пакетов — см. следующую главу).

3. При создании объекта подкласса сначала вызывается конструктор суперкласса, причем вызов конструктора суперкласса необходимо в явном виде прописать в конструкторе подкласса. Для этого используется ключевое слово `super`. В круглых скобках после этого ключевого слова указываются аргументы, передаваемые конструктору суперкласса. Команда вызова конструктора суперкласса должна быть первой в коде конструктора подкласса.
4. Ключевое слово `super` используется также для ссылки на элементы суперкласса (обычно при дублировании наследуемых членов).
5. Наследуемые методы можно переопределять. В этом случае в подклассе описывается метод с соответствующей сигнатурой. Старая (исходная) версия метода также доступна с помощью ключевого слова `super`. Переопределение и перегрузка методов могут использоваться одновременно.
6. В Java множественное наследование (когда подкласс создается на основе нескольких суперклассов) не поддерживается, но есть многоуровневое наследование. В этом случае подкласс служит суперклассом для другого подкласса.
7. Объектные переменные суперкласса могут ссылаться на объекты подкласса. В этом случае через такие переменные доступны только члены, описанные в суперклассе.
8. В Java существуют абстрактные методы и абстрактные классы. Абстрактным является класс, который содержит хотя бы один абстрактный метод. Абстрактный метод в классе не описывается. Класс содержит только сигнатуру абстрактного метода. Описание абстрактных классов и методов выполняется с помощью ключевого слова `abstract`. Абстрактный класс в силу очевидных причин не может иметь объектов.
9. Чтобы защитить класс от наследования, метод от переопределения и поле от изменения используют ключевое слово `final`.

Часть II. Нетривиальные возможности Java

Глава 7. Пакеты и интерфейсы

Излишек — вещь крайне необходимая.

Вольтер

В данной главе речь идет о пакетах и интерфейсах. Рассказано, как создаются пакеты, описаны принципы их применения и предназначение. Здесь же можно найти полезную информацию об интерфейсах и тех широких возможностях, которые открываются перед программистом при условии их разумного применения.

Пакеты в Java

Господин полковник, вам пакет от графа Мерзляева!

Из к/ф «О бедном гусаре замолвите слово»

В известном смысле пакет — это контейнер для классов, их пространство имен. Современные тенденции программирования таковы, что обычно приходится создавать достаточно объемные проекты, имея дело с большим количеством классов. В принципе, каждый из этих классов должен иметь уникальное имя. Хотя гипотетически придумать названий для классов можно бесконечно много, наступает момент, когда подобный подход означал бы выход за рамки разумного.

В соответствие с концепцией пакетов, все классы проекта разбиваются по группам, которые и называются пакетами. Имя класса должно быть уникальным в пределах его пакета. При этом не важно, есть ли в другом пакете класс с таким же именем. Такой подход прост, понятен и удобен. Познакомимся с ним поближе.

Для определения пакета необходимо в файле с описанием класса, включаемого в пакет, первой командой указать инструкцию `package` и имя пакета, например:

```
package mypack;
```

В данном случае `mypack` — это имя пакета. Если пакет с таким именем уже существует, соответствующий класс (или классы) из файла добавляется в этот пакет. Если такого пакета нет, он создается. Таким образом, одна и та же инструкция

`package` может использоваться в нескольких файлах. Однако в файле может быть только одна инструкция `package` или не быть вовсе. В последнем случае классы попадают в так называемый *пакет по умолчанию*.

Пакет, кроме классов, может содержать интерфейсы (описываются далее в этой главе), а также подпакеты (то есть другие пакеты). При указании имени подпакета (пакета, находящегося в другом пакете) используется точечный синтаксис — имени подпакета предшествует имя пакета, а в качестве разделителя указывается точка. При этом в Java действует жесткое правило: иерархия пакетов должна строго соответствовать структуре файловой системы. Если, например, файл содержит начальную инструкцию `package java.awt.image`, это означает, что файлы подпакета `image` размещены в каталоге `java/awt/image`.

С практической точки зрения наиболее важный аспект в работе с пакетами — это, пожалуй, схема доступа к членам классов с учетом их размещения по пакетам. В табл. 7.1 перечислены уровни доступа к членам класса с учетом использованного при их объявлении спецификатора уровня доступа.

Таблица 7.1. Уровни доступа членов класса

Идентификатор доступа	private	Нет	protected	public
Тот же класс	+	+	+	+
Подкласс того же пакета	-	+	+	+
Не подкласс того же пакета	-	+	+	+
Подкласс в другом пакете	-	-	+	+
Не подкласс в другом пакете	-	-	-	+

Символ плюс (+) означает, что член доступен, а символ минус (-) — что нет. Нетривиальность ситуации усугубляется еще и тем, что подклассы и суперклассы могут размещаться в разных пакетах. Существует несколько правил, которые достаточно полно отражают ситуацию с доступностью различных членов класса.

- Наличие идентификатора `public` у члена класса означает, что он доступен везде: как в классе, так и за его пределами и даже в классах других пакетов.
- Наличие идентификатора `private` означает, что член доступен только в пределах класса, где он объявлен.
- Если у члена нет идентификатора доступа, он доступен в пределах пакета.
- Члены касса, объявленные с идентификатором `protected`, доступны в пакете и в подклассах вне пакета.

Поскольку ранее мы использовали один стандартный пакет по умолчанию, между открытыми членами и членами, для которых идентификатор доступа не указан, принципиальной разницы не было.

У классов также есть уровни доступа. Точнее, в описании класса можно в сигнатуре указать ключевое слово `public` (а можно не указывать). Если класс объявлен как `public`, он доступен отовсюду. Если идентификатор `public` не указан, класс доступен только в пределах своего пакета.

Если в программе выполняется обращение к классам, размещенным во внешних пакетах, необходимо указывать полное имя класса: через точку перечисляется вся иерархия пакетов, где размещен нужный класс. Например, если класс `MyClass` находится в подпакете `subpack` пакета `mypack`, то обращение к этому классу будет иметь вид:

```
mypack.subpack.MyClass
```

Чтобы можно было ссылаться на классы внешних пакетов в упрощенной форме, прибегают к импорту пакетов. При этом используется ключевое слово `import` (соответствующая команда размещается в начале файла после команды подключения пакета). Файл может содержать несколько инструкций импорта. Можно подключать (импортировать) отдельные классы пакета или весь пакет. В частности, для импорта класса после ключевого слова `import` указывают полное имя класса (то есть с учетом иерархии пакетов), например:

```
import mypack.subpack.MyClass
```

Для импорта всего пакета после имени пакета ставят звездочку (*), например:

```
import mypack.subpack.*
```

Существуют некоторые ограничения, накладываемые на импорт пакетов.

- Импортировать можно только открытые классы.
- Пакет `java.lang` (базовая библиотека) можно не импортировать — он и так доступен.
- Имя файла должно совпадать с именем открытого класса, если такой класс существует в файле.
- Если в пакете несколько открытых классов, они должны размещаться в разных файлах.

Интерфейсы

Из всего, что мне говорили, последнее, что я понял,
было «Здравствуйте».

Дж. Буш-старший

Ранее неоднократно отмечалось, что в Java запрещено множественное наследование. Причина отказа от множественного наследования связана с теми потен-

циальными проблемами, которые могут при этом возникать. Однако множественное наследование открывает широкие перспективы для составления эффективных программных кодов и значительно повышает гибкость программ. Выход был найден в использовании интерфейсов.

Интерфейсы во многом напоминают классы. Принципиально от класса интерфейс отличается тем, что содержит только сигнатуры методов без описания, а также поля-константы (поля, значения которых постоянны и не могут изменяться).

Описание интерфейса аналогично к описанию класса, только ключевое слово `class` необходимо заменить ключевым словом `interface`. Как отмечалось, для методов интерфейса указываются только сигнатуры. Описываемые в интерфейсе поля по умолчанию считаются неизменяемыми (как если бы они были описаны с ключевым словом `final`) и статическими (то есть `static`). Таким образом, поля интерфейса играют роль глобальных констант.

Практическое использование интерфейса подразумевает его реализацию. Эта процедура напоминает наследование абстрактных классов. Реализуется интерфейс в классе. Класс, который реализует интерфейс, должен содержать описание всех методов интерфейса. Методы интерфейса при реализации описываются как открытые. Один и тот же класс может реализовать одновременно несколько интерфейсов, равно как один и тот же интерфейс может реализовываться несколькими классами.

Для реализации интерфейса в классе в сигнатуре заголовка класса указывается инструкция `implements()`. С учетом того, что реализующий интерфейс класс может одновременно наследовать еще и суперкласс, общий синтаксис объявления класса, который наследует суперкласс и реализует несколько интерфейсов, имеет следующий вид:

```
class имя [extends суперкласс] implements интерфейс1,интерфейс2,...{
// тело класса
}
```

Если имеет место наследование классов, то после имени класса через ключевое слово `extends` указывается имя наследуемого суперкласса, затем идет ключевое слово `implements`. После ключевого слова `implements` через запятую перечисляются реализуемые в классе интерфейсы, а дальше все стандартно — указывается непосредственно тело класса. Напомним также, что перед ключевым словом `class` может размещаться ключевое слово `public`, определяющее уровень доступа класса.

В листинге 7.1 приведен пример программы, в которой используется интерфейс.

Листинг 7.1. Реализация интерфейса

```
// Интерфейс:
interface MyMath{
```

продолжение

Листинг 7.1 *(продолжение)*

```
// Сигнатура метода:
double Sinus(double x);
// Константа:
double PI=3.14159265358979;
}
// Класс реализует интерфейс:
class MyClass implements MyMath{
// Реализация метода (вычисление синуса):
public double Sinus(double x){
int i,n=1000;
double z=0,q=x;
for(i=1;i<=n;i++){
z+=q;
q*=(-1)*x*x/(2*i)/(2*i+1);}
return z;}
}
class MyMathDemo{
public static void main(String args[]){
MyClass obj=new MyClass();
// Использование константы:
double z=MyClass.PI/6;
// Вызов метода:
System.out.println("sin("+z+")="+obj.Sinus(z));}
}
```

В программе использован интерфейс `MyMath`, а также классы `MyClass` и `MyMathDemo`. Класс `MyClass` реализует интерфейс `MyMath`. Класс `MyMathDemo` содержит главный метод программы.

В интерфейсе `MyMath` объявлен метод (только сигнатура) с названием `Sinus()`. Метод имеет один аргумент типа `double` и возвращает результат того же типа. Кроме того, в интерфейсе объявлена константа `PI` типа `double`. Это приближенное значение числа π. Обращаем внимание, что хотя поле не содержит ни идентификатора `final`, ни идентификатора `static` в своем описании, оно является статической константой. Хотя это и не обязательно, названия полей интерфейсов принято записывать в верхнем регистре.

Класс `MyClass`, как отмечалось, реализует интерфейс `MyMath`. Поскольку в интерфейсе объявлен всего один метод, его и следует описать в классе `MyClass`. В данном случае использован ряд Тейлора для синуса:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

В главном методе программы создается объект `obj` класса `MyClass` и объявляется переменная `z` типа `double`. Этой переменной присваивается значение π/6. При этом используется константа `PI`, объявленная в интерфейсе `MyMath`. Константа статическая и наследуется в классе `MyClass`, поэтому ссылка на нее в главном

методе программы выглядит как `MyClass.PI`. В самом же классе к этому полю можно обращаться просто по имени, то есть как `PI`. Далее для аргумента `z` вычисляется значение синуса и результат выводится на экран. В итоге получаем сообщение:

```
sin(0.5235987755982984)=0.4999999999999995
```

Это достаточно близкий результат к точному значению 1/2.

Для читателей, знакомых с языком программирования C++, концепция интерфейсов может показаться на первый взгляд несколько странной, а сам подход к использованию интерфейсов нерациональным. Но это далеко не так. Еще раз подчеркнем, что главное назначение интерфейсов в Java — реализация множественного наследования (точнее, это — альтернативная технология по отношению к множественному наследованию классов). Дело в том, что основная масса проблем, возникающих при множественном наследовании классов, связана со спецификой реализации наследуемых методов. Другими словами, проблемы обычно появляются при попытке реализации конкретного программного кода методов и не связаны с самой структурой наследования. Через реализацию интерфейсов проблема «конечного кода» сводится к минимуму, поскольку интерфейсы содержат только объявления методов, а конкретная реализация этих методов выполняется в классе на последнем уровне иерархической структуры классов и интерфейсов.

Интерфейсные ссылки

Мы будем проводить иностранную политику
иностранными руками.

Дж. Буш-младший

При создании объектов класса в качестве типа объектной переменной может указываться имя реализованного в классе интерфейса. Другими словами, если класс реализует интерфейс, то ссылку на объект этого класса можно присвоить интерфейсной переменной — переменной, в качестве типа которой указано имя соответствующего интерфейса. Ситуация очень напоминает ту, что рассматривалась в предыдущей главе при наследовании, когда объектная переменная суперкласса ссылалась на объект подкласса. Как и в случае с объектными ссылками суперкласса, через интерфейсную ссылку можно сослаться не на все члены объекта реализующего интерфейс класса. Доступны только те методы, которые объявлены в соответствующем интерфейсе. С учетом того, что класс может реализовать несколько интерфейсов, а один и тот же интерфейс может быть реализован в разных классах, ситуация представляется достаточно пикантной.

В листинге 7.2 приведен пример программы, в которой используются интерфейсные ссылки.

Листинг 7.2. Интерфейсные ссылки

```
// Интерфейс:
interface Base{
int F(int n);
}
// Класс A реализует интерфейс Base:
class A implements Base{
// Двойной факториал числа:
public  int F(int n){
if(n==1||n==2) return n;
else return n*F(n-2);}
}
// Класс B реализует интерфейс Base:
class B implements Base{
// Факториал числа:
public int F(int n){
if(n<1) return 1;
else return n*F(n-1);}
}
class ImplDemo{
public static void main(String args[]){
// Интерфейсные переменные и создание объектов:
Base refA=new A();
Base refB=new B();
// Объектные переменные и создание объектов:
A objA=new A();
B objB=new B();
// Проверка работы методов:
System.out.println("1: "+refA.F(5));
System.out.println("2: "+refB.F(5));
System.out.println("3: "+objA.F(5));
System.out.println("4: "+objB.F(5));
// Изменение интерфейсных ссылок:
refA=objB;
refB=objA;
// Проверка результата:
System.out.println("5: "+refA.F(5));
System.out.println("6: "+refB.F(5));}
}
```

В интерфейсе `Base` объявлен всего один метод с названием `F()`, целочисленным аргументом и целочисленным результатом. Классы `A` и `B` реализуют интерфейс `Base`, причем каждый по-своему. В классе `A` метод `F()` описан так, что им возвращается в качестве результата двойной факториал от целочисленного аргумента (напомним, что по определению двойной факториал числа n есть произведение натуральных чисел до этого числа включительно «через два», то есть

$n!! = n(n-2)(n-4)...$). В классе B методом F() вычисляется факториал числа-аргумента метода (произведение натуральных чисел от 1 до числа n включительно, то есть $n! = n(n-1)(n-2)...1$). При описании метода F() в обоих классах использована рекурсия.

В главном методе программы командами Base refA=new A() и Base refB=new B() создаются два объекта классов A и B, причем ссылки на эти объекты записываются в интерфейсные переменные refA и refB. В качестве типа этих переменных указано имя интерфейса Base, а сами объекты создаются вызовом конструкторов соответствующих классов.

Затем создаются еще два объекта классов A и B, и ссылки на них записываются в объектные переменные objA и objB соответственно. Для этого используются команды A objA=new A() и B objB=new B().

После этого с помощью объектных и интерфейсных переменных несколько раз вызывается метод F() с аргументом 5. Отметим, что 5!!=15 и 5!=120. Поэтому если вызывается версия метода, описанная в классе A, результатом является число 15, а для версии метода, описанной в классе B, результат есть число 120. В частности, при вызове метода через переменные objA и refA вызывается версия метода, описанная в классе A, а при вызове метода через переменные objB и refB — версия, описанная в классе B.

После этого командами refA=objB и refB=objA ссылки «меняются местами»: интерфейсная переменная refA ссылается на объект класса B, а интерфейсная переменная refB — на объект класса A. После этого инструкцией refA.F(5) вызывается версия метода F() из класса B, а инструкцией refB.F(5) — версия метода F(), описанная в классе A. В результате выполнения программы получаем следующее:

```
1: 15
2: 120
3: 15
4: 120
5: 120
6: 15
```

В листинге 7.3 приведен другой пример, в котором один класс реализует несколько интерфейсов.

Листинг 7.3. Реализация нескольких интерфейсов

```
// Первый интерфейс:
interface One{
void setOne(int n);
}
// Второй интерфейс:
interface Two{
void setTwo(int n);
}
```

продолжение

Листинг 7.3 *(продолжение)*

```
// Суперкласс:
class ClassA{
int number;
void show(){
System.out.println("Поле number: "+number);}
}
// Подкласс наследует суперкласс и реализует интерфейсы:
class ClassB extends ClassA implements One,Two{
int value;
// Метод первого интерфейса:
public void setOne(int n){
number=n;}
// Метод второго интерфейса:
public void setTwo(int n){
value=n;}
// Переопределение метода суперкласса:
void show(){
super.show();
System.out.println("Поле value: "+value);}
}
class MoreImplDemo{
public static void main(String[] args){
// Интерфейсные переменные:
One ref1;
Two ref2;
// Создание объекта:
ClassB obj=new ClassB();
// Интерфейсные ссылки:
ref1=obj;
ref2=obj;
// Вызов методов:
ref1.setOne(10);
ref2.setTwo(-50);
// Проверка результата:
obj.show();}
}
```

Результат выполнения этой программы имеет вид:

```
Поле number: 10
Поле value: -50
```

Кратко поясним основные этапы реализации алгоритма. Итак, имеются два интерфейса `One` и `Two`, которые реализуются классом `ClassB`. Кроме того, класс `ClassB` наследует класс `ClassA`. В каждом из интерфейсов объявлено по одному методу: в интерфейсе `One` метод `setOne()`, а в интерфейсе `Two` метод `setTwo()`. Оба метода не возвращают результат и имеют один целочисленный аргумент.

У суперкласса ClassA объявлено поле int number и определен метод show(), который выводит значение поля на экран. При наследовании в подклассе ClassB этот метод переопределяется так, что выводит значения двух полей: наследуемого из суперкласса поля number и поля int value, описанного непосредственно в подклассе.

В классе ClassB методы setOne() и setTwo() реализованы так, что первый метод присваивает значение полю number, второй — полю value.

В главном методе программы создаются две интерфейсные переменные: переменная ref1 типа One и переменная ref2 типа Two. Кроме того, создается объект obj класса ClassB. В качестве значений интерфейсным переменным присваиваются ссылки на объект obj. Это возможно, поскольку класс ClassB реализует интерфейсы One и Two. Однако в силу того же обстоятельства через переменную ref1 можно получить доступ только к методу setOne(), а через переменную ref2 — только к методу setTwo(). Командами ref1.setOne(10) и ref2.setTwo(-50) полям объекта obj присваиваются значения, а командой obj.show() значения полей выводятся на экран.

Расширение интерфейсов

Я унаследовал всех врагов своего отца
и лишь половину его друзей.

Дж. Буш-младший

Подобно классам, один интерфейс может наследовать другой интерфейс. В этом случае говорят о расширении интерфейса. Как и при наследовании классов, при расширении интерфейсов указывается ключевое слово extends. Синтаксис реализации расширения интерфейса фактически такой же, как и синтаксис реализации наследования классов:

```
interface имя1 extends имя2{
// тело интерфейса
}
```

В листинге 7.4 приведен пример расширения интерфейса.

Листинг 7.4. Расширение интерфейса

```
// Интерфейс:
interface BaseA{
int FunA(int n);
}
// Расширение интерфейса:
interface BaseB extends BaseA{
int FunB(int n);
}
```

продолжение

Листинг 7.4 *(продолжение)*

```
// Реализация интерфейса:
class MyClass implements BaseB{
public int FunA(int n){
if(n<1) return 1;
else return n*FunA(n-1);}
public int FunB(int n){
if(n==1||n==2) return n;
else return n*FunB(n-2);}
}
class ImplExtDemo{
public static void main(String args[]){
MyClass obj=new MyClass();
System.out.println("1: "+obj.FunA(5));
System.out.println("2: "+obj.FunB(5));}
}
```

В результате выполнения этой программы получаем:

```
1: 120
2: 15
```

Что касается самого программного кода, то он достаточно прост. Интерфейс `BaseA` содержит объявление метода `FunA()`. У метода целочисленный аргумент и результат метода — тоже целое число. Интерфейс `BaseB` расширяет (наследует) интерфейс `BaseA`. Непосредственно в интерфейсе `BaseB` объявлен метод `FunB()` с целочисленным результатом и целочисленным аргументом. Учитывая наследуемый из интерфейса `BaseA` метод `FunA()`, интерфейс `BaseB` содержит сигнатуры двух методов `FunA()` и `FunB()`. Поэтому в классе `MyClass`, который реализует интерфейс `BaseB`, необходимо описать оба эти метода. Метод `FunA()` описывается как возвращающий в качестве значения факториал числа-аргумента метода, а метод `FunB()` — как возвращающий в качестве значения двойной факториал числа. В главном методе программы создается объект `obj` класса `MyClass` и последовательно вызываются методы `FunA()` и `FunB()`. Думается, результат этих вызовов особых комментариев не требует.

Резюме

1. В Java существуют специальные контейнеры для классов — пакеты. При определении пакета в файле с описанием класса, включаемого в пакет, первой командой следует инструкция `package`, после которой указывается имя пакета.
2. Взаимное расположение классов по пакетам влияет на доступность членов этих классов.
3. Интерфейс напоминает класс, но содержит только сигнатуры методов (без описания), а также поля-константы (значения полей не могут изменяться).

Интерфейс описывается так же, как класс, но только с использованием ключевого слова `interface`.

4. Интерфейсы реализуются в классах. Соответствующий класс должен содержать описание всех методов интерфейса, которые описываются как открытые (`public`). Один класс может реализовать несколько интерфейсов. Для реализации интерфейса в сигнатуре заголовка класса используется инструкция `implements`.
5. При создании объектов класса, который реализует интерфейс, в качестве типа объектной переменной может указываться имя этого интерфейса. При этом через такую интерфейсную ссылку доступны только те методы, которые объявлены в интерфейсе.
6. Существует расширение интерфейсов, когда один интерфейс наследует другой интерфейс. По аналогии с наследованием классов, в этом случае используется ключевое слово `extends`.

Глава 8. Работа с текстом

«В начале было Слово». С первых строк
Загадка. Так ли понял я намек?

И. Гёте. Фауст

В некоторых примерах из предыдущих глав текст уже использовался. По крайней мере, мы имели дело с текстовыми литералами — собственно текстом, заключенным в двойные кавычки. В Java текст — это объект. Для работы с текстом служат два встроенных Java-класса: `String` и `StringBuffer`. Поэтому с формальной точки зрения создание текста сводится к созданию объекта одного из этих классов. В этой главе рассмотрены оба эти класса. Каждый из них имеет свои особенности, хотя у них больше сходств, чем различий. Главное принципиальное различие состоит в том, что объекты класса `String` изменять нельзя, а объекты класса `StringBuffer` — можно. По большому счету, содержание этой главы ограничивается описанием свойств и возможностей классов `String` и `StringBuffer`. Кроме этого, в конце главы кратко описываются способы обработки аргументов командной строки.

Классы `String` и `StringBuffer` определены в базовом пакете `java.lang`, который доступен по умолчанию, поэтому для создания объекта класса `String` или `StringBuffer` импорт пакетов выполнять не нужно. Оба класса определены как неизменяемые (`final`), то есть они не могут быть суперклассами для наследования.

Объекты класса String

Все те вопросы, которые были поставлены,
мы их все соберем в одно место.

В. Черномырдин

Один из способов создания «текстовой переменной» подразумевает создание объекта класса `String`. Чтобы создать объект класса, необходимо, как минимум, знать, какие у этого класса есть конструкторы. Что касается класса `String`, то имеет смысл выделить следующие конструкторы.

- *Конструктор создания пустой строки*. В этом случае конструктору аргументы не передаются. Пример команды создания объекта класса `String` со значением в виде пустой строки имеет вид:

```
String s=new String();
```

- *Конструктор создания текстовой строки на основе символьного массива*. В этом случае аргументом конструктору передается имя массива символов. Результатом является текст, составленный из всех символов массива в порядке их размещения в массиве. Пример создания текстовой строки на основе символьного массива:

```
char symbols[]={'a','b','c'};
String s=new String(symbols);
```

  В этом конструкторе, помимо имени массива, можно указать индекс элемента массива, начиная с которого будет извлекаться строка, а также длину строки в символах. Например, так:

```
char symbols={'a','b','c','d','e','f'};
String s=new String(symbols,2,3);  // s="cde"
```

- *Конструктор копирования объекта*. Аргументом конструктора указывается переменная текстового типа, ссылающаяся на уже существующий текстовый объект или текстовый литерал. В результате создается новый объект с таким же текстовым значением, как и исходный. Например:

```
String obj=new String("Текстовая строка");
String s=new String(obj);
```

Это — далеко не весь список доступных конструкторов. В частности, существует конструктор, принимающий в качестве аргумента массив типа `byte` с кодами символов, которые автоматически преобразуются в буквы, а буквы — в текст. В этом конструкторе также можно указывать второй и третий аргументы — соответственно начальный индекс элемента массива, с которого начинается формирование текстовой строки, и длину строки в символах.

В листинге 8.1 приведен простой пример создания текстовой строки на основе символьного массива, а также на основе уже существующего текстового объекта.

Листинг 8.1. Создание текстовой строки на основе символьного массива

```
class MakeString{
public static void main(String args[]){
char symbs[]={'J','a','v','a'};
String strA=new String(symbs);
String strB=new String(strA);
System.out.println(strA);
System.out.println(strB);}
}
```

В результате выполнения этой программы на экран дважды выводится слово `Java`. Пример этот очень простой: сначала создается символьный массив `symbs`,

затем на его основе создается текстовая строка strA, после чего создается еще одна текстовая строка strB с таким же значением (слово Java), как и строка strA. Обращаем внимание, что в данном случае объектные переменные strA и strB класса String ссылаются на разные объекты, но текстовые значения в этих объектах записаны одинаковые.

По-иному обстояли бы дела, если одной объектной переменной, например srtA, в качестве значения была присвоена другая объектная переменная (команда вида strA=strB). В последнем случае обе переменные ссылались бы на один и тот же объект. В рассмотренном примере объекты разные.

Еще один способ создания текстовой строки проиллюстрирован в листинге 8.2. Здесь текстовая строка создается на основе числового массива. Каждое число в массиве элементов типа byte интерпретируется как код символа в кодировке ASCII.

Листинг 8.2. Создание текстовой строки на основе числового массива

```
class SubStringConstr{
public static void main(String args[]){
byte ascii[]={65,66,67,68,69,70};
String strBig=new String(ascii);
System.out.println(strBig);
String strSmall=new String(ascii,2,3);
System.out.println(strSmall);}
}
```

Создаются две строки, обе на основе массива ascii. В первом случае используется весь массив — он передается аргументом конструктору класса String. Во втором случае извлекается только часть массива: три элемента, начиная с элемента с индексом 2 (здесь нелишне напомнить, что индексация элементов массива начинается с нуля). Результат выполнения программы следующий:

```
ABCDEF
CDE
```

Как уже отмечалось, значения объектов класса String после их создания изменены быть не могут. Это обстоятельство накладывает некоторые ограничения на методы работы со строками. Например, чтобы изменить уже существующий текст, необходимо создавать новый объект. Некоторые методы, с помощью которых выполняется обработка текстовых строк, рассматриваются в следующих разделах этой главы.

Бывает важно знать длину текстовой строки (в символах). Для определения длины строки, записанной в текстовом объекте, используют метод length(). Этот метод вызывается из объекта, для которого определяется длина строки. Например, если str является объектом класса String, то определить длину текстовой строки, на которую ссылается объектная переменная str, можно командой str.length(). Более того, текстовые литералы в Java реализуются в виде объектов класса String. Это означает, что метод length() также может быть вызван

и из текстового литерала. В этом смысле вполне корректной является, например, следующая команда (результат равен 12):

```
"Всем привет!".length()
```

Что касается базовых операций, то по отношению к текстовым строкам можно применять только операцию сложения (с оператором +), которая интерпретируется как объединение (конкатенация) соответствующих строк. Пример конкатенации текстовых строк:

```
String str="Евгению Петрову "+18+" лет.";
```

В результате получаем текст "Евгению Петрову 18 лет.". Если при сложении кроме текстовых операндов присутствуют и операнды других типов (например, числа), то выполняется их автоматическое приведение к текстовому формату (преобразование в объект типа String). При этом могут складываться довольно неожиданные ситуации. Например:

```
String str="Число три: "+1+2;
```

В результате выполнения этой команды текст, записанный в объект класса String, на который ссылается переменная str, получается таким: "Число три: 12". Причина кроется в способе вычисления выражения "Число три: "+1+2. Поскольку в нем кроме двух числовых операндов 1 и 2 присутствует еще и текстовый операнд "Число три: ", причем в выражении он идет первым, а выражение вычисляется слева направо, то к тексту "Число три: " добавляется (объединение строк) текстовое значение "1", после чего к полученному тексту добавляется текстовое значение "2". Чтобы предотвратить такое экзотическое сложение, вместо приведенной нужно воспользоваться командой:

```
String str="Число три: "+(1+2);
```

Здесь с помощью скобок изменен порядок вычисления выражения: сначала вычисляется сумма чисел, а уже после этого полученное число преобразуется в текстовый формат. В результате переменная str ссылается на текст "Число три: 3".

При сложении текстовых значений одним из операндов может быть не только текст или число, но и объект, в том числе класса, определенного пользователем. Правила преобразования объекта в текстовый формат в этом случае определяются методом toString(). В силу особой важности этого метода, рассмотрим его отдельно.

В завершение раздела сделаем несколько замечаний относительно реализации команды вида

```
String str="Текст";
```

Формально результатом команды является создание объектной переменной str типа String, которая ссылается на текст "Текст". На первый взгляд кажется, что такой способ создания текстового объекта отличается от рассматривавшихся ранее. Однако это не совсем так, особенно если учесть, что в Java текстовые литералы являются объектами класса String.

Если вернуться к представленной команде, то ее удобно разбить на две:

```
String str;
str="Текст";
```

Первой командой объявляется объектная переменная `str` класса `String`. Даже если нет объекта, объектную переменную объявить можно. Напомним, что значением объектной переменной является ссылка на объект соответствующего класса. Значение объектной переменной присваивается второй командой. Поскольку текстовый литерал `"Текст"` является объектом класса `String`, ссылка на этот объект командой `str="Текст"` присваивается переменной `str`. Сам же объект текстового литерала создается автоматически, программисту в этом процессе принимать участия не нужно — достаточно, чтобы тестовый литерал появился в программном коде.

Метод toString()

Мы продолжаем то, что мы уже много наделали.

В. Черномырдин

Метод `toString()` определен в классе `Object`, находящемся на вершине иерархии классов Java (это общий суперкласс Java). Метод вызывается по умолчанию при преобразовании объекта в текстовый формат. Благодаря тому, что метод можно перегружать, открывается целый ряд достаточно интересных возможностей. Рассмотрим их.

Для перегрузки метода `toString()`, как минимум, необходимо знать его сигнатуру. Метод в качестве значения возвращает объект класса `String` и не имеет аргументов. Как отмечалось, метод вызывается по умолчанию при преобразовании объекта в текстовый тип. Примеры таких ситуаций: сложение объекта с текстовой строкой или использование объекта в качестве аргумента метода `println()`.

Самый простой и наглядный способ познакомиться с методикой переопределения и использования метода `toString()` — рассмотреть пример. Обратимся к листингу 8.3.

Листинг 8.3. Переопределение метода toString()

```
class ComplNums{
// Поля класса:
double Re;
double Im;
// Конструктор:
ComplNums(double x,double y){
Re=x;
Im=y;}
// Переопределение метода toString():
public String toString(){
String result="",sign="",ImPart="",RePart="";
```

```
if(Re!=0||(Re==0&&Im==0)) RePart+=Re;
if((Im>0)&&(Re!=0)) sign+="+";
if(Im!=0) ImPart+=Im+"i";
result=RePart+sign+ImPart;
return result;}
}
class toStringDemo{
public static void main(String[] args){
for(int i=1;i<=3;i++){
   for(int j=1;j<=5;j+=2){
       ComplNums z=new ComplNums(i-2,j-3);
// Автоматический вызов метода toString():
       System.out.println(z);
   }
}
}
}
```

В программе создается некое подобие класса для реализации комплексных чисел. У класса `ComplNums` два поля `Re` и `Im` типа `double`. Это действительная и мнимая части комплексного числа. Конструктор класса принимает два аргумента — значения полей `Re` и `Im`. Кроме этого, в классе переопределен метод `toString()`. Метод, описанный как `public`, возвращает в качестве значения объект класса `String` и не имеет аргументов. На метод `toString()` возложена обязанность сформировать текстовое представление комплексного числа с заданными действительной и мнимой частями. Правила представления комплексного числа следующие.

- Если действительная часть равна нулю, а мнимая отлична от нуля, то действительная часть не отображается.
- Если и мнимая, и действительная части числа равны нулю, отображается действительная (нулевая) часть.
- Если действительная часть отлична от нуля, а мнимая равна нулю, мнимая часть не отображается.
- Если действительная часть не отображается, то не отображается и знак плюс перед мнимой частью.

Программный код метода `toString()` с помощью нескольких условных инструкций реализует эти правила. В частности, результат метода предварительно записывается в объектную переменную `result` класса `String`. Переменная `result`, в свою очередь, представляется как объединение трех текстовых строк: `RePart` (текстовое представление действительной части комплексного числа), `ImPart` (текстовое представление мнимой части комплексного числа с учетом мнимой единицы *i*) и `sign` (знак между действительной и мнимой частями комплексного числа). Все четыре переменные инициализируются пустой строкой.

Командой `if(Re!=0||(Re==0&&Im==0)) RePart+=Re` изменяется, если необходимо, текстовое представление действительной части. Это происходит, если действительная часть отлична от нуля или если и действительная, и мнимая части равны нулю.

Командой `if((Im>0)&&(Re!=0)) sign+="+"` определяется текстовое представление для знака перед мнимой частью. Это плюс, если мнимая часть положительна, а действительная отлична от нуля. В противном случае значение переменной `sign` остается неизменным, а знак минус автоматически добавляется в текстовое представление мнимой части за счет отрицательного поля `Im`.

Текстовое представление для мнимой части определяется командой `if(Im!=0) ImPart+=Im+"i"`. В представление мнимой части, кроме числового значения, добавляется текстовое представление мнимой единицы `"i"`. Мнимая часть отображается, если она отлична от нуля.

В главном методе программы с помощью двойного цикла перебираются различные значения для действительной и мнимой частей создаваемого в теле объекта `z` класса `ComplNums`. Всего перебирается девять вариантов: разные комбинации положительной, отрицательной и нулевой действительной и мнимой частей. Результат выполнения программы имеет вид:

```
-1.0-2.0i
-1.0
-1.0+2.0i
-2.0i
0.0
2.0i
1.0-2.0i
1.0
1.0+2.0i
```

Обращаем внимание на способ отображения текстового представления объекта `z`. Используется команда `System.out.println(z)`, то есть объект указывается аргументом метода `println()`. Метод `toString()` вызывается автоматически при попытке преобразовать объект `z` в текстовый формат (в объект класса `String`).

Переопределение метода `toString()` — очень удобный прием, который позволяет экономить не только усилия по реализации вывода информации об объектах на экран, но и создавать компактные и продуктивные программы.

Методы для работы со строками

— Да, как эксперимент это интересно.
Но какое практическое применение?
— Господи, именно практическое!

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В классе `String` есть ряд методов, предназначенных для обработки текстовых строк. Некоторые из них кратко обсуждаются в этом разделе. Обратимся к примеру, представленному в листинге 8.4.

Листинг 8.4. Методы для работы со строками

```
class getCharsDemo{
public static void main(String args[]){
// Текстовая строка:
String s="Пример текстовой строки - язык Java";
// Символьный массив:
char buf[]=new char[9];
s.getChars(s.length()-9,s.length(),buf,0);
System.out.println(buf);
// Символ:
char symbol;
symbol=s.charAt(21);
System.out.println(symbol);
// Разделитель:
System.out.println("-------------------------");
// Массив чисел:
byte nums[];
nums=s.getBytes();
for(int i=0;i<s.length();i++){
System.out.print(nums[i]+" ");
if((i+1)%6==0) System.out.println();}
// Разделитель:
System.out.println("\n-------------------------");
char chars[]=new char[s.length()];
chars =s.toCharArray();
for(int i=0;i<s.length();i++){
if(s.charAt(i)==' ') System.out.println("-->");
else System.out.print(chars[i]+" * ");}
}
}
```

В программе объявляется объектная переменная `s` с одновременной инициализацией текстовым значением. После этого объявляется символьный массив `buf` из 9 элементов. В этот массив посимвольно считывается подстрока строки `s`. Используется для этого метод `getChars()` (команда `s.getChars(s.length()-9,s.length(),buf,0)`). Аргументами методу передаются: начальный индекс элемента считываемой подстроки, индекс первого не считываемого в подстроку символа, имя массива, в который посимвольно записывается считываемая подстрока, и индекс элемента в этом массиве, начиная с которого массив заполняется символами подстроки. В данном случае считывается подстрока из элементов с индексами от `s.length()-9` до `s.length()-1` включительно, подстрока посимвольно заносится в массив `buf`, начиная с первого элемента массива `buf[0]`. Элементы массива выводятся на экран командой `System.out.println(buf)` — для вывода символьного массива достаточно указать его имя аргументом метода `println()`.

Для считывания символа с определенным индексом из текстовой строки используется метод charAt(). Аргументом метода указывается индекс символа в текстовой строке (индексация символов, как и в массиве, начинается с нуля). Результатом является символ в соответствующей позиции текстовой строки. Исходная строка при этом не меняется.

В программе объявлена переменная массива nums (тип byte[]). С помощью метода getBytes() создается массив и заполняется кодами символов исходной текстовой строки, а результат присваивается переменной nums. Аргументы методу не передаются, а результатом метода является массив числовых кодов символов строки, из которой вызывается метод. Для удобства при выводе значений массива nums числа распечатываются в несколько строк. Размер числового массива nums равен количеству символов в текстовой строке s.

Аналогично дела обстоят с символьным массивом chars. Только в данном случае массив заполняется символами текстовой строки, а не числовыми кодами, и делается это методом toCharArray(). Метод не имеет аргументов и возвращает в качестве значения символьный массив из букв тестовой строки, из которой вызывается метод.

При выводе на экран содержимого массива chars в местах пробелов осуществляется переход на новую строку вывода, между буквами вставляются символы «звездочка», а строки вывода завершаются импровизированными стрелками. Результат выполнения программы такой:

```
язык Java
к
-------------------------
-49 -16 -24 -20 -27 -16
32 -14 -27 -22 -15 -14
-18 -30 -18 -23 32 -15
-14 -16 -18 -22 -24 32
-106 32 -1 -25 -5 -22
32 74 97 118 97
-------------------------
П * р * и * м * е * р * -->
т * е * к * с * т * о * в * о * й * -->
с * т * р * о * к * и * -->
- * -->
я * з * ы * к * -->
J * a * v * a *
```

Разумеется, это далеко не все встроенные методы, предназначенные для работы с текстовыми строками. При необходимости читатель может обратиться к справочной системе используемого компилятора или специальной справочной литературе.

Сравнение строк

Какое глубокое проникновение в суть вещей!
Впрочем, принц всегда очень тонко анализировал самую
сложную ситуацию.

Из к/ф «Приключения принца Флоризеля»

При работе с текстом нередко возникает необходимость сравнить разные текстовые строки на предмет совпадения. Используют в этом случае методы `equals()` и `equalsIgnoreCase()`. Разница между этими методами состоит в том, что первый метод сравнивает строки с учетом регистра, а второй состояние регистра игнорирует. Метод `equals()` возвращает в качестве значения `true`, если строки состоят из одинаковых символов, размещенных на одинаковых позициях в строке. При этом строчная и прописная буквы интерпретируются как разные символы. Если полного совпадения нет, в качестве результата методом возвращается значение `false`. Метод `equalsIgnoreCase()` при сравнении строк интерпретирует строчную и прописную буквы как один и тот же символ.

Каждый из упомянутых методов вызывается из объекта строки, аргументом методу также передается строка. Это именно те строки, которые сравниваются методами.

Причина, по которой для сравнения строк нельзя использовать операторы сравнения «равно» (`==`) и «не равно» (`!=`), думается очевидна, однако на ней стоит все же остановиться. Вообще говоря, эти операторы использовать можно, но результат может быть несколько неожиданным.

Предположим, необходимо сравнить текстовые строки `strA` и `strB`. Объектные переменные описаны как `String strA` и `String strB`, после чего им присвоены значения. Если для сравнения использовать команду `strA==strB` или `strA!=strB`, то в соответствии с этими командами сравниваются значения объектных переменных `strA` и `strB`, а не текстовое содержание объектов, на которые эти переменные ссылаются. Значениями переменных `strA` и `strB` являются ссылки (адреса) на соответствующие объекты. Например, результатом выражения `strA==strB` является значение `true`, если переменные `strA` и `strB` ссылаются на один и тот же объект. Если же переменные ссылаются на разные объекты, значение выражения равно `false`. При этом разные объекты могут иметь одинаковые текстовые значения.

В листинге 8.5 приведен пример программы, в которой выполняется сравнение текстовых объектов.

Листинг 8.5. Сравнение строк

```
class equalStringsDemo{
public static void main(String args[]){
String strA="Алексей Васильев";
```

продолжение

Листинг 8.5 *(продолжение)*

```
String strB=new String("Алексей Васильев");
String strC="Васильев Алексей";
String strD="АЛЕКСЕЙ ВАСИЛЬЕВ";
System.out.println(strA+" то же самое, что "+strB+" --> "+strA.equals(strB));
System.out.println(strA+" то же самое, что "+strC+" --> "+strA.equals(strC));
System.out.println(strA+" то же самое, что "+strD+" --> "+strA.equals(strD));
System.out.println(strA+" то же самое, что "+strD+" --> "+strA.
equalsIgnoreCase(strD));
System.out.println(strA+" то же самое, что "+strB+" --> "+(strA==strB));
}}
```

Результат выполнения программы имеет вид:

```
Алексей Васильев то же самое, что Алексей Васильев --> true
Алексей Васильев то же самое, что Васильев Алексей --> false
Алексей Васильев то же самое, что АЛЕКСЕЙ ВАСИЛЬЕВ --> false
Алексей Васильев то же самое, что АЛЕКСЕЙ ВАСИЛЬЕВ --> true
Алексей Васильев то же самое, что Алексей Васильев --> false
```

В программе создаются четыре текстовые строки. Все четыре переменные `strA`, `strB`, `strC` и `strD` ссылаются на разные объекты. При этом текстовое значение объектов, на которые ссылаются переменные `strA` и `strB`, одинаково. Текстовое значение объекта, на который ссылается переменная `strD`, совпадает с первыми двумя с точностью до состояния регистра. Объект, на который ссылается переменная `strC`, имеет текстовое значение, отличное от значений прочих объектов.

Текстовые значения созданных объектов сравниваются с помощью методов `equals()`, `equalsIgnoreCase()` и оператора равенства `==`. Результат сравнения выводится на экран.

При сравнении значений объектов переменных `strA` и `strB` с помощью метода `equals()` подтверждается равенство строк. Этот же метод при сравнении строк `strA` и `strC` дает значение `false` — строки не совпадают. То же происходит при сравнении строк `strA` и `strD`. Если строки `strA` и `strD` сравнивать с помощью метода `equalsIgnoreCase()`, результатом является совпадение строк. Разное состояние регистра букв сравниваемых строк при этом игнорируется. Наконец, если сравнивать переменные `strA` и `strB` с помощью оператора сравнения «равно» (командой `strA==strB`), получаем значение `false`, поскольку переменные ссылаются на разные объекты.

Отметим, что если вместо команды `String strB=new String("Алексей Васильев")` воспользоваться командой `String strB="Алексей Васильев"` (практически такой же, как и для переменной `strA`), результатом выражения `strA==strB` будет `true`. Причина проста: обеим переменным `strA` и `strB` в качестве значения присваивается ссылка на один и тот же литерал, то есть на один объект класса `String`. Поэтому значения переменных (адреса объекта ссылки) совпадают, что и подтверждает сравнение их с помощью оператора сравнения «равно».

Поиск подстрок и индексов

Здесь были люди, и я их найду!

Из к/ф «Чародеи»

Еще одна распространенная задача связана с поиском подстроки в строке или, в более простом варианте, поиском символа в строке. Полезными при этом могут оказаться методы `indexOf()` и `lastIndexOf()`. Первым аргументом обоих методов указывается искомый символ (значение типа `char`) или подстрока (объект класса `String`). Может указываться и второй целочисленный аргумент. Он определяет начальную точку поиска (индекс символа в строке, с которого начинается поиск). Для метода `indexOf()` поиск выполняется от точки поиска до конца строки, а методом `lastIndexOf()` выполняется поиск от точки поиска до начала строки. Результатом обоих методов является индекс первого появления символа в строке или первого вхождения подстроки в строку. Если совпадений не найдено, возвращается значение –1. В листинге 8.6 приведен пример программы, в которой используются эти методы.

Листинг 8.6. Поиск символов и подстрок

```
class indexOfDemo{
public static void main(String args[]){
String s="Всегда слова обдумывая чьи-то\n"+
"Ты видеть должен, что за ними скрыто.\n"+
"И помни, что уменье что-то скрыть\n"+
"Порой ценней уменья говорить!";
System.out.println(s);
System.out.println("1: "+s.indexOf('a'));
System.out.println("2: "+s.lastIndexOf('a'));
System.out.println("3: "+s.indexOf("то"));
System.out.println("4: "+s.indexOf('a',10));
System.out.println("5: "+s.indexOf("то",10));}
}
```

В программе объявляется переменная `s` класса `String`, и в качестве значения этой переменной присваивается текстовый литерал. Поскольку литерал достаточно большой, он разбит на отдельные текстовые фрагменты, а нужное текстовое значение получается объединением этих фрагментов.

Полученная строка выводится на экран. Далее с помощью методов `indexOf()` и `lastIndexOf()` определяются индексы вхождения символа `'a'` и подстроки `"то"`, начиная с начала и конца строки, а также с десятой (индексация начинается с нуля) позиции в строке. Результат выполнения программы имеет вид:

```
Всегда слова обдумывая чьи-то
Ты видеть должен, что за ними скрыто.
И помни, что уменье что-то скрыть
```

```
Порой ценней уменья говорить!
1: 5
2: 53
3: 27
4: 11
5: 27
```

В исходном тексте использовалась инструкция `\n` для перехода на новую строку. При подсчете позиции символов наличие этой инструкции также принимается в расчет — она обрабатывается как символ строки.

Изменение текстовых строк

На самом деле я не говорил ничего из того,
что я говорил.

Й. Берра

Как уже неоднократно отмечалось, объекты класса `String` не могут изменяться. Однако могут изменяться ссылки на эти объекты. Поэтому если необходимо изменить текст, связанный с какой-то текстовой переменной (объектной переменной класса `String`), то, во-первых, создается новый объект класса `String` с нужными свойствами, во-вторых, значение объектной переменной меняется так, чтобы она ссылалась на этот объект. Есть еще один способ изменения текстовых строк. Заключается он в использовании объектов класса `StringBuffer`, но об этом рассказывается несколько позже.

В табл. 8.1 представлены некоторые методы, полезные при внесении изменений в текстовые строки.

Таблица 8.1. Методы для внесения изменений в текстовые строки

Метод	Назначение
`substring()`	Методом `substring()` возвращается в качестве результата текстовая подстрока (объект класса `String`) строки, из которой вызывается метод. Аргументами метода указывают индекс начала подстроки в строке и индекс первого не входящего в подстроку символа строки. Можно указывать только первый аргумент
`concat()`	Методом `concat()` выполняется объединение строк: текстовая строка, указанная аргументом метода, добавляется в конец текстовой строки, из которой вызывается метод. Получаемый в результате объединения текстовый объект класса `String` возвращается в качестве результата метода
`replace()`	У метода `replace()` два аргумента символьного (`char`) типа. В качестве результата методом возвращается текстовая строка, которая получается заменой в строке вызова первого символа-аргумента вторым

Метод	Назначение
`trim()`	Результатом метода `trim()` является текстовый объект, который получается из строки вызова удалением начальных и конечных пробелов. Метод не имеет аргументов
`toLowerCase()`	Методом `toLowerCase()` в качестве результата возвращается текстовая строка (объект класса `String`), которая получается из строки вызова переводом всех букв в нижний регистр (все буквы строчные). Метод аргументов не имеет
`toUpperCase()`	Методом `toUpperCase()` в качестве результата возвращается текстовая строка (объект класса `String`), которая получается из строки вызова переводом всех букв в верхний регистр (все буквы прописные). Метод аргументов не имеет

Примеры внесения изменений в текстовые строки приведены в программе из листинга 8.7.

Листинг 8.7. Изменение текстовых строк

```
class StringReplace{
public static void main(String args[]){
String str="Мы программируем на C++";
String s,s1,s2,s3,s4;
// Извлечение подстроки:
s=str.substring(3,21);
System.out.println(s);
// Объединение строк:
s1=str.concat(" и Java");
System.out.println(s1);
// Замена символов:
s2=s1.replace(' ','_');
System.out.println(s2);
// Перевод в нижний регистр:
s3=s1.toLowerCase();
System.out.println(s3);
// Перевод в верхний регистр:
s4=s1.toUpperCase();
System.out.println(s4);
}}
```

В результате выполнения этой программы получаем такую последовательность сообщений:

```
программируем на C
Мы программируем на C++ и Java
Мы_программируем_на_C++_и_Java
мы программируем на c++ и java
МЫ ПРОГРАММИРУЕМ НА C++ И JAVA
```

Рассмотрим представленный программный код более детально и проанализируем результат его выполнения. В главном методе создается строка str со значением:

```
"Мы программируем на C++"
```

Кроме того, объявляются еще пять объектных переменных типа String. Командой s=str.substring(3,21) из строки str извлекается подстрока с 4-го по 21-й символы включительно, и результат (строка "программируем на C") записывается в переменную s. Значение этой переменной выводится на экран.

Строка s1 получается объединением строки str и текстового литерала " и Java" — для этого использована команда:

```
s1=str.concat(" и Java")
```

В результате после вывода строки s1 на экран мы получаем сообщение:

```
Мы программируем на C++ и Java
```

Команда s2=s1.replace(' ','_') означает, что строка s2 получается из строки s1 заменой всех пробелов символами подчеркивания. При выводе строки s2 на экран получаем:

```
Мы_программируем_на_C++_и_Java
```

С помощью команды s3=s1.toLowerCase() формируется строка s3, состоящая из строчных символов строки s1. Аналогично командой s4=s1.toUpperCase() создаем строку, состоящую из прописных символов строки s1. При выводе строк s3 и s4 на экран получаем соответственно сообщения:

```
мы программируем на c++ и java
МЫ ПРОГРАММИРУЕМ НА C++ И JAVA
```

Класс StringBuffer

В этот ресторан больше никто не ходит,
потому что он всегда переполнен.

Й. Берра

Текстовые строки могут быть реализованы не только как объекты класса String, но и как объекты класса StringBuffer. Принципиальное отличие этих объектов состоит в том, что объекты класса StringBuffer можно изменять. Другими словами, если текст реализован в виде объекта класса StringBuffer, в этот текст можно вносить изменения, причем без создания нового объекта. В частности, при работе с объектами класса StringBuffer можно добавлять подстроки в средину и конец строки. Делается это за счет выделения дополнительной памяти при создании объекта класса StringBuffer.

У класса StringBuffer несколько конструкторов, среди которых можно выделить конструктор без аргументов StringBuffer(), конструктор с числовым аргументом и конструктор с текстовым аргументом (типа String или StringBuffer).

При использовании конструктора без аргумента создается объект класса StringBuffer со значением в виде пустой текстовой строки, а также автоматически резервируется память еще для 16-ти символов (буфер памяти). Чтобы в явном виде указать размер буфера памяти при создании объекта класса StringBuffer, используют конструктор с числовым аргументом. Для создания копии уже существующего текстового объекта применяют конструктор с текстовым аргументом.

Некоторые методы для работы с объектами класса StringBuffer перечислены и кратко описаны в табл. 8.2.

Таблица 8.2. Методы для работы с классом StringBuffer

Метод	Описание
length()	Метод возвращает текущую длину текстовой строки
capacity()	Методом возвращается выделенный для данной текстовой переменной объем памяти (в символах, то есть количество символов, которые можно записать в текстовую строку)
ensureCapacity()	Метод выделения памяти для уже созданного объекта. Размер выделяемой памяти указывается аргументом метода
setLength()	Методом устанавливается длина текстовой строки (аргумент метода)
charAt()	Методом возвращается символ в строке с указанным индексом (аргумент метода)
setCharAt()	У метода два аргумента: индекс символа в строке и символьное значение. Символ строки с заданным первым аргументом индексом заменяется символом, указанным вторым аргументом метода. Изменяется исходная строка
getChars()	Копирование строки в символьный массив. Аргументы метода: начальный индекс подстроки и индекс первого не входящего в подстроку символа, массив, в который выполняется копирование, а также индекс элемента в этом массиве, начиная с которого в массив производится посимвольное копирование подстроки
append()	Методом в конец строки вызова добавляется текст, указанный аргументом метода
insert()	Методом в строку вызова выполняется вставка текста, указанного вторым аргументом метода. Первым аргументом метода указывается индекс начала вставки подстроки
reverse()	Метод меняет порядок следования символов в строке вызова. Аргументов у метода нет

Метод	Описание
`delete()`	Методом из строки вызова удаляется подстрока. Первым аргументом метода указывается индекс начала удаляемой подстроки, вторым — индекс первого после удаляемой подстроки символа
`deleteCharAt()`	Методом из строки вызова удаляется символ с индексом, указанным аргументом метода
`replace()`	Методом из строки вызова удаляется подстрока и на ее место вставляется другой текст. Первым аргументом метода указывается индекс начала удаляемой подстроки, вторым — индекс первого после удаляемой подстроки символа. Третий аргумент метода — текст, вставляемый вместо удаленной подстроки

Некоторые из этих методов аналогичны тем, что рассматривались для класса `String`. Примеры использования нескольких из перечисленных методов приведены в листинге 8.8.

Листинг 8.8. Работа с объектами класса StringBuffer

```
class StringBufferDemo{
public static void main(String args[]){
// Базовая строка - объект класса StringBuffer:
StringBuffer str=new StringBuffer("Мы программируем на C++");
// Длина строки:
System.out.println(str.length());
// Размер строки (максимальная длина в символах):
System.out.println(str.capacity());
// Вставка подстроки:
str.insert(20,"Java и ");
// Вывод строки на экран:
System.out.println(str);
// Замена подстроки:
str.replace(27,30,"Pascal");
// Вывод подстроки на экран:
System.out.println(str);
// Инверсия строки:
str.reverse();
// Вывод строки на экран:
System.out.println(str);}
}
```

В программе командой `StringBuffer str=new StringBuffer("Мы программируем на C++")` создается объект класса `StringBuffer` с текстовым значением `Мы программируем на C++`, и ссылка на этот объект записывается в переменную `str`. Командой `System.out.println(str.length())` длина (в символах) данной текстовой строки выводится на экран. Размер (в символах) для того же тестового объекта отображается командой `System.out.println(str.capacity())`. Несложно проверить, что

в строке Мы программируем на C++ всего 23 символа. Размер объекта на 16 символов больше и составляет 39 символов. Причина весьма проста — при создании объекта класса StringBuffer на основе текстового значения по умолчанию выделяется 16 дополнительных позиций для внесения изменений в строку в последующем.

Командой str.insert(20,"Java и ") в строку str, начиная с 20-го индекса (это 21-й символ), выполняется вставка текста "Java и ". В результате строка получает новое значение:

```
Мы программируем на Java и C++
```

Замена подстроки в строке str реализуется с помощью команды str.replace(27,30,"Pascal"). В частности, удаляются символы с индексами с 27-го по 29-й включительно (текст C++), и вместо этой подстроки вставляется слово Pascal. В результате значение строки str становится равным:

```
Мы программируем на Java и Pascal
```

Наконец, после выполнения команды str.reverse() строка str инвертируется — меняется порядок следования символов в строке. Результат выполнения программы таков:

```
23
39
Мы программируем на Java и C++
Мы программируем на Java и Pascal
lacsaP и avaJ ан меуриммаргорп ыМ
```

Обращаем внимание читателя на разницу между длиной текстовой строки, реализованной объектом StringBuffer, и размером этой строки. Длина строки определяется фактическим количеством символов в строке, а размер — объемом памяти (в символах), выделенным для хранения значения объекта. Обычно занята не вся память и часть позиций «вакантна». Этим объясняется необходимость в использовании двух методов: length() и capacity(). При работе с объектами класса String такой проблемы не возникает. Там длина строки совпадает с размером.

Аргументы командной строки

Ведь я так высоко не ставлю слова,
Чтоб думать, что оно всему основа.

И. Гёте. Фауст

Аргументы командной строки — это параметры, которые передаются программе при ее выполнении. При запуске консольных программ аргументы командной строки указываются через пробел после имени запускаемого на выполнение файла.

В Java аргументы командной строки автоматически преобразуются в текстовый формат и передаются в виде текстового массива (элементы массива — текстовые представления параметров командной строки) в метод `main()`. В листинге 8.9 приведен пример простой программы, с помощью которой аргументы командной строки построчно выводятся на экран.

Листинг 8.9. Аргументы командной строки

```
class CommandLine{
public static void main(String args[]){
for(int i=0;i<args.length;i++)
System.out.println("args["+i+"]: "+args[i]);
}}
```

Массив аргументов командной строки — это массив `args`, указанный аргументом метода `main()`. Индексная переменная `i` получает значения от 0 до индекса `args.length-1` включительно (напомним, что свойство `length` определяет длину массива, а индексация массивов в Java начинается с нуля). Каждый элемент массива `args` — это текст (объект класса `String`). Он может быть выведен на экран, что и делается командой `System.out.println("args["+i+"]: "+args[i])` в рамках цикла. Например:

```
java CommandLine это аргументы для программы: 100 и -1
```

Если бы программа запускалась такой строкой, то результатом выполнения программы было бы следующее:

```
args[0]=это
args[1]=аргументы
args[2]=для
args[3]=программы:
args[4]=100
args[5]=и
args[6]=-1
```

В случае если аргументы командной строки должны обрабатываться не как текст, а, например, как числа, то после их считывания в текстовый массив, являющийся аргументом метода `main()`, выполняется преобразование в нужный формат, для чего вызываются встроенные методы класса `String` или разрабатываются собственные.

Резюме

1. Для работы с текстом в Java предусмотрены классы `String` и `StringBuffer`.
2. Принципиальная разница между строками, реализованными в виде объектов классов `String` и `StringBuffer`, состоит в том, что в первом случае созданная строка изменена быть не может, во втором — может. Под изменением строки в данном случае подразумевается изменение объекта, через который

реализована строка. При реализации строки объектом класса `String` для изменения строки создается новый объект, и ссылка на него присваивается соответствующей объектной переменной. Поскольку при реализации строк объектами класса `StringBuffer` предусмотрено автоматическое выделение дополнительного буфера для записи текста, в объекты этого класса можно вносить изменения.

3. Для работы со строками (объектами классов `String` и `StringBuffer`) в Java имеются специальные встроенные функции, которые позволяют выполнять все базовые операции со строками.
4. При преобразовании объектов в тип `String` (например, если объект передается аргументом методу `println()`) автоматически вызывается метод `toString()`. Путем переопределения этого метода для класса можно создать простой и эффективный механизм вывода на консоль информации об объектах класса.
5. Аргументы командной строки передаются в виде текстового массива в метод `main()`. Элементы массива, являющегося аргументом метода `main()`, — это текстовые представления параметров командной строки.

Глава 9. Обработка исключительных ситуаций

Это безобразие так оставлять нельзя!

Из к/ф «Карнавальная ночь»

Программы пишут для того, чтобы они работали, работали быстро и, самое главное, правильно. К сожалению, программы не всегда работают правильно. Причем эта проблема не сводится к уровню подготовки программиста. Другими словами, бывают ситуации, причем нередко, когда принципиально невозможно или практически затруднительно обеспечить безошибочную работу программы. В таких случаях желательно хотя бы свести к минимуму негативные последствия от возникшей ошибки.

Самая большая неприятность при возникновении ошибочной ситуации состоит в том, что обычно это приводит к экстренному завершению программы. Во многих языках программирования, в том числе и в Java, предусмотрены механизмы, позволяющие «сохранить лицо» даже в достаточно сложных ситуациях. Об этих механизмах и идет речь в данной главе.

Исключительные ситуации

Если на клетке слона написано «буйвол»,
не верь глазам своим.

Козьма Прутков

Исключительная ситуация — это ошибка, которая возникает в результате выполнения программы. Исключение в Java — это объект, который описывает исключительную ситуацию (ошибку).

При возникновении исключительной ситуации в процессе выполнения программы автоматически создается объект, описывающий эту исключительную ситуацию. Этот объект передается для обработки методу, в котором возникла исключительная ситуация. Говорят, что исключение выбрасывается в метод. По получении объекта исключения метод может обработать его или передать для обработки дальше (куда именно — другой вопрос).

Ранее было сказано, что исключения (объекты, описывающие исключительные ситуации) генерируются автоматически, однако их также можно генерировать «вручную», то есть специальными программными методами. На первый взгляд, такая возможность кажется лишней и ненужной, но это не так. Дальше мы увидим, что механизм обработки исключительных ситуаций, в том числе искусственное генерирование исключений, нередко позволяет сделать программный код более компактным и, если хотите, элегантным, значительно упрощая решение сложных, на первый взгляд, задач.

Для того чтобы метод мог обработать исключительную ситуацию, необходимо предусмотреть программный код обработки этой ситуации — на случай ее возникновения. Во-первых, нужно выделить фрагмент кода, который должен контролироваться на предмет генерирования исключительной ситуации. Во-вторых, необходимо создать программный код, непосредственно обрабатывающий исключительную ситуацию, то есть код, который выполняется в случае возникновения исключительной ситуации.

В Java для обработки исключительных ситуаций используется блок `try-catch-finally`. В блок `try` помещается программный код, который отслеживается на случай, если возникнет исключительная ситуация. Если исключительная ситуация возникает, то управление передается блоку `catch`. Программный код в этом блоке выполняется, только если возникает исключительная ситуация, причем не любая, а определенного типа. Аргумент, определяющий, какого типа исключительные ситуации обрабатываются в блоке `catch`, указывается после ключевого слова `catch` в круглых скобках, то есть в том же формате, что и аргумент метода.

Поскольку в блоке `try` могут возникать исключения разных типов, для каждого из них можно предусмотреть свой блок `catch`. Если блоков `catch` несколько, при возникновении исключительной ситуации они перебираются последовательно до совпадения типа исключительной ситуации с аргументом блока `catch`.

После блоков `try` и `catch` можно указать блок `finally` с кодом, который выполняется в любом случае вне зависимости от того, возникла исключительная ситуация или нет.

Общая схема использования блока `try-catch-finally` для обработки исключительных ситуаций выглядит так:

```
try{
// код, который генерирует исключение
}
catch(Тип_исключения_1 объект){
// код для обработки исключения
}
catch(Тип_исключения_2 объект){
// код для обработки исключения
}
...
```

```
finally{
// код, который выполняется обязательно
}
```

Если при исполнении программного кода в блоке try{} ошибок не возникает, после выполнения этого блока выполняется блок finally (если он имеется), затем управление передается следующей после конструкции try-catch-finally команде.

При возникновении ошибки в процессе выполнения кода в блоке try выполнение кода в этом блоке останавливается и начинается поиск подходящего блока catch. Если подходящий блок найден, выполняется его программный код, после чего выполняется код блока finally (при наличии такого). На этом все — далее выполняется код, следующий после блока try-catch-finally.

Может случиться, что в блоке try возникла ошибка, но подходящего блока catch для ее обработки нет. В этом случае исключение выбрасывается из метода и должно быть обработано внешним к методу программным кодом. Согласно правилам языка Java, исключения, которые не обрабатываются в методе и выбрасываются из метода, указываются в сигнатуре метода после ключевого слова throws. То есть указываются классы выбрасываемых из метода исключений. Правда, далеко не все классы выбрасываемых исключений нужно указывать — только так называемые неконтролируемые исключения. Мы рассмотрим их позже.

Если возникает ошибка, обработка которой в программе не предусмотрена, используется обработчик исключительной ситуации по умолчанию. Самое трагическое последствие вызова обработчика по умолчанию состоит в том, что программа завершает работу.

Есть еще одно ключевое слово, которое достаточно часто используется при обработке исключительных ситуаций, а точнее, при генерировании исключительной ситуации. Это ключевое слово throw.

Классы исключений

Человек редко ошибается дважды.
Обычно раза три или больше.

Дж. Барлоу

В Java существует целая иерархия классов, предназначенных для обработки исключительных ситуаций. В вершине этой иерархии находится суперкласс Throwable. У этого суперкласса есть два подкласса: Exception и Error. К классу Error относятся «катастрофические» ошибки, которые невозможно обработать в программе, например переполнение стека памяти. У класса Exception есть подкласс RuntimeException. К классу RuntimeException относятся ошибки времени выполнения программы, которые перехватываются программами пользователя.

Исключения для класса RuntimeException определяются автоматически. К ним относятся, например, деление на ноль, выход за пределы массива (недопустимая индексация массива).

В листинге 9.1 приведен пример программы, в которой происходит обработка исключительной ситуации, заключающейся в делении на ноль.

Листинг 9.1. Обработка ошибки деления на ноль

```
class ExceptionDemo{
public static void main(String args[]){
int a,b;
// Блок контроля исключительной ситуации:
try{
b=0;
// Деление на ноль:
a=100/b;
}catch(ArithmeticException e){
// Обработка исключительной ситуации:
System.out.println("Деление на ноль!");
}
System.out.println("Выполнение программы продолжено!");}
}
```

Что касается алгоритма, реализованного в программе, то он прост и непритязателен. В главном методе объявляются две целочисленные переменные a и b. Переменной b присваивается нулевое значение, а переменной a — некое значение командой a=100/b. Другими словами, без всяких обиняков выполняется деление на ноль! Поэтому нет никаких сомнений в том, что при выполнении этого кода возникнет ошибка деления на ноль. Если не предусмотреть ее обработки, на команде a=100/b работа программы, фактически, прекратится, поскольку будет вызван обработчик ошибки по умолчанию, который и довершит начатый при делении на ноль декаданс.

Ошибка деления на ноль относится к классу ArithmeticException, который является подклассом класса RuntimeException. Для отслеживания этой ошибки код, который ее вызывает, заключается в блок try, а для обработки ошибки после блока try размещается блок catch. Аргументом в блок catch передается объект e класса ArithmeticException (объект исключения). В данном случае при обработке ошибки напрямую объект исключения e не используется, но в принципе такая ситуация возможна и часто встречается.

Код, выполняемый при обработке исключительной ситуации деления на ноль, состоит всего из одной команды:

```
System.out.println("Деление на ноль!")
```

То есть при делении на ноль на экран выводится сообщение Деление на ноль!. Здесь важно другое — работа программы при этом не завершается. После

выполнения блока `catch` выполняется следующая после этого блока команда `System.out.println("Выполнение программы продолжено!")`. В результате мы получаем два сообщения:

```
Деление на ноль!
Выполнение программы продолжено!
```

Еще раз отметим, что в данном случае важно то, что после попытки деления на ноль программа продолжает работу.

Другой пример обработки исключительной ситуации деления на ноль представлен в листинге 9.2.

Листинг 9.2. Еще одно деление на ноль

```
// Импорт класса Random:
import java.util.Random;
class ArithExcepionDemo{
public static void main(String args[]){
int a=0,b=0,c=0;
// Объект для генерирования случайных чисел:
Random r=new Random();
for(int i=0;i<32000;i++){
try{
b=r.nextInt(200);
c=r.nextInt(100);
// Возможно деление на ноль:
a=10000/b/c;
}catch(ArithmeticException e){
// Обработка ошибки:
System.out.println("Деление на ноль!");
a=0;}
System.out.println("a="+a);}
}
}
```

Этот пример более реалистичен, хотя и несколько запутан. Здесь имеет место генерирование случайных чисел. Для этого создается объект класса `Random`. В свою очередь, чтобы класс стал доступен, необходимо его импортировать с помощью команды `import java.util.Random` — класс принадлежит пакету `java.util`.

В главном методе создаются три целочисленные переменные `a`, `b` и `c` с нулевыми начальными значениями. Командой `Random r=new Random()` создается объект `r` класса `Random`. Для генерирования целого числа служит метод `nextInt()`, который вызывается из объекта `r`. Аргументом метода указывается верхняя граница диапазона генерируемых чисел. Нижняя граница диапазона генерируемых чисел равна нулю.

Далее запускается цикл с достаточно большим количеством итераций. В рамках каждого цикла последовательно выполняются команды `b=r.nextInt(200)`, `c=r.nextInt(100)` и `a=10000/b/c`. Поскольку переменные `b` и `c` получают случайные значения, в том числе это может быть ноль, то гипотетически при выполнении

команды a=10000/b/c возможно деление на ноль. Поэтому соответствующие команды заключены в блок try.

В блоке catch{}, предназначенном для обработки исключительной ситуации деления на ноль, выполняются команды System.out.println("Деление на ноль!") и a=0. В результате при попытке деления на ноль выводится соответствующее сообщение, а переменная a получает нулевое значение. Работа цикла при этом продолжается. В частности, значение переменной a выводится на экран.

Результат выполнения программы, в силу очевидных причин, полностью привести невозможно, но один из фрагментов мог бы выглядеть так:

```
...
a=1
Деление на ноль!
a=0
a=18
a=2
...
```

Ранее упоминалось, что передаваемый в блок catch аргумент (объект исключения) может применяться непосредственно при обработке ошибки. Нередко используется информация об ошибке, заложенная в объект исключительной ситуации.

Описание исключительной ситуации

Ваше высочество, здесь вам никто ничего не скажет.

Из к/ф «Приключения принца Флоризеля»

В классе Throwable переопределяется метод toString(), который, как известно, определен в общем суперклассе Object, причем переопределяется он так, что в качестве результата возвращает строку, описывающую соответствующую ошибку. Напомним, что метод toString() вызывается автоматически, например, при передаче объекта исключения методу println() в качестве аргумента. Соответствующий пример приведен в листинге 9.3.

Листинг 9.3. Описание ошибки

```
class MoreExceptionDemo{
public static void main(String args[]){
int a,b;
try{
b=0;
// Деление на ноль:
a=100/b;
```

продолжение

Листинг 9.3 *(продолжение)*

```
}catch(ArithmeticException e){
// При обработке ошибки использован объект исключения:
System.out.println("Ошибка: "+e);}
System.out.println("Выполнение программы продолжено!");}
}
```

По сравнению с рассмотренным ранее особенность этого примера состоит в том, что в команде `System.out.println("Ошибка: "+e)` в качестве аргумента методу `println()` передается объект исключительной ситуации `e`. Результат выполнения программы в этом случае имеет такой вид:

```
Ошибка: java.lang.ArithmeticException: / by zero
Выполнение программы продолжено!
```

В первой текстовой строке, выведенной на экран, текст после слова `Ошибка:` появился в результате преобразования объекта исключения `e` в текстовый формат.

Множественный блок catch{}

Если сразу не разберешь,
Плох он или хорош...

В. Высоцкий

Как отмечалось в начале главы, для каждого типа исключений можно предусмотреть свой блок `catch` для обработки. Блоки размещаются один за другим и им передаются разные аргументы (объекты исключений разных классов) в соответствии с типом обрабатываемой исключительной ситуации. В листинге 9.4 приведен пример программы, в которой помимо ошибки деления на ноль обрабатывается также и ошибка неверной индексации массива.

Листинг 9.4. Несколько блоков catch

```
import java.util.Random;
class MultiCatchDemo{
public static void main(String args[]){
Random r=new Random();
int MyArray[]={0,2};
int a,b;
for(int i=1;i<10;i++){
try{
a=r.nextInt(3);
b=10/MyArray[a];
System.out.println(b);
}catch(ArithmeticException e){
System.out.println("Деление на ноль!");}
catch(ArrayIndexOutOfBoundsException e){
System.out.println("Выход за границы массива!");}
```

```
finally{System.out.println("**************");}}
System.out.println("Цикл for завершен!");}
}
```

Как и в одном из предыдущих примеров, здесь используется генератор случайных чисел, для чего командой `import java.util.Random` импортируется класс `Random`, а с помощью команды `Random r=new Random()` создается объект `r` этого класса для генерирования случайных чисел. Кроме того, создается целочисленный массив `MyArray` из двух элементов (значения 0 и 2), также описываются две целочисленные переменные `a` и `b`.

В цикле командой `a=r.nextInt(3)` присваивается значение переменной `a` — случайное целое число в диапазоне от 0 до 2 включительно. Переменная `a`, другими словами, может принимать значения 0, 1 и 2. Далее командой `b=10/MyArray[a]` присваивается значение переменной `b`. При выполнении этой команды могут возникать неприятности двух видов. Во-первых, если значение переменой `a` равно 0, то выполняется деление на ноль, поскольку элемент массива `MyArray[0]` равен нулю. Во-вторых, если значение переменой `a` равно 2, то имеет место ошибка выхода за границы массива, поскольку элемента `MyArray[2]` не существует. Если же значение переменной `a` равно 1, то значение переменной `b` вычисляется как 5 и выводится на экран командой `System.out.println(b)`.

Для обработки ошибки деления на ноль используется блок `catch` с аргументом класса `ArithmeticException`. В этом случае выводится сообщение о том, что произошла попытка деления на ноль.

Исключительная ситуация, связанная с неправильной индексацией элементов массива, описывается исключением класса `ArrayIndexOutOfBoundsException`. Аргумент этого класса передается во второй блок `catch`. Обработка этой ошибки сводится к тому, что выводится сообщение о выходе за границы массива.

Наконец, блок `finally` содержит команду вывода разделительной «звездной линии», которая отображается независимо от того, произошла какая-либо ошибка или нет. Результат выполнения программы мог бы выглядеть следующим образом:

```
Выход за границы массива!
**************
Деление на ноль!
**************
Выход за границы массива!
**************
5
**************
Деление на ноль!
**************
5
**************
Выход за границы массива!
```

```
**************
5
**************
Выход за границы массива!
**************
Цикл for завершен!
```

В конце работы программы выводится сообщение о том, что работа цикла завершена.

Вложенные блоки try

До определенного момента никто не знает, на что способен.

П. Сирус

Один блок try может размещаться внутри другого блока try. В этом случае, если во внутреннем блоке try возникает ошибка и этот блок try не содержит блока catch для ее обработки, исключение выбрасывается во внешний блок try и начинается последовательный просмотр его блоков catch на предмет обработки возникшей ошибки.

Может сложиться и более нетривиальная ситуация, например, когда метод, который вызывается в блоке try, сам содержит блок try. Если в блоке try метода возникает ошибка, не обрабатываемая методом, ее перехватывает внешний блок try, в котором вызывается метод.

Вообще же общий принцип обработки ситуаций при сложной схеме включения блоков try состоит в том, что при входе в очередной блок try контексты обрабатываемых этим блоком исключений записываются в стек. При возникновении ошибки этот стек начинает «раскручиваться» — контексты исключений просматриваются в обратном порядке (то есть последний занесенный в стек контекст исключения просматривается первым).

В листинге 9.5 приведен пример использования вложенных блоков try.

Листинг 9.5. Вложенные блоки try

```
import java.util.Random;
class NestTryDemo{
public static void main(String args[]){
Random r=new Random();
int a,b;
int c[]={-1,1};
for(int i=1;i<10;i++){
try{
a=r.nextInt(3);        // значение 0,1 или 2
b=100/a;               // возможно деление на ноль
System.out.println("b="+b);
```

```
    try{
    if(a==1) a=a/(a-1);    // деление на ноль
    else c[a]=200;         // выход за границы массива
    }catch(ArrayIndexOutOfBoundsException e){
     System.out.println("Выход за границы массива: "+e);}
}catch(ArithmeticException e){
System.out.println("Деление на ноль: "+e);}
System.out.println("******************************");}
}}
```

Как и ранее, в программе генерируются случайные числа (с помощью объекта `r` класса `Random`), объявляются две целочисленные переменные `a` и `b`, а также целочисленный массив `c`, состоящий всего из двух элементов (со значениями –1 и 1).

В цикле внешнего блока `try` последовательное выполнение команд `a=r.nextInt(3)` и `b=100/a` может закончиться генерированием исключения, поскольку среди возможных значений переменной `a` есть и нулевое, что, в свою очередь, означает ошибку деления на ноль. На этот случай предусмотрен блок `catch` внешнего блока `try`. В случае ошибки выполняется команда:

```
System.out.println("Деление на ноль: "+e)
```

Здесь объект `e` класса `ArithmeticException` является аргументом блока `catch`.

Если в указанном месте программы ошибка деления на ноль не возникает, значение переменной `b` выводится на экран (эта переменная может принимать всего два значения: 100 при значении переменной `a` равном 1 и 50 при значении переменной `a` равном 2), после чего выполняется серия команд, заключенных во внутренний блок `try`. Сразу отметим, что этот блок обрабатывает только исключение, связанное с выходом за границы массива (объект класса `ArrayIndexOutOfBoundsException`). В случае возникновения соответствующей ошибки выполняется команда:

```
System.out.println("Выход за границы массива: "+e)
```

Что касается самого программного кода во внутреннем блоке `try`, то он может вызывать исключения двух типов. В частности, там с помощью условной инструкции проверяется условие равенства значения переменной `a` единице. Если условие соблюдается, то при выполнении команды `a=a/(a-1)` происходит ошибка деления на ноль. В противном случае (то есть если значение переменной `a` отлично от единицы) выполняется команда `c[a]=200`. Обращаем внимание, что внутренний блок `try` выполняется, только если значение переменной a равно 1 или 2, поскольку если значение этой переменной равно 0, еще раньше возникнет ошибка деления на ноль, которая перехватывается блоком `catch` внешнего блока `try`. Поэтому если во внутреннем блоке `try` значение переменной `a` отлично от единицы, это автоматически означает, что значение переменной `a` равно 2. В результате при выполнении команды `c[a]=200` возникает ошибка выхода за границы массива, поскольку в массиве `c` всего два элемента, а элемента `c[2]` там просто нет.

Таким образом, во внутреннем блоке try обрабатывается ошибка выхода за границы диапазона. Если во внутреннем блоке try возникает ошибка деления на ноль, она обрабатывается блоком catch внешнего блока try. Результат выполнения программы мог бы быть следующим:

```
b=50
Выход за границы массива: java.lang.ArrayIndexOutOfBoundsException: 2
*******************************
b=100
Деление на ноль: java.lang.ArithmeticException: / by zero
*******************************
b=50
Выход за границы массива: java.lang.ArrayIndexOutOfBoundsException: 2
*******************************
Деление на ноль: java.lang.ArithmeticException: / by zero
*******************************
b=50
Выход за границы массива: java.lang.ArrayIndexOutOfBoundsException: 2
*******************************
b=50
Выход за границы массива: java.lang.ArrayIndexOutOfBoundsException: 2
*******************************
b=100
Деление на ноль: java.lang.ArithmeticException: / by zero
*******************************
b=100
Деление на ноль: java.lang.ArithmeticException: / by zero
*******************************
Деление на ноль: java.lang.ArithmeticException: / by zero
*******************************
```

Другой пример вложенных блоков try приведен в листинге 9.6. На этот раз в блоке try вызывается метод, у которого есть собственный блок try.

Листинг 9.6. Метод с блоком try

```
import java.util.Random;
class MethWithTryDemo{
static void nesttry(int a){
int c[]={-1,1};
try{
    if(a==1) a=a/(a-1); // деление на ноль
    else c[a]=200;      // выход за границы массива
    }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Выход за границы массива: "+e);}
}
public static void main(String args[]){
Random r=new Random();
```

```
int a,b;
for(int i=1;i<10;i++){
try{
a=r.nextInt(3);     // значения 0, 1 или 2
b=100/a;            // возможно деление на ноль
System.out.println("b="+b);
nesttry(a);
}catch(ArithmeticException e){
System.out.println("Деление на ноль: "+e);}
System.out.println("******************************");}
}}
```

Фактически, это та же программа, что и в предыдущем примере, только код внутреннего блока try реализован в виде статического метода nesttry(), содержащего блок try. Там, где раньше был внутренний блок try, теперь вызывается метод nesttry(). Если возникает ошибка выхода за границы массива, она обрабатывается в самом методе, а ошибка деления на ноль обрабатывается во внешнем блоке try. Результат выполнения этой программы аналогичен предыдущему.

Искусственное генерирование исключений

Дурные примеры действуют сильнее хороших правил.

Дж. Локк

Как уже отмечалось, исключения можно генерировать «вручную», то есть создавать видимость ошибки там, где ее и в помине нет. Для генерирования исключения используется ключевое слово throw. Команда генерирования исключения имеет следующий синтаксис:

```
throw объект_исключения;
```

После инструкции throw необходимо указать объект исключения, то есть объект, описывающий создаваемую исключительную ситуацию. Причем предварительно этот объект нужно создать. Напомним, что объект исключения — это объект класса Throwable или его подкласса. Существует два способа создания объекта исключения. Во-первых, можно воспользоваться аргументом блока catch, во-вторых, можно создать новый объект с помощью оператора new. При этом прибегают к помощи конструктора класса соответствующего исключения. Все исключения времени выполнения программы (класса RuntimeException) имеют конструкторы без аргументов и с текстовым аргументом. В последнем случае текст, переданный конструктору при создании объекта, отображается затем при описании объекта, если последний приводится к текстовому формату (например, при передаче объекта методам print() и println()).

После выполнения оператора throw поток выполнения останавливается, и следующая команда не выполнятся. Вместо этого начинается поиск подходящего для обработки сгенерированного исключения блока catch. Если такой блок не

обнаруживается, используется обработчик по умолчанию. Пример программы с явным выбрасыванием исключения приведен в листинге 9.7.

Листинг 9.7. Явное выбрасывание исключения

```
class ThrowDemo{
static void demoproc(){
try{
// Создание объекта исключения:
NullPointerException ExObj=new NullPointerException("Ошибка!");
throw ExObj;  // выбрасывание исключения
}catch(NullPointerException e){
System.out.println("Перехват исключения в методе demoproc!");
throw e;                    // повторное выбрасывание исключения
}}
public static void main(String args[]){
try{
demoproc();
}catch(NullPointerException e){
System.out.println("Повторный перехват: "+e);}
System.out.println("Работа программы завершена!");}
}
```

Результат выполнения программы такой:

```
Перехват исключения в методе demoproc!
Повторный перехват: java.lang.NullPointerException: Ошибка!
Работа программы завершена!
```

В классе `ThrowDemo`, помимо главного метода программы `main()`, описывается метод `demoproc()`, в котором явно выбрасывается исключение. Для начала создается объект исключения командой:

```
NullPointerException ExObj= new NullPointerException("Ошибка!")
```

Точнее, это объект `ExObj` класса `NullPointerException` (ошибка операций с указателем). Имеется конструктор класса `NullPointerException` с текстовым аргументом `"Ошибка!"`. Этот текст впоследствии используется при выводе на экран описания возникшей ошибки. Командой `throw ExObj` производится выбрасывание исключения. Поскольку все это происходит в блоке `try`, то начинается поиск подходящего блока `catch` для обработки исключения. В данном случае блок `catch` всего один, и это именно тот блок, который нужен. В этом блоке выводится сообщение о том, что исключение перехвачено в методе `demoproc()`. Делается это командой

```
System.out.println("Перехват исключения в методе demoproc!")
```

Однако затем командой `throw e` снова выбрасывается исключение. Для того чтобы обработать это исключение, нужен внешний блок `try` с соответствующим блоком `catch` для обработки исключения. Поскольку в главном методе программы метод `demoproc()` вызывается в блоке `try` и для исключения класса `NullPointerException`

описан обработчик (выполняется команда System.out.println("Повторный перехват: "+e)), то выброшенное из метода demoproc() исключение перехватывается и обрабатывается.

Хочется обратить внимание читателя на два немаловажных момента. Во-первых, вместо явного создания в методе demoproc() объекта исключения ExObj можно было ограничиться анонимным объектом, объединив команды создания объекта исключения и его выбрасывания в одну команду вида:

```
throw new NullPointerException("Ошибка!")
```

Обычно так и поступают, поскольку это экономит место и время, а результат в принципе тот же. Во-вторых, сообщение программы Повторный перехват: java.lang.NullPointerException: Ошибка! возникает в результате обработки повторно выброшенного исключения вне метода demoproc(), в то время как объект исключения с текстовым параметром Ошибка! создавался в этом методе. Последовательность действий, которые приводят к такому результату, следующая.

При выбрасывании исключения в методе demoproc() объект исключения (то есть объект ExObj) передается аргументом в блок catch. Аргумент в этом блоке обозначен как e, но это формальное название аргумента. Такие же формальные названия для аргументов используются при описании методов. Реально в блок передается объект ExObj. Далее в блоке catch есть команда throw e, которая означает выброс исключения, переданного аргументом в блок catch. То есть это опять объект ExObj. Поскольку это второе исключение в методе demoproc() не обрабатывается, а передается во внешний блок try для перехвата и далее в соответствующий блок catch для обработки, то аргумент внешнего блока catch — это все тот же метод ExObj, который создавался конструктором с текстовым аргументом Ошибка!. Именно это описание ошибки и появляется на экране после передачи аргументом методу println() объекта исключения. Кроме этого описания автоматически отображается трасса стека ошибки (сообщение java.lang.NullPointerException:).

Выбрасывание исключений методами

Дело умных — предвидеть беду, пока она не пришла.

Питтак

Ранее отмечалось, что если метод выбрасывает или может выбросить исключение, которое в методе не обрабатывается, этот факт нужно отразить при описании метода. В сигнатуре метода после ключевого слова throws перечисляются классы исключений, которые может выбрасывать метод. Причина такой предупредительности состоит в том, что внешним методам нужно сообщить, к каким неприятностям следует быть готовым при вызове данного метода. Ранее мы не использовали ключевое слово throws, хотя некоторые методы и выбрасывали исключения. Дело в том, что перечисляются далеко не все выбрасываемые

методом исключения. В частности, не нужно явно указывать исключения класса Error, а также класса RuntimeException и его подклассов. Рассматривавшиеся ранее исключения как раз относились к подклассам RuntimeException.

Общий синтаксис описания метода, выбрасывающего исключения (которые не обрабатываются в методе), выглядит так:

```
тип_результата имя_метода(аргументы) throws исключение1,исключение2,...{
// тело метода
}
```

Если метод может выбрасывать несколько исключений, их классы перечисляются через запятую. Пример описания метода, выбрасывающего необрабатываемое исключение, приведен в листинге 9.8.

Листинг 9.8. Метод выбрасывает исключение

```
class ThrowsDemo{
// Описание метода:
static void throwOne() throws IllegalAccessException{
System.out.println("Ошибка в методе throwOne!");
// Выбрасывание исключения:
throw new IllegalAccessException("Большая ошибка!");}
public static void main(String args[]){
try{
throwOne(); // Метод выбрасывает исключение
}catch(IllegalAccessException e){ // Обработка исключения
System.out.println("Случилась неприятность: "+e);}
}}
```

В результате выполнения этой программы получаем два сообщения:

```
Ошибка в методе throwOne!
Случилась неприятность: java.lang.IllegalAccessException: Большая ошибка!
```

Программа достаточно простая и ее особенность лишь в том, что при описании метода throwOne() явно указано, что метод может выбрасывать исключение класса IllegalAccessException (ошибка доступа).

Методом throwOne() выводится на экран сообщение об ошибке, затем выбрасывается исключение командой:

```
throw new IllegalAccessException("Большая ошибка!")
```

В данном случае исключение — это анонимный объект класса IllegalAccessException, для создания которого использовался конструктор с текстовым аргументом. В методе это исключение не отслеживается и не обрабатывается, о чем и свидетельствует наличие в сигнатуре метода ключевого слова throws и названия класса исключения IllegalAccessException. Отметим, что если бы это исключение в методе обрабатывалось, необходимости указывать в сигнатуре метода ключевое слово throws (и класса исключения) не было бы.

В главном методе программы вызывается метод `throwOne()`, выбрасываемое методом исключение отслеживается и обрабатывается. Объект (анонимный) выброшенного методом исключения передается в блок `catch`.

Контролируемые и неконтролируемые исключения

Ситуация в стране не требует такого глубокого кризиса.

Г. Явлинский

Встроенные исключения в Java делятся на контролируемые и неконтролируемые. Фактически, перечислять выбрасываемые методом исключения в сигнатуре метода нужно только в том случае, если они неконтролируемые. В табл. 9.1 приведены некоторые контролируемые исключения в Java.

Таблица 9.1. Контролируемые исключения в Java

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка (например, деление на ноль)
`ArrayIndexOutOfBoundsException`	Индекс массива за пределами допустимых границ
`ArrayStoreException`	Присваивание элементу массива недопустимого значения
`ClassCastException`	Недопустимое приведение типов
`IllegalArgumentException`	Методу передан недопустимый аргумент
`IndexOutOfBoundsException`	Некоторый индекс находится за пределами допустимых для него границ
`NegativeArraySizeException`	Создание массива отрицательного размера
`NullPointerException`	Недопустимое использование нулевого указателя
`NumberFormatException`	Недопустимое преобразование текстовой строки в числовой формат
`StringIndexOutOfBoundsException`	Попытка индексирования вне пределов строки
`UnsupportedOperationException`	Недопустимая операция

Ряд неконтролируемых исключений в Java приведен в табл. 9.2.

Таблица 9.2. Неконтролируемые исключения в Java

Исключение	Описание
`ClassNotFoundException`	Класс не найден
`IllegalAccessException`	В доступе к классу отказано

продолжение

Таблица 9.2 *(продолжение)*

Исключение	Описание
InstantiationException	Попытка создать объект абстрактного класса или интерфейса
InterruptedException	Один поток прерван другим потоком
NoSuchFieldException	Поле не существует
NoSuchMethodException	Метод не существует

Еще раз отметим, что даже если метод не обрабатывает и выбрасывает контролируемое исключение, отражать этот факт в сигнатуре метода не нужно. Неконтролируемые исключения указываются в сигнатуре метода, если они в этом методе не обрабатываются.

Создание собственных исключений

Как же меня можно исключать?
Я за гараж родину продал!

Из к/ф «Гараж»

Классы встроенных исключений в Java описывают только наиболее общие ошибки, поэтому в некоторых случаях нужно или полезно описать собственное исключение. В Java такая возможность имеется.

Технически создание собственного исключения сводится к созданию подкласса класса Exception, который, в свою очередь, является подклассом класса Throwable. В создаваемом подклассе ничего не нужно реализовывать. Что касается класса Exception, то он не определяет собственных методов, а наследует их из класса Throwable. Некоторые методы класса Throwable представлены в табл. 9.3.

Таблица 9.3. Методы класса Throwable

Метод	Описание
fillInStackTrace()	Метод в качестве результата возвращает объект Throwable, который содержит полную трассу стека. Метод не имеет аргументов
getLocalizesMessage()	В качестве результата метод возвращает строку (объект класса String) с локализованным описанием исключения. Метод не имеет аргументов
getMessage()	Методом возвращается строка (объект класса String) с описанием исключения. Метод не имеет аргументов
printStackTrace()	Методом отображается трасса стека. Метод не имеет аргументов
toString()	Метод в качестве значения возвращает объект класса String, содержащий описание исключения. Метод не имеет аргументов

В листинге 9.9 приведен пример программы, в которой создается пользовательский класс для исключения.

Листинг 9.9. Программа с пользовательским классом исключения

```
// Класс исключения:
class MyException extends Exception{
private double min;
private double max;
private String error;
// Конструктор:
MyException(double x,double y,String str){
min=x;
max=y;
error=str;}
// Переопределение метода toString():
public String toString(){
return "Произошла ошибка ("+error+"): попадание в диапазон ["+min+","+max+"]";}
}
class MyExceptionDemo{
// Метод выбрасывает исключение пользовательского типа:
static double MyLog(double x) throws MyException{
if(x<0||x>1) return Math.log(x*(x-1));
else throw new MyException(0,1,"неверный аргумент");
}
public static void main(String args[]){
double x=-1.2,y=1.2,z=0.5;
try{
System.out.println("ln("+x+")="+MyLog(x));
System.out.println("ln("+y+")="+MyLog(y));
System.out.println("ln("+z+")="+MyLog(z));
}catch(MyException e){// Обработка исключения
System.out.println(e);}
}
}
```

Результат выполнения программы такой:

```
ln(-1.2)=0.9707789171582248
ln(1.2)=-1.427116355640146
Произошла ошибка (неверный аргумент): попадание в диапазон [0.0,1.0]
```

В программе описывается класс пользовательского исключения `MyException`, который наследует класс `Exception`. У класса есть три закрытых поля: поля `min` и `max` типа `double` и поле `error`, являющееся объектом класса `String`. Также в классе переопределяется метод `toString()`. В качестве результата методом

возвращается текстовая строка, в которой содержится информация всех трех полей класса.

В главном методе программы определяется статический метод `MyLog()` с аргументом типа `double`. В качестве результата метод возвращает значение типа `double`. Метод описан как способный выбрасывать исключение пользовательского класса `MyException`.

В теле метода для значения аргумента `x` вычисляется натуральный логарифм по формуле:

$$\ln(x \cdot (x-1)).$$

Если аргумент, переданный методу `MyLog()`, лежит вне пределов диапазона $[0,1]$, методом возвращается значение `Math.log(x*(x-1))` (в этой команде имеет место обращение к статическому методу вычисления натурального логарифма `log()`, описанному в классе `Math`). Если же аргумент метода `MyLog()` попадает в диапазон $[0,1]$, приведенное выражение вычислено быть не может, поскольку у натурального логарифма аргумент отрицательный. В этом случае методом `MyLog()` генерируется и выбрасывается исключение пользовательского типа `MyException`. Аргументами конструктору при этом передаются границы диапазона $[0,1]$ и описание ошибки (`неверный аргумент`).

В главном методе программы выполняется попытка вычислить значение методом `MyLog()` и вывести его на экран. При этом отслеживается возможность появления исключения класса `MyException`. В случае если соответствующая ошибка возникает, выполняется ее обработка, которая состоит в том, что на экран выводится описание объекта исключения. Этот объект передается методу `println()` в блоке `catch`, а наблюдаемый при этом результат отражает способ переопределения метода `toString()` в классе `MyException`.

Резюме

1. В Java при возникновении ошибочных ситуаций автоматически создаются объекты, которые их описывают, — исключения. Подобный объект (исключение) передается методу, вызвавшему ошибку, для обработки. Метод может обработать объект (исключение) или передать его дальше.
2. Исключения также можно генерировать специально — в некоторых случаях это имеет смысл делать. В таком случае используется инструкция `throw`, после которой указывается объект генерируемого исключения.
3. Для обработки исключений служит блок `try-catch-finally`. В блок `try` помещается отслеживаемый программный код, блоки `catch` содержат код обработки исключений (их может быть несколько), а необязательный блок `finally` содержит код, который должен выполняться в любом случае.
4. В Java для описания исключений есть иерархия классов, на вершине которой находится класс `Throwable`. Встроенные исключения делятся на контролируе-

мые и неконтролируемые. Если метод может сгенерировать неконтролируемое исключение и в самом методе обработка этого исключения не предусмотрена, в сигнатуре метода через ключевое слово `throws` необходимо указать класс данного исключения.

5. Помимо использования встроенных исключений, можно создавать собственные классы исключений. Для этого необходимо создать подкласс класса `Exception` (это подкласс класса `Throwable`).

Глава 10. Многопоточное программирование

Необычные случаи обычно повторяются.

К. Чапек

Язык Java поддерживает многопоточное программирование. Многопоточная программа содержит две или больше частей, которые могут выполняться одновременно. Каждая такая часть программы называется потоком.

С понятием многопоточности тесно связано понятие многозадачности. Многозадачность реализуется, как правило, либо на потоках, либо на процессах. Различие между потоками и процессами достаточно зыбкое. Обычно для процессов выделяется отдельная область памяти, которая доступна только для этих процессов. Это повышает безопасность, но снижает скорость выполнения программы. На процессах основана работа операционных систем.

При многопоточности обычно память по потокам не разбивается. Хотя такая ситуация может сказаться на стабильности программы, системные ресурсы используются экономнее и программа работает быстрее.

Обычно многопоточное программирование применяют для сведения к минимуму времени простоя системы, поскольку сразу несколько задач могут выполняться одновременно.

Поточная модель Java

Нужно, чтобы она тебя брала. Нужно, чтобы она тебя вела.
Но в то же время и не уводила!

Из к/ф «Карнавальная ночь»

При однопоточном программировании в бесконечном цикле выполняется один поток управления, который опрашивает единую очередь событий и принимает решение, какое действие выполнять следующим. Примером может быть процесс считывания информации из файла. После получения сигнала о готовности файла к считыванию управление передается соответствующему обработчику, и пока из этого обработчика не будет получен ответ, никаких новых действий

не предпринимается. В противовес этому при многопоточном программировании один поток может делать паузу, не прерывая выполнение других потоков. Именно такой подход реализован в Java.

Как и все в Java, поточная модель реализуется посредством иерархии классов, описывающих потоки. Основу этой иерархии составляют класс `Thread` и интерфейс `Runnable`. Для создания потока необходимо либо расширить класс `Thread`, либо реализовать интерфейс `Runnable`. При этом класс `Thread` инкапсулирует поток исполнения.

При запуске Java-программы начинает выполняться главный поток. Особенность главного потока состоит в том, что в нем порождаются все дочерние потоки. Главный поток отождествляется с программой. Программа начинается с выполнения главного потока и должна завершаться с завершением главного потока.

В отличие от дочерних потоков главный поток создается автоматически. Поэтому в предыдущих примерах никаких дополнительных действий для создания главного потока не применялось.

Главным потоком можно управлять. Делается это с помощью поточного объекта с использованием методов класса `Thread`. Некоторые из этих методов приведены в табл. 10.1.

Таблица 10.1. Методы класса Thread

Метод	Описание
`currentThread()`	Методом в качестве результата возвращается ссылка на поток, из которого вызывается метод
`getName()`	Метод в качестве результата возвращает имя потока (текстовую строку)
`getPriority()`	Метод в качестве результата возвращает приоритет потока (целое число)
`isAlive()`	Метод позволяет выяснить, используется поток или нет
`join()`	Методом дается команда ожидания завершения потока
`run()`	Метод определения точки входа в поток
`sleep()`	Метод для приостановки потока на определенный промежуток времени (аргумент метода, в миллисекундах)
`start()`	Метод для запуска потока путем вызова его метода `run()`

В частности, ссылку на поток получают с помощью метода `currentThread()`, который является членом класса `Thread` и имеет атрибуты `public` и `static`. Метод имеет сигнатуру `public static Thread currenThread()`. Пример использования метода `currentThread()` для получения доступа к главному потоку приведен в листинге 10.1.

Листинг 10.1. Главный поток программы

```
class CurrentThreadDemo{
public static void main(String args[]){
// Объектная переменная t класса Thread:
Thread t;
// Объектная переменная t ссылается на главный поток программы:
t=Thread.currentThread();
// Информация о потоке:
System.out.println("Активный поток: "+t);
// Потоку присвоено (изменено) имя:
t.setName("Самый главный поток");
// Информация о потоке:
System.out.println("После изменения имени: "+t);
try{
    for(int n=5;n>0;n--){
    System.out.println(n);
    // Приостановка потока:
    Thread.sleep(1000);}
}catch(InterruptedException e){ // Обработка исключения "прерывание потока"
System.out.println("Поток завершен!");}}
}
```

В главном методе программы командой `Thread t` объявляется объектная переменная `t` класса `Thread`. Значение этой переменной, то есть ссылка на поток, присваивается командой `t=Thread.currentThread()` (кстати, можно было две команды объединить в одну: `Thread t=Thread.currentThread()`). Поскольку метод `currentThread()` статический, его можно вызывать, не создавая объект, а указав имя класса, что, собственно, и было сделано. В качестве значения метод возвращает ссылку на тот поток, из которого метод вызывался. В данном случае это главный поток программы. В результате в объектную переменную `t` записывается ссылка на главный поток программы. Теперь, если нам понадобится обратиться к главному потоку, мы можем воспользоваться переменной `t`. Хочется обратить внимание читателя на уже упоминавшийся факт: главный поток создается автоматически. Он существует безотносительно того, объявляем мы переменную `t` или нет. Эта переменная нужна лишь для того, чтобы идентифицировать поток, так сказать, поймать его «за уши».

Командой `System.out.println("Активный поток: "+t)` на экран выводится информация о главном потоке. Объект `t`, переданный аргументом методу `println()`, приводится к текстовому формату (благодаря переопределенному для класса `Thread` методу `toString()`). В результате на экран выводится сообщение:

```
Активный поток: Thread[main,5,main]
```

Часть сообщения `Thread[main,5,main]` является результатом приведения объекта потока к текстовому формату. В квадратных скобках после ключевого слова `Thread` соответственно указываются: имя потока, приоритет и группа потока. Для главного потока по умолчанию именем является `main`, приоритет равен 5,

а поток относится к группе с именем main. Имя потока — это его уникальный текстовый идентификатор. Приоритет — целое число. Само значение приоритета особой важности не имеет, важно только, у какого потока оно больше. Приоритет определяет, какому потоку отдается предпочтение при выполнении программы, когда один поток прерывается другим. Поток с низким приоритетом может быть остановлен потоком с более высоким приоритетом. Все потоки разбиваются на группы. Приоритеты потоков сравниваются в пределах групп.

Командой t.setName("Самый главный поток") меняется имя главного потока. Новое имя потока "Самый главный поток" указывается аргументом метода setName(), который вызывается из объекта главного потока. Поэтому после выполнения команды System.out.println("После изменения имени: "+t) на экране появляется сообщение:

```
После изменения имени: Thread[Самый главный поток,5,main]
```

Далее в рамках цикла на экран в столбик выводятся цифры от 5 до 1. При этом использована команда Thread.sleep(1000) для приостановки потока в каждом цикле. В результате выполнения программы получаем:

```
Активный поток: Thread[main,5,main]
После изменения имени: Thread[Самый главный поток,5,main]
5
4
3
2
1
```

Причем цифровой ряд выводится на экран с заметной задержкой (порядка одной секунды).

Поскольку метод sleep() может выбрасывать исключение InterruptedException (прерывание потока) и это исключение неконтролируемое, то в программе предусмотрена его обработка. В противном случае пришлось бы отразить в сигнатуре метода main() тот факт, что он выбрасывает исключение класса InterruptedException.

Создание потока

Бабу Ягу со стороны приглашать не будем.
Воспитаем в своем коллективе!

Из к/ф «Карнавальная ночь»

Как уже отмечалось, для создания потока (кроме главного) следует либо расширить класс Thread, либо реализовать интерфейс Runnable. Сначала рассмотрим создание потока путем реализации интерфейса Runnable.

Создать поток можно на базе любого класса, который реализует интерфейс Runnable. При реализации интерфейса Runnable достаточно определить всего один

метод run(). Программный код этого метода — это тот код, который выполняется в рамках создаваемого потока. Говорят, что метод run() определяет точку входа в поток. Метод run() имеет следующую сигнатуру:

```
public void run()
```

Для начала выполнения потока вызывают метод start().

Общая последовательность действий при создании нового потока путем реализации интерфейса Runnable следующая.

1. Определяется класс, реализующий интерфейс Runnable. В этом классе определяется метод run().
2. В этом классе создается объект класса Thread. Конструктору класса передается два аргумента: объект класса, реализующего интерфейс Runnable, и текстовая строка — название потока.
3. Для запуска потока из объекта класса Thread вызывается метод start().

Другими словами, для того чтобы определить программный код, выполняемый в новом потоке, необходимо расширить интерфейс Runnable, причем указанный код потока — это, фактически, код метода run(), определяемого в классе, расширяющем интерфейс Runnable. Поток в Java — это объект класса Thread. Поэтому для создания потока необходимо создать объект этого класса. В то же время при создании потока необходимо указать код этого потока (то есть код соответствующего метода run()). Код потока определяется в классе, реализующем интерфейс Runnable. Объект этого класса передается аргументом конструктору класса Thread при создании объекта нового потока. Поскольку создание потока не означает его запуск, поток запускается с помощью метода start(), вызываемого из объекта потока (объект класса Thread).

Часто процесс создания нового потока реализуется по следующей схеме.

1. При расширении интерфейса Runnable в соответствующем классе (для удобства назовем его внешним) не только определяется метод run(), но и описывается поле — объект класса Thread.
2. Создание объекта класса Thread (объекта потока), ссылка на который присваивается полю Thread, выполняется в конструкторе внешнего класса. При создании этого объекта вызывается конструктор класса Thread, первым аргументом которому передается ссылка this. Таким образом, одновременно с созданием объекта внешнего класса создается и объект потока, причем объект потока создается на основе объекта внешнего класса.
3. В конструкторе внешнего класса после команды создания объекта потока (объекта класса Thread) из этого потока вызывается метод start(). Это приводит к запуску потока.
4. Для создания и запуска потока в главном методе программы создается объект описанного ранее класса, расширяющего интерфейс Runnable. Поскольку для запуска потока достаточно самого факта создания объекта, нередко этот создаваемый объект является анонимным, то есть ссылка на него ни в какие объектные переменные не записывается.

Пример создания потока на основе реализации интерфейса `Runnable` приведен в листинге 10.2.

Листинг 10.2. Создание потока реализацией интерфейса Runnable

```
// Класс, расширяющий интерфейс Runnable:
class NewThread implements Runnable{
// Поле - ссылка на объект потока:
Thread t;
// Конструктор класса:
NewThread(){
// Создание объекта потока:
t=new Thread(this,"Новый поток");
// Вывод сведений о потоке:
System.out.println("Дочерний поток: "+t);
t.start();        // Запуск потока
}
// Определение метода run():
public void run(){
try{
for(int i=5;i>0;i--){
System.out.println("Дочерний поток: "+i);
// Приостановка потока:
Thread.sleep(500);}
}
// Обработка исключительной ситуации прерывания потока
catch(InterruptedException e){
System.out.println("Прерывание дочернего потока!");}
System.out.println("Завершение дочернего потока!");
}
}
class ThreadDemo{
public static void main(String args[]){
// Создание анонимного объекта класса NewThread:
new NewThread();     // Создание нового потока
try{
for(int i=5;i>0;i--){
System.out.println("Главный поток: "+i*100);
// Приостановка главного потока:
Thread.sleep(1000);}
}
// Обработка исключительной ситуации прерывания главного потока:
catch(InterruptedException e){
System.out.println("Прерывание главного потока!");}
System.out.println("Завершение главного потока!");}
}
```

В программе создается класс NewThread, реализующий интерфейс Runnable. Полем этого класса описана переменная t класса Thread. В конструкторе класса NewThread командой t=new Thread(this,"Новый поток") полю t в качестве значения присваивается ссылка на создаваемый объект класса Thread. Объект создается на основе объекта класса NewThread, полем которого является объектная переменная t (первый аргумент конструктора — ключевое слово this), а имя потока задается вторым аргументом конструктора — в данном случае это строка "Новый поток". Командой System.out.println("Дочерний поток: "+t) выводятся сведения о созданном потоке, а командой t.start() поток запускается. Еще раз отметим, что все эти действия описаны в конструкторе класса NewThread, то есть выполняются они при создании объекта этого класса.

В классе NewThread описан метод run(). Этим методом с интервалом задержки в 0,5 секунды выводится сообщение "Дочерний поток: " и натуральное число от 5 до 1 с шагом дискретности 1. Для приостановки потока использована команда Thread.sleep(500). Кроме того, в методе run() обрабатывается исключение InterruptedException (прерывание потока) для выполняемого потока.

В главном методе программы в классе ThreadDemo командой new NewThread() создается анонимный объект класса NewThread, чем автоматически запускается дочерний поток. После этого в рамках главного потока с интервалом задержки в одну секунду выводится сообщение "Главный поток: " и число от 500 до 100 с интервалом дискретности 100. Как и в случае дочернего потока, приостановка главного потока осуществляется вызовом метода sleep(), также отслеживается и обрабатывается исключение InterruptedException. В результате выполнения программы получаем:

```
Дочерний поток: Thread[Новый поток,5,main]
Главный поток: 500
Дочерний поток: 5
Дочерний поток: 4
Дочерний поток: 3
Главный поток: 400
Дочерний поток: 2
Дочерний поток: 1
Главный поток: 300
Завершение дочернего потока!
Главный поток: 200
Главный поток: 100
Завершение главного потока!
```

Фактически, при выполнении программы накладываются друг на друга два процесса (потока): главным потоком сообщения выводятся с интервалом одна секунда, а дочерним потоком сообщения выводятся с интервалом 0,5 секунды. Поскольку количество выводимых в каждом из потоков сообщений одинаково, а интервал между сообщениями главного потока больше, чем интервал между сообщениями дочернего потока, первым заканчивается дочерний поток, а его сообщения появляются «кучнее».

Практически также создаются потоки наследованием класса Thread. Здесь уместно отметить, что класс Thread сам наследует интерфейс Runnable. Поэтому принцип создания потока остается неизменным, просто вместо непосредственной реализации в создаваемом классе интерфейса Runnable этот интерфейс реализуется опосредованно, путем расширения (наследования) класса Thread. Реализация метода run() в классе Thread не предполагает каких-либо действий. Выход из ситуации можно найти, расширив класс Thread путем создания подкласса. Как и в предыдущем случае с интерфейсом Runnable, в подклассе, создаваемом на основе класса Thread, необходимо описать (переопределить) метод run() и запустить его унаследованным из Thread методом start(). Правда, здесь есть одно отличие, которое, в принципе, упрощает ситуацию. Дело в том, что при создании объекта подкласса, расширяющего класс Thread, нет необходимости создавать объект класса Thread, как это было в предыдущем примере, когда в реализующем интерфейс Runnable классе определялось поле-объект класса Thread. Поток вызывается прямо из объекта подкласса.

В листинге 10.3 приведен пример создания нового потока путем расширения класса Thread.

Листинг 10.3. Создание потока расширением класса Thread

```
// Класс NewThread расширяет класс Thread:
class NewThread extends Thread{
// Конструктор класса:
NewThread(){
// Вызов конструктора класса Thread:
super("Новый поток");
// Вывод сведений о потоке:
System.out.println("Дочерний поток: "+this);
// Запуск потока на выполнение:
start();
}
// Переопределение метода run():
public void run(){
try{
for(int i=5;i>0;i--){
System.out.println("Дочерний поток: "+i);
// Приостановка потока:
Thread.sleep(500);}
}
// Обработка исключения прерывания потока:
catch(InterruptedException e){
System.out.println("Прерывание дочернего потока!");}
System.out.println("Завершение дочернего потока!");}
}
class ExtendsThreadDemo{
public static void main(String args[]){
new NewThread();
```

продолжение

Листинг 10.3 *(продолжение)*

```
try{
for(int i=5;i>0;i--){
System.out.println("Главный поток: "+i*100);
Thread.sleep(1000);}
}catch(InterruptedException e){
System.out.println("Прерывание главного потока!");}
System.out.println("Завершение главного потока!");
}}
```

В результате выполнения программы получаем следующее:

```
Дочерний поток: Thread[Новый поток,5,main]
Главный поток: 500
Дочерний поток: 5
Дочерний поток: 4
Дочерний поток: 3
Главный поток: 400
Дочерний поток: 2
Дочерний поток: 1
Главный поток: 300
Завершение дочернего потока!
Главный поток: 200
Главный поток: 100
Завершение главного потока!
```

Программа, фактически, такая же, как и в предыдущем случае. Однако создание потока реализовано по-иному. В классе `NewThread`, который наследует класс `Thread`, определяются конструктор и метод `run()`. В конструкторе командой `super("Новый поток")` вызывается конструктор класса `Thread` с аргументом — названием создаваемого потока. Вывод на экран информации о потоке осуществляется командой:

```
System.out.println("Дочерний поток: "+this)
```

Причем здесь в качестве ссылки на объект потока использована ссылка `this` на создаваемый объект. Запускается поток вызовом метода `start()` объекта. Во всем остальном программный код схож с кодом из рассмотренного ранее примера и, думается, особых комментариев не требует.

Создание нескольких потоков

Здесь сотни мелких ручейков
В могучую впадают реку.

С. Маевский

По той же схеме, что создание одного дочернего потока, реализуется создание нескольких потоков. В листинге 10.4 приведен пример программы, в которой создаются три дочерних потока.

Листинг 10.4. Создание нескольких дочерних потоков

```
// Импорт класса Date:
import java.util.Date;
// Класс NewThread наследует класс Thread:
class NewThread extends Thread{
// Параметры потока (название, время задержки, количество итераций):
private String name;
private int time;
private int count;
// Конструктор:
NewThread(String name,int time,int count){
super(name);
this.name=name;
System.out.print("Создан новый поток: "+name+". ");
// Отображение даты и времени:
System.out.println("Время: "+new Date()+".");
this.time=time;
this.count=count;
// Запуск потока:
start();}
// Переопределение метода run():
public void run(){
try{
for(int i=1;i<=count;i++){
System.out.print("Поток: "+name+". Сообщение "+i+" из "+count+". ");
// Отображение даты и времени:
System.out.println("Время: "+new Date()+".");
// Приостановка потока:
Thread.sleep(time);}
}catch(InterruptedException e){
System.out.println("Прерывание потока"+name);}
finally{
System.out.print("Поток \""+name+"\" работу завершил! ");
// Отображение даты и времени:
System.out.println("Время: "+new Date()+".");}
}}
class MultiThreadDemo{
// Исключение InterruptedException в методе main() не обрабатывается:
public static void main(String args[]) throws InterruptedException{
System.out.print("Начало работы! ");
// Отображение даты и времени:
System.out.println("Время: "+new Date()+".");
// Создание трех дочерних потоков:
new NewThread("Красный",5000,5);
new NewThread("Желтый",6000,4);
new NewThread("Зеленый",7000,3);
```

продолжение

Листинг 10.4 *(продолжение)*

```
// Приостановка главного потока:
Thread.sleep(30000);
System.out.print("Работа программы завершена! ");
// Отображение даты и времени:
System.out.println("Время: "+new Date()+".");
}}
```

Несмотря на значительно больший объем программного кода по сравнению с предыдущим примером, в данной программе принципиальных изменений практически нет. В основном речь идет о «косметических» мерах. В первую очередь следует отметить, что конструктор класса `NewThread`, наследующего класс `Thread`, видоизменен так, что принимает три аргумента: имя создаваемого потока и два целочисленных параметра. Первый параметр определяет величину задержки при выводе потоком сообщений, второй — количество таких сообщений. Эти значения (аргументы конструктора) записываются в закрытые поля класса `name`, `time` и `count` соответственно.

Основу метода `run()`, который переопределяется в классе `NewThread` и задает функциональность потока (то есть выполняемый в потоке код), составляет цикл. Количество итераций задается полем `count` (которое, в свою очередь, передается третьим аргументом конструктору). За каждую итерацию выводится сообщение с названием потока (параметр `name`), номером сообщения, ожидаемым количеством сообщений этого потока, а также текущими датой и временем. Для получения даты и времени используется анонимный объект класса `Date`. Класс `Date` импортируется командой `import java.util.Date` в заголовке программы, а безымянный объект создается командой `new Date()`. При передаче этого объекта аргументом функции `println()` выводятся системные дата и время.

После вывода сообщения на экран командой `Thread.sleep(time)` производится приостановка выполнения потока на время `time` (в миллисекундах) — параметр передается вторым аргументом конструктору класса `NewThread`.

В конце выполнения потока выводится соответствующее сообщение с именем потока и временем завершения работы. Обращаем внимание, что в этом сообщении имя потока заключается в двойные кавычки, а сам символ двойных кавычек вводится в текст с помощью предваряющей косой черты.

В главном методе программы в классе `MultiThreadDemo` выводится сообщение о начале работы с указанием времени начала. Затем следующими командами создаются три анонимных объекта класса `NewThread`, каждый из которых запускает дочерний поток:

```
new NewThread("Красный",5000,5);
new NewThread("Желтый",6000,4);
new NewThread("Зеленый",7000,3);
```

После этого командой `Thread.sleep(30000)` на 30 секунд приостанавливается выполнение главного потока, чтобы успели закончить работу дочерние потоки. Затем главным потоком выводится сообщение о завершении работы программы

(с указанием времени завершения). Результат выполнения программы может иметь следующий вид:

```
Начало работы! Время: Sat Sep 19 23:39:36 EEST 2009.
Создан новый поток: Красный. Время: Sat Sep 19 23:39:36 EEST 2009.
Создан новый поток: Желтый. Время: Sat Sep 19 23:39:36 EEST 2009.
Создан новый поток: Зеленый. Время: Sat Sep 19 23:39:36 EEST 2009.
Поток: Красный. Сообщение 1 из 5. Время: Sat Sep 19 23:39:36 EEST 2009.
Поток: Желтый. Сообщение 1 из 4. Время: Sat Sep 19 23:39:36 EEST 2009.
Поток: Зеленый. Сообщение 1 из 3. Время: Sat Sep 19 23:39:36 EEST 2009.
Поток: Красный. Сообщение 2 из 5. Время: Sat Sep 19 23:39:41 EEST 2009.
Поток: Желтый. Сообщение 2 из 4. Время: Sat Sep 19 23:39:42 EEST 2009.
Поток: Зеленый. Сообщение 2 из 3. Время: Sat Sep 19 23:39:43 EEST 2009.
Поток: Красный. Сообщение 3 из 5. Время: Sat Sep 19 23:39:46 EEST 2009.
Поток: Желтый. Сообщение 3 из 4. Время: Sat Sep 19 23:39:48 EEST 2009.
Поток: Зеленый. Сообщение 3 из 3. Время: Sat Sep 19 23:39:50 EEST 2009.
Поток: Красный. Сообщение 4 из 5. Время: Sat Sep 19 23:39:51 EEST 2009.
Поток: Желтый. Сообщение 4 из 4. Время: Sat Sep 19 23:39:54 EEST 2009.
Поток: Красный. Сообщение 5 из 5. Время: Sat Sep 19 23:39:56 EEST 2009.
Поток "Зеленый" работу завершил! Время: Sat Sep 19 23:39:57 EEST 2009.
Поток "Желтый" работу завершил! Время: Sat Sep 19 23:40:00 EEST 2009.
Поток "Красный" работу завершил! Время: Sat Sep 19 23:40:01 EEST 2009.
Работа программы завершена! Время: Sat Sep 19 23:40:06 EEST 2009.
```

Наличие момента времени, в который выводится сообщение, позволяет провести достаточно скрупулезный анализ особенностей «наложения» потоков друг на друга. Желающие могут заняться этим самостоятельно.

Еще одна особенность представленного кода состоит в том, что в методе `main()` не производится обработка исключения `InterruptedException`. Поэтому в сигнатуру метода вынесено сообщение о том, что метод может выбросить такое исключение (`throws InterruptedException`).

Синхронизация потоков

Возьмемся за руки, друзья!
Пусть видят все, что мы едины!

С. Маевский

Нередко при многопоточном программировании приходится решать проблему синхронизации потоков. Проблема эта обычно возникает, если разные потоки имеют доступ к одному и тому же ресурсу. Пояснить возникающие при этом сложности можно на следующем примере, не относящемся напрямую к программированию.

Предположим, имеется банковский счет, на который могут вноситься суммы и с которого могут сниматься суммы, причем выполняться могут сразу несколько

операций — это обычная ситуация, когда доступ к счету имеют несколько субъектов финансово-экономической деятельности. Процесс изменения состояния счета можно отождествить с потоком. Таким образом, может выполняться сразу несколько потоков.

Непосредственно процесс изменения состояния счета состоит из двух этапов. Сначала сумма, находящаяся на счету, считывается. Затем со считанным значением выполняется нужная операция, после чего новое значение вносится как новое состояние счета. Если в процесс изменения состояния счета между считыванием и записью суммы счета вклинится другой поток, последствия могут быть катастрофическими. Например, пусть есть счет в размере 10 000 рублей. Одним потоком сумма на счету увеличивается на 5000 рублей, а другим — уменьшается на 3000 рублей. Несложно понять, что новое значение счета должно быть равным 12 000 рублей. А теперь проанализируем такую ситуацию. Первым процессом считана сумма в 10 000 рублей. После этого, но до записи первым потоком нового значения, второй поток также считывает сумму на счету. Затем первый поток записывает новое значение счета, то есть 15 000 рублей. После этой оптимистичной процедуры второй поток также записывает свое значение, но это 7000 рублей, поскольку 10 000 – 3000 = 7000. Понятно, что для банка это хорошо, но никак не для обладателя счета. Другой пример: продажа железнодорожных билетов из разных касс. В этом случае из базы данных считывается информация о наличествующих свободных местах, и на одно из них выписывается билет (или не выписывается). Соответствующее место помечается как занятое (то, на которое продан билет). Понятно, что если подобные операции выполняются сразу несколькими потоками, возможны неприятности, поскольку на одни и те же места могут продаваться по несколько билетов, если при выписке билета другой поток успеет считать из базы данных старую информацию, в которой не отражены вносимые при покупке билета изменения. Поэтому при необходимости потоки синхронизируют, что делает невозможным ситуации, подобные описанным.

Существует два способа создания синхронизированного кода:

- создание синхронизированных методов;
- создание синхронизированных блоков.

В обоих случаях используется ключевое слово `synchronized`. Если создается синхронизированный метод, ключевое слово `synchronized` указывается в его сигнатуре. При вызове синхронизированного метода потоком другие потоки на этом методе блокируются — они не смогут его вызвать, пока работу с методом не завершит первый вызвавший его поток.

Можно синхронизировать объект в блоке команд. Для этого блок выделяется фигурными скобками, перед которыми указывается ключевое слово `synchronized`, а в скобках после этого слова — синхронизируемый объект. Пример программы с синхронизированным методом приведен в листинге 10.5.

Листинг 10.5. Синхронизация потоков

```
class MySource{
// Синхронизированный метод:
synchronized void showName(String msg1,String msg2,int time){
try{
// Приостановка потока, из которого вызван метод:
Thread.sleep(time);
// Вывод значения поля msg1:
System.out.print(" Фамилия: "+msg1);
// Еще одна приостановка потока:
Thread.sleep(2*time);
// Вывод значения поля msg2:
System.out.println(" Имя: "+msg2);
}catch(InterruptedException e){// Обработка исключения
System.out.println("Прерывание потока: "+e);}
}}
// Класс, реализующий интерфейс Runnable:
class MakeThread implements Runnable{
// Поле объекта потока:
Thread t;
// Поле-объект MySource:
MySource src;
// Текстовые поля:
String name;
String surname;
int time;
// Конструктор:
MakeThread(String s1,String s2,int time, MySource obj){
surname=s1;
name=s2;
src=obj;
this.time=time;
// Создание потока:
t=new Thread(this);
// Запуск потока:
t.start();}
// Определение метода run():
public void run(){
src.showName(surname,name,time);}
}
class SynchThreads{
public static void main(String args[]){
// Объект "ресурса":
MySource obj=new MySource();
```

продолжение

Листинг 10.5 *(продолжение)*

```
// Создание потоков:
MakeThread fellow1=new MakeThread("Иванов","Иван",1000,obj);
MakeThread fellow2=new MakeThread("Петров","Петр",450,obj);
MakeThread fellow3=new MakeThread("Сидоров","Сидор",1450,obj);
try{                        // Ожидать завершения потоков
fellow1.t.join();
fellow2.t.join();
fellow3.t.join();
}catch(InterruptedException e){ // Обработка исключения
System.out.println("Прерывание потока: "+e);}
}}
```

Идея, положенная в основу алгоритма программы, достаточно проста. В главном потоке создаются и запускаются три дочерних потока, и каждый из них с временной задержкой выводит два сообщения: одно с именем и другие с фамилией. Однако для вывода сообщений используется один и тот же объект класса `MySource`, а точнее, метод `showName` этого объекта. Таким образом, три потока в процессе своего выполнения в разное время обращаются к одному и тому же объекту, который играет роль общего ресурса. Этот объект создается в главном методе программы `main()` в классе `SynchThreads` с помощью команды `MySource obj=new MySource()`. Описание класса `MySource` можно найти в начале листинга 10.5. В этом классе описан всего один метод `showName()`, причем метод синхронизирован — об этом свидетельствует инструкция `synchronized` в сигнатуре метода. У метода три аргумента: два текстовых и один целочисленный. Текстовые аргументы определяют фамилию и имя виртуального пользователя, а третий числовой аргумент определяет значение задержки между выводимыми методом сообщениями. В частности, перед выводом первого сообщения задержка равна, в миллисекундах, значению третьего аргумента, а интервал между первым и вторым сообщениями в два раза больше. Для приостановки выполнения потока используется статический метод `sleep()`. Также в методе `showName()` обрабатывается исключение класса `InterruptedException` — «прерывание потока».

Потоки создаются с помощью класса `MakeThread`, реализующего интерфейс `Runnable`. У класса пять полей: поле `t` класса `Thread`, на котором реализуется поток, объектная ссылка `src` класса `MySource`, два текстовых поля `name` и `surname` и целочисленное поле `time`. Поле через объектную ссылку `src` ссылается на внешний объект класса `MySource`, посредством которого осуществляется вывод информации на экран. В поле `name` записывается имя, а в поле `surname` — фамилия виртуального пользователя. Целочисленный аргумент `time` содержит значение базовой задержки вывода сообщений. Конструктор класса `MakeThread` имеет четыре аргумента, которыми задаются значения полей класса `surname`, `name`, `time` и `src` соответственно. Командой `t=new Thread(this)` создается объект для потока, а командой `t.start()` поток запускается.

В методе run() командой src.showName(surname,name,time) из объекта src запускается метод showName(). Аргументами методу передаются значения полей объекта класса MakeThread, из которого запускается поток.

В главном методе программы, кроме объекта obj, создаются три объекта fellow1, fellow2 и fellow3 класса MakeThread (с разными аргументами). Затем в главном потоке с помощью метода join(), который вызывается из объекта-поля t каждого из трех объектов fellow1, fellow2 и fellow3, дается указание ожидать окончания выполнения каждого из трех потоков. В результате выполнения программы получаем:

```
Фамилия: Иванов Имя: Иван
Фамилия: Сидоров Имя: Сидор
Фамилия: Петров Имя: Петр
```

Таким образом, имеет место соответствие между фамилиями и именами виртуальных пользователей. Такое соответствие достигается благодаря синхронизации метода showName(). Убедиться в последнем просто — достаточно убрать ключевое слово synchronized из сигнатуры метода showName(). Результат выполнения программы после этого изменится радикально:

```
Фамилия: Петров Фамилия: Иванов Имя: Петр
Фамилия: Сидоров Имя: Иван
Имя: Сидор
```

Проблема в том, что каждый поток из-за задержки перед выводом сообщений какое-то время работает с объектом obj, через который выводятся сообщения. При отсутствии синхронизации в работу одного потока вклинивается другой поток, в результате сообщения появляются достаточно хаотично (хотя по строгой логической схеме, в соответствии с программным кодом).

В завершение отметим, что реализовать синхронизацию общего ресурса в данном случае можно было и по-другому. Например:

```
public void run(){
src.showName(surname,name,time);
}
```

Вместо подобной реализации метода run() в классе MakeThread можно было сделать следующее:

```
public void run(){
synchronized(src){
src.showName(surname,name,time);
}
}
```

В данном случае синхронизируемый код выделен в блок, а перед ним в скобках после ключевого слова synchronized указан синхронизируемый объект src.

Резюме

1. В Java поддерживается многопоточное программирование, когда несколько частей программы выполняются одновременно.
2. Поточная модель реализуется через иерархию классов, основу которой составляют класс `Thread` и интерфейс `Runnable`.
3. Для создания потока необходимо либо расширить класс `Thread`, либо реализовать интерфейс `Runnable`.
4. Запуск Java-программы заключается в выполнении главного потока. В главном потоке порождаются все дочерние потоки. Главный поток отождествляется с программой.
5. Для управления потоками используются специальные встроенные методы (в основном методы класса `Thread`). Метод `run()` содержит код потока, а метод `start()` служит для запуска потока.
6. В некоторых случаях потоки необходимо синхронизировать. Обычно это требуется для правильной последовательности операций с ресурсом, если потоки имеют доступ к одному ресурсу. Синхронизация выполняется с использованием ключевого слова `synchronized`. Пока с ресурсом работает синхронизированный метод, доступ других методов к этому ресурсу блокируется.

Глава 11. Система ввода-вывода

А с обратной стороны я вас попрошу сделать дверь.
Чтобы можно было войти и, когда надо, выйти.

Из к/ф «Человек с бульвара Капуцинов»

До этого мы только выводили информацию на экран, но никогда не считывали ее с клавиатуры. Дело в том, что в Java достаточно нетривиальная система ввода-вывода. И если с выводом информации на экран особых проблем не возникает, то ввод информации в программы (в частности, с клавиатуры) — задача не самая простая. Для реализации ввода приходится решать несколько вспомогательных задач, в том числе прибегать к созданию потоков и обрабатывать возможные исключительные ситуации.

Причина такого состояния дел с вводом и выводом в Java объясняется достаточно просто и прозаично. Язык Java создавался не для того, чтобы писать консольные программы. Писать консольную программу на Java — все равно, что съездить на танке на рынок, чтобы купить десяток-другой яиц. Танки предназначены для других целей, хотя с их помощью успешно можно решить и упомянутую задачу.

Ввод и вывод данных в Java реализуется через *потоки ввода-вывода* (stream)[1]. Выделяют два типа потоков: байтовые и символьные. На самом деле, в обоих случаях речь идет о байтах, но поскольку символ — это два байта, то при работе с символьными потоками байты обрабатываются, условно говоря, парами.

Для работы с символьными и байтовыми потоками используются специально разработанные для этого классы. Чтобы эти классы стали доступными в программе, необходимо подключить (импортировать) пакет `java.io`.

---

[1] Не путать с программными потоками (thread), рассматриваемыми в главе 10.

Байтовые и символьные потоки

Лучше обнаруживать свой ум в молчании,
нежели в разговорах.

А. Шопенгауэр

На вершине иерархии байтовых потоков находятся два абстрактных класса: `InputStream` и `OutputStream`. В этих классах определены методы `read()` и `write()`, предназначенные для чтения данных из потока и записи данных в поток соответственно.

Некоторые другие классы байтовых потоков перечислены в табл. 11.1.

Таблица 11.1. Классы байтовых потоков

Класс байтового потока	Описание
`InputStream`	Абстрактный класс, который описывает поток ввода
`OutputStream`	Абстрактный класс, который описывает поток вывода
`FilterInputStream`	Класс, который реализует абстрактный класс `InputStream`
`FilterOutputStream`	Класс, который реализует абстрактный класс `OutputStream`
`BufferedInputStream`	Класс буферизованного потока ввода
`BufferedOutputStream`	Класс буферизованного потока вывода
`ByteArrayInputStream`	Класс потока ввода для считывания из массива
`ByteArrayOutputStream`	Класс потока вывода для записи в массив
`FileInputStream`	Класс потока ввода для считывания из файла
`FileOutputStream`	Класс потока вывода для записи в файл
`DataInputStream`	Класс потока ввода с методами для считывания данных стандартных типов Java
`DataOutputStream`	Класс потока вывода с методами для записи данных стандартных типов Java
`PrintStream`	Класс потока вывода, который поддерживает методы `print()` и `println()`

Иерархия классов для символьных потоков ввода-вывода начинается с абстрактных классов `Reader` и `Writer`. В этих классах определены методы `read()` для считывания символьных данных из потока и `write()` для записи символьных данных в поток. Некоторые из классов для символьных потоков представлены и кратко описаны в табл. 11.2.

Таблица 11.2. Классы символьных потоков Java

Класс символьного потока	Описание
`Reader`	Абстрактный класс, который описывает поток ввода
`Writer`	Абстрактный класс, который описывает поток вывода

Класс символьного потока	Описание
FilterReader	Класс, который описывает отфильтрованный поток ввода
FilterWriter	Класс, который описывает отфильтрованный поток вывода
InputStreamReader	Класс потока ввода, который переводит байты в символы
OutputStreamWriter	Класс потока вывода, который переводит символы в байты
StringReader	Класс потока ввода для считывания из текстовой строки
StringWriter	Класс потока вывода для записи в текстовую строку
FileReader	Класс потока ввода для считывания из файла
FileWriter	Класс потока вывода для записи в файл
BufferedReader	Класс буферизованного потока ввода
BufferedWriter	Класс буферизованного потока вывода
PrintWriter	Класс потока вывода, который поддерживает методы print() и println()
CharArrayReader	Класс потока ввода для считывания из массива
CharArrarWriter	Класс потока вывода для записи в массив
LineNumberReader	Класс потока ввода для подсчета текстовых строк

Часть возможностей ввода-вывода может быть реализована посредством класса System. Класс System содержит три переменных потока: in, out и err. Эти поля имеют атрибуты public и static. В частности:

- Поле System.out — поток стандартного вывода. По умолчанию он связан с консолью. Поле System.out является объектом класса PrintStream.
- Поле System.in — это поток стандартного ввода. По умолчанию он связан с клавиатурой. Поле является объектом класса InputStream.
- Поле System.err — это стандартный поток ошибок. По умолчанию поток связан с консолью. Поле является объектом класса PrintStream.

Методы для работы с классами потоков ввода-вывода мы рассмотрим на конкретных примерах.

Консольный ввод с использованием объекта System.in

То, что кажется странным, редко остается необъяснимым.

Г. Лихтенберг

Консольный ввод в Java реализуется посредством считывания объекта System.in. При этом используется класс BufferedReader — класс буферизованного входного потока. Применение этого класса требует подключения (импорта) пакета java.io (командой import java.io.* в заголовке файла программы). Класс BufferedReader имеет конструктор с аргументом — объектом подкласса класса Reader. В свою

очередь, класс Reader — абстрактный класс, подклассом которого является класс InputSrtreamReader. Класс InputSrtreamReader предназначен для преобразования файлов в символы. Аргументом конструктора InputSrtreamReader указывается объект класса InputStream. Напомним, что к этому классу относится и объект System.in.

Далее приведен классический пример команды создания объекта буферизованного символьного потока ввода (объекта br):

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

Объект создается как относящийся к классу BufferedReader. При этом конструктору класса BufferedReader указан анонимный объект класса InputStreamReader. Этот анонимный объект создается конструктором с аргументом — объектом System.in.

Такая нетривиальная цепочка инструкций связана с тем, что класс InputStreamReader переводит байты в символы, но позволяет считывать при этом только один символ. Поэтому объект класса InputStreamReader «вкладывается» в класс BufferedReader, который позволяет считывать несколько символов. Технически это сводится к тому, что символы заносятся в буфер, откуда и считываются (отсюда название — буферизованный поток).

Для считывания символов служит метод read() класса BufferedReader. Метод имеет следующую сигнатуру:

```
int read() throws IOException
```

Таким образом, метод выбрасывает исключение IOException (ошибка ввода-вывода). Обращаем внимание на то, что в качестве результата метод возвращает число — код символа. Поэтому считанные с помощью этого метода значения необходимо через механизм явного приведения типов преобразовывать в тип char.

Для считывания текстовых строк используют метод readLine() класса BufferedReader. У метода следующая сигнатура:

```
String read() throws IOException
```

Методом также выбрасывается исключение IOException, а результатом является считанная из буфера текстовая строка. В листинге 11.1 приведен пример организации консольного ввода.

Листинг 11.1. Консольный ввод символов

```
// Подключение пакета:
import java.io.*;
class MySymbInput{
public static void main(String args[]) throws IOException{
// Символьная переменная для записи считываемого значения:
char symb;
// Буферизованный поток ввода:
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
// Вывод начального сообщения на экран:
```

```
System.out.println("Введите несколько символов и нажмите <Enter>:");
do{
// Считывание символа (с явным преобразованием типа):
symb=(char)br.read();
// Вывод считанного символа на экран:
System.out.println("Вы ввели: "+symb);
}while(symb!='Ы');
}}
```

Первой инструкцией `import java.io.*` подключается пакет `java.io`, в котором размещены классы ввода-вывода. Главный метод программы в классе `MySymbInput` объявлен как выбрасывающий исключение `IOException`. Это исключение выбрасывается использованным в методе `main()` методом `read()` и в классе `main()` оно не обрабатывается. Поэтому в сигнатуре метода `main()` есть инструкция `throws IOException`.

В методе `mian()` объявляется переменная `symb` типа `char`, в которую планируется записывать считываемые символьные значения. Объект `br` буферизованного символьного потока создается уже знакомой нам (см. ранее) командой:

```
BufferedReader br= new BufferedReader(new InputStreamReader(System.in))
```

Далее выводится сообщение с приглашением ввести несколько символов с клавиатуры и запускается цикл `do-while`. Тело этого цикла состоит из двух команд. Первой командой `symb=(char)br.read()` с клавиатуры считывается символ и записывается в переменную `symb`. Для считывания служит метод `read()`, вызываемый из объекта буферизованного потока `br`. Поскольку методом возвращается код символа, для преобразования этого кода в символ используем явное приведение типов — перед инструкцией считывания значения указываем в круглых скобках тип приведения — `(char)`.

Второй командой `System.out.println("Вы ввели: "+symb)` считанный символ выводится на экран. Цикл выполняется до тех пор, пока не будет считана буква `Ы`. Далее приведен результат выполнения программы, если с клавиатуры пользователем вводится фраза `Операция Ы и другие приключения Шурика` с последующим нажатием клавиши Enter:

```
Введите несколько символов и нажмите Enter:
Операция Ы и другие приключения Шурика
Вы ввели: О
Вы ввели: п
Вы ввели: е
Вы ввели: р
Вы ввели: а
Вы ввели: ц
Вы ввели: и
Вы ввели: я
Вы ввели:
Вы ввели: Ы
```

Фактически, на экран посимвольно выводится введенный пользователем текст (по одному символу в строке) до буквы ы включительно. Обращаем внимание, что если во веденной строке буквы ы нет, программа продолжит работу даже после нажатия клавиши Enter. При этом нажатие клавиши Enter воспринимается системой как ввод символа с кодом 10.

Пример консольного считывания текстовых строк приведен в листинге 11.2.

Листинг 11.2. Консольный ввод текстовых строк

```
import java.io.*;
class MyStringInput{
public static void main(String args[]) throws IOException{
String str="Ваш заказ: ";
String s;
int count=0;
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Что-то желаете?");
s=br.readLine();
while(!s.equalsIgnoreCase("Нет")){
count++;
str=str+"\n"+count+": "+s.toLowerCase();
System.out.println(str+"\n Еще что-то?");
s=br.readLine();}
System.out.println("Спасибо! Ваш заказ принят!");
}}
```

Программой моделируется процесс приема заказа от клиента в кафе. Начальная «приветственная» фраза выводится командой:

```
System.out.println("Что-то желаете?")
```

В ответ пользователь может ввести наименование заказа или слово Нет. Командой s=br.readLine() введенный пользователем текст считывается и заносится в качестве значения в переменную s класса String. При этом используется объект br буферизованного символьного потока ввода. Создается он, как и в предыдущем примере, командой

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in))
```

В инструкции while проверяется условие !s.equalsIgnoreCase("Нет"). Если пользователем введена фраза, отличная от Нет (без учета регистра букв), значение переменной count, которая нумерует позиции заказа, увеличивается на единицу и к строке str добавляется новый пункт заказа. В частности, командой str=str+"\n"+count+": "+s.toLowerCase() к уже существующему текстовому значению str добавляется символ-инструкция перехода к новой строке, номер заказа, а также сам заказ. Причем заказ записывается в нижнем регистре символов (строчными буквами), для чего из объекта s вызывается метод toLowerCase(). После этого командой System.out.println(str+"\n Еще что-то?") выводится очередной запрос

и командой s=br.readLine() считывается ввод пользователя. Так продолжается до тех пор, пока пользователь не введет слово Нет. В конце выполнения программы выводится фраза:

```
"Спасибо! Ваш заказ принят!"
```

Результат выполнения программы может иметь следующий вид (жирным шрифтом выделен ввод пользователя):

```
Что-то желаете?
Мороженое
Ваш заказ:
1: мороженое
 Еще что-то?
Лимонад
Ваш заказ:
1: мороженое
2: лимонад
 Еще что-то?
Булочка
Ваш заказ:
1: мороженое
2: лимонад
3: булочка
 Еще что-то?
Кофе со сливками
Ваш заказ:
1: мороженое
2: лимонад
3: булочка
4: кофе со сливками
 Еще что-то?
Нет
Спасибо! Ваш заказ принят!
```

Как и в предыдущих случаях, для работы с классами ввода-вывода необходимо подключить пакет java.io.

Консольный ввод с помощью класса Scanner

Самое смешное желание — это желание нравиться всем.

И. Гёте

Описанный способ реализации потока ввода достаточно громоздок и не очень удобен. Начиная с версии JDK 5, в Java появилась возможность реализовать консольный ввод-вывод намного проще — через класс Scanner. Для работы

с этим классом необходимо включить в заголовок файла программы инструкцию `import java.util.*`, то есть подключить (импортировать) пакет `java.util`.

Общая схема реализации процесса введения данных с консоли посредством класса `Scanner` такова: на основе стандартного потока ввода `System.in` создается объект класса `Scanner`, через который и осуществляется консольный ввод. При этом полезными могут оказаться методы класса `Scanner`, среди которых имеет смысл выделить следующие:

- `nextLine()` — считывание текстовой строки;
- `next()` — считывание одного слова;
- `nextInt()` — считывание целого числа;
- `nextDouble()` — считывание действительного числа.

Пример использования класса `Scanner` и его методов для реализации в программе консольного ввода приведен в листинге 11.3.

Листинг 11.3. Консольный ввод на основе класса Scanner

```
// Импорт пакета:
import java.util.*;
class MyNewScanner{
public static void main(String args[]){
// Объект класса Scanner создается на основе объекта System.in:
Scanner inp=new Scanner(System.in);
// Текстовое поле (имя):
String name;
// Числовое поле (возраст):
int age;
// Задаем вопрос:
System.out.println("Как Вас зовут?");
// Считываем текст (имя):
name=inp.nextLine();
// Приветствие:
System.out.println("Добрый день, "+name+"!");
// Задаем вопрос:
System.out.println("Сколько Вам лет?");
// Считываем число (возраст):
age=inp.nextInt();
// Вывод сообщения:
System.out.println(name+", Вам "+age+" лет!");
}}
```

Инструкцией `import java.util.*` импортируется пакет `java.util` для работы с классом `Scanner`. В главном методе программы командой `Scanner inp=new Scanner(System.in)` на основе объекта стандартного ввода `System.in` создается объект `inp` класса `Scanner`. С помощью объекта `inp` в данной программе реализуется консольный ввод. В частности, с клавиатуры считываются имя и возраст пользователя. Для

записи текстового значения имени пользователя объявляется поле name класса String. Для записи возраста объявляется целочисленное поле age.

После вывода командой System.out.println("Как Вас зовут?") вопроса об имени пользователя введенный пользователем текст считывается командой name=inp.nextLine(). При этом применяется метод nextLine(), вызываемый из объекта inp, а результат записывается в поле name. Командой System.out.println("Добрый день, "+name+"!") выводится приветствие, причем в этом приветствии указано считанное на предыдущем этапе имя пользователя. Затем командой System.out.println("Сколько Вам лет?") выводится вопрос о возрасте пользователя. Введенное пользователем значение считывается как число командой age=inp.nextInt(), и результат заносится в поле age. Это считывание выполняется вызовом метода nextInt() из объекта inp. Наконец, командой System.out.println(name+", Вам "+age+" лет!") выводится информация о возрасте пользователя с указанным именем. Результат выполнения программы может иметь следующий вид (жирным шрифтом выделен ввыод пользователя):

```
Как Вас зовут?
Алексей Васильев
Добрый день, Алексей Васильев!
Сколько Вам лет?
35
Алексей Васильев, Вам 35 лет!
```

Если сравнивать консольный ввод на основе объекта класса Scanner с тем способом, что рассматривался в предыдущем разделе, то сравнение явно не в пользу последнего.

Использование диалогового окна

Искусство нравиться — это умение обманывать.

Вовенарг

Введение данных во время выполнения программы можно реализовать не через консоль, а с помощью специального диалогового окна, которое является компонентом графической библиотеки Swing. Для подключения библиотеки необходимо в заголовок файла программы включить инструкцию import javax.swing.*. Само диалоговое окно вызывается командой вида JOptionPane.showInputDialog(аргумент), где аргументом статического метода showInputDialog() класса JOptionPane указывается текст, отображаемый над полем ввода диалогового окна. В качестве результата методом showInputDialog() возвращается текст, который вводится пользователем в поле ввода диалогового окна.

Поскольку при вызове диалогового окна создается новый поток, который автоматически не завершается при закрытии окна, для завершения программы со всеми ее потоками требуется использовать инструкцию System.exit(0).

Обращаем внимание читателя, что результатом вызова метода showInputDialog() является текст, что не всегда соответствует формату исходных данных, например пользователь может ввести в поле ввода диалогового окна число. В таких случаях применяют методы преобразования форматов. В частности, для перевода текстового представления чисел в числовые форматы вызывают статический метод parseInt() класса-оболочки Integer и статический метод parseDouble() класса-оболочки Double. Аргументами методов указывают текстовое представление числа. Метод parseInt() предназначен для работы с целыми числами, а parseDouble() — с действительными.

Пример программы, в которой для ввода данных используется диалоговое окно, приведен в листинге 11.4.

Листинг 11.4. Ввод с помощью диалогового окна

```
// Импорт пакета:
import javax.swing.*;
class MyOptionPane{
public static void main(String args[]){
// Текстовое поле (имя):
String name;
// Числовое поле (возраст):
int age;
// Вопрос:
System.out.println("Как Вас зовут?");
// Открываем диалоговое окно:
name=JOptionPane.showInputDialog("Укажите Ваше имя");
// Выводим резльтат считывания:
System.out.println("Считано имя: "+name);
// Приветствие:
System.out.println("Добрый день, "+name+"!");
// Вопрос:
System.out.println("Сколько Вам лет?");
// Открываем диалоговое окно:
age=Integer.parseInt(JOptionPane.showInputDialog("Укажите Ваш возраст"));
// Выводим результат считывания:
System.out.println("Считан возраст: "+age+" лет.");
// Сообщение:
System.out.println(name+", Вам "+age+" лет!");
// Завершение всех потоков:
System.exit(0);}
}
```

Фактически, представленная программа является модификацией рассмотренного ранее примера. Только теперь для ввода имени и возраста пользователя используется не консоль, а диалоговое окно.

Для того чтобы можно было воспользоваться утилитами библиотеки Swing, командой `import javax.swing.*` в заголовке файла программы выполняем импорт соответствующего пакета.

Как и ранее, в главном методе программы объявляется текстовое поле `name` для записи имени пользователя и числовое поле `age` для записи возраста пользователя. Далее выводится приглашение пользователю указать свое имя, однако для считывания имени используется команда

```
name=JOptionPane.showInputDialog("Укажите Ваше имя")
```

В результате открывается диалоговое окно, содержащее текстовое поле ввода (рис. 11.1).

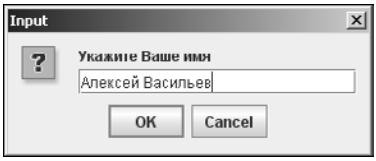

Рис. 11.1. Диалоговое окно с полем для ввода имени пользователя

Переданный методу `showInputDialog()` текст `"Укажите Ваше имя"` отображается над полем ввода в открытом диалоговом окне. В поле вводится текст (в данном случае `Алексей Васильев`), после чего щелкают на кнопке OK в диалоговом окне. В результате введенный в поле диалогового окна текст присваивается в качестве значения полю `name`.

Для считывания возраста пользователя используем команду

```
age=Integer.parseInt(JOptionPane.showInputDialog("Укажите Ваш возраст"))
```

В данном случае команда считывания значения из поля диалогового окна `JOptionPane.showInputDialog("Укажите Ваш возраст")` передается аргументом методу `Integer.parseInt()` для преобразования считанного из поля текста в числовой формат — предполагается, что пользователь введет в поле целое число. Открывающееся при этом диалоговое окно для ввода возраста пользователя представлено на рис. 11.2.

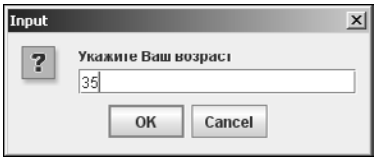

Рис. 11.2. Диалоговое окно с полем для ввода возраста пользователя

Поскольку теперь в окне консоли вводимые пользователем в диалоговом окне значения не отображаются, программа дополнена командами вывода на консоль введенных пользователем имени (команда `System.out.println("Считано имя:`

"+name)) и возраста (команда System.out.println("Считан возраст: "+age+" лет.")). Консольный вывод программы при этом имеет следующий вид:

```
Как Вас зовут?
Считано имя: Алексей Васильев
Добрый день, Алексей Васильев!
Сколько Вам лет?
Считан возраст: 35 лет.
Алексей Васильев, Вам 35 лет!
```

Представленный способ ввода данных с помощью диалогового окна достаточно прост и в то же время удобен. Подробнее методы работы с графическими утилитами библиотеки Swing описываются в следующих главах.

Работа с файлами

Пытался я лукавить и хитрить,
Но каждый раз судьба мой посрамляла опыт.

Омар Хайям

Практически важной является задача ввода данных в файлы и вывода данных из файла. Благодаря концепции потоков ввода-вывода, ввод и вывод информации в файлы и из файлов осуществляется достаточно просто. Для этого создается поток ввода или вывода и подключается к соответствующему файлу. При этом используют классы файловых потоков FileInputStream (файловый поток ввода) и FileOutputStream (файловый поток вывода). Конструкторы классов FileInputStream и FileOutputStream выбрасывают исключение FileNotFoundException (файл не найден). Имя (полное) файла, с которым связывается поток ввода или вывода, в виде текстовой строки передается аргументом конструктору.

После завершения работы с файлом поток необходимо закрыть, для чего используют метод close(). Пример программы, в которой реализована работа с файлами, приведен в листинге 11.5.

Листинг 11.5. Работа с файлами

```
// Подключение пакета:
import java.io.*;
class WriteToFile{
public static void main(String args[]) throws IOException{
// Целочисленное поле:
int a;
try{
// Поток файлового вывода:
FileOutputStream fout=new FileOutputStream("F:/Java_2/Files/mydata.txt");
// Поток файлового ввода:
```

```
FileInputStream fin=new FileInputStream("F:/Java_2/Files/base.txt");
// Считывание из файла:
a=fin.read();
while(a!=-1){
// Замена пробелов символами подчеркивания:
if(a==(int)' ') a=(int)'_';
// Запись в файл:
fout.write(a);
// Считывание из файла:
a=fin.read();}
// Закрыт поток вывода:
fout.close();
// Закрыт поток ввода:
fin.close();
}catch(FileNotFoundException e){ // Обработка исключения "файл не найден":
System.out.println("Нет доступа к файлу: "+e);}
// Сообщение программы:
System.out.println("Работа программы завершена!");}
}
```

Программа достаточно проста: посимвольно считывается информация из исходного текстового файла `base.txt` и записывается в файл `mydata.txt`. При этом все пробелы заменяются символами подчеркивания. На консоль программой по завершении работы выводится сообщение `"Работа программы завершена!"`. Основные события происходят в файлах. Вот содержимое исходного файла `base.txt`:

```
Для изучения любого языка программирования необходимо несколько составляющих.
1. Учебник
2. Компьютер
3. Программное обеспечение
4. Желание учиться
```

После выполнения программы содержимое файла `mydata.txt` будет следующим:

```
Для_изучения_любого_языка_программирования_необходимо_несколько_составляющих.
1._Учебник
2._Компьютер
3._Программное_обеспечение
4._Желание_учиться
```

Рассмотрим подробнее, как выполняется программа. В первую очередь для использования классов, через которые реализована система ввода-вывода (в том числе и в файлы), командой `import java.io.*` подключаем пакет `java.io`.

В главном методе программы описывается целочисленное поле `a` для посимвольного считывания из файлов. Поскольку использоваться при этом будет метод `read()`, который считывает символ и возвращает его числовой код, выбор типа поля `a` как имеющего тип `int` вполне оправдан.

Файловый поток вывода fout создается командой

```
FileOutputStream fout=new FileOutputStream("F:/Java_2/Files/mydata.txt")
```

Файловый поток fout — это объект класса FileOutputStream. Поток подключается к файлу mydata.txt, причем аргументом конструктора класса FileOutputStream указывается текстовая строка с полным именем файла. Поток fout используется в дальнейшем для вывода данных в файл mydata.txt.

Поток файлового ввода fin создается командой

```
FileInputStream fin=new FileInputStream("F:/Java_2/Files/base.txt")
```

Файловый поток fin — это объект класса FileInputStream. Поток подключается к файлу base.txt. Аргументом конструктора класса FileInputStream указывается текстовая строка с полным именем файла. С помощью потока fin будет осуществляться ввод информации из файла base.txt в программу.

Далее командой a=fin.read() выполняется считывание символа из файла base.txt. Для этого из объекта потока файлового ввода fin (связанного, напомним, с файлом base.txt) вызывается метод read(), возвращающий код считанного из файла символа. Этот код записывается в поле a.

В цикле while проверяется условие a!=-1, означающее, на самом деле, что считанный символ не является символом конца файла. Если конец файла не достигнут, условной инструкцией if(a==(int)' ') a=(int)'_' в случае необходимости пробел заменяется символом подчеркивания, и командой fout.write(a) этот символ записывается в файл mydata.txt. Для записи символа в файл из объекта файлового потока вывода fout (связанного с файлом mydata.txt) вызывается метод write(), аргументом которого указывается код записываемого в соответствующий файл символа. После этого командой a=fin.read() считывается следующий символ из файла base.txt и его код записывается в поле a.

После завершения цикл while потоки вывода и ввода закрываются командами fout.close() и fin.close() соответственно. Также в методе main() при создании файловых потоков и работе с ними отслеживается исключение FileNotFoundException (нет доступа к файлу). В случае возникновения такого исключения выводится текстовое сообщение. Для проверки, как программа реагирует на исключительную ситуацию, достаточно изменить название базового файла base.txt и запустить программу. Результат будет следующим:

```
Нет доступа к файлу: java.io.FileNotFoundException: F:\Java_2\Files\base.txt
(Не удается найти указанный файл)
Работа программы завершена!
```

При этом файл mydata.txt оказывается пустым, даже если до этого там что-то было записано. Дело в том, что в программе сначала создается файловый поток вывода, а затем файловый поток ввода. Исключение возникает именно при создании потока ввода, когда файл mydata.txt уже открыт для ввода и очищен от содержимого.

Другой пример работы с системой ввода-вывода, в том числе файловой, приведен в листинге 11.6.

Листинг 11.6. Система ввода-вывода

```
// Подключение пакетов:
import java.io.*;
import javax.swing.*;
class FindFellow{
public static void main(String args[]) throws IOException{
// Текстовое поле (имя файла):
String fileName;
// Текстовое поле (фамилия сотрудника):
String name;
// Текстовое поле (для считывания текста):
String s;
// Считывание имени файла:
fileName=JOptionPane.showInputDialog("Укажите имя файла:");
try{
// Создание файлового потока ввода:
FileInputStream fin=new FileInputStream("F:/Java_2/Files/"+fileName);
// Создание буферизованного символьного потока (из файла):
BufferedReader br=new BufferedReader(new InputStreamReader(fin));
// Считывание фамилии сотрудника:
name=JOptionPane.showInputDialog("Укажите фамилию сотрудника:");
// Формально "бесконечный" цикл:
while(true){
// Считывание строки из файла:
s=br.readLine();
try{
if(s.equalsIgnoreCase(name)){// Вывод "послужного списка"
System.out.println("Фамилия   : "+name);
System.out.println("Имя       : "+br.readLine());
System.out.println("Отчество  : "+br.readLine());
System.out.println("Возраст   : "+br.readLine());
System.out.println("Тел.      : "+br.readLine());
break;}
}catch(NullPointerException e){// Обработка исключения
System.out.println("Такого сотрудника нет!");
// Выход из цикла:
break;}
}
// Закрываем файловый поток:
fin.close();
```

продолжение

Листинг 11.6 *(продолжение)*

```
}catch(FileNotFoundException e){// Обработка исключения
System.out.println("Ошибка доступа к файлу: "+e);}
// Завершение всех потоков:
System.exit(0);}}
```

Программой по введенной пользователем фамилии производится поиск сотрудника в импровизированной базе данных, представленной в виде текстового файла. Текстовый файл `personal.txt`, используемый в качестве базы поиска, содержит записи всего о трех сотрудниках. Содержимое файла `personal.txt` выглядит так:

```
Петров
Иван
Сергеевич
52
526-44-12

Сидоров
Игорь
Степанович
46
526-00-13

Иванов
Семен
Николаевич
61
522-16-48
```

В частности, после фамилии сотрудника построчно указываются его имя, отчество, возраст и телефон.

Программой выводится запрос (диалоговое окно с полем ввода) для ввода имени файла, в котором производится поиск. После этого во втором диалоговом окне необходимо указать фамилию сотрудника для поиска в файле. Если совпадение по фамилии найдено, для данного сотрудника выводится весь «послужной список»: фамилия, имя, отчество, возраст и телефон. Если сотрудника с указанной фамилией нет, на консоль выводится соответствующее сообщение.

Инструкциями `import java.io.*` и `import javax.swing.*` подключаются пакеты `java.io` (для использования классов системы ввода-вывода) и `javax.swing` (для использования утилит библиотеки `Swing` и, в частности, диалогового окна с полем ввода).

В главном методе программы объявляются три текстовых поля класса `String`: поле `fileName` для записи имени файла, в котором осуществляется поиск, поле `name` для записи фамилии сотрудника и поле `s` для записи строк, считываемых из файла.

Диалоговое окно с полем ввода имени файла выводится командой

```
fileName= JOptionPane.showInputDialog("Укажите имя файла:")
```

В окно вводится только имя файла без указания каталога. Каталог размещения добавляется к имени файла при создании файлового потока ввода командой

```
FileInputStream fin=new FileInputStream("F:/Java_2/Files/"+fileName)
```

Недостаток объекта `fin` состоит в том, что он позволяет производить посимвольное считывание, а в данном случае необходимо считывать текстовые строки. Поэтому файловый поток приходится буферизовать, для чего на основе объекта `fin` создается объект `br` буферизованного потока командой

```
BufferedReader br=new BufferedReader(new InputStreamReader(fin)).
```

Точнее, на основе объекта `fin` создается анонимный объект класса `InputStreamReader`, а уже на его основе создается объект буферизованного потока. Аналогичная ситуация имела место при консольном вводе, только там буферизованный поток создавался по той же схеме на основе объекта стандартного ввода `System.in`.

Фамилия сотрудника для поиска вводится в диалоговом окне, которое открывается командой

```
name=JOptionPane.showInputDialog("Укажите фамилию сотрудника:")
```

Результат ввода записывается в поле `name`.

После этого запускается формально бесконечный цикл `while` — условием в нем указано логическое значение `true`. В цикле командой `s=br.readLine()` выполняется построчное считывание из файла. Затем в условной инструкции командой `s.equalsIgnoreCase(name)` проверяется условие совпадения считанной строки и фамилии, записанной в поле `name`. Если совпадение найдено, последовательно выводятся все параметры записи для данного сотрудника. При этом неявно предполагается, что в файле после строки с фамилией следуют строки с именем, отчеством, возрастом и номером телефона. Завершается блок команд вывода сведений о пользователе инструкцией `break`, благодаря чему работа цикла завершается.

Обработка ситуации, когда в цикле совпадений не обнаруживается, реализована через обработку исключения `NullPointerException` — это исключение (ошибка операции с нулевым указателем) возникает при попытке считать строку после того, как достигнут конец файла. В этом случае в обработчике исключения в блоке `catch` выводится сообщение о том, что сотрудника с указанной фамилией нет, и инструкцией `break` прекращается выполнение цикла.

После инструкции цикла командой `fin.close()` закрывается файловый поток, а командой `System.exit(0)` завершается работа всех потоков.

В программе также обрабатывается исключительная ситуация `FileNotFoundException` (отсутствие доступа к файлу). В этом случае выводится сообщение соответствующего содержания с описанием ошибки.

Если при запуске программы в первое окно ввести имя файла `personal.txt` (рис. 11.3) и фамилию `Сидоров` (рис. 11.4), результат выполнения программы будет следующим:

```
Фамилия   : Сидоров
Имя       : Игорь
Отчество  : Степанович
Возраст   : 46
Тел.      : 526-00-13
```

Если фамилию сотрудника указать неправильно (то есть такую, которой нет в файле `personal.txt`), появится сообщение

```
Такого сотрудника нет!
```

Если неверно указано имя файла, будет получено сообщение

```
Ошибка доступа к файлу: java.io.FileNotFoundException: F:\Java_2\Files\df
(Не удается найти указанный файл)
```

При этом до ввода фамилии дело не доходит — работа программы завершается. Хотя при желании можно сделать так, чтобы этого не происходило.

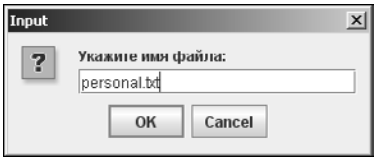

Рис. 11.3. Диалоговое окно для ввода имени файла

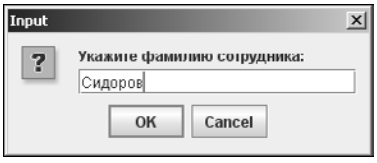

Рис. 11.4. Диалоговое окно для ввода фамилии сотрудника

Резюме

1. В Java ввод и вывод данных реализуется через байтовые и символьные потоки ввода-вывода.
2. Иерархия байтовых потоков основана на абстрактных классах `InputStream` и `OutputStream` (в них определены методы `read()` и `write()` для чтения-записи данных).
3. На вершине иерархии классов символьных потоков находятся абстрактные классы `Reader` и `Writer` (с методами `read()` и `write()`).

4. Некоторые возможности ввода-вывода реализованы через класс `System`. В частности, консольный ввод реализуется считыванием объекта `System.in`. Также может использоваться класс `Scanner`.
5. Для ввода данных можно использовать специальное диалоговое графическое окно (компонент библиотеки `Swing`).
6. При работе с файлами создается поток ввода или вывода и подключается к соответствующему файлу. С этой целью используют классы файловых потоков `FileInputStream` (файловый поток ввода) и `FileOutputStream` (файловый поток вывода).

Глава 12. Создание программ с графическим интерфейсом

Что вы на это скажете, мой дорогой психолог?

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

В Java для создания графического интерфейса обычно используются библиотеки AWT и Swing. Исторически первой и базовой была библиотека AWT (Abstract Window Toolkit). Библиотека Swing появилась несколько позже, хотя она во многом базируется на библиотеке AWT. В этих библиотеках по-разному решается проблема универсальности программного кода, краеугольная для концепции, положенной в основу Java.

В библиотеке AWT универсальность программного кода обеспечивается за счет использования разных инструментальных средств с целью реализации программного кода для разных операционных систем. Удобство такого подхода состоит в том, что на разных системах программы работают одинаково и имеют единый программный интерфейс. Однако такой подход имеет и недостаток: он применим только при написании относительно небольших программ. Кроме того, при тестировании программного кода нередко оказывается, что один и тот же код генерирует разные ошибки для разных операционных систем.

Решить эти фундаментальные проблемы призвана библиотека Swing. В основу ее положена разработка компании Netscape, которая в 1996 году выпустила библиотеку IFC (Internet Foundation Classes). В этой библиотеке реализована следующая идея: элементы интерфейса (кнопки, меню, поля и т. д.) отображаются в пустом окне, а особенности конкретной платформы принимаются во внимание только при выборе способа отображения этого окна. Впоследствии на основе библиотеки IFC компании Sun Microsystems и Netscape создали библиотеку Swing, в которой реализован тот же принцип, что и в IFC. При этом библиотека Swing не заменяет библиотеку AWT, а дополняет ее. Механизм обработки событий в библиотеке Swing тот же, что и в AWT.

Создание простого окна

Трудные задачи выполняем немедленно,
невыполнимые — чуть погодя.

Девиз ВВС США

Рассмотрение методов создания и работы с графическим интерфейсом начнем с задачи создания фрейма. Фрейм — это такое окно, которое не содержит в себе других окон (иногда называемых дочерними).

Для создания фрейма в Java необходимо расширить один из базовых классов: класс `Frame` из библиотеки AWT или класс `JFrame` из библиотеки Swing (причем класс `JFrame` является подклассом класса `Frame`). Вообще говоря, следует отметить, что для очень многих классов библиотеки Swing название класса этой библиотеки отличается от названия суперкласса из библиотеки AWT наличием первой прописной литеры *J*.

Однако просто создать окно — это полдела. Необходимо предусмотреть возможность для этого окна реагировать хоть на какие-то события, в частности на попытку это окно закрыть. Для этого создается специальный обработчик события закрытия окна. С помощью метода `addWindowListener()` ссылка на этот обработчик добавляется в класс, реализующий окно.

Для использования библиотек AWT и Swing, а также классов обработки событий подключают пакеты `javax.swing` (библиотека Swing), `java.awt` и `java.awt.event` (библиотека AWT). Пример программы, в которой средствами AWT создается и отображается графическое окно, представлен в листинге 12.1.

Листинг 12.1. Создание графического окна средствами AWT

```
// Подключение библиотеки AWT:
import java.awt.*;
import java.awt.event.*;
// Класс обработчика закрытия окна:
class MyAdapter extends WindowAdapter{
public void windowClosing(WindowEvent we){
System.exit(0);}
}
// Класс окна:
class JustAFrame extends Frame{
// Конструктор:
JustAFrame(int a,int b){
// Аргумент конструктора суперкласса - имя окна:
super("Новое окно");
// Объект обработчика закрытия окна:
MyAdapter adapter=new MyAdapter();
// Размеры окна:
setSize(a,b);
```

продолжение

Листинг 12.1 *(продолжение)*

```
// Отображение окна:
setVisible(true);
// Добавлен обработчик:
addWindowListener(adapter);}
}
class MyAWTFrame{
public static void main(String args[]){
// Создание окна:
JustAFrame frame=new JustAFrame(400,300);
}}
```

Кроме команд подключения пакетов для работы с утилитами библиотеки AWT, в программе описываются три класса: класс обработчика события закрытия окна программы `MyAdapter`, класс главного окна (фрейма) программы `JustAFrame`, а также класс `MyAWTFrame` с главным методом программы, в котором непосредственно и создается графическое окно (фрейм). Окно, открываемое при выполнении программы, представлено на рис. 12.1.

Рис. 12.1. В результате выполнения программы открывается окно

Окно имеет название `Новое окно`, и все, что можно полезного сделать с этим окном (не считая, разумеется, операций свертывания-развертывания, перемещения по экрану и изменения размеров перетаскиванием границ) — это закрыть его щелчком на системной кнопке в правом верхнем углу строки заголовка окна. Собственно, для обеспечения этой минимальной функциональности (имеется в виду возможность закрыть в нормальном режиме окно) и нужен класс обработчика события закрытия окна.

Для определения главного и единственного окна программы предназначен класс `JustAFrame`, который создается наследованием класса `Frame` библиотеки AWT. Все описание класса состоит из конструктора, у которого два целочисленных аргумента — они определяют размеры создаваемого окна. Первой командой `super("Новое окно")` в конструкторе вызывается конструктор суперкласса с текстовым аргументом, который определяет название создаваемого окна — оно отображается в строке заголовка. Командой `setSize(a,b)` с помощью унаследованного из класса `Frame` метода `setSize()` задаются (в пикселях) размеры окна по горизонтали и вертикали.

Поскольку создание окна не означает его отображения на экране, командой `setVisible(true)` окно выводится на экран (становится видимым). Как и в предыдущем случае, использован унаследованный от класса `Frame` метод `setVisible()`, аргументом которого указано значение `true` — свойство видимости окна устанавливается истинным.

Командой `MyAdapter adapter=new MyAdapter()` создается объект обработчика события закрытия окна, то есть объект `adapter` класса `MyAdapter`. Команда `addWindowListener(adapter)` означает, что объект `adapter` используется для обработки событий, происходящих с окном. В свою очередь, что именно будет происходить в рамках обработки событий, определяется кодом класса `MyAdapter`. В этом классе описан всего один метод `windowClosing()`, наследуемый от класса `WindowAdapter`. Аргументом методу передается объект события (генерируется автоматически при возникновении события). Метод `windowClosing()` определяет последовательность действий при попытке закрыть окно. Действие всего одно — командой `System.exit(0)` завершается работа всех процессов программы, в результате чего окно закрывается (аргумент метода напрямую не используется).

Следует отметить, что в Java применяется достаточно нетривиальная, но эффективная модель обработки событий. Подробнее она описывается в следующем разделе, а здесь отметим только основные ее особенности. В частности, для обработки происходящих в окне событий используются объекты-обработчики. Объект, в котором может произойти событие, должен иметь ссылку на объект, обрабатывающий это событие. Для обработки событий создаются классы, на основе которых затем создаются объекты-обработчики событий. Объект `adapter`, создаваемый в конструкторе класса фрейма `JustAFrame`, является примером такого объекта. С помощью метода `addWindowListener()` осуществляется связывание объекта, в котором происходит событие (это фрейм), с объектом, обрабатывающим событие.

Еще одно замечание относится к использованному в данном примере способу реализации описанной схемы. Дело в том, что поскольку окно в программе создается всего одно, которое реагирует всего на одно событие (точнее, одно событие обрабатывается), код можно было бы несколько упростить. Один из способов состоит в том, чтобы не создавать в явном виде объект-обработчик, и вместо команд `MyAdapter adapter=new MyAdapter()` и `addWindowListener(adapter)` использовать всего одну команду `addWindowListener(new MyAdapter())`.

Другой способ упрощения программного кода состоит в применении анонимного класса обработчика события. В этом случае класс `MyAdapter` не описывается вообще, в конструкторе класса `JustAFrame` команды `MyAdapter adapter=new MyAdapter()` и `addWindowListener(adapter)` отсутствуют, а вместо них используется команда

```
addWindowListener(new WindowAdapter(){
                    public void windowClosing(WindowEvent we){
                    System.exit(0);
                    }
});
```

Здесь аргументом метода addWindowListener() указан анонимный объект анонимного класса (создаваемого на основе класса WindowAdapter), содержащий описание метода windowClosing().

В главном методе программы создается объект frame класса JustAFrame командой

```
JustAFrame frame=new JustAFrame(400,300)
```

При создании объекта вызывается конструктор класса, в результате на экране отображается окно размером 400 на 300 пикселей, представленное на рис. 12.1.

Несколько проще создать окно с помощью утилит библиотеки Swing. Пример такой программы приведен в листинге 12.2.

Листинг 12.2. Создание графического окна средствами Swing

```
// Подключение библиотеки:
import javax.swing.*;
// Расширение класса JFrame:
class JustAFrame extends JFrame{
// Конструктор класса:
public JustAFrame(int a,int b){
// Заголовок окна - аргумент конструктора суперкласса:
super("Простое графическое окно");
// Размеры окна:
setSize(a,b);
// Реакция на попытку закрыть окно:
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Отображение окна:
setVisible(true);}
}
class MyFrame{
public static void main(String args[]){
// Создание окна:
JustAFrame frame=new JustAFrame(300,200);
}}
```

Класс фрейма JustAFrame создается на основе класса JFrame библиотеки Swing. Как и в предыдущем примере, класс содержит описание конструктора с двумя целочисленными аргументами — размерами окна. Все команды в конструкторе практически такие же, как в примере из листинга 12.1, с тем же назначением: метод setSize() служит для определения размеров окна, метод setVisible() — для отображения окна. Название окна передается текстовым аргументом конструктору суперкласса. Изменился только способ обработки события закрытия окна. В данном случае этой цели служит команда

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

Аргументом методу setDefaultCloseOperation(), определяющему реакцию на попытку закрыть окно, передается статическая константа EXIT_ON_CLOSE класса JFrame, означающая, что работа программы будет завершена и окно закрыто.

Для создания окна в главном методе программы выполняется команда `JustAFrame frame=new JustAFrame(300,200)`, открывающая окно размером 300 на 200 пикселей с названием `Простое графическое окно` (аргумент конструктора суперкласса). Окно показано на рис. 12.2.

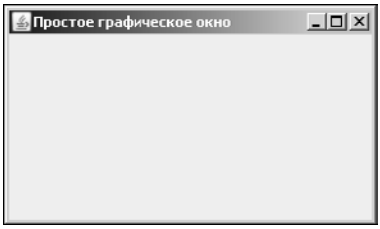

Рис. 12.2. Окно создано средствами библиотеки Swing

Несложно заметить, что внешне окна на рис. 12.1 и 12.2 отличаются (имеется в виду стиль окон). Легкие компоненты, то есть те, что созданы на основе библиотеки Swing, обычно больше соответствуют стилю окон используемой операционной системы.

Обработка событий

— По-вашему это не интересно?
— Интересно. Для любителей древности.

Из к/ф «Приключения Шерлока Холмса и доктора Ватсона»

Обработка событий при создании приложений с графическим интерфейсом является, пожалуй, краеугольным камнем любого графического приложения. Взаимодействие пользователя с графическим интерфейсом реализуется через обработку событий. Под событием подразумевают любое действие, производимое с графическим компонентом, например щелчок на кнопке, изменение размеров окна, выделение текстового поля для ввода значения с клавиатуры и т. д. Объект, в котором возникло событие, называется источником события. При возникновении события в Java автоматически создается объект события (для каждого события существует класс, описывающий его, и эти классы образуют иерархию). Объект события содержит описание этого события. Фактически, обработка события подразумевает обработку его объекта.

Как уже отмечалось, в Java используется схема обработки событий, реализованная в библиотеке AWT. То есть независимо от того, легкие (библиотека Swing) или тяжелые (библиотека AWT) компоненты используются, схема обработки событий одна и та же.

На вершине иерархии классов, описывающих события, находится класс `EventObject`, который описан в пакете `java.util` и является в Java расширением общего

суперкласса Object. В свою очередь, класс EventObject наследуется абстрактным классом AWTEvent. Этот класс описан в библиотеке AWT и, соответственно, находится в пакете java.awt. Все остальные классы обработки событий в библиотеке AWT являются подклассами класса AWTEvent и описаны в пакете java.awt.event. Иерархия классов обработки событий библиотеки AWT показана на рис. 12.3.

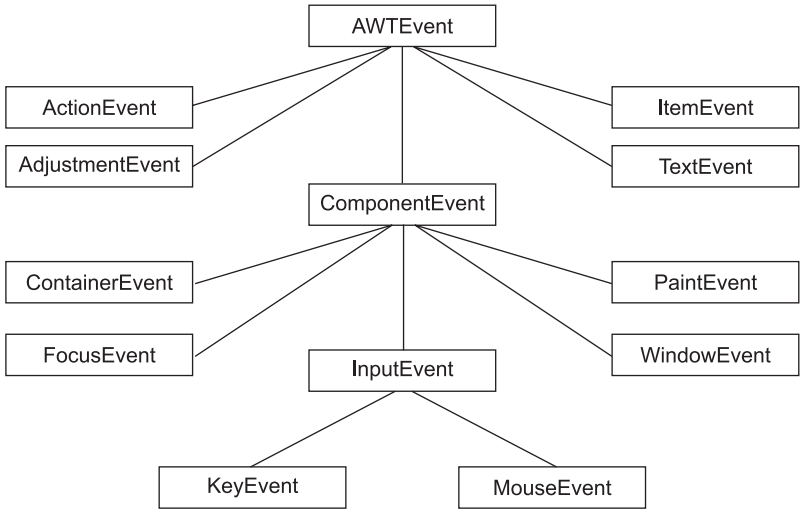

Рис. 12.3. Иерархия классов для описания событий

События (классы событий библиотеки AWT) кратко описаны в табл. 12.1.

Таблица 12.1. Классы событий библиотеки AWT

Класс события	Описание	Может возникать
ActionEvent	Генерируется при щелчке мышью на кнопке	Возникает в компонентах классов Button, List и TextField
AdjustmentEvent	Возникает при изменении положения ползунка полосы прокрутки	Возникает в компонентах класса Scrollbar
ComponentEvent	Возникает при перемещении компонента, изменении его размеров, отображении и скрытии компонента	Возникает во всех компонентах
ItemEvent	Возникает при выборе или отказе от выбора элемента в соответствующих компонентах	Возникает в компонентах классов Checkbox, Choice и List
TextEvent	Происходит при изменении текста	Возникает в компонентах классов TextComponent, TextArea и TextField

Класс события	Описание	Может возникать
ContainerEvent	Возникает, если в контейнер добавляется компонент или компонент из контейнера удаляется	Возникает в компонентах классов Container, Dialog, FileDialog, Frame, Panel, ScrollPane и Window
FocusEvent	Возникает, если соответствующий компонент получает или теряет фокус	Возникает во всех компонентах
InputEvent	Абстрактный класс, являющийся суперклассом для классов KeyEvent и MouseEvent. В классе определяются восемь целочисленных констант для получения информации о событии	Возникает при операциях ввода для компонентов
PaintEvent	Происходит при перерисовке компонента	Возникает в основных компонентах
WindowEvent	Происходит при открытии, закрытии, сворачивании, разворачивании окна, получении и передаче окном фокуса	Возникает в компонентах классов Dialog, FileDialog, Frame и Window
KeyEvent	Возникает при нажатии клавиши, отпускании клавиши, ввода символа	Возникает во всех компонентах
MouseEvent	Возникает при манипуляциях мышью с компонентом, таких как щелчок, перемещение указателя, появление указателя мыши на компоненте и т. д.	Возникает во всех компонентах

Общая схема обработки событий, используемая в Java, называется схемой с *делегированием*. В отличие от другой популярной схемы, где при появлении события опрашиваются все доступные обработчики событий, в схеме с делегированием для всех объектов, в которых могут происходить события (точнее, в тех объектах, в которых предполагается обрабатывать события), явно указывается обработчик события. Удобство такого подхода состоит в значительной экономии времени на поиск нужного обработчика. Недостатки также очевидны — приходится писать больше кода, а сам код становится слегка «заумным».

Перечисленные в табл. 12.1 классы описывают события. Для обработки этих событий используют другие классы. Точнее, для каждого из событий существует специальный интерфейс создания класса обработки события. Для обработки события необходимо, как минимум, расширить интерфейс обработчика этого события. Названия интерфейсов для обработки событий связаны с названиями классов соответствующих событий. Чтобы получить название интерфейса, в имени соответствующего класса необходимо вместо слова Event вставить

слово Listener. Например, для события класса WindowEvent интерфейс обработчика имеет название WindowListener. Исключением из этого правила является класс InputEvent — для этого события собственного интерфейса нет.

Таким образом, для обработки события необходимо расширением соответствующего интерфейса создать класс обработки события, а затем объект этого класса. Но этого мало. При создании компонента, в котором предполагается обрабатывать данное событие, необходимо оставить ссылку на объект, позволяющий обработать событие. Делается это с помощью специальных методов. Названия методов формируются, как и названия интерфейсов, на основе названий классов событий. Имя метода начинается словом add, далее следует имя события (это имя соответствующего класса без слова Event) и, наконец, слово Listener. Например, для связывания компонента с обработчиком события WindowEvent используют метод addWindowListener(). Аргументом метода указывается объект обработчика события.

Вкратце это вся схема обработки событий. Особенности ее применения рассмотрим на конкретных примерах.

Приложение с кнопкой

Нажми на кнопку — получишь результат...

Из песни группы «Технология»

В качестве простой иллюстрации использования в главном окне компонентов и обработки базовых событий рассмотрим пример программы с графическим интерфейсом, в которой, помимо фрейма, имеется еще и кнопка. Функциональность этой кнопки реализуется путем создания обработчиков событий. Программный код приведен в листинге 12.3.

Листинг 12.3. Окно с кнопкой создано средствами Swing

```
// Подключение пакетов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
// Класс фрейма:
class MyFrame extends JFrame{
// Счетчик окон:
public static int count=0;
// Конструктор:
MyFrame(int a,int b){
count++;  // Количество открытых окон
// Название окна:
setTitle("Окно с кнопкой: "+count);
```

```
// Создание панели:
MyPanel panel=new MyPanel();
setSize(300,200); // Размер окна
// Закрытие окна:
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocation(a,b);  // Положение окна
add(panel);        // Добавление панели
setVisible(true);  // Отображение окна
}}
// Класс панели:
class MyPanel extends JPanel{
// Конструктор:
MyPanel(){
// Создание кнопки:
JButton button=new JButton("Создать новое окно");
add(button);          // Добавление кнопки на панель
button.addActionListener(listener);} // Регистрация обработчика
// Обработчик для кнопки - объект анонимного класса:
ActionListener listener=new ActionListener(){
public void actionPerformed(ActionEvent event){
Random rnd=new Random();
// Создание окна со случайными координатами размещения на экране:
MyFrame frame=new MyFrame(rnd.nextInt(800),rnd.nextInt(500));}};
}
class FrameAndButton{
public static void main(String args[]){
// Создание первого окна:
MyFrame frame=new MyFrame(100,100);}
}
```

Действие программы состоит в следующем: при запуске программы открывается окно с кнопкой и названием Окно с кнопкой: 1, показанное на рис. 12.4.

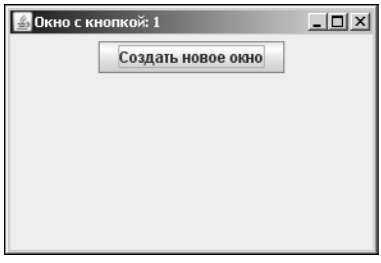

Рис. 12.4. Первое окно, которое открывается при запуске программы

Кнопка имеет название Создать новое окно. При щелчке на кнопке открывается еще одно окно — практически копия первого. Отличается только номер в названии. Расположение нового окна на экране выбирается случайным образом.

Далее при щелчке на любой из кнопок в первом или втором окне открывается такое же третье окно (с порядковым номером три), причем его положение на экране также выбирается случайно и т. д. Если закрыть хотя бы одно окно щелчком на системной кнопке в строке заголовка окна, закрываются все окна. Результат выполнения программы показан на рис. 12.5.

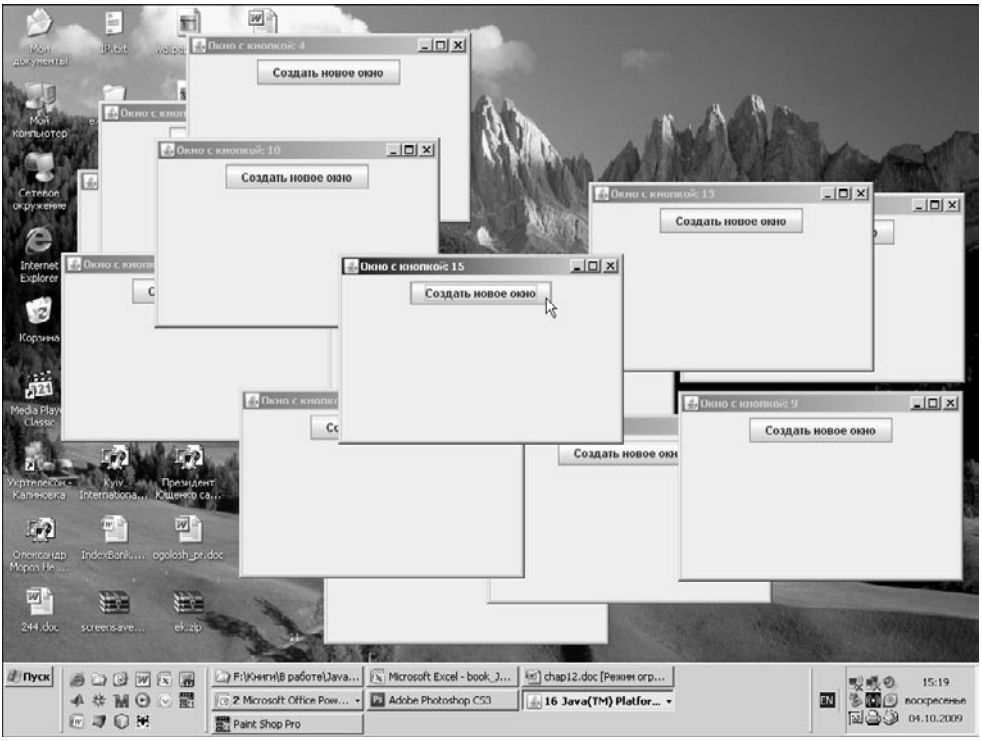

Рис. 12.5. Вид экрана в процессе выполнения программы

Класс фрейма `MyFrame` создается путем расширения класса `JFrame`. В классе объявляется статическое поле `count` с нулевым начальным значением. Поле служит для подсчета количества открытых окон. Конструктор класса имеет два целочисленных аргумента, определяющих положение окна на экране. При создании очередного окна значение поля `count` увеличивается на единицу (командой `count++` в конструкторе класса), после чего методом `setTitle()` задается название созданного окна: к тексту Окно с кнопкой добавляется значение поля `count`. В результате создаваемые окна нумеруются последовательно. Здесь же, в конструкторе класса фрейма, командой `MyPanel panel=new MyPanel()` создается объект панели `panel` на основе класса `MyPanel` (описывается далее). Во фрейм панель добавляется командой `add(panel)`, для чего вызывается стандартный метод `add()`, аргументом которого указывается добавляемый в контейнер объект. Сам контейнер определяется как объект, из которого вызывается метод `add()`.

Размер создаваемого окна задается командой `setSize(300,200)`, расположение на экране — командой `setLocation(a,b)` (аргументы конструктора `a` и `b` определяют координату левого верхнего угла фрейма). Реакция фрейма на щелчок на системной кнопке, как и в предыдущем примере, определяется командой

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

Отображается окно командой `setVisible(true)`. На этом описание класса фрейма заканчивается. Особенность. в данном случае класса, связана с явным определением позиции окна на экране и добавлением на фрейм панели.

Класс панели `MyPanel` создается на основе класса `JPanel` библиотеки Swing. В конструкторе класса на панель добавляется кнопка. Объект кнопки `button` (объект класса `JButton` библиотеки Swing) создается командой

```
JButton button=new JButton("Создать новое окно")
```

Название кнопки передается аргументом конструктору класса `JButton`. Добавление кнопки на панель осуществляется командой `add(button)`. Командой `button.addActionListener(listener)` для кнопки регистрируется обработчик — объект `listener` анонимного класса, созданного на основе класса `ActionListener`. При создании объекта описывается метод `actionPerformed()`, в котором создается окно, чье положение на экране определяется случайным образом. В главном методе приложения в классе `FrameAndButton` создается первое окно командой

```
MyFrame frame=new MyFrame(100,100)
```

Практически тот же подход реализован в следующем примере, представленном в листинге 12.4.

Листинг 12.4. Окно с кнопкой создано средствами AWT

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
import java.util.Random;
// Класс фрейма:
class MyFrame extends Frame{
public static int count=0; // Количество открытых окон
// Конструктор:
MyFrame(int a,int b){
count++;                    // Количество открытых окон
setTitle("Окно с кнопкой: "+count);
setLayout(null);        // Отключение менеджера размещения элементов
Font f=new Font("Arial",Font.BOLD,11);  // Определение шрифта
setFont(f);                             // Применение шрифта
Button btn=new Button("Создать новое окно");
btn.setBounds(50,100,200,30);  // Размеры и положение кнопки
add(btn);                      // Добавление кнопки
setBounds(a,b,300,200);        // Положение и размер окна
```

продолжение

Листинг 12.4 *(продолжение)*

```
// Регистрация обработчика в окне:
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ve){
System.exit(0);}               // Закрытие окна
});
// Регистрация обработчика в кнопке:
btn.addActionListener(new ButtonPressed());
setVisible(true);              // Отображение окна
}}
// Класс обработчика для кнопки:
class ButtonPressed implements ActionListener{
private Random rnd;
// Конструктор:
ButtonPressed(){
rnd = new Random();}
public void actionPerformed(ActionEvent ae){
// Создание окна со случайным положением на экране:
new MyFrame(rnd.nextInt(800),rnd.nextInt(500));}
}
class FrameAndButtonAWT{
public static void main(String args[]){
// Создание первого окна:
MyFrame frame=new MyFrame(100,100);}
}
```

Однако в данном случае задействованы «тяжелые» компоненты из библиотеки AWT. В частности, на основе стандартного класса `Frame` библиотеки AWT создается, путем наследования, класс фрейма `MyFrame`. Для последовательной нумерации создаваемых окон служит статическое поле `count` (начальное значение поля равно нулю). Конструктор класса имеет два целочисленных аргумента, которые определяют положение окна на экране. В теле конструктора командой `count++` на единицу увеличивается значение поля `count`, а командой `setTitle("Окно с кнопкой: "+count)` определяется название для создаваемого окна (каждое новое окно — это объект класса `MyFrame`). Командой `setLayout(null)` отключается менеджер размещения элементов: расположение элементов (точнее, кнопки) в окне предполагается указать в явном виде. При создании окна явно определяется шрифт, используемый для отображения надписей на кнопке. Командой `Font f=new Font("Arial",Font.BOLD,11)` создается объект шрифта `f`, который соответствует типу шрифта *Arial* размера 11 полужирного начертания. Применяется шрифт командой `setFont(f)`. Командой `Button btn=new Button("Создать новое окно")` создается объект `btn` кнопки (объект класса `Button` библиотеки AWT), а командой `btn.setBounds(50,100,200,30)` задается размещение кнопки в окне (левый верхний край кнопки находится на расстоянии 50 пикселей от левой границы окна и 100 пикселей от верхней границы окна) и ее размеры

(200 пикселей в ширину и 30 пикселей в высоту). Добавление кнопки в окно выполняется командой `add(btn)`. Положение и размер создаваемого окна задает команда `setBounds(a,b,300,200)`. В этом случае параметры `a` и `b` (аргументы конструктора) определяют координату левого верхнего угла окна (расстояние в пикселях соответственно от левого и верхнего края экрана), ширина окна равна 300 пикселей, высота — 200 пикселей. Регистрация обработчика для закрытия окна (щелчок на системной кнопке в строке заголовка окна) выполняется командой

```
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ve){
System.exit(0);}
});
```

Этот код, как и команда `setVisible(true)`, обсуждался в предыдущих примерах. Кроме того, в кнопке выполняется регистрация обработчика щелчка на кнопке, для чего служит команда

```
btn.addActionListener(new ButtonPressed())
```

При этом используется анонимный объект класса `ButtonPressed`. Сам класс обработчика щелчка на кнопке `ButtonPressed` создается как реализация интерфейса `ActionListener`. В классе объявляется закрытое поле `rnd` — объект класса `Random` (объект нужен для генерирования случайных чисел), описывается конструктор и метод `actionPerformed()`. В конструкторе создается объект класса `Random`, и ссылка на него присваивается в качестве значения полю `rnd`. В методе `actionPerformed()` командой `new MyFrame(rnd.nextInt(800),rnd.nextInt(500))` создается анонимный объект класса `MyFrame`, то есть создается и отображается новое окно. Его координаты (в пикселях) на экране — по горизонтали случайное число от 0 до 800, а по вертикали от 0 до 500. Условные координатные оси на экране направлены по горизонтали слева направо, а по вертикали — сверху вниз. Начало отсчета находится в левом верхнем углу экрана.

В главном методе программы в классе `FrameAndButtonAWT` командой `MyFrame frame=new MyFrame(100,100)` создается первое окно (напомним, конструктору аргументами передаются координаты этого окна на экране). Это окно показано на рис. 12.6.

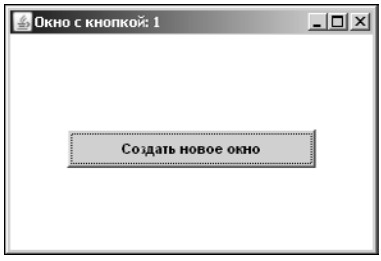

Рис. 12.6. Первое окно, которое открывается при запуске программы

На рис. 12.7 показано, как мог бы выглядеть экран в процессе работы приложения.

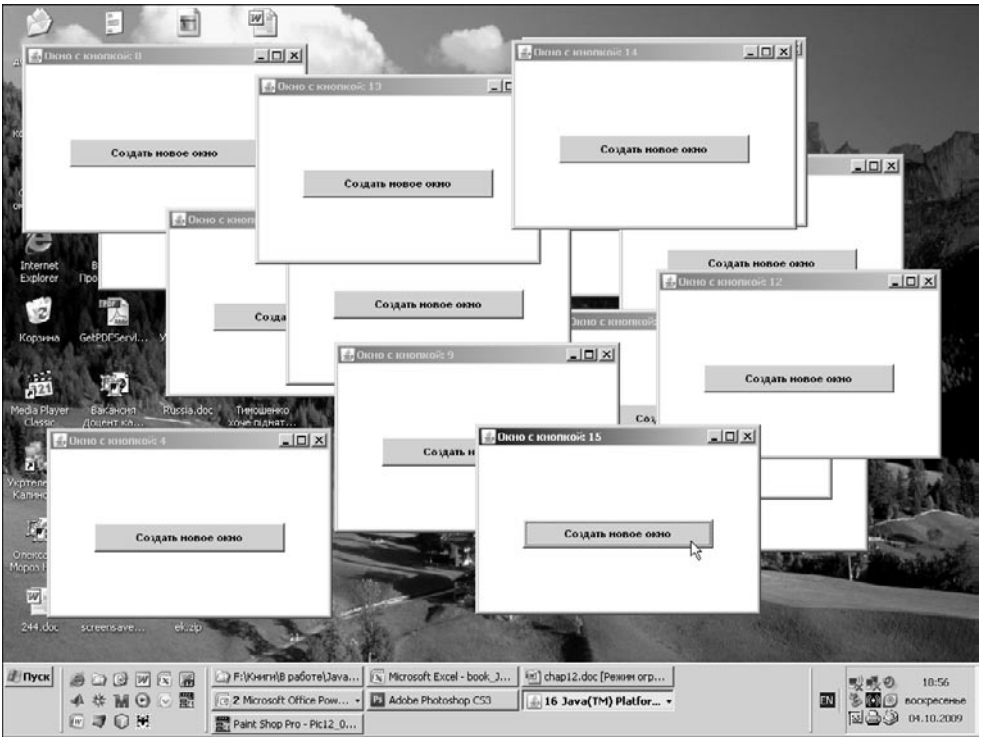

Рис. 12.7. Вид экрана при выполнении программы

В рассмотренных примерах, кроме непосредственно базового окна (фрейма), рассматривались только два компонента — кнопка и панель. Разумеется, компонентов существует намного больше и некоторые из них рассматриваются далее.

Классы основных компонентов

Огласите весь список, пожалуйста.

Из к/ф «Операция „Ы“ и другие приключения Шурика»

В этом разделе дается краткий обзор некоторых компонентов, наиболее часто используемых при создании приложений с графическим интерфейсом. На рис. 12.8 приведена несколько упрощенная структура классов базовых компонентов Java из библиотеки AWT.

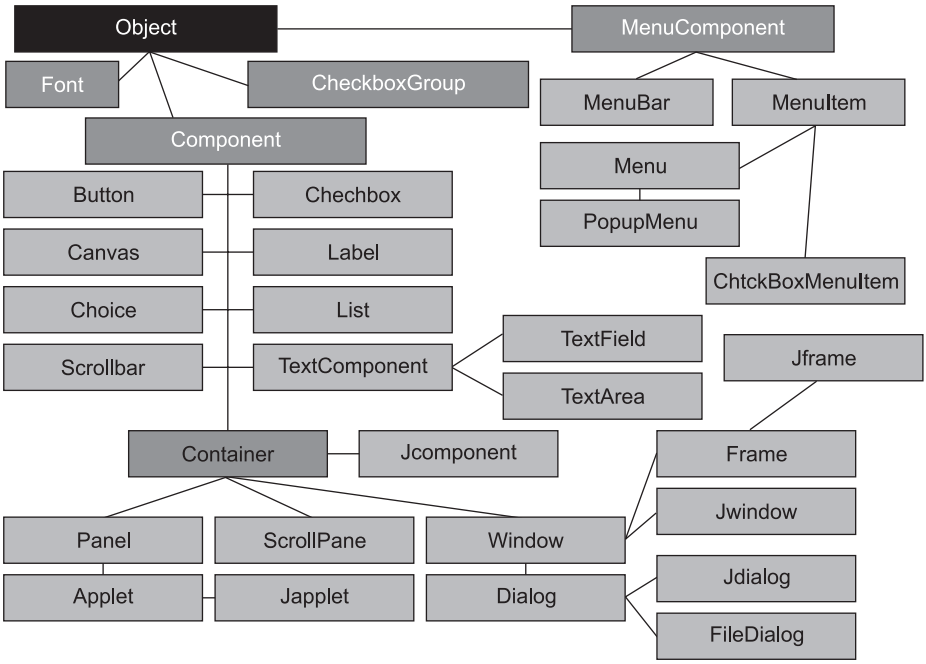

Рис. 12.8. Структура классов компонентов библиотеки AWT

Наиболее важные классы перечислены и кратко описаны в табл. 12.2.

Таблица 12.2. Классы компонентов библиотеки AWT

Класс	Компонент	Описание
`Button`	Стандартная для данной операционной системы кнопка	Конструктор класса: `Button()` — создание кнопки без надписи, `Button(`*название*`)` — создание кнопки с надписью *название*. Некоторые методы: `getLabel()` — возвращает *название* кнопки, `setLabel(`*название*`)` — задает название кнопки. События: `ComponentEvent` (перемещение компонента, изменение размера, удаление и появление на экране), `FocusEvent` (получение и утрата фокуса), `KeyEvent` (нажатие и отпускание клавиши при наличии фокуса), `MouseEvent` (манипуляции мышью на области компонента), `ActionEvent` (возникает при воздействии на кнопку)
`Checkbox`	Флажок с квадратной областью для установки (активное состояние) и сброса (неактивное состояние)	Конструктор класса: `Checkbox()` — создание компонента без надписи, `Checkbox(`*название*`)` — создание компонента с *названием*, `Checkbox(`*название*`, `*состояние*`)` — создание компонента с *названием* и *состоянием* (активный — `true` или неактивный — `false`). Некоторые методы: `getLabel()` — возвращает *название* (надпись)

продолжение

Таблица 12.2 *(продолжение)*

Класс	Компонент	Описание
`Checkbox` (*продолжение*)		компонента, `setLabel(`*название*`)` — задает *название* компонента, `getState()` — возвращает *состояние* компонента, `setState(`*состояние*`)` — задает *состояние* компонента. События: `ItemEvent` (возникает при изменении *состояния* компонента), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`CheckboxGroup`	Группа переключателей	Конструктор класса: `CheckboxGroup()` — создание компонента (группы переключателей). В частности, создается объект класса `CheckboxGroup`, после чего с помощью конструкторов `CheckboxGroup(`*название*, *группа*, *состояние*`)` и `CheckboxGroup(`*название*, *состояние*, *группа*`)` создаются переключатели группы. При этом аргументами передаются *название* компонента (переключателя), его *состояние* и *группа* (объект) переключателей. Только один переключатель в группе может иметь *состояние*, равное `true`. Некоторые методы: `getSelectedCheckbox()` — возвращает активный (выбранный) объект в группе, `setSelectedCheckbox(`*объект*`)` — делает активным (переводит в активное состояние) *объект*
`Choice`	Раскрывающийся список	Конструктор класса: `Choice()` — создание пустого списка. Некоторые методы: `add(`*текст*`)` — добавление *текста* пунктом в список (нумерация пунктов списка начинается с нуля и выполняется в соответствии с порядком добавления в список пунктов), `insert(`*текст*`,`*позиция*`)` — вставка пункта *текст* в *позицию* в списке, `select(`*текст*`)` или `select(`*позиция*`)` — выбор пункта в списке, `remove(`*текст*`)` или `remove(`*позиция*`)` — удаление пункта из списка, `removeAll()` — очистка списка, `getItem(`*позиция*`)` — возвращает *текст* пункта с данной *позицией*, `getSelectedIndex()` — возвращает позицию выбранного пункта, `getSelectedItem()` — возвращает *текст* выбранного пункта, `getItemCount()` — возвращает количество пунктов в списке. События: `ItemEvent` (возникает при выборе пункта из списка), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`List`	Список с полосой прокрутки, в котором можно выбирать сразу несколько пунктов	Конструктор класса: `List()` — создание пустого списка с четырьмя пунктами для отображения, `List(`*число*`)` — создание пустого списка с *числом* пунктов для отображения, `List(`*число*`,`*состояние*`)` — создание пустого

Класс	Компонент	Описание
`List` (*продолжение*)		списка с *числом* пунктов для отображения (если *состояние* равно `true`, то можно выбирать сразу несколько пунктов). Некоторые методы: `add(`*текст*`)` — добавление *текста* в качестве пункта в конец списка, `add(`*текст*`,`*позиция*`)` — добавление *текста* пунктом списка с заданной *позицией*, `remove(`*текст*`)` или `remove(`*позиция*`)` — удаление пункта из списка, `removeAll()` — очистка списка, `getItem(`*позиция*`)` — возвращает текст пункта с данной *позицией*, `getSelectedIndex()` — возвращает позицию выбранного пункта, `getSelectedItem()` — возвращает текст выбранного пункта, `getSelectedIndexes()` — возвращает позиции (массив) выбранных пунктов, `getSelectedItems()` — возвращает текст (массив) выбранных пунктов, `getItemCount()` — возвращает количество пунктов в списке. События: `ActionEvent` (возникает при двойном щелчке мышью на выбранном пункте), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`TextComponent`	Абстрактный класс	Самостоятельно не используется (используются подклассы этого класса). Некоторые методы: `getText()` — возвращает текст в поле ввода, `setEditable(`*состояние*`)` — задает *состояние*, определяющее возможность редактирования текста в поле ввода, `isEditable()` — возвращает *состояние*, определяющее возможность редактирования, `getCaretPosition()` — возвращает индекс позиции курсора в поле ввода, `setCaretPosition()` — устанавливает индекс позиции курсора в поле ввода, `select(`*начало*`,`*конец*`)` — выделение текста от позиции *начало* до позиции *конец*, `selectAll()` — выделение всего текста в поле ввода, `setSelectionStart(`*позиция*`)` — определение *позиции* начала выделения текста, `setSelectionEnd(`*позиция*`)` — определение *позиции* конца выделения текста, `getSelectedText()` — возвращает выделенный текст, `getSelectionStart()` — возвращает начальный индекс выделения текста, `getSelectionEnd()` — возвращает конечный индекс выделения текста. События: `TextEvent` (возникает при изменении текста в поле), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`

продолжение

Таблица 12.2 *(продолжение)*

Класс	Компонент	Описание
TextField	Текстовое поле (одна строка)	Подкласс класса TextComponent. Конструкторы класса: TextField() — создание пустого поля шириной в одну колонку, TextField(*число*) — создание пустого поля шириной в *число* колонок, TextField(*текст*) — создание поля с *текстом*, TextField(*текст*,*число*) — создание поля с *текстом* шириной в *число* колонок. Некоторые методы: getColumns() — возвращает количество колонок поля, setColumns(*число*) — задает *число* колонок поля, setEchoChar(*символ*) — задает *символ* отображения в поле ввода пароля, echoCharIsSet() — проверяет, установлен ли символ отображения в поле ввода пароля, getEchoChar() — возвращает символ отображения в поле ввода пароля, setEchoChar(0) — перевод поля в нормальный режим. События: ActionEvent (возникает при нажатии клавиши Enter), TextEvent (возникает при изменении текста в поле), ComponentEvent, FocusEvent, KeyEvent и MouseEvent
TextArea	Текстовая область (несколько строк)	Подкласс класса TextComponent. Конструкторы класса: TextArea(*текст*,*строки*, *столбцы*,*полосы*) — создание области ввода текста из заданного количества *строк* и *столбцов* с *текстом* и *полосами* прокрутки (значения: SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY, SCROLLBARS_BOTH). Можно указывать не все аргументы. Некоторые методы: append(*текст*) — добавление *текста* в конец области, insert(*текст*,*позиция*) — вставка текста в позицию, replaceRange(*текст*, *начало*,*конец*) — вместо текста в области от позиции *начало* до позиции *конец* вставляется *текст*. События: TextEvent (возникает при изменении текста в области), ComponentEvent, FocusEvent, KeyEvent и MouseEvent
Scrollbar	Полоса прокрутки (и ползунок)	Конструкторы класса: Scrollbar() — создается вертикальная полоса прокрутки в диапазоне значений от 0 до 100 и текущим значением 0, Scrollbar(*ориентация*) — в зависимости от значения аргумента *ориентация* (HORIZONTAL или VERTICAL) создается соответственно горизонтальная или вертикальная полоса прокрутки (с тем же диапазоном прокрутки), Scrollbar(*ориентация*, *значение*, *размер*,*мин*,*макс*) — задается *ориентация* полосы, текущее *значение* (положение ползунка),

Класс	Компонент	Описание
Scrollbar (*продолжение*)		диапазон прокрутки от *мин* до *макс* и *размер* блока. Основной метод: getValue() — возвращает текущее *значение* ползунка. События: AdjustmentEvent (происходит при изменении позиции ползунка), ComponentEvent, FocusEvent, KeyEvent и MouseEvent
Panel	Панель (контейнер) — невидимый компонент интерфейса	Конструкторы класса: Panel() — создается контейнер для размещения компонентов с менеджером размещения компонентов по умолчанию FlowLayout, Panel(*менеджер*) — создается контейнер для размещения компонентов с указанным *менеджером* размещения компонентов
ScrollPane	Контейнер для больших компонентов. Содержит только один компонент	Контейнеры класса: ScrollPane() — создается контейнер в режиме отображения полос прокрутки по необходимости, ScrollPane(*полосы*) — аргумент определяет наличие полос прокрутки и может принимать значения SCROLLBARS_ALWAYS (отображаются всегда), SCROLLBARS_AS_NEEDED (по необходимости) и SCROLLBARS_NEVER (никогда). Методы класса: getHAdjustable() — возвращает положение горизонтальной полосы прокрутки, getVAdjustable() — возвращает положение вертикальной полосы прокрутки, getScrollPosition() — возвращает через объект класса Point координаты точки компонента, находящейся в левом верхнем углу контейнера, setScrollPosition(*объект*) или setScrollPosition(*x*,*y*) — прокручивает компонент в позицию с координатами *x* и *y*. Координаты задаются напрямую или через *объект* класса Point
Window	Пустое окно	Конструктор класса: Window(*окно*) — создается окно, владельцем которого является уже существующее *окно* (объект класса Window или Frame). Некоторые методы: show() — отображает окно, hide() — убирает окно с экрана, isShowing() — проверяет, отображается ли окно, toFront() — окно переводится на передний план, toBack() — окно переводится на задний план, dispose() — уничтожение окна. События: WindowEvent (происходит при перемещении, изменении размеров окна, удалении с экрана или выводе на экран), ComponentEvent, FocusEvent, KeyEvent и MouseEvent

продолжение

Таблица 12.2 *(продолжение)*

Класс	Компонент	Описание
`Frame`	Окно со строкой заголовка	Конструкторы класса: `Frame()` — создается окно без названия, `Frame(`*название*`)` — создается окно с *названием*. Некоторые методы: `setTitle(`*название*`)` — задает *название* окна, `setMenuBar(`*объект*`)` — добавление строки меню (*объект* класса `MenuBar`), `setIconImage(`*объект*`)` — задается значок (*объект* класса `Image`). События: `WindowEvent` (происходит при перемещении, изменении размеров окна, удалении с экрана или выводе на экран), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`Dialog`	Диалоговое окно (обычно окно фиксированного размера)	Конструкторы класса: `Dialog(`*владелец*`,`*название*`,`*состояние*`)` — если *состояние* равно `false` или не указано, создается немодальное окно (пока не закроется, работу продолжать нельзя) с *названием* и *владельцем* (объект класса `Dialog` или `Frame`). Обязательным является только первый аргумент. Некоторые методы: `isModal()` — проверяет окно на предмет модальности, `setModal(`*состояние*`)` — задает *состояние* модальности окна. События: `WindowEvent` (происходит при перемещении, изменении размеров окна, удалении с экрана или выводе на экран), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`FileDialog`	Стандартное окно выбора файла	Конструкторы класса: `FileDialog(`*владелец*`,`*название*`,`*режим*`)` – создается модальное окно с *названием*, *владельцем* (объект класса `Frame`) для открытия (значение параметра *режим* равно `FileDialog.LOAD` или не указано) или сохранения (значение параметра *режим* равно `FileDialog.SAVE`) файла. Обязательным является только первый аргумент. Некоторые методы: `getDirectory()` — возвращает имя выбранной папки, `getFile()` — возвращает имя выбранного файла, `setDirectory(`*папка*`)` — устанавливает *папку* для поиска файла, `setFile(`*файл*`)` — задает имя *файла*. События: `WindowEvent` (происходит при перемещении, изменении размеров окна, удалении с экрана или выводе на экран), `ComponentEvent`, `FocusEvent`, `KeyEvent` и `MouseEvent`
`Canvas`	Пустой компонент	Класс содержит конструктор по умолчанию `Canvas()` и пустую реализацию метода `paint()`. Используется для создания «тяжелых» компонентов пользователя

Основные классы компонентов библиотеки Swing перечислены в табл. 12.3.

Таблица 12.3. Некоторые классы компонентов библиотеки Swing

Класс	Описание
`AbstractButton`	Абстрактный суперкласс для классов-компонентов кнопок в библиотеке Swing
`ButtonGroup`	Класс для создания групп кнопок, например групп переключателей
`ImageIcon`	Класс для инкапсуляции изображения (значка). Конструктору класса при создании объекта передается текстовая строка с именем файла изображения или url-адрес соответствующего ресурса
`JApplet`	Класс для работы с апплетами. Расширяет класс `Applet`
`JButton`	Класс для работы с элементами-кнопками
`JCheckBox`	Класс для работы с элементами-флажками
`JComboBox`	Класс для работы с элементами-раскрывающимися текстовыми списками
`JLabel`	Класс для работы с текстовыми метками
`JRadioButton`	Класс для работы с переключателями
`JScrollPane`	Класс для работ с полосами прокрутки
`JTabbedPane`	Класс для работы с панелями, содержащими вкладки
`JTable`	Класс для работы с таблицами
`JTextField`	Класс для работы с текстовыми полями
`JTree`	Класс для работы с деревьями

Пример использования некоторых классов AWT и обработчиков событий для соответствующих компонентов приведен в следующем разделе.

Создание графика функции

В любой науке столько истины, сколько в ней математики.

И. Кант

Процесс создания приложения с не очень сложным графическим интерфейсом рассмотрим на примере программы, предназначенной для вывода графика функции. Для конкретности будем строить график при положительных значениях аргумента x функции

$$y(x) = \frac{1 + \sin(x)}{1 + |x|}.$$

Левая граница диапазона отображения графика равняется нулю, а правая определяется в окне интерфейса программы. Сама функция возвращает значения

в диапазоне от 0 до 1, что с прикладной точки зрения достаточно удобно. При построении графика предусматривается возможность некоторой дополнительной настройки. Обратимся к программному коду, представленному в листинге 12.5.

Листинг 12.5. Программа для отображения графика функции

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
// Класс фрейма:
class PlotFrame extends Frame{
// Конструктор (аргументы - высота и ширина окна):
PlotFrame(int H,int W){
// Заголовок окна:
setTitle("График функции");
setBounds(100,50,W,H);        // Положение и размер окна
setBackground(Color.GRAY);    // Цвет фона окна
setLayout(null);        // Отключение менеджера размещения элементов
Font f=new Font("Arial",Font.BOLD,11);  // Определение шрифта
setFont(f);                             // Применение шрифта
BPanel BPnl=new BPanel(6,25,W/4,H-30);   // Создание панели с кнопками
add(BPnl);                               // Добавление панели в главное окно
// Панель для отображения графика (создание):
PPanel PPnl=new PPanel(W/4+10,25,3*W/4-15,H-120,BPnl);
// Добавление панели в главное окно:
add(PPnl);
// Третья панель для отображения справки:
HPanel HPnl=new HPanel(W/4+10,H-90,3*W/4-15,85);
// Добавление панели в главное окно:
add(HPnl);
// Регистрация обработчика в окне (закрытие окна):
addWindowListener(new WindowAdapter(){
public void windowClosing(WindowEvent ve){
System.exit(0);}               // Закрытие окна
});
// Регистрация обработчика для первой кнопки:
BPnl.B1.addActionListener(new Button1Pressed(BPnl,PPnl));
// Регистрация обработчика для второй кнопки:
BPnl.B2.addActionListener(new Button2Pressed());
// Регистрация обработчика для флажка вывода сетки:
BPnl.Cb[3].addItemListener(new cbChanged(BPnl));
// Размер окна (фрейма) не изменяется:
setResizable(false);
// Значок для окна программы:
setIconImage(getToolkit().getImage("C:/icons/icon.png"));
```

```
setVisible(true);              // Отображение окна
}}
// Класс панели с кнопками:
class BPanel extends Panel{
// Метки панели:
public Label[] L;
// Группа переключателей панели:
public CheckboxGroup CbG;
// Переключатели панели:
public Checkbox[] Cb;
// Раскрывающийся список:
public Choice Ch;
// Текстовое поле:
public TextField TF;
// Кнопки панели:
public Button B1,B2;
// Конструктор
// (аргументы - координаты и размеры панели):
BPanel(int x,int y,int W,int H){
// Отключение менеджера размещения элементов на панели:
setLayout(null);
setBounds(x,y,W,H);   // Положение и размер панели
setBackground(Color.LIGHT_GRAY);    // Цвет фона панели
// Массив меток:
L=new Label[3];
// Текстовая метка:
L[0]=new Label("Выбор цвета:",Label.CENTER);
// Шрифт для текстовой метки:
L[0].setFont(new Font("Arial",Font.BOLD,12));
// Размеры метки:
L[0].setBounds(5,5,getWidth()-10,30);
// Добавление метки на панель:
add(L[0]);
// Группа переключателей:
CbG=new CheckboxGroup();
Cb=new Checkbox[4];
// Переключатели группы:
Cb[0]=new Checkbox(" красный ",CbG,true); // Красный
Cb[1]=new Checkbox(" синий ",CbG,false);  // Синий
Cb[2]=new Checkbox(" черный ",CbG,false); // Черный
// Флажок вывода сетки:
Cb[3]=new Checkbox(" Сетка ",true);
// Размеры переключателей и флажка и добавление их на панель:
for(int i=0;i<4;i++){
Cb[i].setBounds(5,30+i*25,getWidth()-10,30); // Размер
add(Cb[i]);
```

продолжение

Листинг 12.5 *(продолжение)*

```
}
// Раскрывающийся список выбора цвета для линий сетки:
Ch=new Choice();
// Добавление элемента "Зеленый":
Ch.add("Зеленый");
// Добавление элемента "Желтый":
Ch.add("Желтый");
// Добавление элемента "Серый":
Ch.add("Серый");
// Размер и положение раскрывающегося списка:
Ch.setBounds(20,140,getWidth()-25,30);
// Добавление списка на панель:
add(Ch);
// Вторая текстовая метка:
L[1]=new Label("Интервал по x:",Label.CENTER);
// Шрифт для метки:
L[1].setFont(new Font("Arial",Font.BOLD,12));
// Размер и положение метки:
L[1].setBounds(5,220,getWidth()-10,30);
// Добавление метки на панель:
add(L[1]);
// Третья текстовая метка:
L[2]=new Label("От x=0 до x=",Label.LEFT);
// Размер и положение метки:
L[2].setBounds(5,250,70,20);
// Добавление метки на панель:
add(L[2]);
// Текстовое поле для ввода границы интервала:
TF=new TextField("10");
// Размер и положение поля:
TF.setBounds(75,250,45,20);
// Добавление поля на панель:
add(TF);
// Первая кнопка ("Нарисовать"):
B1=new Button("Нарисовать");
// Вторая кнопка ("Закрыть"):
B2=new Button("Закрыть");
// Размеры и положение первой кнопки:
B1.setBounds(5,getHeight()-75,getWidth()-10,30);
// Размер и положение второй кнопки:
B2.setBounds(5,getHeight()-35,getWidth()-10,30);
add(B1);      // Добавление первой кнопки на панель
add(B2);      // Добавление второй кнопки на панель
}}
// Класс панели для отображения графика:
class PPanel extends Panel{
```

```
// Ссылка на объект реализации графика функции:
public Plotter G;
// Внутренний класс для реализации графика функции:
class Plotter{
// Границы диапазона изменения координат:
private double Xmin=0,Xmax,Ymin=0,Ymax=1.0;
// Состояние флажка вывода сетки:
private boolean status;
// Цвет для линии графика:
private Color clr;
// Цвет для отображения линий сетки:
private Color gclr;
// Конструктор класса
// (аргументы - панель с кнопками и панель для отображения графика):
Plotter(BPanel P){
// Считывание значения текстового поля и преобразование в число:
try{
Xmax=Double.valueOf(P.TF.getText());}
catch(NumberFormatException e){
P.TF.setText("10");
Xmax=10;}
status=P.Cb[3].getState();
// Определение цвета линий сетки:
switch(P.Ch.getSelectedIndex()){
    case 0:
    gclr=Color.GREEN;
    break;
    case 1:
    gclr=Color.YELLOW;
    break;
    default:
    gclr=Color.GRAY;}
// Цвет линии графика:
String name=P.CbG.getSelectedCheckbox().getLabel();
if(name.equalsIgnoreCase(" красный ")) clr=Color.RED;
else if(name.equalsIgnoreCase(" синий ")) clr=Color.BLUE;
else clr=Color.BLACK;
}
// Отображаемая на графике функция:
private double f(double x){
return (1+Math.sin(x))/(1+Math.abs(x));}
// Метод для считывания и запоминания настроек:
public Plotter remember(BPanel P){
return new Plotter(P);}
// Метод для отображения графика и сетки
// (Fig - объект графического контекста):
```

продолжение

Листинг 12.5 *(продолжение)*

```
public void plot(Graphics Fig){
// Параметры области отображения графика:
int H,W,h,w,s=20;
H=getHeight();
W=getWidth();
h=H-2*s;
w=W-2*s;
// Очистка области графика:
Fig.clearRect(0,0,W,H);
// Индексная переменная и количество линий сетки:
int k,nums=10;
// Цвет координатных осей - черный:
Fig.setColor(Color.BLACK);
// Отображение координатных осей:
Fig.drawLine(s,s,s,h+s);
Fig.drawLine(s,s+h,s+w,s+h);
// Отображение засечек и числовых значений на координатных осях:
for(k=0;k<=nums;k++){
Fig.drawLine(s+k*w/nums,s+h,s+k*w/nums,s+h+5);
Fig.drawLine(s-5,s+k*h/nums,s,s+k*h/nums);
Fig.drawString(Double.toString(Xmin+k*(Xmax-Xmin)/nums),s+k*w/nums-5,s+h+15);
Fig.drawString(Double.toString(Ymin+k*(Ymax-Ymin)/nums),s-17,s+h-1-k*h/nums);
}
// Отображение сетки (если установлен флажок):
if(status){
Fig.setColor(gclr);
// Отображение линий сетки:
for(k=1;k<=nums;k++){
Fig.drawLine(s+k*w/nums,s,s+k*w/nums,h+s);
Fig.drawLine(s,s+(k-1)*h/nums,s+w,s+(k-1)*h/nums);
}}
// Отображение графика:
Fig.setColor(clr);     // Установка цвета линии
// Масштаб на один пиксель по каждой из координат:
double dx=(Xmax-Xmin)/w,dy=(Ymax-Ymin)/h;
// Переменные для записи декартовых координат:
double x1,x2,y1,y2;
// Переменные для записи координат в окне отображения графика:
int h1,h2,w1,w2;
// Начальные значения:
x1=Xmin;
y1=f(x1);
w1=s;
h1=h+s-(int)Math.round(y1/dy);
// Шаг в пикселях для базовых точек:
int step=5;
```

```
// Отображение базовых точек и соединение их линиями:
for(int i=step;i<=w;i+=step){
x2=i*dx;
y2=f(x2);
w2=s+(int)Math.round(x2/dx);
h2=h+s-(int)Math.round(y2/dy);
// Линия:
Fig.drawLine(w1,h1,w2,h2);
// Базовая точка (квадрат):
Fig.drawRect(w1-2,h1-2,4,4);
// Новые значения для координат:
x1=x2;
y1=y2;
w1=w2;
h1=h2;}
}}
// Конструктор панели
// (аргументы - координаты и размеры панели,
// а также ссылка на панель с кнопками):
PPanel(int x,int y,int W,int H,BPanel P){
// Создание объекта реализации графика функции:
G=new Plotter(P);
// Цвет фона панели:
setBackground(Color.WHITE);
// Размер и положение панели:
setBounds(x,y,W,H);
}
// Переопределение метода перерисовки панели:
public void paint(Graphics g){
// При перерисовке панели вызывается метод
// для отображения графика:
G.plot(g);
}}
// Класс для панели справки:
class HPanel extends Panel{
// Метка:
public Label L;
// Текстовая область:
public TextArea TA;
// Конструктор создания панели
// (аргументы - координаты и размеры панели):
HPanel(int x,int y,int W,int H){
// Цвет фона панели:
setBackground(Color.LIGHT_GRAY);
// Размер и положение панели:
setBounds(x,y,W,H);
```

продолжение

Листинг 12.5 *(продолжение)*

```
// Отключения менеджера размещения компонентов панели:
setLayout(null);
// Метка для панели справки:
L=new Label("СПРАВКА",Label.CENTER);
// Размер и положение метки:
L.setBounds(0,0,W,20);
// Добавление метки на панель:
add(L);
// Текстовая область для панели справки:
TA=new TextArea(" График функции y(x)=(1+sin(x))/(1+|x|)");
// Шрифт для текстовой области:
TA.setFont(new Font("Serif",Font.PLAIN,15));
// Размер и положение текстовой области:
TA.setBounds(5,20,W-10,60);
// Область недоступна для редактирования:
TA.setEditable(false);
// Добавление текстовой области на панель справки:
add(TA);
}}
// Класс обработчика для первой кнопки:
class Button1Pressed implements ActionListener{
// Панель с кнопками:
private BPanel P1;
// Панель для отображения графики:
private PPanel P2;
// Конструктор класса (аргументы - панели):
Button1Pressed(BPanel  P1,PPanel P2){
this.P1=P1;
this.P2=P2;}
// Метод для обработки щелчка на кнопке:
public void actionPerformed(ActionEvent ae){
// Обновление параметров (настроек) для отображения графика:
P2.G=P2.G.remember(P1);
// Реакция на щелчок (прорисовка графика):
P2.G.plot(P2.getGraphics());
}}
// Класс обработчика для второй кнопки:
class Button2Pressed implements ActionListener{
// Метод для обработки щелчка на кнопке:
public void actionPerformed(ActionEvent ae){
// Реакция на щелчок:
System.exit(0);
}}
// Класс обработчика для флажка вывода сетки:
class cbChanged implements ItemListener{
```

```
// Список выбора цвета для сетки:
private Choice ch;
// Конструктор класса (аргумент - панель с кнопками):
cbChanged(BPanel P){
this.ch=P.Ch;}
// Метод для обработки изменения состояния флажка:
public void itemStateChanged(ItemEvent ie){
// Реакция на изменение состояния флажка:
ch.setEnabled(ie.getStateChange()==ie.SELECTED);
}}
// Класс с главным методом программы:
class PlotDemo{
public static void main(String args[]){
// Создание окна:
new PlotFrame(400,500);}
}
```

При запуске программы открывается графическое окно, представленное на рис. 12.9.

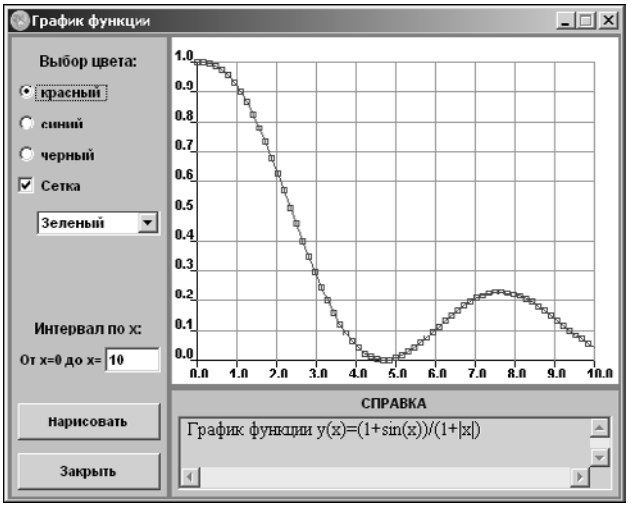

Рис. 12.9. Графическое окно приложения

Прежде чем приступить к анализу программного кода, кратко рассмотрим основные принципы, заложенные в основу представленной программы, и кратко опишем ее функциональные возможности. В частности, в результате запуска программы открывается окно, состоящее из трех панелей. Первая панель с кнопками и другими элементами управления расположена в левой части главного окна и занимает примерно четверть ширины главного окна. Правую часть главного окна делят две панели: большая белая вверху предназначена для отображения графика, а панель поменьше внизу содержит краткую справку (фактически,

выражение для отображаемой на графике функции). Для каждой из этих трех панелей создается собственный класс (путем наследования класса `Panel`).

Вся функциональность приложения «спрятана» в компоненты, размещенные на первой панели (в левой части главного окна). Эта панель, в частности, содержит группу Выбор цвета с тремя переключателями (красный, синий и зеленый), предназначенными для выбора цвета кривой графика. Также там есть флажок Сетка, установив который можно включить режим вывода координатной сетки на графике. При установленном флажке снизу доступен раскрывающийся список с пунктами Зеленый, Желтый и Серый для выбора цвета линий сетки. Если флажок Сетка сброшен, список недоступен.

В поле в нижней части панели вводится значение верхней границы для диапазона изменения аргумента функции. По умолчанию значение в поле равно 10. Кнопка Нарисовать предназначена для отображения графика при заданных параметрах, а кнопка Закрыть служит для закрытия окна.

На рис. 12.10 представлено окно приложения при выводе графика функции без линий сетки на диапазоне значений аргумента функции от 0 до 30.

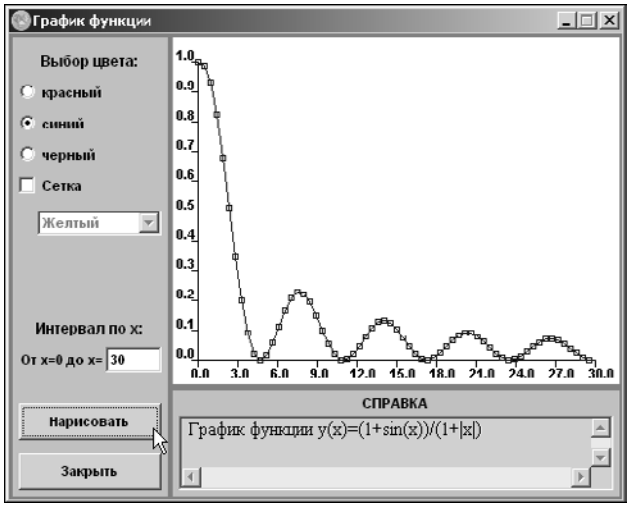

Рис. 12.10. Графическое окно приложения

Панель справки содержит статическую информацию, которая не меняется в процессе выполнения программы.

Далее рассмотрим, как описанные возможности реализованы через программный код (см. листинг 12.5). В первую очередь обращаем внимание на то, что программа состоит из нескольких классов. Среди них: класс `PlotFrame` предназначен для реализации главного окна программы, по одному классу — для каждой из трех панелей (то есть всего три класса: панель с кнопками `BPanel`, панель для отображения графика `PPanel` и справочная панель `HPanel`), класс `Button1Pressed` — для обработки щелчка на первой кнопке, класс `Button2Pressed` — для обработки

щелчка на второй кнопке, класс cbChanged — для обработки изменения состояния флажка вывода сетки, внутренний класс Plotter (в классе PPanel) — для отображения графика функции и запоминания заданных параметров, а также класс PlotDemo с главным методом программы.

Класс панели BPanel создается на основе класса Panel путем наследования. Класс имеет в качестве полей массив меток L, ссылку на группу переключателей CbG, массив Cb элементов типа Checkbox (три из них являются переключателями и принадлежат группе CbG, а один — это флажок), раскрывающийся список Ch, текстовое поле TF и две кнопки B1 и B2. Это ссылки на соответствующие объекты панели — сами объекты создаются при вызове конструктора класса. Аргументами конструктору передаются координаты левого верхнего угла панели (по отношению к контейнеру — в данном случае к главному окну, в которое будет добавляться панель), а также ширина и высота панели. Напомним, что точка начала отсчета координат в окне находится в левом верхнем углу. По горизонтали координата вычисляется в пикселях слева направо, по вертикали — сверху вниз.

В конструкторе класса BPanel командой setLayout(null) отключается менеджер компоновки элементов. Размер и положение панели задаются командой setBounds(x,y,W,H). При этом методу setBounds() аргументами передаются аргументы конструктора. Цвет панели задается командой setBackground(Color.LIGHT_GRAY). Здесь использовано статическое поле LIGHT_GRAY класса Color для определения цвета (светло-серый).

Командой L=new Label[3] создается массив из трех меток, и ссылка на этот массив присваивается в качестве значения полю L класса. Обращаем внимание читателя, что элементами массива L являются ссылки на метки. Сами метки нужно создавать отдельно. Поэтому, например, командой L[0]=new Label("Выбор цвета:",Label.CENTER) создается первая метка. Текстом метки является фраза "Выбор цвета:", и этот текст размещается по центру в области метки (использовано статическое поле CENTER класса Label). Шрифт для текста метки задается командой L[0].setFont(new Font("Arial",Font.BOLD,12)). В этой команде аргументом методу setFont() передается анонимный объект класса Font, который создается командой new Font("Arial",Font.BOLD,12). Размеры метки задаются командой L[0].setBounds(5,5,getWidth()-10,30). Здесь использован метод getWidth() для получения значения ширины панели. Наконец, командой add(L[0]) метка добавляется на панель. Созданная метка предназначена для группы переключателей, которые размещаются под меткой.

Группа переключателей создается командой CbG=new CheckboxGroup(). Командой Cb=new Checkbox[4] создается массив из четырех элементов — ссылок на объекты класса Checkbox. Первые три объекта — это переключатели для выбора цвета кривой графика, четвертый — флажок для выбора режима отображения сетки. Переключатели группы создаются командами:

```
Cb[0]=new Checkbox(" красный ",CbG,true)
Cb[1]=new Checkbox(" синий ",CbG,false)
Cb[2]=new Checkbox(" черный ",CbG,false).
```

Аргументами конструктору класса Checkbox() передаются соответственно отображаемый возле переключателя текст, группа, к которой принадлежит переключатель, и состояние переключателя (true, если переключатель установлен, и false, если сброшен). Таким образом, при создании окна установлен переключатель, отвечающий за отображение графика красным цветом. Флажок создается командой Cb[3]=new Checkbox(" Сетка ",true). Поскольку флажок к группе переключателей не принадлежит, второй аргумент пропускается — указываются только текст возле флажка и его состояние (установлен или сброшен). По умолчанию используется режим отображения координатной сетки. Поскольку три переключателя и флажок располагаются упорядоченно один под другим, процедуры определения размера и положения переключателей и флажка, а также добавления их на панель реализуются в рамках цикла.

Объект раскрывающегося списка создается командой Ch=new Choice(). Вначале список пустой. Добавление элементов в список осуществляется командами Ch.add("Зеленый"), Ch.add("Желтый") и Ch.add("Серый"). Пункты добавляются в список один за другим в соответствии с порядком следования команд. Размер и положение списка на панели (и вообще в контейнере) задается командой Ch.setBounds(20,140,getWidth()-25,30). Левый верхний угол элемента списка находится на расстоянии 20 пикселей вправо от левого верхнего угла панели и на 140 пикселей вниз. Высота поля составляет 30 пикселей, а ширина на 25 пикселей меньше ширины панели. Добавляется список на панель командой add(Ch).

Далее добавляются еще две текстовые метки L[1] и L[2] (надписи "Интервал по x" и "От x=0 до x="). Метки добавляются практически так же, как и первая метка на панели, поэтому хочется верить, что особых комментариев эта часть кода не требует.

Текстовое поле создается командой TF=new TextField("10"). При вызове конструктора класса TextField аргументом указано текстовое значение "10". Именно оно отображается в поле по умолчанию. Размеры и положение поля на панели задаются командой TF.setBounds(75,250,45,20) (поле шириной 45 пикселей и высотой 20 пикселей расположено на 75 пикселей вправо и 250 пикселей вниз от верхнего левого угла панели), а добавляется поле на панель с помощью команды add(TF).

Наконец, две кнопки (объекты класса Button) создаются командами B1=new Button("Нарисовать") и B2=new Button("Закрыть"). Текст на кнопках передается аргументом конструктору класса Button. Размеры и положение кнопок задаются методом setBounds(), а добавляются кнопки на панель с помощью метода add().

Достаточно прост класс HPanel для панели справки. Он создается на основе класса Panel и имеет метку (поле Label L) и текстовую область (поле TextArea TA). Соответствующие объекты создаются в конструкторе класса. При этом используются методы и приемы, описывавшиеся ранее. Обращаем внимание, что для текстовой области командой TA.setEditable(false) устанавливается режим, не позволяющий редактировать ее содержимое. Таким образом, элементы справочной панели (метка и текстовая область) являются статическими. В принципе,

можно было бы несколько усовершенствовать код так, чтобы на справочной панели отображались и текущие параметры режима вывода графика. Но это задание оставим читателю.

С формальной точки зрения не очень сложно описан и класс `PPanel` для панели вывода графика. Класс создается путем наследования класса `Panel`. У этого класса есть всего одно поле `G` — объект класса `Plotter`. Как уже отмечалось, это внутренний класс. Через него реализована процедура прорисовки графика функции, поэтому остановимся на нем подробнее.

У класса четыре поля типа `double`: `Xmin` (значение `0`), `Xmax`, `Ymin` (значение `0`) и `Ymax` (значение `1.0`). Эти поля определяют границы диапазонов значений по каждой из координатных осей. Три поля имеют начальные значения и в процессе выполнения программы не меняются. Значение поля `Xmax` вычисляется на основе состояния элементов управления рабочего окна приложения. Поле логического типа `status` предназначено для записи состояния флажка, задающего режим вывода сетки. Поля `clr` и `gclr` являются объектами класса `Color` и предназначены для записи значений цвета кривой графика функции и цвета линий сетки соответственно.

Конструктору класса `Plotter` передается объект класса `BPanel`, то есть панель, содержащая элементы управления. Собственно, в конструкторе выполняется считывание состояния элементов управления, переданных аргументом конструктору. В первую очередь считывается значение текстового поля со значением верхней границы интервала значений для аргумента функции. При этом учитываем, что в поле может быть введено (по ошибке или специально) не число. Поэтому соответствующий фрагмент кода заключен в блок `try`. Командой `Xmax=Double.valueOf(P.TF.getText())` присваивается значение полю `Xmax`: сначала методом `getText()` объекта поля `TF` панели `P` считывается содержимое поля, а затем методом `valueOf()` класса-оболочки `Double` текстовое представление числа преобразуется в формат `double`. При некорректном значении в поле или его отсутствии возникает ошибка (исключение типа `NumberFormatException`). Это исключение обрабатывается в блоке `catch`. В частности, командой `P.TF.setText("10")` в поле заносится значение 10. Также командой `Xmax=10` соответствующее значение присваивается полю `Xmax`.

Полю `status` значение присваивается командой `status=P.Cb[3].getState()`. В этом случае использован метод `getState()` объекта флажка `Cb[3]` (который, в свою очередь, является полем объекта панели `P`). В результате поле `status` получает значение `true`, если флажок установлен, и `false`, если сброшен.

Командой `P.Ch.getSelectedIndex()` возвращается индекс выбранного элемента в раскрывающемся списке `Ch`. Это значение используется в инструкции `switch()` для определения цвета линий координатной сетки. Индексация элементов раскрывающегося списка начинается с нуля. Индекс 0 соответствует зеленому цвету (значение `Color.GREEN`), индекс 1 — желтому цвету (значение `Color.YELLOW`), а индекс 2 — серому цвету (значение `Color.GRAY`). Соответствующее значение записывается в переменную `gclr`. Несколько по иному определяется

цвет кривой для графика функции. Для этого командой `String name=P.CbG.getSelectedCheckbox().getLabel()` объявляется текстовая переменная `name` и в качестве значения этой переменной присваивается текст установленного пользователем переключателя. Объект этого переключателя в группе `CbG` возвращается методом `getSelectedCheckbox()`. Из этого объекта вызывается метод `getLabel()`, которым в качестве значения возвращается текст переключателя. Затем с помощью вложенных условных инструкций проверяется считанное значение, и полю `clr` присваивается соответствующее значение (`Color.RED`, `Color.BLUE` или `Color.BLACK`).

Также в классе `Plotter` описан метод `f()` с одним аргументом типа `double`. Этот метод определяет функциональную зависимость, отображаемую на графике.

Метод класса `remember()` с аргументом — объектом класса `BPanel` в качестве значения возвращает объект класса, созданный на основе панели, переданной в качестве аргумента методу. Этот метод используется в тех случаях, когда необходимо запомнить состояние элементов управления панели, чтобы на его основе можно было нарисовать картинку с графиком функции.

Отображение графика функции и сопутствующих ему атрибутов реализуется с помощью метода `plot()` класса `Plotter`. В этом методе аргументом является объект `Fig` класса `Graphics`. Это графический контекст — объект, через который реализуется графическое представление компонента. Обычно для создания графического контекста компонента (в данном случае панели) используется метод `getGraphics()` этого компонента.

Командами `H=getHeight()` и `W=getWidth()` определяются размеры панели, в которой будет отображаться график. Командами `h=H-2*s` и `w=W-2*s` определяются фактические размеры области отображения графика (при этом переменная `s` определяет ширину поля вокруг графика функции).

Командой `Fig.clearRect(0,0,W,H)` выполняется очистка области панели. Это необходимо делать для того, чтобы при выводе графика старое изображение убиралось. Метод вызывается из объекта графического контекста компонента, а аргументами ему передаются координаты левой верхней точки области очистки и ее ширина и высота.

Сначала отображаются координатные оси. Для этого командой `Fig.setColor(Color.BLACK)` устанавливается черный цвет линий, а командами `Fig.drawLine(s,s,s,h+s)` и `Fig.drawLine(s,s+h,s+w,s+h)` непосредственно отображаются координатные оси. Линии (точнее, отрезки прямых) выводятся с помощью метода `drawLine()`, аргументами которому передаются координаты начальной и конечной точек отрезка. Засечки и текстовые обозначения координатных осей отображаются в рамках цикла. Вывод текста в графическом формате осуществляется методом `drawString()`. Аргументом метода указывается отображаемый текст и координаты для вывода этого текста. Для преобразования действительных чисел в формат текстовой строки используется метод `toString()` класса `Double`. Переменная `nums` определяет количество линий сетки.

Если установлен флажок вывода сетки (значение переменной status равно true), отображается сетка. Для этого командой Fig.setColor(gclr) задается цвет линий сетки, а затем в цикле прорисовываются линии сетки.

Для отображения кривой задается цвет линии графика (командой Fig.setColor(clr)). График строится по базовым точкам. Расстояние по горизонтальной оси (в пикселях) между базовыми точками определяется переменной step. Эти точки соединяются линиями, а также выделяются квадратами. В последнем случае вызывается метод drawRect(), аргументами которому передаются координаты левой верхней точки отображаемого прямоугольника и его размеры (ширина и высота).

На этом описание внутреннего класса Plotter завершается.

Конструктору класса-контейнера PPanel в качестве аргументов передаются координаты верхней левой точки панели во фрейме, ее размеры (ширина и высота), а также объект P класса BPanel (то есть панель с элементами управления). В конструкторе командой G=new Plotter(P) создается новый объект класса Plotter и записывается в поле G. Белый цвет фона устанавливается командой setBackground(Color.WHITE). Границы панели определяются с помощью метода setBounds(), которому аргументом передаются первые четыре аргумента конструктора.

Также в классе PPanel переопределяется метод paint(). Этот метод автоматически вызывается при перерисовке компонентов, например при разворачивании свернутого окна. Если метод не переопределить, то в конечном варианте программы сворачивание или разворачивание окна будет приводить к исчезновению графика функции. Аргументом методу передается графический контекст компонента. В данном случае метод переопределен так, что при его вызове выполнятся команда G.plot(g) (здесь g — аргумент метода paint(), то есть графический контекст перерисовываемого компонента), в результате выводится график функции.

Класс фрейма PlotFrame создается на основе класса Frame. Конструктору класса передаются два числа W и H — размеры окна. Собственно, весь класс состоит фактически из конструктора. В частности, командой setTitle("График функции") в конструкторе задается название окна (отображается в строке заголовка окна). Положение окна на экране и его размеры задаются командой setBounds(100,50,W,H). Серый цвет фона устанавливается с помощью команды setBackground(Color.GRAY), а менеджер размещения компонентов отключается командой setLayout(null). Командой Font f=new Font("Arial",Font.BOLD,11) создается шрифт (объект f класса Font), и этот шрифт устанавливается как шрифт фрейма с помощью команды setFont(f). Также в главное окно добавляются (а сначала создаются) три панели. Панель с кнопками создается командой BPanel BPnl=new BPanel(6,25,W/4,H-30). Добавляется в главное окно панель командой add(BPnl). Панель для отображения графика создается командой PPanel PPnl=new PPanel(W/4+10,25,3*W/4-15,H-120,BPnl), а добавляется в окно командой add(PPnl). Панель для вывода справки создается командой HPanel HPnl=new HPanel(W/4+10,H-90,3*W/4-15,85), а добавляется в окно командой add(HPnl).

Кроме этого, в окне регистрируются четыре обработчика событий: для щелчка на системной кнопке закрытия окна, для каждой из двух кнопок, а также для флажка вывода сетки. В частности, обработчик для кнопки закрытия окна регистрируется методом `addWindowListener()` (соответствующая команда уже рассматривалась).

Регистрация обработчика для первой кнопки выполняется командой `BPnl.B1.addActionListener(new Button1Pressed(BPnl,PPnl))`, регистрация обработчика для второй кнопки — командой `BPnl.B2.addActionListener(new Button2Pressed())`, регистрация обработчика для флажка вывода сетки — командой `BPnl.Cb[3].addItemListener(new cbChanged(BPnl))`. Соответствующие классы обработчиков событий описаны далее. Объекты для классов обработчиков событий предаются аргументами методам регистрации этих обработчиков.

Также для удобства делаем главное окно не масштабируемым, для чего используем команду `setResizable(false)`. Значок для окна создаваемого приложения задается командой

```
setIconImage(getToolkit().getImage("C:/icons/icon.png"))
```

Аргументом методу `setIconImage()` передается изображение значка (объект класса `Image`). Объект изображения создается методом `getImage()`. Аргументом этому методу передается текстовая строка с полным именем графического изображения (файла), а вызывается метод из объекта класса `Toolkit`, который возвращается методом `getToolkit()`.

Наконец, главное окно отображается командой `setVisible(true)`.

Класс обработчика для первой кнопки `Button1Pressed` создается как реализация интерфейса `ActionListener`. У класса есть поля-объекты: панель `P1` (класса `BPanel`) и панель `P2` (класса `PPanel`). Такие же аргументы имеет и конструктор класса. Они определяют значения упомянутых полей. Метод для обработки щелчка на кнопке `actionPerformed()` состоит из команды обновления параметров вывода графика `P2.G=P2.G.remember(P1)` и команды реакции на щелчок рисования графика `P2.G.plot(P2.getGraphics())`. В первом случае фактически происходит обновление объекта `G`, являющегося полем объекта панели `P2`. Считывание параметров выполняется с помощью метода `remember()`, который вызывается из объекта `G` и аргументом которому предается объект панели с элементами управления.

Класс обработчика для второй кнопки `Button2Pressed` также создается на основе интерфейса `ActionListener`. В конструкторе переопределен метод `actionPerformed()`, в котором имеется всего одна команда `System.exit(0)`, которой завершается работа программы (и, соответственно, закрывается главное окно).

Поскольку от состояния флажка качественно зависит, как отображается график (с сеткой или без нее), необходимо предусмотреть реакцию на изменение этого состояния. Для этого на основе интерфейса `ItemListener` создаем класс `cbChanged` обработчика состояния флажка вывода сетки. Класс имеет поле `ch` класса `Choice` (раскрывающийся список) и конструктор, которым этому полю присваивается

значение. Аргументом конструктору передается объект для панели с элементами управления, в том числе с флажком установки режима вывода сетки. Ссылка на этот флажок присваивается в качестве значения полю ch.

Класс cbChanged переопределяет метод для обработки изменения состояния флажка itemStateChanged(). Реакция на изменение состояния флажка определяется командой

```
ch.setEnabled(ie.getStateChange()==ie.SELECTED)
```

В ней для проверки этого состояния использована константа SELECTED (флажок установлен). Метод getStateChange() возвращает в качестве результата значение SELECTED, если флажок установлен, и DESELECTED, если сброшен. И метод, и константа вызываются через объект ie класса события ItemEvent. Методом setEnabled(), вызываемым из объекта ch, состояние (доступность) этого объекта устанавливается такой же, как и состояние флажка. Таким образом, при изменении состояния флажка происходит автоматическое считывание этого состояния и в соответствии с этим определяется доступность или недоступность раскрывающегося списка для выбора цвета линий координатной сетки.

В главном методе программы в классе PlotDemo командой new PlotFrame(400,500) создается анонимный объект для главного окна программы. Аргументами конструктору класса PlotFrame() передаются размеры создаваемого окна.

Еще раз обращаем внимание читателя, что в представленном примере задача состоит не только в том, чтобы нарисовать график, но и чтобы «запомнить» картинку. Последнее нужно для того, чтобы ее заново нарисовать при перерисовке компонента. В данном случае запоминается не сама картинка, а параметры, на основе которых она создавалась. Для этого описывается специальный внутренний класс. При перерисовке компонента картинка отображается заново.

Рассмотренный код является иллюстративным и служит скорее для демонстрации того, как можно в принципе обрабатывать графические компоненты, чем как пример оптимального кода.

Калькулятор

— Чем желают заняться состоятельные кроты?
— Мы пока посчитаем.

Из м/ф «Дюймовочка»

Следующий пример иллюстрирует возможности использования библиотеки Swing для создания приложения с графическим интерфейсом на основе «легких» компонентов. В данном случае это классика жанра — программа-калькулятор. Правда, один из самых простых ее вариантов. Программный код приведен в листинге 12.6.

Листинг 12.6. Калькулятор

```
// Подключение пакетов:
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// Класс главного окна:
class CalculatorFrame extends JFrame{
// Конструктор класса:
CalculatorFrame(){
// Размеры окна:
int w=270,h=240;
// Заголовок окна:
setTitle("Калькулятор");
// Установка размеров и положения окна:
setBounds(100,100,w,h);
// Создание панели с кнопками и полем:
CPanel panel=new CPanel(w,h);
// Добавление панели в окно:
add(panel);
// Режим запрета изменения размеров окна:
setResizable(false);
// Отображение окна:
setVisible(true);
// Обработка щелчка на системной кнопке закрытия окна:
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}}
// Класс панели:
class CPanel extends JPanel{
// Текстовое поле:
public JTextField TF;
// Обработчик щелчка на кнопке:
private BtnAction BtnPressed;
// Конструктор класса (аргументы - размеры панели):
CPanel(int W,int H){
// Размеры кнопок и отступы:
int w=W/5,h=H/8,sx=w/5,sy=h/3;
// Отключение менеджера компоновки:
setLayout(null);
// Установка положения и размеров панели:
setBounds(0,0,W,H);
// Создание текстового поля:
JTextField TF=new JTextField();
// Выравнивание текста в поле по правому краю:
TF.setHorizontalAlignment(JTextField.RIGHT);
// Положение и размер поля:
TF.setBounds(sx,sy,2*sx+3*w,h);
// Отмена возможности редактирования поля:
```

```
TF.setEditable(false);
// Добавление поля на панель:
add(TF);
// Создание обработчика щелчка на кнопке:
BtnPressed=new BtnAction(TF);
// Список названий кнопок:
String[] BtnTxt={"1","2","3","+","4","5","6","-
","7","8","9","/","0",".","=","*"};
// Создание кнопок и добавление их на панель:
for(int i=0;i<BtnTxt.length;i++){
addBtn(sx+(w+sx)*(i%4),(2*sy+h)+(sy+h)*(i/4),w,h,BtnTxt[i],BtnPressed);}
// Создание кнопки сброса параметров:
JButton BtnC=new JButton("C");
// Размер и положение кнопки:
BtnC.setBounds(4*sx+3*w,sy,w,h);
// Добавление обработчика для кнопки:
BtnC.addActionListener(BtnPressed);
// Режим отсутствия выделения названия кнопки при активации:
BtnC.setFocusPainted(false);
// Красный цвет для названия кнопки:
BtnC.setForeground(Color.RED);
// Добавление кнопки на панель:
add(BtnC);}
// Метод для создания и добавления кнопок
// (аргументы - положение и размер кнопки, название и обработчик щелчка):
void addBtn(int i,int j,int w,int h,String txt,ActionListener AcList){
// Создание кнопки:
JButton b=new JButton(txt);
// Размер и положение кнопки:
b.setBounds(i,j,w,h);
// Режим отсутствия выделения названия кнопки при активации:
b.setFocusPainted(false);
// Добавление обработчика для кнопки:
b.addActionListener(AcList);
// Добавление кнопки на панель:
add(b);}
}
// Класс обработчика щелчка на кнопке:
class BtnAction implements ActionListener{
// Текстовое поле для вывода информации:
public JTextField TF;
// Индикатор состояния ввода числа:
private boolean start;
// Индикатор состояния ввода десятичной точки:
private boolean point;
// Текстовое представление последнего введенного оператора:
```

продолжение

Листинг 12.6 *(продолжение)*

```
private String cmnd;
// Поле для записи промежуточного результата:
private double result;
// Метод для сброса параметров:
private void onStart(){
start=true;
point=true;
cmnd="C";
result=0;
TF.setText("0.0");}
// Метод для вычисления результата последней операции:
private void calc(){
// Введенное в поле число:
double x;
x=Double.parseDouble(TF.getText());
// Вычисление результата:
if(cmnd.equals("*")) result*=x;
else if(cmnd.equals("/")) result/=x;
else if(cmnd.equals("-")) result-=x;
else if(cmnd.equals("+")) result+=x;
else result=x;
// Заполнение текстового поля:
TF.setText(Double.toString(result));}
// Конструктор класса (аргумент - текстовое поле):
BtnAction(JTextField TF){
this.TF=TF;
onStart();}
// Реакция на щелчок на кнопке:
public void actionPerformed(ActionEvent ae){
// Считывание текста на кнопке:
String str=ae.getActionCommand();
// Проверка вариантов:
if(str.equals("C")){// Кнопка сброса значений
onStart();
return;}
// Вычисление результата:
if(str.equals("+")|str.equals("-")|str.equals("*")|str.equals("/")|str.
equals("=")){
calc();
cmnd=str;
start=true;
point=true;
return;}
// Ввод числа:
if(start){// Начало ввода числа
    if(str.equals(".")){// Ввод точки в начале ввода числа
```

```
    TF.setText("0.");
    point=false;
    start=false;
    return;}
    else{// Ввод цифры в начале ввода числа
    TF.setText(str);
    start=false;
    return;}
}
else{// Продолжение ввода числа
if(str.equals(".")){// Попытка ввести точку
    str=point?str:"";
    point=false;}
// Добавление цифры к числу:
   // Незначащий первый ноль:
if(TF.getText().equals("0")&!str.equals(".")) TF.setText(str);
else TF.setText(TF.getText()+str);}
}}
// Класс с главным методом программы:
class MyCalculator{
public static void main(String[] args){
// Создание окна:
new CalculatorFrame();
}}
```

На рис. 12.11 представлено графическое окно программы.

Рис. 12.11. Графическое окно программы-калькулятора

Как видим, калькулятор предназначен для выполнения только базовых арифметических операций, таких как сложение, вычитание, умножение и деление. Окно калькулятора представляет собой упорядоченный набор одинаковых по размеру кнопок и поле, используемое для отображения результатов вычислений.

Что касается непосредственно графического окна приложения, то создается оно достаточно просто. Основные проблемы связаны с обработкой манипуляций пользователя различными кнопками в окне.

Окно калькулятора реализуется в виде объекта класса `CalculatorFrame`, который создается наследованием класса `JFrame` библиотеки Swing. Класс состоит, фактически, из конструктора, в котором командой `setTitle("Калькулятор")` задается название окна. Положение окна и его размеры устанавливаются с помощью метода `setBounds()`. Далее создается панель — объект класса `CPanel` (который, в свою очередь, наследует класс `JPanel`). Методом `add()` панель добавляется в окно. С помощью команды `setResizable(false)` устанавливается режим запрета изменения размеров окна, а само окно отображается посредством команды `setVisible(true)`. Наконец, способ обработки щелчка на системной кнопке закрытия окна определяется командой `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`.

Теперь рассмотрим класс `CPanel`, поскольку именно через объект этого класса реализуется все содержимое окна приложения. Как уже отмечалось, класс создается наследованием класса `JPanel` из библиотеки Swing. У класса `CPanel` определены два поля: открытое поле `TF` — объект класса `JTextField` (текстовое поле) и закрытое поле `BtnPressed` — объект класса `BtnAction` (класс обработчика события щелчка на кнопке). Сразу отметим, что хотя в данном случае имеется в общей сложности 17 кнопок, все они используют один и тот же обработчик, и ссылка на этот объект записывается в поле `BtnPressed`. Класс `BtnAction` рассматривается далее.

Конструктору класса панели в качестве аргументов передаются размеры панели. Командой `setLayout(null)` отключается менеджер компоновки элементов (кнопки размещаются на панели в ручном режиме с явным указанием их положения). Как и для окна приложения, положение и размеры панели устанавливаются методом `setBounds()` (положение панели определяется в пределах окна приложения). Командой `JTextField TF=new JTextField()` создается текстовое поле `TF`. Данное текстовое поле предназначено для вывода результатов вычислений. Для того чтобы текст в поле выравнивался по правому краю, используется инструкция `TF.setHorizontalAlignment(JTextField.RIGHT)`. Положение и размер поля задаются методом `setBounds()`, который вызывается из объекта `TF`. Командой `TF.setEditable(false)` поле переводится в режим невозможности редактирования (иначе пришлось бы писать обработчики событий нажатия клавиш на клавиатуре). На панель поле добавляется вызовом команды `add(TF)`.

Объект для обработчика щелчка на кнопке создается командой `BtnPressed=new BtnAction(TF)`. Этой же командой ссылка на объект обработчика записывается в поле `BtnPressed`. Аргументом конструктору класса `BtnAction` передается созданное на предыдущем этапе текстовое поле `TF`. В обработчике это поле используется для вывода результатов вычислений и считывания введенных в поле значений (см. далее).

Командой `String[] BtnTxt={"1","2","3","+","4","5","6","-","7","8","9","/","0",".","=","*"}` создается текстовый массив с названиями для кнопок на панели (для всех, кроме кнопки сброса значений). Сами кнопки создаются и размещаются на панели в рамках цикла с индексной переменной `i`, которая получает значения от 0 до `BtnTxt.length-1` включительно. Основу цикла составляет команда вызова метода `addBtn()` (описание метода см. далее), аргументами которому передаются координаты новой создаваемой кнопки, ее размеры, название

и обработчик события, регистрируемого для кнопки. Обращаем внимание, что ссылки на создаваемые кнопки никуда не записываются. Другими словами, если бы при таком подходе впоследствии понадобилось изменить атрибуты какой-то кнопки, то «отловить» ее было бы крайне проблематично.

Отдельно создается кнопка сброса параметров, для чего используется команда `JButton BtnC=new JButton("C")`. В данном случае `BtnC` — это локальная переменная, которая создается в конструкторе. После вызова конструктора она удаляется, но кнопка при этом остается. Размеры и положение кнопки задаются вызовом из объекта `BtnC` метода `setBounds()`. Обработчик для кнопки добавляется командой `BtnC.addActionListener(BtnPressed)`. Для того чтобы при активации кнопки не появлялась рамка выделения вокруг текстового названия кнопки, командой `BtnC.setFocusPainted(false)` отключаем этот режим. Кроме того, для кнопки сброса значений устанавливаем красный цвет отображения названия кнопки командой `BtnC.setForeground(Color.RED)`. Наконец, добавляется кнопка на панель командой `add(BtnC)`.

Метод `addBtn()` класса `CPanel` предназначен для создания и размещения кнопок на панели. Непосредственно создается очередная кнопка командой `JButton b=new JButton(txt)`, при этом аргументом конструктору класса `JButton` передается текстовое название кнопки — пятый аргумент метода `addBtn()`. Размер и положение кнопки определяются командой `b.setBounds(i,j,w,h)`, в которой использованы первые четыре аргумента метода `addBtn()`. Переход в режим отсутствия выделения названия кнопки при активации осуществляется командой `b.setFocusPainted(false)`. Командой `b.addActionListener(AcList)` для кнопки добавляется обработчик (шестой аргумент метода `addBtn()`). Хотя, откровенно говоря, учитывая то обстоятельство, что обработчик всего один, можно было бы его использовать напрямую, без передачи аргументом методу `addBtn()`. При желании читатель может внести в программный код необходимые изменения самостоятельно. Добавляется кнопка на панель командой `add(b)`.

На этом, собственно, все, что касается организации графического интерфейса как такового, заканчивается. Прочий код относится в основном к реализации взаимодействия между разными элементами интерфейса. Основная нагрузка лежит на методах класса обработчика `BtnAction`, который создается расширением интерфейса `ActionListener`. В этом классе объявлено, как уже упоминалось, текстовое поле `JTextField TF` и ряд закрытых «технических» полей. В частности, это логическое поле `start`, играющее роль индикатора начала ввода числа в текстовое поле `TF`, логическое поле `point`, используемое в качестве индикатора при вводе десятичной разделительной точки, текстовое поле `cmnd`, в которое записывается текстовый символ выполняемой операции, а также поле `double result`, предназначенное для записи промежуточных результатов вычислений. Также класс имеет несколько закрытых методов. Среди них метод `onStart()`, используемый для сброса параметров: при вызове метода значение индикаторов `start` и `point` устанавливается равным `true`, поле `cmnd` получает значение `"C"`, обнуляется значение поля `result`, а значение в текстовом поле командой `TF.setText("0.0")` устанавливается равным `"0.0"`.

Метод calc() предназначен для вычисления результата последней операции. В этом методе командой x=Double.parseDouble(TF.getText()) значение из текстового поля считывается, преобразуется в формат double и записывается в локальную переменную x. Далее в зависимости от того, какой оператор записан в переменную cmnd, выполняется соответствующее арифметическое действие. Поскольку все реализуемые в данном проекте арифметические операции бинарные, нужен, кроме переменной x, еще один операнд. Его роль играет поле result. Предполагается, что на момент вызова метода calc() это поле содержит результат предыдущих вычислений. Оператор, который записан в поле cmnd, вводится после вычисления этого результата и перед вводом числа, записанного в переменную x. В соответствии с этой схемой использована группа вложенных условных инструкций, с помощью которой вычисляется результат арифметической операции и в качестве нового значения присваивается полю result. После этого командой TF.setText(Double.toString(result)) полученное значение заносится в текстовое поле. При этом для преобразования числового значения поля result в текстовое представление служит метод toString(), который вызывается из класса Double.

Конструктор класса BtnAction получает в качестве аргумента текстовое поле, ссылка на которое заносится в поле TF. После этого выполняется метод onStart(), переводящий все поля и индикаторы в начальное состояние.

В классе BtnAction переопределяется метод actionPerformed(), которым и определяется реакция каждой из кнопок на щелчок. Первой командой String str=ae.getActionCommand() в методе производится считывание текста кнопки, на которой выполнен щелчок. Результат (текст кнопки) записывается в текстовую переменную str. Далее проверяются разные варианты для конкретных кнопок и их последовательностей. Так, если щелчок был произведен на кнопке сброса (условие str.equals("C")), то выполняется метод onStart() и завершается работа метода actionPerformed() (инструкцией return). Если щелчок был выполнен на одной из кнопок с символом арифметической операции или знаком равенства (условие str.equals("+")|str.equals("-")|str.equals("*")|str.equals("/")|str.equals("=")), то вызывается метод calc() (вычисление результата), символ нажатой клавиши командой cmnd=str записывается в поле cmnd, индикаторам start и point присваивается значение true, и работа метода завершается. Значение true для индикатора start означает, что далее будет вводиться число, а для индикатора point — что десятичная точка еще не вводилась.

При вводе числа важно разделять начальный этап ввода числа (когда вводится первая цифра) и его продолжение. В этом случае используется поле start. Так, если число только начинает вводиться (условие start), предусмотрен случай, когда нажата кнопка с десятичной точкой (условие str.equals(".")) и в поле TF вводится текст "0." (ноль с точкой). Индикаторам point и start присваивается значение false, и работа метода завершается (инструкцией return). Если же первой вводится цифра (условие str.equals(".") не выполнено), то она заносится в поле TF командой TF.setText(str), после чего выполняются команды start=false и return.

Невыполнение условия start (значение переменой start при вызове метода actionPerformed() равно false) означает, что продолжается ввод числа в поле TF. В этом случае при попытке ввести точку значение переменной str переопределяется командой str=point?str:"" и выполняется команда point=false. В результате первой из этих двух команд переменная str не изменит своего значения (то есть "."), если точка еще не вводилась. В противном случае она станет пустой строкой. Таким способом предотвращается попытка ввести больше десятичных точек, чем положено. Кроме того, отслеживается ситуация, когда вводится первый незначащий ноль: то есть когда сначала вводится ноль, а затем цифра. В этом случае первый ноль нужно игнорировать и не отображать его в поле TF. Однако если после нуля вводится точка, то ноль нужно оставить. Поэтому проверяется условие TF.getText().equals("0")&!str.equals("."), которое истинно, если в поле TF записан ноль, а следующей вводится не точка. Если это так, то выполняется команда TF.setText(str). В результате первый ноль пропадает с экрана — вместо него вводится цифра (которая, кстати, тоже может быть нулем). Наконец, если это не так, то выполняется команда TF.setText(TF.getText()+str), которой введенная цифра добавляется в конец текстового представления вводимого числа.

В главном методе программы в классе MyCalculator всего одна команда — new CalculatorFrame(), которая создает анонимный объект для окна приложения.

Основы создания апплетов

Метод важнее открытия.

Л. Ландау

До этого мы рассматривали программы, в которых в обязательном порядке присутствовал метод main(). Такие программы выполняются виртуальной Java-машиной под управлением операционной системы. Однако с помощью Java можно создавать и программы иного рода. Такие программы, называемые *апплетами*, находятся на сервере, загружаются клиентом через Интернет и выполняются под управлением браузера. С формальной точки зрения апплет представляет собой панель, на которой могут размещаться различные компоненты, взаимодействующие по той или иной схеме. Важной особенностью апплетов является отсутствие метода main(). Есть и другие особенности. К ним можно отнести бесперспективность использования конструкторов в апплетах. Вместо них используется метод init(). Этот метод выполняется один раз сразу после загрузки апплета браузером. Метод не возвращает результата и наследуется из класса Applet. Вообще же для создания апплета нужно создать класс, расширяющий класс Applet. При этом нередко переопределяют, кроме упомянутого метода init(), метод destroy(), выполняемый при завершении работы апплета, метод start(), выполняемый каждый раз при отображении апплета на экране, и метод stop(), выполняемый в случае, если апплет убирается с экрана. Хотя необходимости в переопределении этих методов нет.

Выполняются апплеты тоже по-особенному. Поскольку апплет, как отмечалось, выполняется под управлением браузера, то кроме непосредственно апплета нужен еще и браузер. Браузер просматривает файлы с гипертекстовой разметкой (HTML-код). Как известно, размеченный таким образом текст содержит теги, которые, собственно, и являются инструкциями для браузера по выводу текста (и не только) в определенном виде или формате. Для включения в HTML-документ инструкции по выполнению апплета используют тег <applet>. В этом теге для параметра code указывается имя откомпилированного файла апплета (файл с расширением class). Пример файла HTML-кода с тегом <applet> приведен в листинге 12.7.

Листинг 12.7. HTML-код с тегом включения апплета

```
<html>
<head><title> Апплет выводит строку</title></head>
<body>
Следующая строка выводится апплетом:<br>
<applet code="ShowStringApplet.class" width="100%" height="100">
</applet>
</body>
</html>
```

Это содержимое файла ShowStringApplet.html, находящегося в той же папке, что и файл ShowStringApplet.class, — результат компиляции Java-файла ShowStringApplet.java, содержащего код из листинга 12.8.

Листинг 12.8. Код простого апплета

```
import java.applet.*;
import java.awt.*;
public class ShowStringApplet extends Applet{
public void paint(Graphics g){
g.drawString("Это очень простой апплет!",10,50);
}}
```

Хотя описание HTML-кода выходит за рамки тематики книги, краткий анализ все же проведем в первую очередь в расчете на тех читателей, кто не знаком с особенностями создания HTML-документов. Для понимания работы апплетов важно знать, что документ с гипертекстовой разметкой содержит теги — инструкции в угловых скобках. Обычно теги используются парами из открывающего и закрывающего тегов. Закрывающий тег содержит символ обратной косой черты. Например, открывающему тегу HTML-страницы <html> соответствует закрывающий тег </html>. Содержимое HTML-документа размещается между этими тегами. Прочие теги выделяют различные блоки HTML-документа. Так, шапка документа размещается между тегами <head> и </head>. В свою очередь, заголовок страницы заключается в теги <title> и </title>. Основное содержание страницы размещается между тегами <body> и </body>.

Большинство тегов могут содержать параметры. Параметры указываются внутри открывающего тега после идентификатора, но перед угловой скобкой. Способ определения параметра следующий: имя параметра и через знак равенства значение (обычно в двойных кавычках). Например, тег `<applet>` содержит параметры `code="ShowStringApplet.class"` (имя откомпилированного файла апплета), `width="100%"` (ширина области апплета устанавливается равной 100 % от ширины окна браузера) и `height="100"` (высота области апплета в 100 пикселей).

Основное тело гипертекстовой страницы содержит текст `Следующая строка выводится апплетом:`, после которого следует инструкция перехода к новой строке `<br>` (закрывающего тега нет). Собственно, это сообщение и отображается в окне браузера. Все остальное делает апплет. Если открыть файл ShowStringApplet.html с помощью браузера, получим результат, представленный на рис. 12.12.

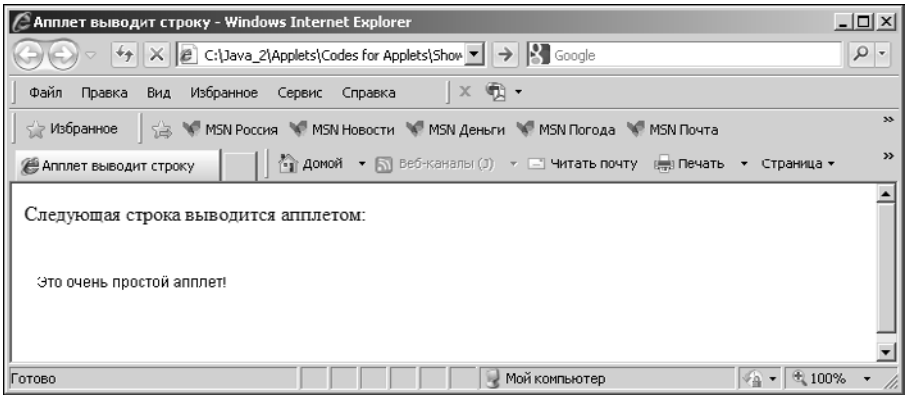

Рис. 12.12. Результат выполнения апплета в окне браузера

Для понимания происходящего обратимся теперь к коду апплета из листинга 12.8. Инструкцией `import java.applet.*` подключается пакет для работы с апплетами. Библиотеку AWT мы подключаем командой `import java.awt.*`. Это необходимо для переопределения метода `paint()`. Метод вызывается автоматически каждый раз при восстановлении изображения апплета. В переопределенном методе `paint()` вызывается метод `drawString()`, отображающий текстовую строку, указанную первым аргументом, в позиции, которая определяется вторым и третьим аргументами (координаты точки вывода текста). Метод `drawString()` вызывается из объекта графического контекста, который автоматически передается аргументом методу `paint()`.

Если код апплета несколько модифицировать, заново откомпилировать и перезапустить браузер (именно перезапустить, а не обновить), результат будет таким, как на рис. 12.13.

Код измененного апплета представлен в листинге 12.9.

Листинг 12.9. Код апплета после внесения изменений

```
import java.applet.*;
import java.awt.*;
public class ShowStringApplet extends Applet{
public void init(){
setBackground(Color.YELLOW);
setFont(new Font("Serief",Font.BOLD,18));
}
public void paint(Graphics g){
g.drawString("Это очень простой апплет!",10,50);
}}
```

Рис. 12.13. Результат после изменения кода апплета

Помимо уже рассмотренного кода добавлена реализация метода `init()`. В этом методе командой `setBackground(Color.YELLOW)` устанавливается желтый цвет фона для апплета. Кроме того, командой `setFont(new Font("Serief",Font.BOLD,18))` задается шрифт вывода текста. В результате в окне браузера выделяется область апплета, а текст изменяет внешний вид.

Апплет может содержать практически все стандартные элементы управления графического интерфейса. В листинге 12.10 приведен код апплета, содержащего текстовую метку и две кнопки, с помощью которых можно изменять размер шрифта текста метки.

Листинг 12.10. Апплет с меткой и кнопками

```
// Подключение пакетов:
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Класс апплета:
```

```
public class MyApplet extends Applet{
// Метка для отображения текста:
private Label L;
// Кнопки:
private Button BtnH;
private Button BtnL;
// Основной текст:
private String txt="Изменяющий размеры текст: сейчас размер ";
// Размер шрифта:
private int size;
// Обработчик щелчка на кнопке:
private BtnPressed BP;
// Метод для инициализации параметров:
public void init(){
// Белый цвет фона апплета:
setBackground(Color.WHITE);
// Начальное значение для размера шрифта:
size=20;
// Размеры метки:
int W,H;
// Ширина метки определяется на основе ширины апплета:
W=getWidth();
// Высота метки определяется на основе высоты апплета:
H=2*getHeight()/3;
// Отключение менеджера компоновки:
setLayout(null);
// Создание метки:
L=new Label();
// Выравнивание текста по центру:
L.setAlignment(Label.CENTER);
// Размеры метки (применение):
L.setSize(W,H);
// Положение метки:
L.setLocation(0,0);
// Белый цвет фона метки:
L.setBackground(Color.WHITE);
// Шрифт для метки:
L.setFont(new Font("Serief",Font.BOLD,size));
// Текст метки:
L.setText(txt+L.getFont().getSize());
// Первая кнопка (создание):
BtnH=new Button("Больше >>");
// Вторая кнопка (создание):
BtnL=new Button("<< Меньше");
// Положение и размеры первой кнопки:
BtnH.setBounds(W/4,H+1,W/4,H/4);
```

продолжение

Листинг 12.10 *(продолжение)*

```
// Положение и размеры второй кнопки:
BtnL.setBounds(W/2+1,H+1,W/4,H/4);
// Шрифт для первой кнопки:
BtnH.setFont(new Font("Arial",Font.BOLD,11));
// Шрифт для второй кнопки:
BtnL.setFont(new Font("Arial",Font.BOLD,11));
// Создание обработчика для щелчка на кнопке:
BP=new BtnPressed();
// Регистрация обработчика для первой кнопки:
BtnH.addActionListener(BP);
// Регистрация обработчика для второй кнопки:
BtnL.addActionListener(BP);
// Добавление первой кнопки в апплет:
add(BtnH);
// Добавление второй кнопки в апплет:
add(BtnL);
// Добавление метки в апплет:
add(L);}
// Внутренний класс для обработчика щелчка на кнопке:
class BtnPressed implements ActionListener{
// Закрытый метод, через который реализуется обработка:
private void pressed(boolean btn){
if(btn) size++;
else size--;
L.setFont(new Font("Serief",Font.BOLD,size));
L.setText(txt+L.getFont().getSize());
if(size>15) BtnL.setEnabled(true);
else BtnL.setEnabled(false);
if(size<25) BtnH.setEnabled(true);
else BtnH.setEnabled(false);}
// Реакция на щелчок на кнопке:
public void actionPerformed(ActionEvent ae){
if(ae.getSource()==BtnH) pressed(true);
if(ae.getSource()==BtnL) pressed(false);}
}}
```

Код соответствующего HTML-документа приведен в листинге 12.11.

Листинг 12.11. Код HTML-документа с апплетом

```
<html>
<head><title> Пример простого апплета</title></head>
<body>
Текст может изменять свой размер. Щелкните на кнопке:<br>
<applet code="MyApplet.class" width="600" height="200">
</applet>
</body>
</html>
```

Окно браузера с открытой в нем HTML-страницей представлено на рис. 12.14.

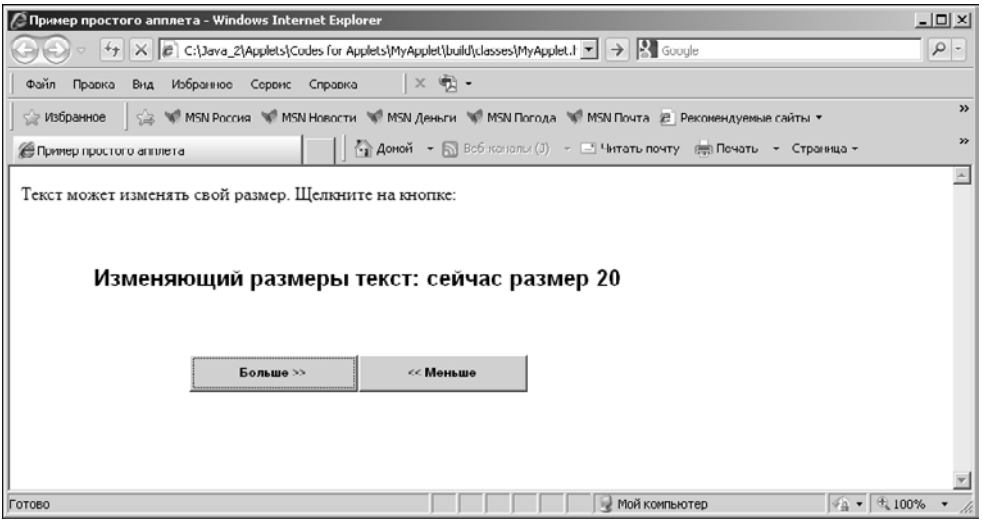

Рис. 12.14. Окно браузера с апплетом, реализующим метку и две кнопки

При открытии страницы апплетом отображается текст размером 20 пунктов. Щелчок на кнопке Больше >> приводит к увеличению размера текста на единицу, щелчок на кнопке << Меньше — к уменьшению размера текста на единицу. При этом значение размера указывается в тексте метки. Размер текста не может быть больше 25 и меньше 15. Если размер шрифта принимает граничное значение, соответствующая кнопка становится недоступной, как показано на рис. 12.15.

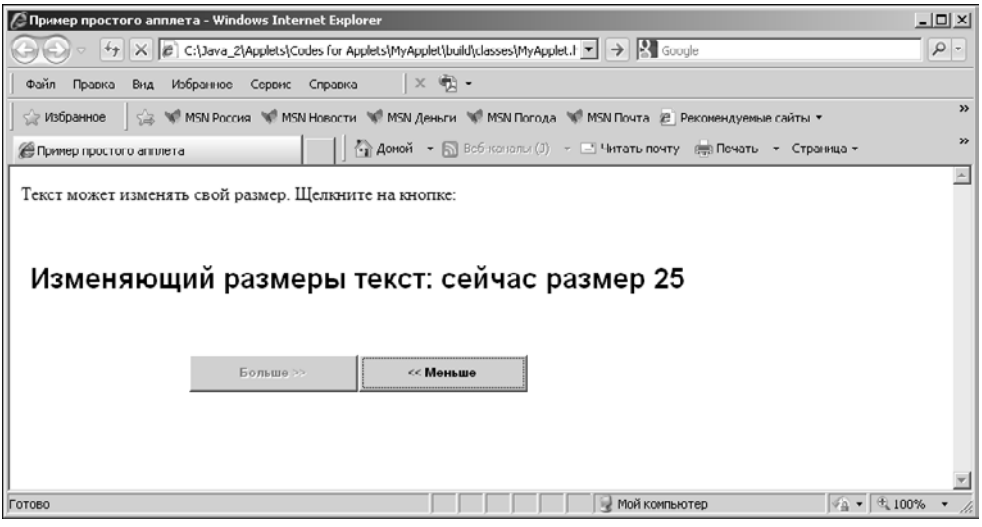

Рис. 12.15. Одна из кнопок недоступна

Рассмотрим программный код апплета более подробно. Класс апплета MyApplet наследует стандартный класс Applet, имеет ряд закрытых полей, переопределяет метод init() и внутренний класс обработчика щелчка на кнопке BtnPressed. Среди закрытых полей (в скобках указана принадлежность соответствующего объекта классу): метка L (класса Label), две кнопки BtnH и BtnL (класса Button), текстовое поле txt с основным отображаемым апплетом текстом (класса String), целочисленное (типа int) поле size, определяющее текущий размер шрифта, а также обработчик щелчка на кнопке BP (внутренний класс BtnPressed).

В методе init() командой setBackground(Color.WHITE) устанавливается белый цвет фона апплета, а командой size=20 — начальное значение для размера шрифта, которым отображается текст апплета. Командами W=getWidth() и H=2*getHeight()/3 на основании ширины и высоты апплета задаются размеры метки (она занимает весь апплет по ширине и две трети апплета по высоте). Также обращаем внимание, что размеры непосредственно апплета определяются кодом HTML-документа (см. листинг 12.11). Отключение менеджера компоновки выполняется, как и прежде, командой setLayout(null).

Для создания метки служит инструкция L=new Label(), в результате выполнения которой ссылка на созданную метку записывается в поле L. Текст метки выравнивается по центру, для чего используется команда L.setAlignment(Label.CENTER). После выполнения команд L.setSize(W,H) и L.setLocation(0,0) соответственно размеры применяются к метке и задается ее положение в апплете. Белый цвет фона метки устанавливается командой L.setBackground(Color.WHITE). Шрифт для метки задается инструкцией L.setFont(new Font("Serief",Font.BOLD,size)), в которой в качества значения размера шрифта использовано поле size. После этого определяется текст метки. В команде L.setText(txt+L.getFont().getSize()) инструкцией L.getFont() возвращается объект текущего шрифта метки (объект класса Font). Из этого анонимного объекта вызывается метод getSize(), который в качестве значения возвращает размер шрифта. Таким образом, текст метки состоит из стандартной фразы, записанной в поле txt, и текстового представления текущего размера шрифта. Затем создаются кнопки.

Первая кнопка создается командой BtnH=new Button("Больше >>"), вторая — командой BtnL=new Button("<< Меньше"). Положение и размеры кнопок определяются соответственно командами BtnH.setBounds(W/4,H+1,W/4,H/4) и BtnL.setBounds(W/2+1,H+1,W/4,H/4). Для применения шрифта, которым отображается текст кнопок, используются команды BtnH.setFont(new Font("Arial",Font.BOLD,11)) и BtnL.setFont(new Font("Arial",Font.BOLD,11)).

Обработчик щелчка на кнопке создается командой BP=new BtnPressed() (объект внутреннего класса BtnPressed, описание которого приведено далее), после чего созданный обработчик регистрируется в кнопках (командами BtnH.addActionListener(BP) и BtnL.addActionListener(BP)). Добавление кнопок и метки на панель осуществляется последовательно инструкциями add(BtnH), add(BtnL) и add(L).

Внутренний класс BtnPressed для обработчика щелчка на кнопке создается расширением интерфейса ActionListener. Основу класса составляют два метода.

Закрытый метод pressed() получает в качестве аргумента логическое значение, являющееся индикатором того, какая кнопка нажата. Значение аргумента true соответствует первой кнопке и означает увеличение на единицу размера шрифта, а значение аргумента false соответствует второй кнопке и означает уменьшение на единицу размера шрифта. Проверка того, какая кнопка нажата, выполняется в условной инструкции. После изменения поля size командой L.setFont(new Font("Serief",Font.BOLD,size)) обновляется шрифт метки и командой L.setText(txt+L.getFont().getSize()) изменяется текст метки. В следующих далее условных инструкциях проверяется, не достигло ли значение размера шрифта граничного значения. Если точнее, то проверяется обратное условие: если размер шрифта больше граничного значения 15, доступной для щелчка делается кнопка уменьшения размера (в противном случае она делается недоступной для щелчка), а если размер шрифта меньше граничного значения 25, доступной для щелчка делается кнопка увеличения размера шрифта (в противном случае она делается недоступной для щелчка). Таким образом, метод pressed() может обрабатывать щелчок как на первой, так и на второй кнопке.

Также в классе BtnPressed переопределяется метод actionPerformed(), который, собственно, и вызывается для обработки щелчка на кнопке. В методе две условных инструкции. С помощью метода getSource(), вызываемого из аргумента метода, возвращается объект, вызвавший это событие (событие — объект класса ActionEvent). Если этот объект — кнопка BtnH, то метод pressed() вызывается с аргументом true. Если событие вызывала кнопка BtnL, то метод pressed() вызывается с аргументом false.

Апплеты могут создаваться путем наследования класса JApplet, который, в свою очередь, наследует класс Applet. Обычно к помощи класса JApplet прибегают в случае, если в апплете размещаются «легкие» компоненты библиотеки Swing. Возможности класса JApplet существенно шире тех, что представлены в классе Applet.

В заключение хочется отметить одну очень важную возможность, которая состоит в том, что апплету могут передаваться аргументы (параметры) через инструкции в соответствующем HTML-документе в теге <applet>. В блоке тега <applet> указываются теги <param> с параметрами тега name и value. Значения этих параметров указываются через знак равенства: для параметра тега name — это имя параметра, передаваемого апплету, а для параметра тега value — значение параметра, передаваемого апплету. В самом апплете считывание передаваемых апплету параметров осуществляется с помощью метода getParameter(). В листинге 12.12 приведен код HTML-документа, а в листинге 12.13 представлен Java-код соответствующего апплета.

Листинг 12.12. В HTML-документе апплету передаются параметры

```
<html>
<head><title> Передача параметров</title></head>
<body>
Параметры апплету передаются в HTML-документе:<br>
```

продолжение

Листинг 12.12 *(продолжение)*

```
<applet code="ShowText.class" width="600" height="90">
<param name="цвет" value="красный">
<param name="цвет шрифта" value="желтый">
<param name="шрифт" value="Arial">
<param name="стиль" value="жирный">
<param name="размер" value="28">
</applet>
</body>
</html>
```

Листинг 12.13. Код апплета с передачей параметров

```
// Подключение пакетов:
import java.awt.*;
import java.applet.*;
// Класс апплета:
public class ShowText extends Applet{
// Отображаемый текст:
String text;
// Метод для определения цвета:
private Color getColor(String color){
Color clr;
if(color.equals("желтый")) clr=Color.YELLOW;
else if(color.equals("синий")) clr=Color.BLUE;
else if(color.equals("красный")) clr=Color.RED;
else clr=Color.WHITE;
return clr;
}
// Метод инициализации апплета:
public void init(){
// Считывание цвета фона:
String color=getParameter("цвет");
// Применение цвета фона:
setBackground(getColor(color));
// Считывание цвета для отображения текста:
color=getParameter("цвет шрифта");
// Применение цвета для отображения текста:
setForeground(getColor(color));
// Считывание названия шрифта:
String font=getParameter("шрифт");
// Считывание стиля шрифта (текстовое название):
String style=getParameter("стиль");
// Стиль шрифта:
int stl;
// Преобразование текстового значения для стиля в числовое:
if(style.equals("жирный")) stl=Font.BOLD;
else if(style.equals("курсив")) stl=Font.ITALIC;
```

```
else stl=Font.PLAIN;
// Размер шрифта:
int size=Integer.parseInt(getParameter("размер"));
// Текст для отображения:
text="Шрифт "+font+" "+style+" размера "+size;
// Применение атрибутов к шрифту:
setFont(new Font(font,stl,size));}
// Метод отрисовки апплета:
public void paint(Graphics g){
g.drawString(text,30,getHeight()-30);}
}
```

Помимо обычного по сравнению со встречавшимся ранее HTML-кодом в листинге 12.12 появляется группа тегов вида:

```
<param name="значение1" value="значение2">
```

В данном случае значение1 является названием параметра, который передается апплету, а значение2 — значением этого параметра. Например, инструкция <param name="цвет шрифта" value="желтый"> означает, что апплету передается параметр, который называется цвет шрифта, а значение этого параметра равно желтый. Все значения параметров апплета считываются в апплете методом getParameter() в текстовом формате. Аргументом методу передается текстовое название считываемого параметра. Например, чтобы считать значение параметра цвет шрифта, метод getParameter() в коде апплета вызывается в формате getParameter("цвет шрифта"), а результатом выполнения такой команды является текстовое значение "желтый".

Что касается кода апплета (см. листинг 12.13), то класс апплета ShowText состоит всего из трех методов (getColor(), init() и paint()) и одного текстового поля text. Текстовое поле text предназначено для записи текстовой фразы, отображаемой в области апплета. Поскольку этот текст формируется в методе init(), а отображается переопределяемым методом paint() (то есть оба метода должны иметь доступ к соответствующей текстовой переменной), текст реализуется в виде поля класса апплета.

Кроме формирования текстового поля text, в методе init() производится считывание параметров апплета. Командой String color=getParameter("цвет") объявляется текстовая переменная color и в качестве значения ей присваивается значение параметра цвет, указанное в HTML-документе. После этого командой setBackground(getColor(color)) применяется цвет для фона. Здесь использован метод getColor(), с помощью которого выполняется преобразование текстового значения переменной color в объект класса Color (метод getColor() описывается далее). Этот же метод getColor() используется в команде setForeground(getColor(color)), которой задается цвет отображения текста. Предварительно командой color=getParameter("цвет шрифта") меняется значение переменной color — теперь это цвет, указанный в качестве значения параметра

цвет шрифта в HTML-документе. Считывание названия шрифта выполняется командой String font=getParameter("шрифт"). Стиль шрифта считывается командой String style=getParameter("стиль"). Обращаем внимание читателя, что переменная style является текстовой. Для использования ее необходимо преобразовать в целочисленный формат (значения Font.BOLD, Font.ITALIC и Font.PLAIN являются целочисленными константами). Результат преобразования записывается в целочисленную переменную stl. Само преобразование выполняется на основе вложенных условных инструкций, которые последовательно проверяют текстовое значение, записанное в переменную style. Обращаем внимание, что по умолчанию, фактически, применяется стиль PLAIN.

Размер шрифта считывается с помощью команды getParameter("размер"). Полученное текстовое значение передается аргументом методу Integer.parseInt(), в результате чего текстовое представление числа преобразуется в целочисленное значение, которое записывается в переменную size. Все эти действия реализованы в команде

```
int size=Integer.parseInt(getParameter("размер"))
```

Текст для отображения определяется командой text="Шрифт "+font+" "+style+" размера "+size, а с помощью команды setFont(new Font(font,stl,size)) считанные параметры используются для создания нового шрифта в апплете.

Аргументом закрытого метода getColor() передается текстовое название для цвета. Далее это текстовое значение преобразуется в приемлемый для использования формат — в объект класса Color. Для этого в методе предусмотрительно объявляется переменная clr класса Color. Далее с помощью вложенных условных инструкций проверяется значение текстового аргумента метода color. В зависимости от значения аргумента объектной переменной clr присваивается то или иное значение (цветом по умолчанию является белый). Переменная clr возвращается как результат.

Наконец, при переопределении метода paint() с помощью команды g.drawString(text,30,getHeight()-30) (где аргумент метода g — объект контекста апплета) в области апплета выводится сформированная в методе init() текстовая строка, причем с применением атрибутов текста, переданных апплету через код HTML-документа. На рис. 12.16 показано, как будет выглядеть HTML-документ с апплетом.

В данном случае на синем фоне отображается желтый текст с атрибутами, указанными в тексте.

Передача параметров апплету через HTML-документ достаточно удобна, поскольку в этом случае для изменения параметров правка вносится в HTML-документ, причем изменения вступают в силу после сохранения изменений и обновления страницы. Важно в этом случае то, что заново компилировать апплет нет необходимости.

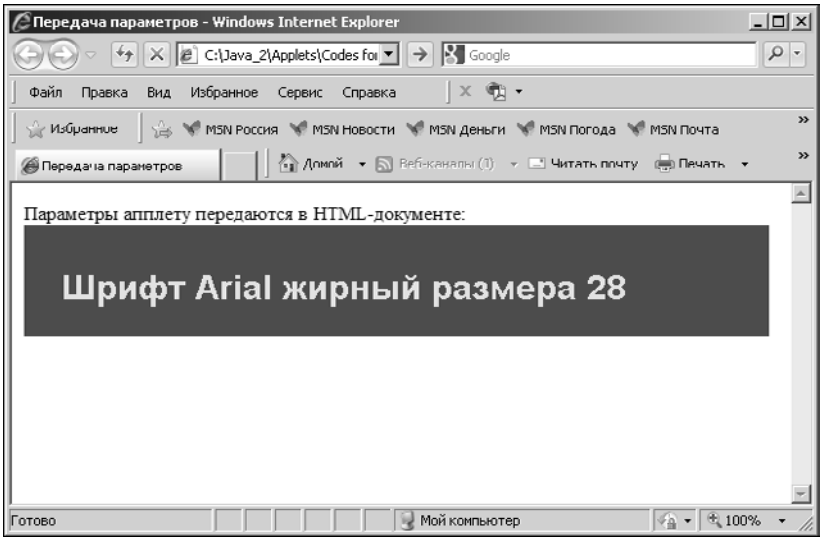

Рис. 12.16. Параметры в апплет передаются через HTML-документ

Резюме

1. Существуют две библиотеки, которые используются для создания приложений с графическим интерфейсом: AWT и Swing. Вторая из них является дополнением к библиотеке AWT.
2. Обычно приложение содержит главное окно или фрейм. Фрейм реализуется как объект класса, наследующего класс `Frame` (библиотека AWT) или класс `JFrame` (библиотека Swing). Существуют специальные классы и для прочих элементов графического интерфейса, таких, например, как кнопки или переключатели.
3. Взаимодействие компонентов графического интерфейса реализуется посредством обработки событий. В Java используется модель обработки событий с делегированием: в компоненте, который может вызвать то или иное событие, регистрируется обработчик этого события. Для обработчиков событий создаются специальные классы путем расширения соответствующего интерфейса.
4. Помимо приложений, которые выполняются под управлением операционной системы, существует особый вид Java-программ — апплеты, выполняющиеся под управлением браузера и обычно загружаемые через Интернет. Апплет создается расширением класса `Applet` или `JApplet`. У апплетов есть некоторые особенности: например, в них отсутствует метод `main()`. По сравнению с обычными приложениями апплеты имеют ограниченные возможности. Однако именно благодаря этому они более безопасны, особенно при загрузке через Интернет.

Заключение

В книге представлена лишь малая часть из того, что можно было бы написать о Java. При написании книги всегда стоит вопрос о том, что в книгу включать, а что — нет. По большому счету, успех книги зависит от того, насколько удачен ответ на этот вопрос.

В данном случае решалась двойная задача. Во-первых, хотелось описать базовые конструкции и основы синтаксиса языка Java. Во-вторых, среди множества тем и подходов были выбраны те, что наиболее интересны и перспективны и при этом вписываются в концепцию книги. Тем не менее список тем, не вошедших в книгу, может быть предметом еще для нескольких книг. Поэтому в конце книги приведен список литературы, который может быть полезен тем, кто желает продолжить свое самообразование и приобщение к миру Java. Список небольшой, но показательный. Даже если в перечисленных изданиях освещены те же или почти те же вопросы, что и в данной книге, всегда полезно узнать, что по поводу одних и тех же проблем думают разные люди. В любом случае, хочется пожелать читателю успехов и выразить благодарность за интерес к книге.

Литература

1. *Ноутон П., Шилдт Г.* Java 2: Наиболее полное руководство. СПб.: БХВ, 2006. 1072 с.
2. *Хабибулин И.* Самоучитель Java. СПб.: БХВ, 2008. 720 с.
3. *Хорстманн К.С., Корнелл Г.* Java 2. Библиотека профессионала. Т. 1, Основы. М.: Вильямс, 2009. 816 с.
4. *Хорстманн К.С., Корнелл Г.* Java 2. Библиотека профессионала. Т. 2, Тонкости программирования. М.: Вильямс, 2010. 992 с.
5. *Монахов В.* Язык программирования Java и среда NetBeans. СПб.: БХВ, 2009. 720 с.
6. *Шилдт Г.* Java: руководство для начинающих. М.: Вильямс, 2008. 720 с.
7. *Шилдт Г.* Java: методики программирования Шилдта. М.: Вильямс, 2008. 512 с.
8. *Фишер Т. Р.* Java. Карманный справочник. М.: Вильямс, 2008. 224 с.
9. *Шилдт Г.* Библиотека SWING для Java: руководство для начинающих. М.: Вильямс, 2007. 704 с.
10. *Шилдт Г.* Полный справочник по Java SE 6. М.: Вильямс, 2010. 1040 с.
11. *Эккель Б.* Философия Java. СПб.: Питер., 2009. 640 с.

Приложение

Программное обеспечение

В этом приложении приведена краткая справка по программному обеспечению, которое может оказаться полезным (а некоторые утилиты и просто необходимыми) для успешной работы с Java. При этом в приложении приведен минимальный объем информации относительно загрузки и использования программного обеспечения — в объеме, необходимом для того, чтобы читатель смог успешно откомпилировать и запустить приведенные в основной части книги программные коды. Читателю, интересующемуся вопросами работы со специальным программным обеспечением для Java, можем порекомендовать обратиться к специальной или справочной литературе по этому вопросу.

Загрузка программного обеспечения

Поиск нужного программного обеспечения имеет смысл начать с посещения страницы www.java.com, предназначенной специально для поддержки языка Java. Эта страница в окне браузера Internet Explorer представлено на рис. П.1.

Хочется верить, что сама процедура загрузки и установки программного обеспечения комментариев не требует — для этого достаточно щелкнуть на соответствующей кнопке и следовать инструкциям по загрузке и установке. С сайта можно загрузить и установить последние версии JDK и JRE. Однако в этом случае компилировать и запускать программы придется, что называется, вручную. Гораздо удобнее и надежнее воспользоваться специальной интегрированной средой разработки. Удачный выбор — среда NetBeans. Именно в этой среде компилировались программные коды из книги. Загрузить файлы установки среды NetBeans можно с сайта www.netbeans.org (рис. П.2).

Существуют различные конфигурации установочных пакетов, в том числе для работы с отличными от Java языками. Сайт организован достаточно удачно, так что проблем с загрузкой программного обеспечения возникнуть не должно.

Еще один неплохой выбор — среда разработки Eclipse. Установочные файлы свободно загружаются со страницы www.eclipse.org (рис. П.3).

Как и в случае с NetBeans, существует несколько вариантов установки Excipse с поддержкой разных платформ и языков программирования, в том числе Java. Далее кратко описываются особенности работы с каждой из упомянутых сред разработки. При этом NetBeans, как предполагаемая основная среда разработки, описывается более детально.

Рис. П.1. Веб-страница www.java.com

Рис. П.2. Веб-страница www.netbeans.org

Рис. П.3. Веб-страница www.eclipse.org

Работа с NetBeans

Окно среды разработки NeBeans версии 6.8 представлено на рис. П.4.

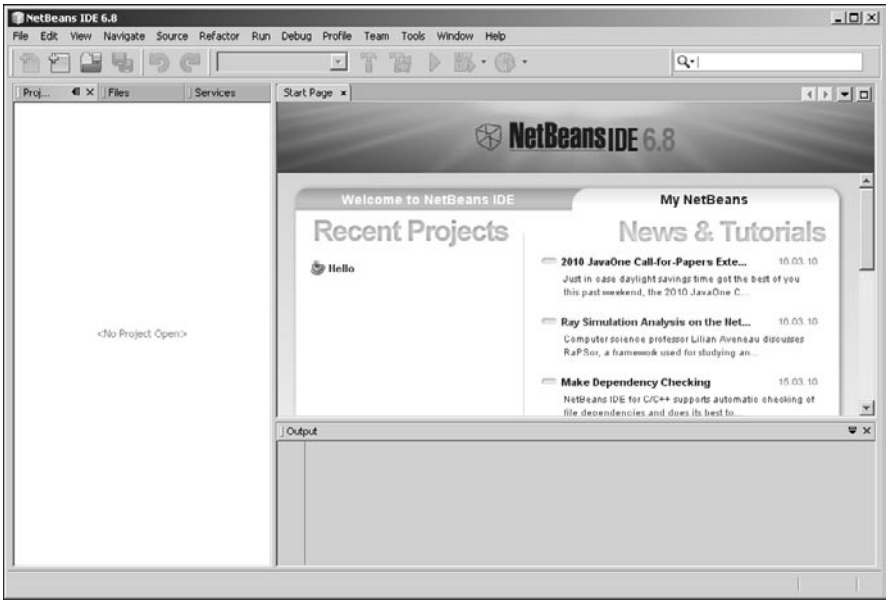

Рис. П.4. Окно среды разработки NetBeans 6.8

Для создания проекта в меню File выбирают команду New Project, как показано на рис. П.5.

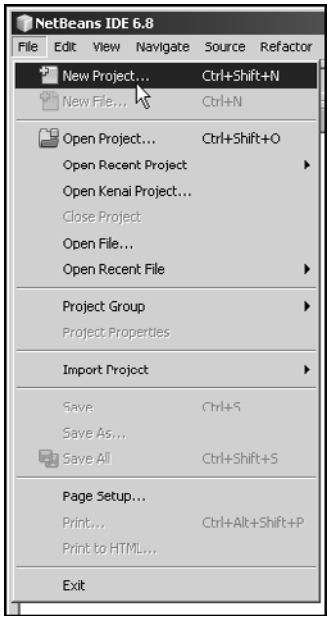

Рис. П.5. Создание нового проекта командой File ▸ New Project

В качестве альтернативы можно воспользоваться кнопкой панели инструментов (по умолчанию она вторая слева) в виде светло-коричневой закрытой папки с зеленым крестом (рис. П.6).

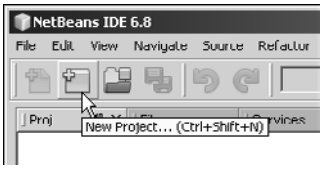

Рис. П.6. Кнопка создания нового проекта

В результате открывается диалоговое окно New Project, представленное на рис. П.7.

В этом окне в разделе Categories следует выбрать вид создаваемого проекта (в нашем случае выбирается позиция Java), а в разделе Projects — тип приложения (позиция Java Application). После щелчка на кнопке Next открывается окно New Java Application (рис. П.8).

В поле Project Name указывается имя создаваемого проекта. Место для сохранения проекта вводится в поле Project Location (для выбора можно воспользоваться кнопкой Browse справа от поля). Кроме того, на рис. П.8 сброшены флажки Create Main Class и Set as Main Project — в результате будет создан пустой (без

пользовательских классов) проект. Окно среды NetBeans со вновь созданным проектом представлено на рис. П.9.

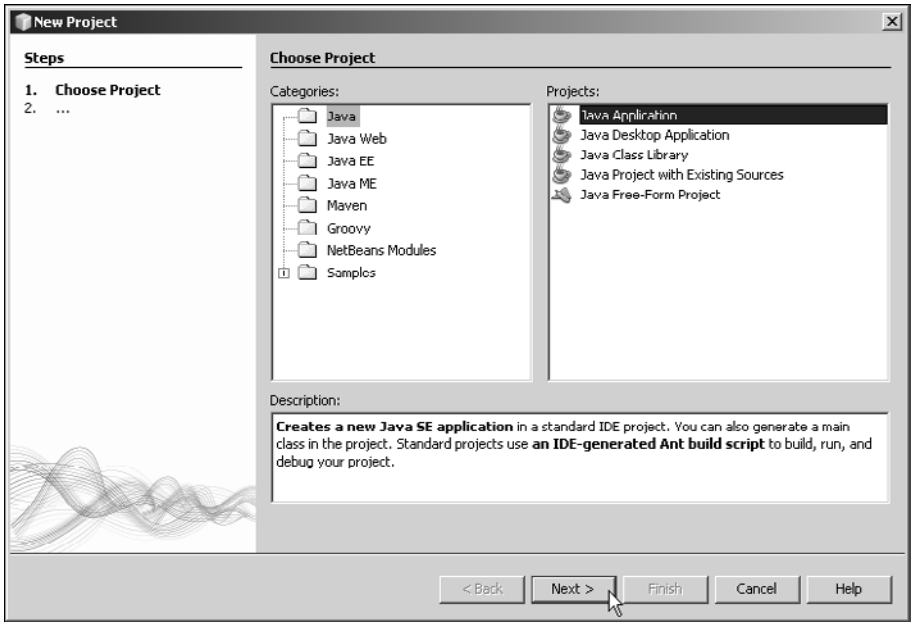

Рис. П.7. Окно выбора типа приложения New Project

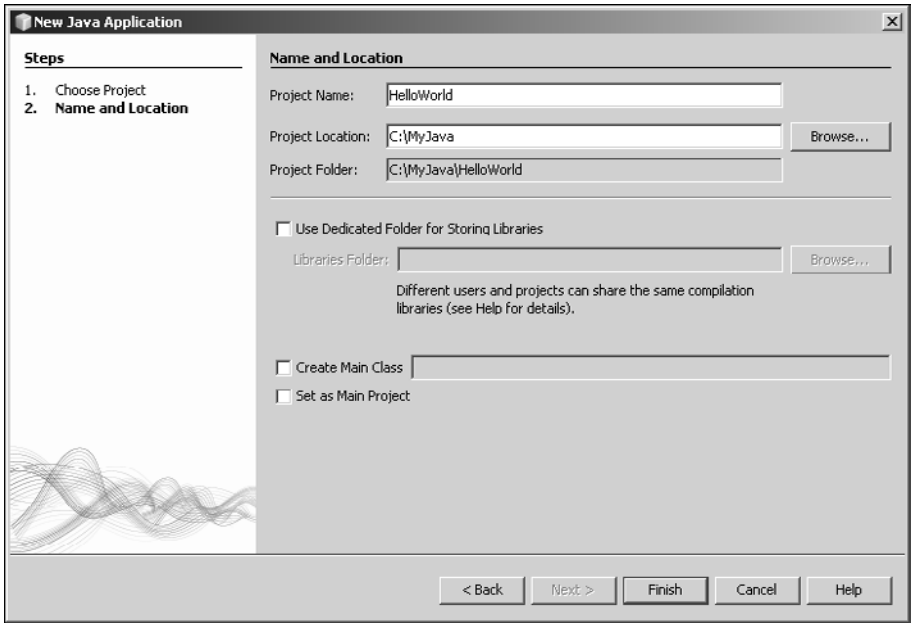

Рис. П.8. Окно выбора параметров приложения New Java Application

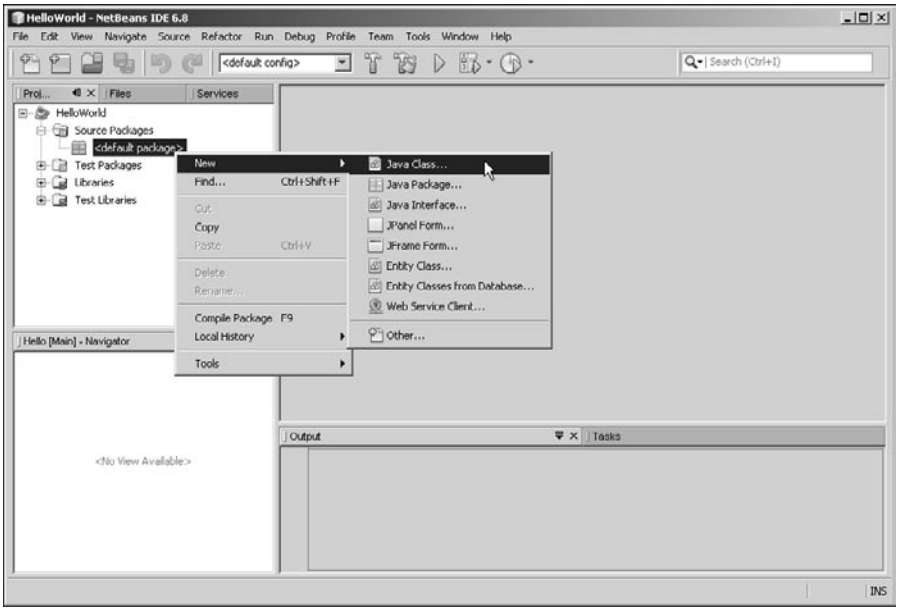

Рис. П.9. Добавление класса в проект

Чтобы добавить в проект класс, можно в контекстном меню проекта выбрать команду New ▸ Java Class или воспользоваться кнопкой New File на панели инструментов (по умолчанию первая слева).

В диалоговом окне New Java Class в поле Class Name вводится имя добавляемого в проект класса (рис. П.10).

По умолчанию код класса содержит только формальные атрибуты начала и окончания описания класса и комментарии в шапке файла (этот шаблон, кстати, можно изменить). В код класса вносятся нужные изменения, например, так, как показано на рис. П.12.

Для компиляции проекта можно воспользоваться командой Run ▸ Build Project или одноименной кнопкой на панели инструментов (рис. П.13).

После успешной компиляции запускаем проект на выполнение. Для этого используем команду Run ▸ Run Project или кнопку, расположенную на панели инструментов справа от кнопки компиляции проекта (рис. П.14).

При первом запуске проекта открывается окно Run Prject, предназначенное для выбора главного класса (рис. П.15). В данном случае с предложенным выбором лучше согласиться.

Окно среды NetBeans с результатом выполнения программы показано на рис. П.16. Результат (текстовое сообщение) выводится в окне Output, которое играет в данном случае роль консоли.

Чтобы закрыть проект, используют команду Close Project меню File или команду Close контекстного меню проекта (рис. П.17).

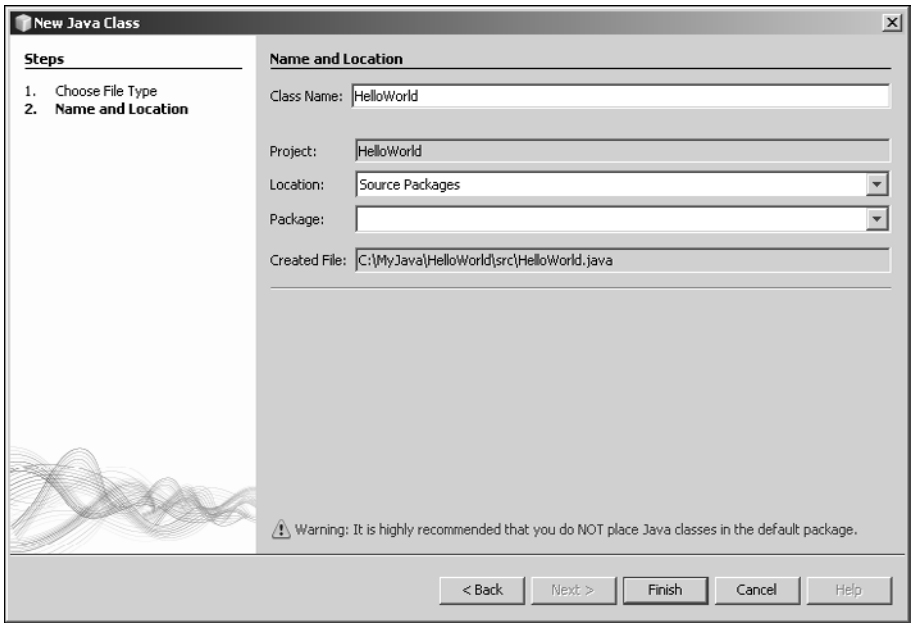

Рис. П.10. Окно задания параметров нового класса

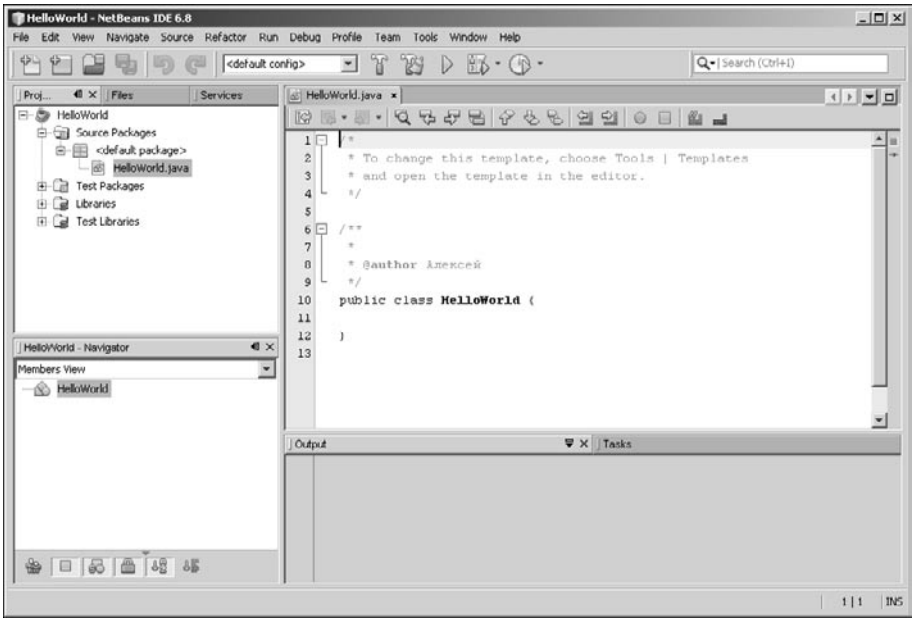

Рис. П.11. Окно среды NetBeans с кодом вновь созданного класса

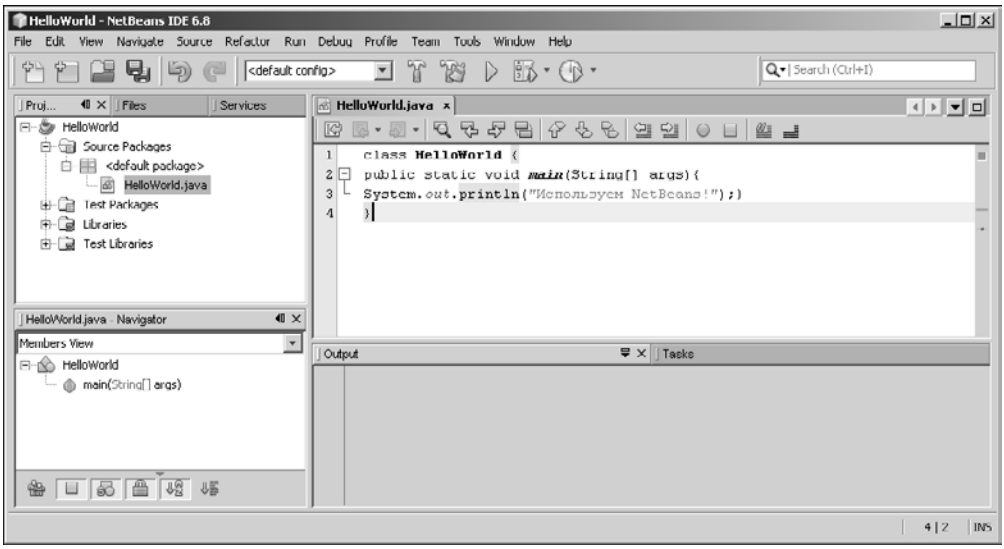

Рис. 14.12. Код класса после внесения изменений

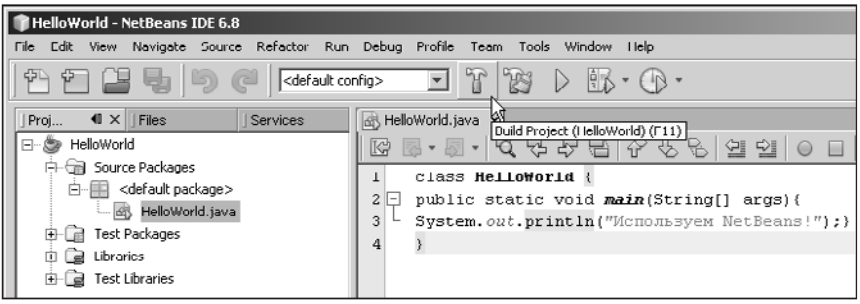

Рис. П.13. Компиляция проекта

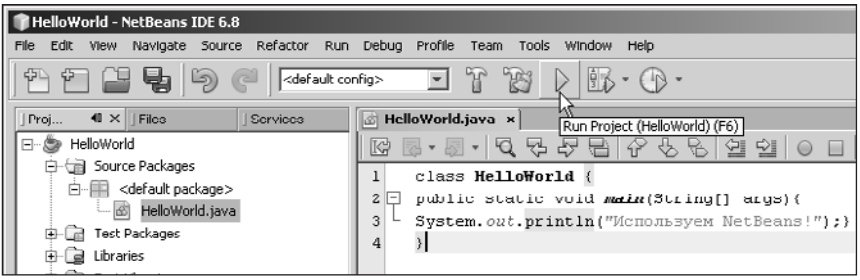

Рис. П.14. Запуск приложения

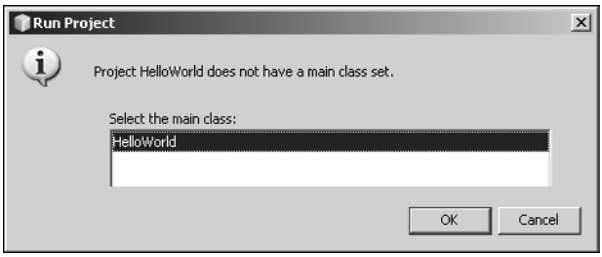

Рис. П.15. Окно выбора главного класса

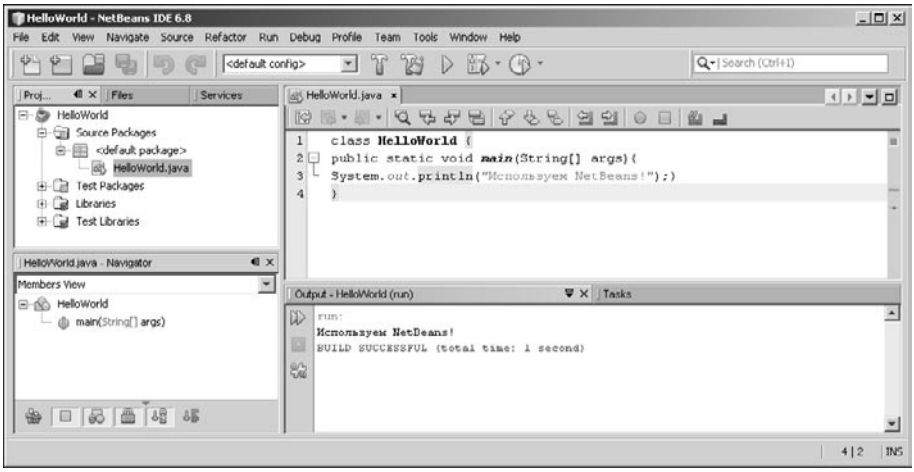

Рис. П.16. Результат выполнения программы отображается в окне Output среды NetBeans

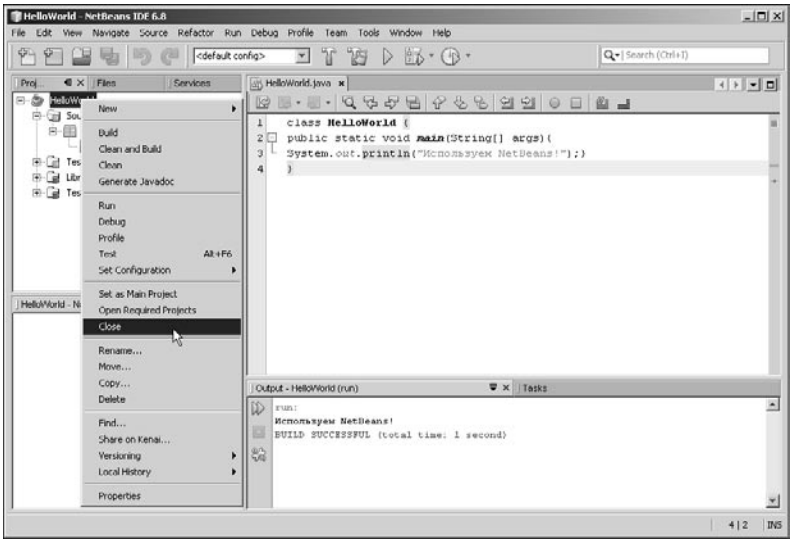

Рис. П.17. Закрытие проекта

Чтобы открыть уже существующий проект, можно прибегнуть, например, к команде Open Project меню File (рис. П.18).

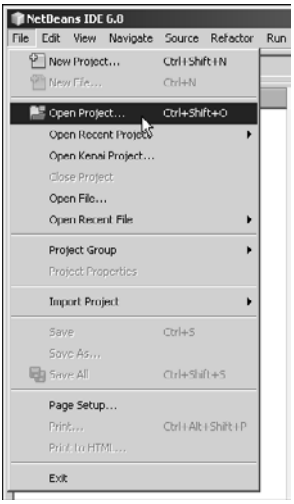

Рис. П.18. Открытие проекта командой меню

Другой способ подразумевает использование соответствующей кнопки на панели инструментов — это третья слева кнопка (рис. П.19).

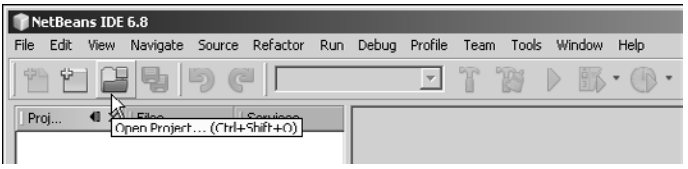

Рис. П.19. Открытие проекта с помощью кнопки панели инструментов

В результате открывается окно Open Project, в котором выбирается папка проекта, после чего выполняется щелчок на кнопке Open Project в нижней части окна (рис. П.20).

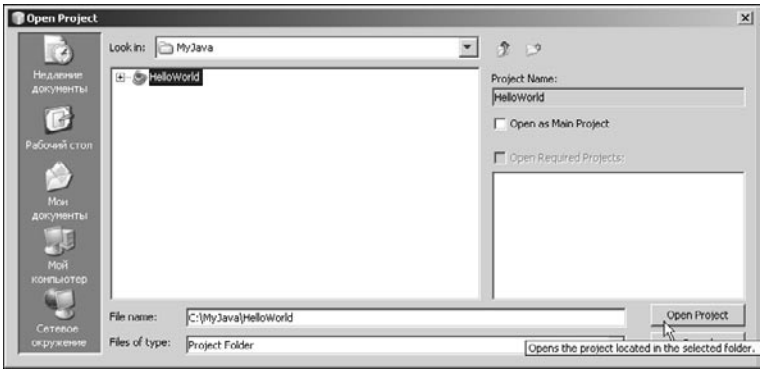

Рис. П.20. Окно выбора папки проекта

После этого в окне среды NetBeans соответствующий проект открывается и его можно редактировать, компилировать и выполнять. Если необходимости в редактировании проекта нет и он уже откомпилирован, проект можно сразу запускать.

Работа с Eclipse

Во многом методы работы в среде Eclipse напоминают методы работы в NetBeans, поэтому в данном случае кратко остановимся на способе создания и запуска Java-приложения с помощью редактора и утилит среды Eclipse. На рис. П.21 представлено окно среды Eclipse, в котором выбрана команда File ▸ New ▸ Java Project.

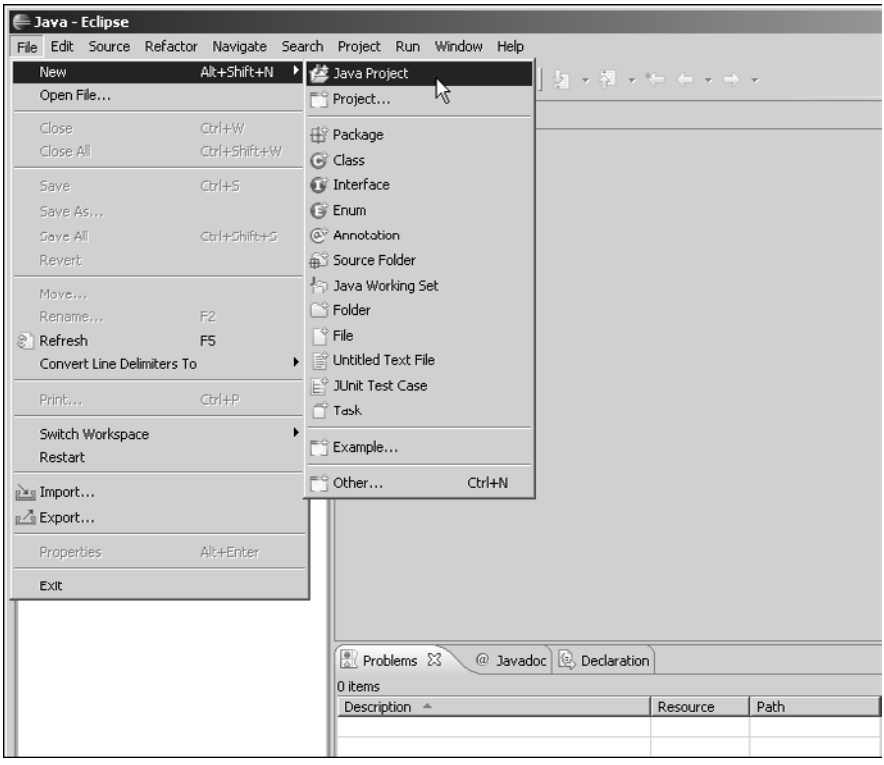

Рис. П.21. Создание проекта с помощью команды меню

Можно также воспользоваться раскрывающимся меню кнопки создания новых элементов, которая по умолчанию является первой слева на панели инструментов (рис. П.22).

В любом случае откроется диалоговое окно настройки нового проекта New Java Project (рис. П.23).

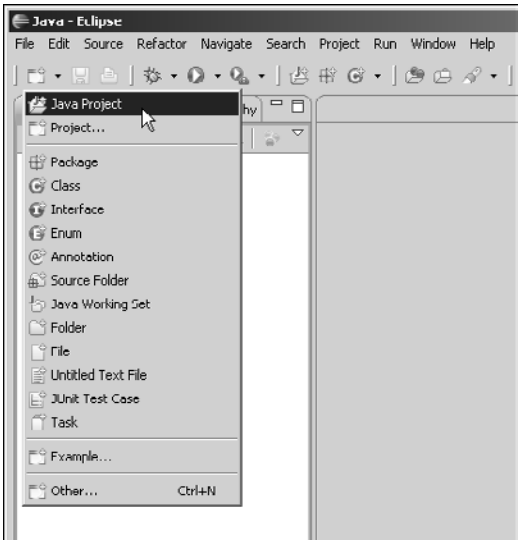

Рис. П.22. Создание нового проекта с помощью кнопки панели инструментов

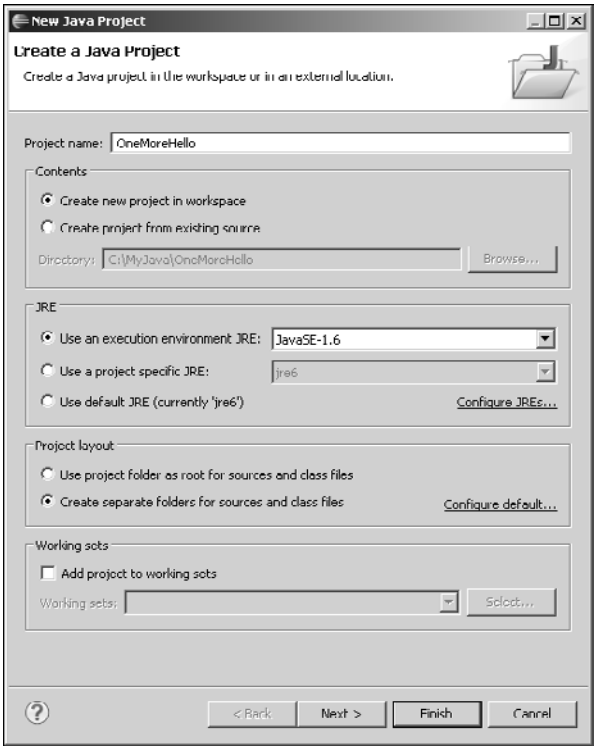

Рис. П.23. Окно настройки создаваемого проекта

В поле Project name указывается название проекта. В группе JRE выбирается среда выполнения (обычно используются предлагаемые по умолчанию параметры), назначение переключателей прочих групп должно быть понятно из их названий. На рис. П.24 показано окно среды Eclipse со вновь созданным проектом, в котором пока нет классов пользователя.

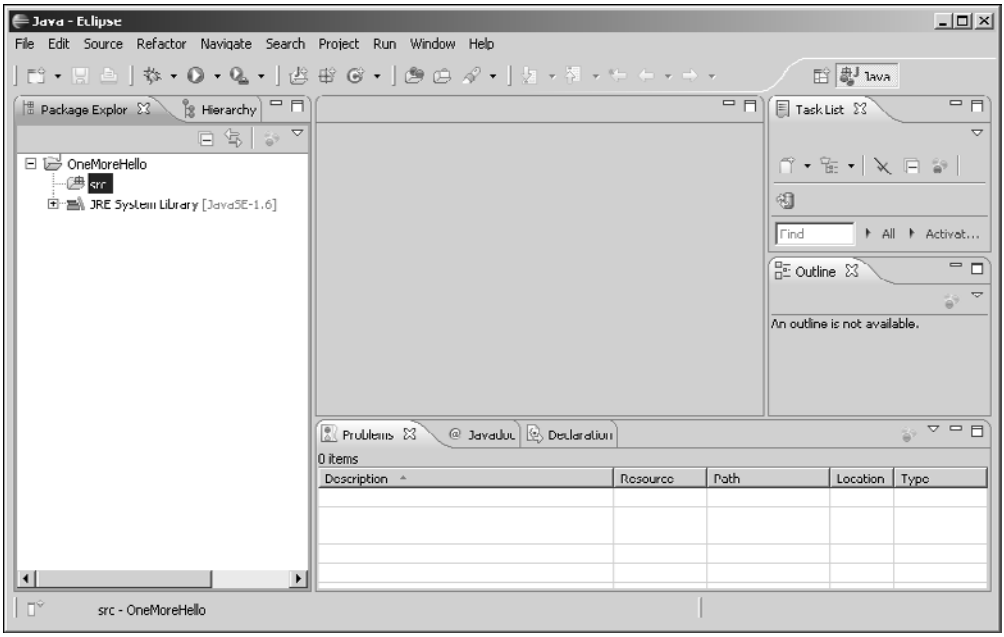

Рис. П.24. Созданный проект не содержит классов пользователя

Для добавления нового класса в проект используем команду File ▸ New ▸ Class (рис. П.25) или уже упоминавшуюся кнопку панели инструментов.

В окне настройки класса New Java Class задается ряд параметров (рис. П.26).

В частности, в поле Name указывается имя класса. Переключатель public в группе Modifiers позволяет добавить к классу одноименный атрибут. При необходимости можно установить флажки для создания абстрактного класса или класса, не допускающего наследование. Полезной может также оказаться флажок public static void main(String[] args) — если его установить, в код вновь созданного класса будет добавлена сигнатура главного метода программы. Результат создания нового класса показан на рис. П.27.

Код класса после внесения изменений представлен на рис. П.28.

Для запуска программы можно воспользоваться специальной кнопкой на панели инструментов (рис. П.29) или командами меню Run.

Перед выполнением программы может появиться окно, как показано на рис. П.30.

С предложенным выбором лучше согласиться, если, конечно, планы по запуску приложения не изменились. Результат выполнения программы показан на рис. П.31.

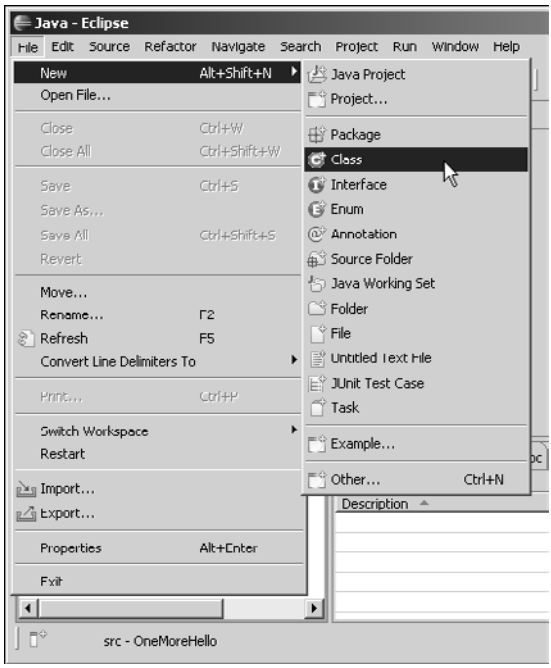

Рис. П.25. Создание нового класса

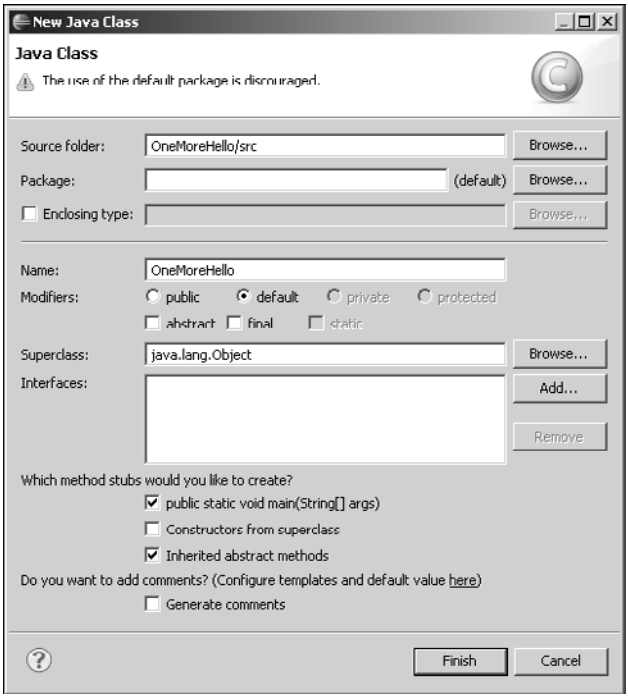

Рис. П.26. Окно настройки создаваемого класса

В частности, результат (выводимое на консоль сообщение) отображается на вкладке Console в нижней части окна среды Eclipse.

Разумеется, в этом приложении были представлены далеко не все возможности как одной, так и другой среды разработки. Однако подробное их описание однозначно выходит за рамки тематики книги. Поэтому заинтересованному в рассмотрении этих вопросов читателю можем порекомендовать обратиться к специальной литературе или справочной системе представленных здесь сред.

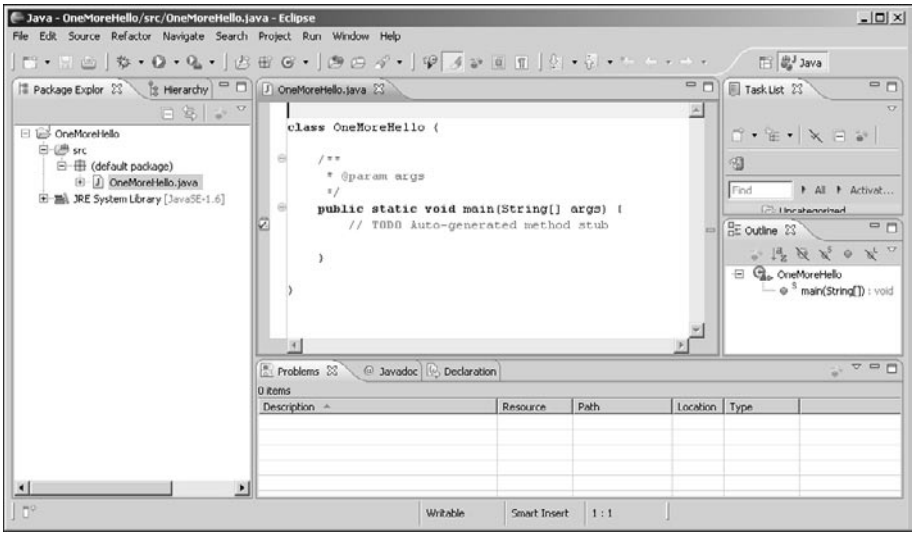

Рис. П.27. Код вновь созданного класса в окне Eclipse

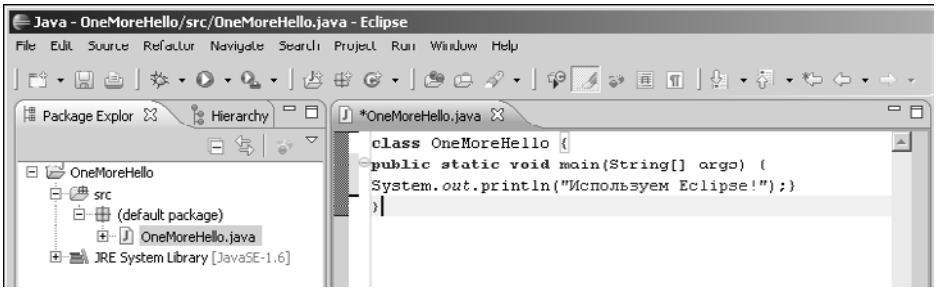

Рис. П.28. Программный код класса в окне среды Eclipse

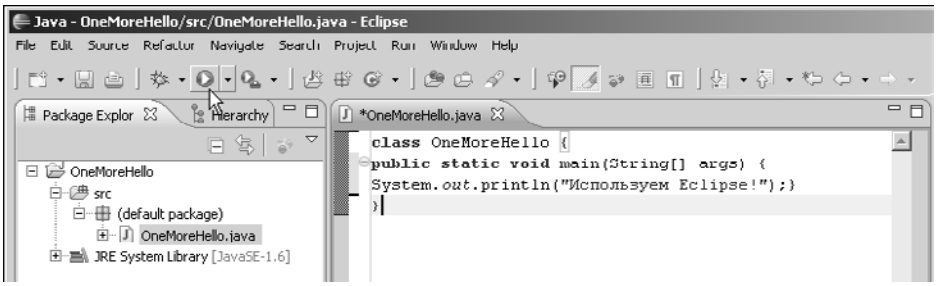

Рис. П.29. Запуск программы

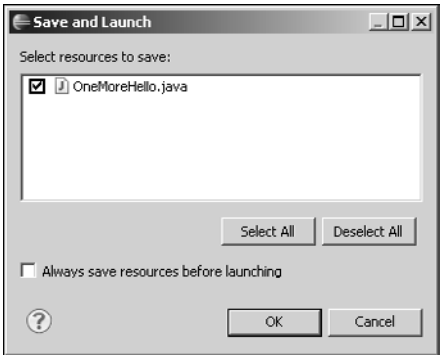

Рис. П.30. Окно сохранения и запуска проекта

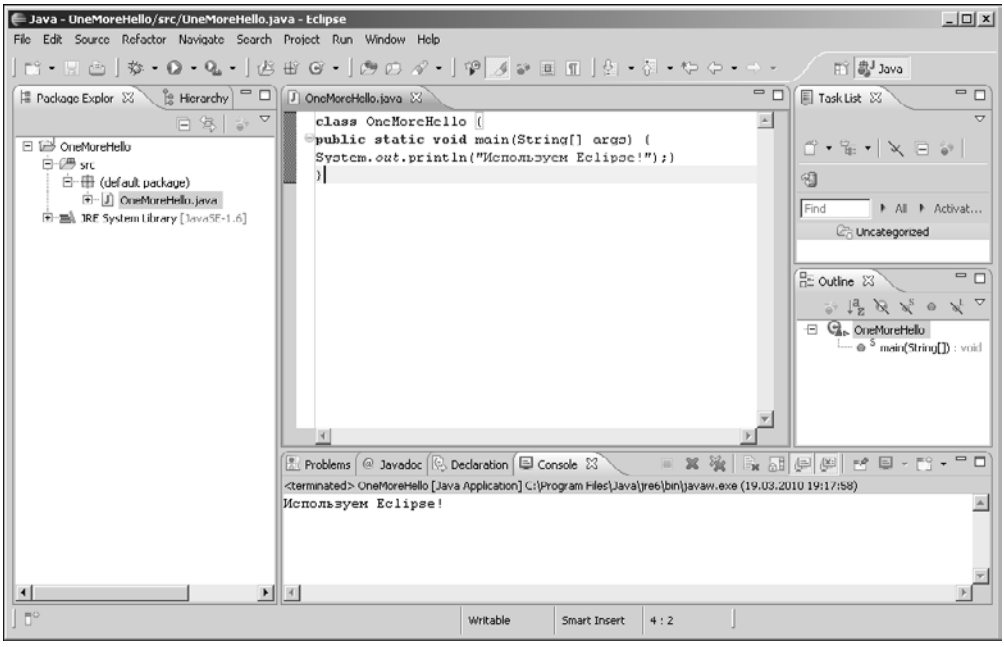

Рис. П.31. Результат выполнения программы

Алфавитный указатель

Алексей Николаевич Васильев

Java. Объектно-ориентированное программирование: Учебное пособие

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Редактор	*А. Жданов*
Художественный редактор	*К. Радзевич*
Корректор	*В. Листова*
Верстка	*Е. Егорова*

Подписано в печать 27.08.10. Формат 70x100/16. Усл. п. л. 32,25. Тираж 2000. Заказ 0000.
ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73**. **E-mail:** fuganov@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству авторов. Обращайтесь по телефонам: **Санкт-Петербург** – **(812) 703-73-72**, **Москва** – **(495) 974-34-50**

Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ У НАШИХ
РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

ДАЛЬНИЙ ВОСТОК

Владивосток
«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

Воронеж, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»
Московское шоссе, д.15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

УРАЛ

Екатеринбург, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

- *Зарегистрируйтесь на нашем сайте в качестве партнера по адресу* ***www.piter.com/ePartners***
- *Получите свой персональный уникальный номер партнера*
- *Выбирайте книги на сайте* ***www.piter.com,*** *размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт* ***www.piter.com****)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM