Справочник по программированию на Java

Методическое пособие

**Центр компьютерного обучения «Специалист», 2016**

Справочник по программированию на Java

Методическое пособие

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм. Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на электронный носитель, если на это нет письменного разрешения автора.

© Центр компьютерного обучения «Специалист» при МГТУ им. Н.Э. Баумана, 2016 г.

Оглавление

[Введение в Java технологии 8](#_Toc451092631)

[Простота 8](#_Toc451092632)

[Объектная ориентированность 8](#_Toc451092633)

[Устойчивость 9](#_Toc451092634)

[Многопоточность 10](#_Toc451092635)

[Архитектурная нейтральность 10](#_Toc451092636)

[Интерпретируемость и высокая производительность 10](#_Toc451092637)

[Распределенность 10](#_Toc451092638)

[Динамический характер 11](#_Toc451092639)

[Обзор языка Java 11](#_Toc451092640)

[Объектно-ориентированное программирование 11](#_Toc451092641)

[Абстракция 12](#_Toc451092642)

[Инкапсуляция 13](#_Toc451092643)

[Наследование 14](#_Toc451092644)

[Полиморфизм 15](#_Toc451092645)

[Совместное использование полиморфизма, инкапсуляции и наследования 15](#_Toc451092646)

[Первый пример простой программы 17](#_Toc451092647)

[Ввод кода программы 17](#_Toc451092648)

[Компиляция программы 17](#_Toc451092649)

[Второй пример короткой программы 21](#_Toc451092650)

[Типы данных и переменные 23](#_Toc451092651)

[Java – строго типизированный язык 23](#_Toc451092652)

[Элементарные типы 23](#_Toc451092653)

[Целочисленные значения 24](#_Toc451092654)

[Типы с плавающей точкой 25](#_Toc451092655)

[Символы 25](#_Toc451092656)

[Булевские значения 27](#_Toc451092657)

[Константы 28](#_Toc451092658)

[Константы с плавающей точкой 28](#_Toc451092659)

[Булевские константы 29](#_Toc451092660)

[Символьные константы 29](#_Toc451092661)

[Строковые константы 30](#_Toc451092662)

[Область определения и время существования переменных 30](#_Toc451092663)

[Преобразование и приведение типов 33](#_Toc451092664)

[Автоматическое преобразование типов в Java 33](#_Toc451092665)

[Приведение несовместимых типов 34](#_Toc451092666)

[Автоматическое повышение типа в выражениях 35](#_Toc451092667)

[Правила повышения типа 36](#_Toc451092668)

[Массивы 37](#_Toc451092669)

[Одномерные массивы 37](#_Toc451092670)

[Многомерные массивы 40](#_Toc451092671)

[Альтернативный синтаксис объявления массивов 44](#_Toc451092672)

[Строки 44](#_Toc451092673)

[Указатели 45](#_Toc451092674)

[Операции 46](#_Toc451092675)

[Арифметические операции 46](#_Toc451092676)

[Побитовые операции 50](#_Toc451092677)

[Побитовые логические операции 52](#_Toc451092678)

[Сдвиг влево 54](#_Toc451092679)

[Сдвиг вправо 56](#_Toc451092680)

[Сдвиг вправо без учета знака 57](#_Toc451092681)

[Побитовые составные операции с присваиванием 59](#_Toc451092682)

[Операции сравнения 60](#_Toc451092683)

[Булевские логические операции 61](#_Toc451092684)

[Замыкающие логические операции 62](#_Toc451092685)

[Операция присваивания 63](#_Toc451092686)

[Операция ? 64](#_Toc451092687)

[Управляющие операторы 64](#_Toc451092688)

[Операторы выбора 65](#_Toc451092689)

[Вложенные операторы if 66](#_Toc451092690)

[Многозвенная структура if-else-if 67](#_Toc451092691)

[Оператор switch 68](#_Toc451092692)

[Вложенные операторы switch 71](#_Toc451092693)

[Операторы цикла 72](#_Toc451092694)

[Цикл while 72](#_Toc451092695)

[Цикл do-while 74](#_Toc451092696)

[Цикл for 77](#_Toc451092697)

[Объявление управляющих переменных цикла внутри цикла for 78](#_Toc451092698)

[Версия “for-each” цикла for 80](#_Toc451092699)

[Итерация в многомерных массивах 84](#_Toc451092700)

[Использование усовершенствованного цикла for 85](#_Toc451092701)

[Вложенные циклы 86](#_Toc451092702)

[Операторы перехода 87](#_Toc451092703)

[Использование оператора break 87](#_Toc451092704)

[Использование оператора break в качестве формы оператора безусловного перехода 89](#_Toc451092705)

[Использование оператора continue 91](#_Toc451092706)

[Оператор return 93](#_Toc451092707)

[Разработка классов на Java 94](#_Toc451092708)

[Общая форма класса 94](#_Toc451092709)

[Простой класс 96](#_Toc451092710)

[Объявление объектов 99](#_Toc451092711)

[Более подробное рассмотрение операции new 100](#_Toc451092712)

[Присваивание переменных объектных ссылок 101](#_Toc451092713)

[Знакомство с методами 101](#_Toc451092714)

[Возвращение значения 104](#_Toc451092715)

[Добавление метода, принимающего параметры 105](#_Toc451092716)

[Конструкторы 107](#_Toc451092717)

[Конструкторы с параметрами 109](#_Toc451092718)

[Ключевое слово this 110](#_Toc451092719)

[Сокрытие переменной экземпляра 111](#_Toc451092720)

[Метод finalize() 112](#_Toc451092721)

[Перегрузка методов 112](#_Toc451092722)

[Перегрузка конструкторов 113](#_Toc451092723)

[Использование объектов в качестве параметров 115](#_Toc451092724)

[Возврат объектов 118](#_Toc451092725)

[Введение в управление доступом 119](#_Toc451092726)

[Ключевое слово static 121](#_Toc451092727)

[Ключевое слово final 123](#_Toc451092728)

[Массивы как классы 124](#_Toc451092729)

[Представление вложенных и внутренних классов 124](#_Toc451092730)

[Работа со строками в Java 128](#_Toc451092731)

[Использование аргументов командной строки 130](#_Toc451092732)

[Аргументы переменной длины 131](#_Toc451092733)

[Параметры переменной длины и неопределенность 133](#_Toc451092734)

[Наследование и полиморфизм 135](#_Toc451092735)

[Основы наследования 135](#_Toc451092736)

[Доступ к членам и наследование 137](#_Toc451092737)

[Переменная суперкласса может ссылаться на объект подкласса 139](#_Toc451092738)

[Использование ключевого слова super 140](#_Toc451092739)

[Создание многоуровневой иерархии 142](#_Toc451092740)

[Порядок вызова конструкторов 143](#_Toc451092741)

[Переопределение методов 144](#_Toc451092742)

[Динамическая диспетчеризация методов 146](#_Toc451092743)

[Использование переопределения методов 148](#_Toc451092744)

[Использование абстрактных классов 150](#_Toc451092745)

[Использование ключевого слова final для предотвращения переопределения 153](#_Toc451092746)

[Использование ключевого слова final для предотвращения наследования 154](#_Toc451092747)

[Класс Object 154](#_Toc451092748)

[Интерфейсы и пакеты 155](#_Toc451092749)

[Пакеты 156](#_Toc451092750)

[Защита доступа 157](#_Toc451092751)

[Импорт пакетов 158](#_Toc451092752)

[Интерфейсы 159](#_Toc451092753)

[Определение интерфейса 160](#_Toc451092754)

[Реализация интерфейсов 161](#_Toc451092755)

[Доступ к реализациям через ссылки на интерфейсы 162](#_Toc451092756)

[Частичные реализации 163](#_Toc451092757)

[Вложенные интерфейсы 164](#_Toc451092758)

[Переменные в интерфейсах 164](#_Toc451092759)

[Возможность расширения интерфейсов 166](#_Toc451092760)

[Аннотации (метаданные) 166](#_Toc451092761)

[Основы аннотирования 166](#_Toc451092762)

[Спецификация политики удержания 167](#_Toc451092763)

[Получение аннотаций во время выполнения с использованием рефлексии 168](#_Toc451092764)

[Получение всех аннотаций 170](#_Toc451092765)

[Встроенные аннотации 172](#_Toc451092766)

[Обработка ошибок в Java 173](#_Toc451092767)

[Типы исключений 174](#_Toc451092768)

[Необработанные исключения 175](#_Toc451092769)

[Использование try и catch 175](#_Toc451092770)

[Множественные операторы catch 177](#_Toc451092771)

[Вложенные операторы try 178](#_Toc451092772)

[throw 179](#_Toc451092773)

[throws 181](#_Toc451092774)

[finally 181](#_Toc451092775)

[Встроенные исключения Java 183](#_Toc451092776)

[Создание собственных подклассов исключений 183](#_Toc451092777)

[Сцепленные исключения 183](#_Toc451092778)

[Потоки данных в Java 184](#_Toc451092779)

[Потоки 185](#_Toc451092780)

[Байтовые и символьные потоки 185](#_Toc451092781)

[Классы байтовых потоков 185](#_Toc451092782)

[Классы символьных потоков 186](#_Toc451092783)

[Предопределенные потоки 187](#_Toc451092784)

[Чтение консольного ввода 188](#_Toc451092785)

[Класс PrintWriter 189](#_Toc451092786)

[Чтение и запись файлов 190](#_Toc451092787)

[Generic типы: параметризованный или обобщенный код 194](#_Toc451092788)

[Простой пример обобщения 195](#_Toc451092789)

[Общая форма обобщенного класса 201](#_Toc451092790)

[Создание обобщенного метода 204](#_Toc451092791)

[Обобщенные конструкторы 206](#_Toc451092792)

[Обобщенные интерфейсы 207](#_Toc451092793)

[Иерархии обобщенных классов 209](#_Toc451092794)

[Обобщенный подкласс 211](#_Toc451092795)

[Ошибки неоднозначности 212](#_Toc451092796)

[Ограничения на статические члены 213](#_Toc451092797)

[Ограничения обобщенных массивов 214](#_Toc451092798)

[Ограничения обобщенных исключений 214](#_Toc451092799)

# Введение в Java технологии

Основные факторы, обусловившие изобретение языка Java ­– необходимость обеспечения переносимости и безопасности, другие факторы также сыграли свою роль в формировании окончательной версии языка. Группа разработки Java обобщила основные понятия в следующем перечне терминов:

* простота;
* безопасность;
* переносимость;
* объектная ориентированность;
* устойчивость;
* многопоточность;
* архитектурная нейтральность;
* интерпретируемость;
* высокая производительность;
* распределенность;
* динамический характер.

Рассмотрим значения этих терминов.

## Простота

Java был задуман в качестве простого в изучении и эффективного в использовании профессиональными программистами языка. Для тех, кто обладает определенным опытом программирования, овладение языком Java не представит особой сложности. Если же вы уже знакомы с базовыми концепциями объектно-ориентированного программирования, изучение Java будет еще проще. А для опытного программиста на C++ переход к Java вообще потребует минимум усилий. Поскольку Java наследует синтаксис C/C++ и многие объектно-ориентированные свойства C++, для большинства программистов изучение Java не представит сложности.

## Объектная ориентированность

Хотя предшественники языка Java и оказали влияние на его архитектуру и синтаксис, при его проектировании задача совместимости по исходному коду с каким-либо другим языком не ставилась. Это позволило группе разработки Java выполнять проектирование, что называется, с чистого листа. Одним из следствий этого явился четкий, практичный, прагматичный подход к объектам. Притом что Java позаимствовал свойства многих удачных объектно-программных сред, разработанных на протяжении нескольких последних десятилетий, в нем удалось достичь баланса между строгим соблюдением концепции “все компоненты программы – объекты” и более прагматичной моделью “прочь с дороги”. Объектная модель Java проста и легко расширяема. В то же время примитивные типы, такие как целые числа, сохраняются в виде высокопроизводительных компонентов, не являющихся объектами.

## Устойчивость

Многоплатформенная среда Web предъявляет к программам повышенные требования, поскольку они должны надежно выполняться в разнообразных системах. Поэтому способность создавать устойчивые программы была одним из главных приоритетов при проектировании Java. Для обеспечения надежности Java накладывает ряд ограничений в нескольких наиболее важных областях, что вынуждает программиста выявлять ошибки на ранних этапах разработки программы. В то же время Java избавляет от беспокойства по поводу многих наиболее часто встречающихся ошибок программирования. Поскольку Java – строго типизированный язык, проверка кода выполняется во время компиляции. Однако проверка кода осуществляется и во время выполнения. В результате многие трудно обнаруживаемые программные ошибки, которые часто ведут к возникновению трудновоспроизводимых ситуаций времени выполнения, в Java-программе попросту невозможны.

Предсказуемость кода в различных ситуациях – одна из основных особенностей Java. Чтобы понять, чем достигается устойчивость Java-программ, рассмотрим две основных причины программных сбоев: ошибки управления памятью и неправильная обработка исключений (т.е. ошибки времени выполнения). В традиционных средах создания программ управление памятью – сложная и трудоемкая задача. Например, в среде C/C++ программист должен вручную резервировать и освобождать всю динамически распределяемую память. Иногда это ведет к возникновению проблем, поскольку программисты либо забывают освободить ранее зарезервированную память, либо, что еще хуже, пытаются освободить участок памяти, все еще используемый другой частью кода. Java полностью исключает такие ситуации, автоматически управляя резервированием и освобождением памяти. (Фактически, освобождение выполняется полностью автоматически, поскольку Java предоставляет функцию сборки мусора в отношении неиспользуемых объектов.)

В традиционных средах условия исключений часто возникают в таких ситуациях, как деление на ноль или “файл не найден”, и управление ими должно осуществляться с помощью громоздких и трудных для понимания конструкций. Java облегчает выполнение той задачи, предлагая объектно-ориентированный механизм обработки исключений. В хорошо написанной Java-программе все ошибки времени выполнения могут – и должны – управляться самой программой.

## Многопоточность

Язык Java был разработан в ответ на потребность создания интерактивных сетевых программ. Для достижения этой цели Java поддерживает написание многопоточных программ, которые могут одновременно выполнять много действий. Система времени выполнения Java содержит изящное, но вместе с тем сложное решение задачи синхронизации множества процессов, которое позволяет создавать действующие без перебоев интерактивные системы. Простой в применении подход к организации многопоточной обработки, реализованный в Java, позволяет программисту сосредоточивать свое внимание на конкретном поведении программы, а не на создании многозадачной подсистемы.

## Архитектурная нейтральность

Основной задачей, которую ставили перед собой разработчики Java, было обеспечение долговечности и переносимости кода. Одна из главных проблем, стоящих перед программистами – отсутствие гарантий того, что код, созданный сегодня, будет успешно выполняться завтра, даже на том же самом компьютере. Операционные системы и процессоры модернизируются, а все изменения в основных системных ресурсах могут приводить к неработоспособности программ. Пытаясь изменить эту ситуацию, проектировщики приняли ряд жестких решений в языке Java и виртуальной машине Java. Они поставили перед собой цель, чтобы “программы создавались лишь однажды, в любой среде, в любое время и навсегда”. В значительной степени эта цель была достигнута.

## Интерпретируемость и высокая производительность

Как уже говорилось, выполняя компиляцию программ в промежуточное представление, называемое байт-кодом, Java позволяет создавать многоплатформенные программы. Этот код может выполняться в любой системе, которая реализует виртуальную машину Java. Самые первые попытки получения многоплатформенных решений добивались поставленной цели за счет снижения производительности. Как пояснялось ранее, байт-код Java был тщательно спроектирован так, чтобы посредством использования JIT-компиляции его можно было с высокой производительностью легко преобразовывать во внутренний машинный код. Системы времени выполнения Java, которые предоставляют эту функцию, сохраняют все преимущества кода, не зависящего от платформы.

## Распределенность

Язык Java предназначен для распределенной среды Internet, поскольку он поддерживает протоколы семейства TCP/IP. Фактически обращение к ресурсу через URL-адрес не очень отличается от обращения к файлу. Java поддерживает также удаленный вызов методов (Remote Method Invocation – RMI). Это свойство позволяет программам вызывать методы по сети.

## Динамический характер

Программы Java содержат значительный объем информации времени выполнения, которая используется для проверки полномочий и предоставления доступа к объектам во время выполнения. Это позволяет выполнять безопасную и технически оправданную динамическое связывание кода. Это обстоятельство исключительно важно для устойчивости среды Java, в которой небольшие фрагменты байт-кода могут динамически обновляться в действующей системе.

# Обзор языка Java

Как и во всех компьютерных языках, элементы Java существуют не сами по себе. Скорее они работают совместно, образуя язык в целом. Однако эта взаимосвязанность может затруднять описание какого-то одного аспекта Java, не касаясь при этом нескольких других аспектов. Часто для понимания одного свойства требуется наличия знаний о другом. Поэтому в настоящей главе представлен краткий обзор нескольких основных свойств языка Java.

## Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) – это основная характеристика языка Java. Фактически все Java-программы являются, по меньшей мере, частично, объектно-ориентированными. Язык Java связан с ООП настолько тесно, что прежде чем приступать к написанию на нем даже простейших программ, следует вначале ознакомиться с базовыми принципами ООП. Поэтому ознакомление с материалом этой главы мы начнем с рассмотрения теоретических аспектов ООП.

Все компьютерные программы состоят из двух элементов: кода и данных. Более того, концептуально программа может быть организована вокруг своего кода или вокруг своих данных. То есть организация некоторых программ определяется тем “что происходит”, а других – тем, “на что оказывается влияние”. Существуют две концепции создания программы. Первый способ называют моделью, ориентированной на процессы. Этот подход характеризует программу в виде последовательностей линейных шагов (т.е. кода).

Модель, ориентированную на процессы, можно рассматривать в качестве кода, воздействующего на данные. Процедурные языки вроде C достаточно успешно используют эту модель. Однако этот подход порождает ряд проблем с увеличением размеров и сложности программ.

Для преодоления увеличивающейся сложности была начата разработка подхода, названного объектно-ориентированным программированием. Объектно-ориентированное программирование организует программу вокруг ее данных (т.е. объектов) и набора подробно определенных интерфейсов к этим данным. Объектно-ориентированную программу можно характеризовать как данные, управляющие доступом к коду.

## Абстракция

Важный элемент объектно-ориентированного программирования – абстракция.

Человеку свойственно представлять сложные явления и объекты посредством абстракции. Например, люди представляют себе автомобиль не в виде набора десятков тысяч отдельных деталей, а в виде совершенно определенного объекта, имеющего собственное уникальное поведение. Эта абстракция позволяет не задумываться о сложности деталей, образующих автомобиль, используя его, скажем, для поездки в магазин. Мы можем не обращать внимания на подробности работы двигателя, коробки передач и тормозной системы. Вместо этого объект можно использовать как единое целое.

Мощное средство применения абстракции – применение иерархических классификаций. Это позволяет упрощать семантику сложных систем, разбивая их на более пригодные для управления фрагменты. Внешне автомобиль выглядит единым объектом. Но стоит заглянуть внутрь, как становится ясно, что он состоит из нескольких подсистем: рулевого управления, тормозов, аудиосистемы, привязных ремней, обогревателя, навигатора и т.п. Каждая из этих подсистем, в свою очередь, собрана из более специализированных узлов. Например, аудиосистема состоит из радиоприемника, проигрывателя компакт-дисков и/или устройства воспроизведения аудиокассет. Суть сказанного в том, что сложную структуру автомобиля (или любой другой сложной системы) можно описать с помощью иерархических абстракций.

Иерархические абстракции сложных систем можно применять и к компьютерным программам. Посредством абстракции данные традиционной ориентированной на процессы программы можно преобразовать в составляющие ее объекты. При этом последовательность шагов процесса может быть преобразована в коллекцию сообщений, передаваемых между этими объектами. Таким образом, каждый и этих объектов описывает свое уникальное поведение. Эти объекты можно считать конкретными элементами, которые отвечают на сообщения, указывающие им о необходимости выполнить что-либо.

Концепции объектной ориентированности лежат в основе языка Java, точно так же, как они лежат в основе восприятия мира человеком. Важно понимать, как эти концепции реализуются в программах. Как вы увидите, объектно-ориентированное программирование – мощная и естественная концепция создания программ, которые способны пережить неизбежные изменения, сопровождающие жизненный цикл любого крупного программного проекта, включая создание концепции, рост и старение. Например, при наличии тщательно определенных объектов и четких, надежных интерфейсов к этим объектам можно безбоязненно и без особых проблем извлекать или заменять части старой системы.

## Инкапсуляция

Инкапсуляция – механизм, который связывает код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений. Один из возможных способов представления инкапсуляции – представление в виде защитной оболочки, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируются тщательно определенным интерфейсом. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует сотни бит информации об автомобиле, такой как степень ускорения, крутизна поверхности, по которой совершается движение и положение рычага переключения скоростей. Пользователь (водитель) может влиять на эту сложную инкапсуляцию только одним методом: перемещая рычаг переключения скоростей. На коробку передач нельзя влиять, например, посредством индикатора поворота или дворников. Таким образом, рычаг переключения скоростей – строго определенный (а в действительности единственный) интерфейс к коробке передач. Более того, происходящее внутри коробки передач, не влияет на объекты, находящиеся вне ее. Например, переключение передач не включает фары! Поскольку функция автоматического переключения передач инкапсулирована, десятки изготовителей автомобилей могут реализовать ее каким угодно способом. Однако с точки зрения водителя все эти коробки передач работают одинаково.

Аналогичную идею можно применять к программированию. Сила инкапсулированного кода в том, что все знают, как к нему можно получить доступ, и, следовательно, могут его использовать независимо от нюансов реализации и не опасаясь неожиданных побочных эффектов. В языке Java основой инкапсуляции является класс. Хотя подробнее мы рассмотрим классы в последующих главах книги, сейчас полезно ознакомиться со следующим кратким описанием. Класс определяет структуру и поведение (данные и код), которые будут совместно использоваться набором объектов. Каждый объект данного класса содержит структуру и поведение, которые определены классом, как если бы объект был “отлит” в форме класса. Поэтому иногда объекты называют экземплярами класса. Таким образом, класс – это логическая конструкция, а объект имеет физическое воплощение.

При создании класса определяют код и данные, которые образуют этот класс. Совокупность этих элементов называют членами класса. В частности, определенные классом данные называют переменными -членами или переменными экземпляра. Код, который выполняет действия по отношению к данным, называют переменными-методами или просто методами. (То, что программисты на Java называют методом, программисты на C/C++ называют функциями.) В правильно написанных Java-программах методы определяют способы использования переменных-членов. Это означает, что поведение и интерфейс класса определяются методами, которые выполняют действия по отношению к данным его экземпляра.

Поскольку назначение класса – инкапсуляция сложной структуры программы, существуют механизмы сокрытия сложной структуры реализации внутри класса. Каждый метод или переменная в классе может быть помечена как приватная или общедоступная. Общедоступный интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. Приватные методы и данные могут быть доступны только для кода, который является членом данного класса. Следовательно, любой другой код, не являющийся членом класса, не может получать доступ к приватному методу или переменной. Поскольку приватные члены класса доступны другим частям программы только посредством общедоступных методов класса, можно быть уверенным в невозможности выполнения неправомерных действий. Конечно, это означает, что общедоступный интерфейс должен быть тщательно спроектирован, открывая не слишком много нюансов внутренней работы класса.

## Наследование

Наследование – процесс, посредством которого один объект получает свойства другого объекта. Это важно, поскольку он поддерживает концепцию иерархической классификации. Как уже отмечалось, большинство сведений становится доступным для понимания посредством иерархических (т.е. нисходящих) классификаций.

Как правило, большинство людей воспринимает окружающий мир в виде иерархически связанных между собой объектов, подобных животным, млекопитающим и собакам. Если требуется привести абстрактное описание животных, можно сказать, что они обладают определенными атрибутами, такими как размеры, уровень интеллекта и тип скелета. Животным присущи также определенные особенности поведения: они едят, дышат и спят. Приведенное описание атрибутов и поведения – определение класса животных.

Если бы требовалось описать более конкретный класс животных, например млекопитающих, нужно было бы указать более конкретные атрибуты, такие как тип зубов и молочных желез. Это определение называют подклассом животных, которые относятся к суперклассу (родительскому классу) млекопитающих.

Поскольку млекопитающие – всего лишь более точно определенные животные, они наследуют все атрибуты животных. Подкласс нижнего уровня иерархии классов наследует все атрибуты каждого из его родительских классов. Наследование связано также с инкапсуляцией. Если данный класс инкапсулирует определенные атрибуты, то любой его подкласс будет иметь эти же атрибуты плюс любые дополнительные атрибуты, являющиеся составной частью его специализации.

Эта ключевая концепция делает возможным возрастание сложности объектно-ориентированных программ в линейной, а не геометрической прогрессии. Новый подкласс наследует все атрибуты всех своих родительских классов. Поэтому он не содержит непредсказуемых взаимодействий с большей частью остального кода системы.

## Полиморфизм

Полиморфизм – свойство, которое позволяет использовать один и тот же интерфейс для общего класса действий. Конкретное действие определяется конкретным характером ситуации.

Рассмотрим стек (представляющий собой список типа “последним вошел, первым вышел”). Может существовать программа, которая требует применения трех типов стеков.

Один стек используется для целочисленных значений, один – для значений с плавающей точкой и один – для символов. Алгоритм реализации каждого из этих стеков остается неизменным, несмотря на различие хранящихся в них данных. В не объектно-ориентированном языке пришлось бы создавать три различных набора подпрограмм стека, каждый из которых должен был бы иметь отдельное имя. Однако в Java, благодаря полиморфизму, можно определить общий набор подпрограмм стека, использующих одни и те же имена.

В более общем виде концепцию полиморфизма часто выражают фразой “один интерфейс, несколько методов”. Это означает, что можно спроектировать общий интерфейс для группы связанных между собой действий. Этот подход позволяет уменьшить сложность программы, поскольку один и тот же интерфейс используется для указания общего класса действий. Выбор же конкретного действия (т.е. метода), применимого к каждой ситуации – задача компилятора. Программисту не нужно осуществлять этот выбор вручную.

Необходимо лишь помнить об общем интерфейсе и применять его.

## Совместное использование полиморфизма, инкапсуляции и наследования

При правильном совместном использовании полиморфизма, инкапсуляции и наследования они создают среду программирования, которая поддерживает разработку более устойчивых и масштабируемых программ, чем в случае применения модели, ориентированной на процессы. Тщательно спроектированная иерархия классов – основа многократного использования кода, на разработку и тестирование которого были затрачены время и усилия. Инкапсуляция позволяет возвращаться к ранее созданным реализациям, не разрушая код, зависящий от общедоступного интерфейса применяемых в приложении классов. Полиморфизм позволяет создавать понятный, чувствительный, удобочитаемый и устойчивый код.

Из двух приведенных примеров реального мира пример с автомобилем полнее иллюстрирует возможности объектно-ориентированного проектирования. Садясь за руль различных типов (подклассов) автомобилей, все водители используют наследование. Независимо от того, является ли автомобиль школьным автобусом, легковым мерседесом, порше или семейным микроавтобусом, все водители могут более-менее легко найти и пользоваться рулем, тормозами и педалью акселератора. Немного помучившись с рычагом переключения передач, большинство людей может даже оценить различия между ручной и автоматической коробками передач – это становится возможным благодаря получению четкого представления об общем родительском классе этих объектов – системе передач.

В процессе использования автомобилей люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность соответствующих объектов за настолько простым интерфейсом, что управлять этими объектами можно простым нажатием на педали.

Последний атрибут, полиморфизм, четко отражает способность компаний-изготовителей автомобилей предлагать широкое множество вариантов, по сути, одного и того же средства передвижения. Например, на автомобиле могут быть установлены система тормозов с защитой от блокировки или традиционные тормоза, рулевая система с гидроусилителем или с реечной передачей и 4-, 6- или 8-цилиндровые двигатели. В любом случае нужно будет жать на педаль тормоза, чтобы остановиться, вращать руль, чтобы повернуть, и жать педаль акселератора, чтобы автомобиль двигался. Один и тот же интерфейс может применяться для управления множеством различных реализаций.

Как видите, именно совместное применение инкапсуляции, наследования и полиморфизма позволяет преобразовать отдельные детали в объект, который мы называем автомобилем. Сказанное применимо также к компьютерным программам. За счет применения объектно-ориентированных принципов различные части сложной программы могут быть собраны воедино, образуя надежную, пригодную для обслуживания программу.

# Первый пример простой программы

Мы начнем с компиляции и запуска представленного в этом разделе короткого примера программы. Вы убедитесь, что эта задача несколько более трудоемкая, чем может казаться.

/\*

Это простая программа Java

\*/

class Example {

// Программа начинается с обращения к main().

public static void main(String args[]) {

System.out.println("Привет, Java!");

}

}

## Ввод кода программы

Для большинства языков программирования имя файла, который содержит исходный код программы, не имеет значения. Однако в Java это не так. Прежде всего, следует твердо усвоить, что присваиваемое исходному файлу имя очень важно. В данном случае именем исходного файла должно быть Example.java.

В Java исходный файл официально называется модулем компиляции. Он представляет собой текстовый файл, который содержит определения одного или более классов. Компилятор Java требует, чтобы исходный файл имел расширение .java. Как видно из кода программы, именем определенного программой класса является также Example. И это не случайно. В Java весь код должен размещаться внутри класса. В соответствии с принятым соглашением имя этого класса должно совпадать с именем файла, содержащего программу. Необходимо также, чтобы употребление строчных и прописных букв имени файла соответствовало их употреблению в имени класса.

Это обусловлено тем, что Java-код чувствителен к регистру символов. Пока что соглашение о соответствии имен файлов и имен классов может казаться произвольным.

Однако оно упрощает поддержку и организацию программ.

## Компиляция программы

Чтобы скомпилировать программу Example, запустите компилятор (javac), указав

имя исходного файла в командной строке:

C:\>javac Example.java

Компилятор javac создаст файл Example.class, содержащий байт-кодовую версию программы. Как мы уже отмечали, байт-код Java – это промежуточное представление программы, содержащее инструкции, которые будет выполнять виртуальная машина Java. Следовательно, результат работы компилятора javac не является непосредственно исполняемым кодом.

Чтобы действительно выполнить программу, необходимо воспользоваться программой запуска приложений Java, которая носит имя java. При этом ей потребуется передать имя класса Example в качестве аргумента командной строки, как показано в следующем примере:

C:\>java Example

При выполнении программы на экране отобразится следующий вывод:

*Привет, Java!*

В процессе компиляции исходного кода каждый отдельный класс помещается в собственный выходной файл, названный по имени класса и получающий расширение .class. Именно поэтому исходным файлам Java целесообразно присваивать имена, совпадающие с именами классов, которые они содержат – имя исходного файла будет совпадать с именем файла с расширением .class. При запуске java описанным способом в командной строке в действительности указывают имя класса, который нужно выполнить. Программа будет автоматически искать файл с указанным именем и расширением .class. Если программа найдет файл, она выполнит код, содержащийся в указанном классе.

Хотя программа Example.java достаточно коротка, с ней связано несколько важных особенностей, характерных для всех Java-программ. Давайте рассмотрим каждую часть этой программы более подробно.

Программа начинается со следующих срок:

/\*

Это простая программа Java

\*/

Этот фрагмент кода – комментарий. Подобно большинству других языков программирования, Java позволяет вставлять примечания в исходный файл программы.

Компилятор игнорирует содержимое комментариев. Эти комментарии служат описанием или пояснением действий программы для любого, кто просматривает исходный код.

В Java поддерживаются три стиля комментариев. Комментарий, приведенный в начале программы, называют многострочным комментарием. Этот тип комментария должен начинаться с символов /\* и заканчиваться символами \*/. Весь текст, помещенный между этими двумя символами комментария, компилятором игнорируется. Как следует из его названия, многострочный комментарий может содержать несколько строк.

Следующая строка программы имеет такой вид:

class Example {

В этой строке ключевое слово class используется для объявления о том, что выполняется определение нового класса. Example – это идентификатор, являющийся именем класса. Все определение класса, в том числе его членов, будет размещаться между открывающей ({) и закрывающей (}) фигурными скобками. В среде Java все действия программы осуществляются внутри класса. В этом состоит одна из причин того, что все Java-программы (по крайней мере, частично) являются объектноориентированными.

Следующая строка программы – однострочный комментарий:

// Программа начинается с обращения к main().

Это второй тип комментариев, поддерживаемый языком Java. Однострочный комментарий начинается с символов // и завершается концом строки. Как правило, программисты используют многострочные комментарии для вставки длинных примечаний, а однострочные комментарии – для помещения коротких, построчных описаний.

Следующая строка кода выглядит так:

public static void main(String args[]) {

Она начинается с метода main(). Как видно из предшествующего ей комментария, выполнение программы начинается с этой строки. Выполнение всех Java-приложений начинается с вызова метода main(). Пока мы не можем подробно раскрыть смысл каждой части этой строки, поскольку для этого требуется четкое представление о подходе Java к инкапсуляции. Однако, поскольку эта строка кода присутствует в большей части примеров первой части настоящей книги, давайте кратко рассмотрим значение каждой ее части.

Ключевое слово public – спецификатор доступа, который позволяет программисту управлять видимостью членов класса. Когда члену класса предшествует ключевое слово public, этот член доступен коду, расположенному вне класса, в котором определен данный член. (Противоположное ему по действию ключевое слово – private, которое препятствует использованию члена класса кодом, определенным вне его класса.) В данном случае метод main() должен быть определен как public, поскольку при запуске программы он должен вызываться кодом, определенным вне его класса. Ключевое слово static позволяет вызывать метод main() без конкретизации экземпляра класса. Это необходимо потому, что метод main() вызывается виртуальной машиной Java до создания каких-либо объектов. Ключевое слово void просто сообщает компилятору, что метод main() не возвращает никаких значений. Как будет показано в дальнейшем, методы могут также возвращать значения. Не беспокойтесь, если все сказанное кажется несколько сложным. Все эти концепции подробно рассматриваются в последующих главах. Как было отмечено, main() – метод, вызываемый при запуске Java-приложений.

Необходимо помнить, что язык Java чувствителен к регистру символов. Следовательно, строка Main не эквивалентна строке main. Важно понимать, что компилятор Java будет выполнять компиляцию классов, не содержащих метод main(). Но программа запуска приложений (java) не имеет средств запуска таких классов. Поэтому, если бы вместо main мы ввели Main, компилятор все равно выполнил бы компиляцию программы. Однако программа запуска приложений java сообщила бы об ошибке, поскольку не смогла бы найти метод main(). Для передачи любой информации, необходимой методу, служат переменные, указываемые в скобках, которые следуют за именем метода. Эти переменные называют параметрами. Если для данного метода никакие параметры не требуются, следует указывать пустые скобки. Метод main() содержит только один параметр, но достаточно сложный. Параметр String args[] объявляет параметр args, который представляет собой массив экземпляров класса String. (Массивы – это коллекции аналогичных объектов.) Объекты типа String хранят символьные строки. В данном случае параметр args принимает любые аргументы командной строки, присутствующие во время выполнения программы. В данной программе эта информация не используется, но в других программах, рассмотренных позже, она будет применяться.

Последним символом строки является символ фигурной скобки ({). Он обозначает начало тела метода main(). Весь код, образующий метод, будет располагаться между открывающей и закрывающей фигурными скобками метода.

Еще один важный момент: метод main() служит всего лишь началом программы.

Сложная программа может включать десятки классов, только один из которых должен содержать метод main(), чтобы выполнение было возможным. Аплеты – Java-программы, внедренные в Web-браузеры, вообще не используют метод main(), поскольку в них применяются другие средства запуска выполнения аплетов.

Следующая строка кода приведена ниже. Обратите внимание, что она помещена внутри метода main():

System.out.println("Простая Java-программа.");

Эта строка выводит на экран строку текста "Простая Java-программа.", за которой следует новая строка. В действительности вывод выполняется встроенным методом println(). В данном случае метод println() отображает переданную ему строку.

Строка начинается с идентификатора System.out. Хотя он слишком сложен, чтобы его можно было подробно пояснить на данном этапе, если говорить вкратце, System – это предопределенный класс, который предоставляет доступ к системе, а out – выходной поток, связанный с консолью.

Как легко догадаться, в реальных программах и аплетах Java консольный вывод (и ввод) используются не очень часто. Поскольку большинство современных компьютерных сред по своей природе являются оконными и графическими, в большинстве случаев консольный ввод-вывод применяется в простых служебных и демонстрационных программах. Позднее мы рассмотрим другие способы генерирования вывода с помощью Javа. А пока продолжим применять методы консольного ввода-вывода.

Обратите внимание, что оператор println() завершается символом точки с запятой. В Java все операторы заканчиваются этим символом. Причина отсутствия символа точки с запятой в конце остальных строк программы в том, что с технической точки зрения они не являются операторами.

Первый символ } завершает метод main(), а последний – определение класса Example.

## Второй пример короткой программы

Вероятно, ни одна другая концепция не является для языка программирования столь важной, как концепция переменных. Как вы, вероятно, знаете, переменная – это именованная ячейка памяти, которой может быть присвоено значение внутри программы. Во время выполнения программы значение переменной может изменяться. Следующая программа демонстрирует способы объявления переменной и присвоения ей значения. Она иллюстрирует также некоторые новые аспекты консольного вывода. Как следует из комментариев в начале программы, этому файлу нужно присвоить имя Example2.java.

/\*

Это еще один короткий пример.

\*/

class Example2 {

public static void main(String args[]) {

int num; // эта строка объявляет переменную по имени num

num = 100; // эта строка присваивает переменной num значение,

//равное 100

System.out.println("Это переменная num: " + num);

num = num \* 2;

System.out.print("Значение переменной num \* 2 равно ");

System.out.println(num);

}

}

При запуске этой программы на экране отобразится следующий вывод:

*Это переменная num: 100*

*Значение переменной num \* 2 равно 200*

Рассмотрим генерацию этого вывода подробнее. Первая, еще не знакомая читателям, строка этой программы:

int num; // эта строка объявляет переменную по имени num

Она объявляет целочисленную переменную num. Java (подобно большинству других языков) требует, чтобы переменные были объявлены до их использования.

В приведенном примере программы строка

num = 100; // эта строка присваивает переменной num значение,

//равное 100

, присваивает переменной num значение 100. В Java символом операции присваивания служит одиночный знак равенства, как и в языках C/C++. Специфику этих операций и операторов подробно мы разбирать не будем, считая, что их работа известна читателю.

Следующая строка кода выводит значение переменной num, которому предшествует текстовая строка "Это переменная num:".

System.out.println("Это переменная num: " + num);

В этом операторе знак плюса вызывает дописывание значения переменной num в конец предшествующей строки и вывод результирующей строки.

В действительности значение переменной num вначале преобразуется из целочисленного в строковый эквивалент, а затем объединяется с предшествующей строкой. Подробнее этот процесс еще будет описан.

Этот подход можно обобщить. Используя символ операции +, внутри одного оператора println() можно объединять любое необходимое количество строк.

Следующая строка кода присваивает переменной num значение этой переменной, умноженное на 2. Как и в большинстве других языков, в Java символ операции \* служит для указания операции умножения. После выполнения этой строки кода переменная num будет содержать значение, равное 200.

Следующие две строки программы выглядят так:

System.out.print("Значение переменной num \* 2 равно ");

System.out.println(num);

В них выполняется несколько новых действий. Во-первых, метод print() используется для отображения строки "Значение переменной num \* 2 равно". За этой строкой не следует символ новой строки. То есть следующий генерируемый вывод будет отображаться в той же строке. Метод print() полностью подобен методу println(), за исключением того, что после каждого вызова он не генерирует символ новой строки.

Теперь рассмотрим вызов метода println(). Обратите внимание, что имя переменной num используется само по себе. И print(), и println() могут применяться для вывода

значений любых встроенных типов Java.

# Типы данных и переменные

Java поддерживает несколько типов данных. Эти типы можно применять для объявления переменных и создания массивов. Как будет показано, подход к использованию этих компонентов, примененный в Java, прост, эффективен и целостен.

## Java – строго типизированный язык

Прежде всего, важно уяснить, что Java – строго типизированный язык. Действительно, в определенной степени безопасность и надежность Java-программ обусловлена именно этим обстоятельством. Давайте разберемся, что это означает. Во-первых, каждая переменная обладает типом, каждое выражение имеет тип, и каждый тип строго определен. Вовторых, все присваивания, как явные, так и посредством передачи параметров в вызовах методов, проверяются на соответствие типов. В Java отсутствуют какие-либо средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках. Компилятор Java проверяет все выражения и параметры на предмет совместимости типов. Любые несоответствия типов являются ошибками, которые должны быть исправлены до завершения компиляции класса.

## Элементарные типы

Java определяет восемь элементарных типов данных: byte, short, int, long, char, float, double и boolean.

Часто элементарные типы называют также простыми типами или дататипам. Элементарные типы можно разделить на четыре группы.

* Целочисленные. Эта группа включает в себя типы byte, short, int и long, которые представляют точные целые числа со знаком.
* Числа с плавающей точкой. Эта группа включает в себя типы float и double, которые представляют числа, определенные с точностью до определенного десятичного знака.
* Символы. Эта группа включает в себя тип char, которая представляет символы символьного набора, такие как буквы и цифры.
* Булевские значения. Эта группа включает в себя тип boolean – специальный тип, предназначенный для представления значений типа истинно/ложно.

Эти типы можно использовать в том виде, как они определены, или же для создания собственных типов классов. Таким образом, они служат основой для всех других типов данных, которые могут быть созданы.

Элементарные типы представляют одиночные значения, а не сложные объекты. Хотя во всех других отношениях Java – полностью объектно-ориентированный язык, элементарные типы данных таковыми не являются. Они аналогичны простым типам, которые можно встретить в большинстве других не объектно-ориентированных языков.

Эта особенность обусловлена стремлением обеспечить максимальную эффективность. Превращение элементарных типов в объекты привело бы к слишком большому снижению производительности.

Элементарные типы определены так, чтобы они обладали явной областью допустимых значений и математически строгим поведением. Языки вроде C и C++ допускают варьирование размеров целочисленных переменных в зависимости от требований среды выполнения. Однако Java отличается в этом отношении. В связи с требованием переносимости, предъявляемым к Java-программам, все типы данных обладают строго определенной областью допустимых значений. Например, независимо от конкретной платформы, значения типа int всегда являются 32-битными. Это позволяет создавать программы, которые гарантированно будут выполняться в любой машинной архитектуре без специального переноса. Хотя в некоторых средах строгое указание размера целых чисел может приводить к незначительному снижению производительности, оно абсолютно необходимо для обеспечения переносимости программ.

## Целочисленные значения

Java определяет четыре целочисленных типа: byte, short, int и long. Все эти типы представляют значения со знаком – положительные и отрицательные. Java не поддерживает только положительные целочисленные значения без знака. Многие другие языки программирования поддерживают целочисленные значения как со знаком, так и без знака. Однако разработчики Java посчитали целочисленные значения без знака ненужными. В частности, они решили, что концепция значений без знака использовалась, в основном, для указания поведения старшего бита, который определяет знак целочисленного значения. Как будет показано в главе 4, в Java управление значением старшего бита осуществляется иначе – посредством применения специальной операции “сдвига вправо без учета знака”. Тем самым потребность в целочисленном типе без знака была исключена. Ширина целочисленного типа представляет не занимаемый объем памяти, а скорее поведение, определяемое им для переменных и выражений этого типа.

## Типы с плавающей точкой

Числа с плавающей точкой, называемые также действительными числами, используются при вычислении выражений, которые требуют получение результата с точностью до определенного десятичного знака. Например, такие вычисления, как вычисление квадратного корня или трансцендентных функций вроде синуса или косинуса, приводят к результату, который требует применения типа с плавающей точкой. В Java реализован стандартный (в соответствии с IEEE-754) набор типов и операций с плавающей точкой.

Существуют два вида типов с плавающей точкой: float и double, которые соответственно представляют числа одинарной и двойной точности.

Приблизительная область допустимых значений

double 64 от 4.9e-324 до 1.8e+308

float 32 от 1.4e-045 до 3.4e+038

Ниже приведен пример короткой программы, в которой переменные типа double используются для вычисления площади круга.

// Вычисление площади круга.

class Area {

public static void main(String args[]) {

double pi, r, a;

r = 10.8; // радиус окружности

pi = 3.1416; // pi, приблизительное значение

a = pi \* r \* r; // вычисление площади

System.out.println("Площадь круга составляет " + a);

}

}

## Символы

В Java для хранения символов используется тип данных char. Однако программистам на C/C++ следует помнить, что тип char в Java не эквивалентен типу char в C или C++. В C/C++ char – это целочисленный тип, имеющий ширину 8 битов. В Java это не так. Вместо этого в нем для представления символов используется Unicode. Unicode определяет международный набор символов, который может представлять все символы, присутствующие во всех известных языках. Он представляет собой унифицированный набор десятков наборов символов, таких как латиница, греческий алфавит, арабский алфавит, кириллица, иврит, японские и тайские иероглифы и множество других. Поэтому для хранения этих символов требуется 16 битов. Таким образом, в Java тип char является 16-битным. Диапазон допустимых значений этого типа – от 0 до 65536. Не существует отрицательных значений типа char. Стандартный набор символов, известный как ASCII, содержит значения от 0 до 127, а расширенный 8-битный набор символов, ISO-Latin-1 – значения от 0 до 255. Поскольку язык Java предназначен для обеспечения возможности создания программ, применимых во всем мире, использование кодировки Unicode для представления символов вполне обосновано. Конечно, применение Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, для представления символов которых вполне достаточно 8 битов. Но это та цена, которую приходится платить за переносимость программ во всемирном масштабе.

// Демонстрация использования типа данных char.

class CharDemo {

public static void main(String args[]) {

char ch1, ch2;

ch1 = 88; // код переменной X

ch2 = 'Y';

System.out.print("ch1 и ch2: ");

System.out.println(ch1 + " " + ch2);

}

}

Эта программа отображает следующий вывод:

*ch1 и ch2: X Y*

Обратите внимание, что переменной ch1 присвоено значение 88, являющееся значением ASCII (и Unicode), которое соответствует букве X. Как уже было сказано, набор символов ASCII занимает первые 127 значений набора символов Unicode. Поэтому, все “старые трюки”, применяемые при работе с символами в других языках, будут работать и в среде Java.

Хотя тип char был разработан для хранения символов Unicode, его можно считать также целочисленным типом, пригодным для выполнения арифметических операций.

Например, он позволяет выполнять сложение символов или уменьшать значение символьной переменной. Рассмотрим следующую программу:

// Символьные переменные ведут себя подобно целочисленным значениям.

class CharDemo2 {

public static void main(String args[]) {

char ch1;

ch1 = 'X';

System.out.println("ch1 содержит " + ch1);

ch1++; // увеличение значения ch1 на единицу

System.out.println("ch1 теперь " + ch1);

}

}

Эта программа генерирует следующий вывод:

*ch1 содержит X*

*ch1 теперь Y*

Вначале программа присваивает переменной ch1 значение X. Затем она увеличивает значение переменной ch1 на единицу. В результате хранящееся в переменной значение становится буквой Y – следующим символом в последовательности ASCII (и Unicode).

## Булевские значения

Java содержит элементарный тип, названный boolean, который предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений: true (истинно) или false (ложно). Этот тип возвращается всеми операциями сравнения, подобными a < b. Тип boolean обязателен для использования также в условных выражениях, которые управляют такими управляющими операторами, как if и for.

Следующая программа служит примером использования типа boolean:

// Демострация использования значений типа boolean.

class BoolTest {

public static void main(String args[]) {

boolean b;

b = false;

System.out.println("b равна " + b);

b = true;

System.out.println("b равна " + b);

// значение типа boolean может управлять оператором if

if(b) System.out.println("Это выполняется.");

b = false;

if(b) System.out.println("Это не выполняется.");

// результат выполнения операции сравнения – значение типа boolean

System.out.println("10 > 9 равно " + (10 > 9));

}

}

Эта программа генерирует следующий вывод:

*b равна false*

*b равна true*

*Это выполняется.*

*10 > 9 равно true*

В приведенной программе особый интерес представляют три момента. Во-первых, как видите при выводе значения типа Boolean методом println() на экране отображается строка “true” или “false”. Во-вторых, самого по себе значения переменной типа boolean достаточно для управления оператором if. Вовсе не обязательно записывать оператор if так, как показано ниже:

if(b == true) ...

В-третьих, результат выполнения операции сравнения вроде < – значение типа boolean. Именно поэтому выражение 10 > 9 приводит к отображению строки “true”.

Более того, выражение 10 > 9 должно быть заключено в дополнительный набор круглых скобок, поскольку операция + обладает более высоким приоритетом, чем операция >.

## Константы

Целочисленные значения – вероятно наиболее часто используемый тип в типичной программе. Любое целочисленное значение является числовой константой. Примерами могут служить значения 1, 2, 3 и 42. Все они – десятичные значения, описывающие числа с основанием 10. В числовых константах могут использоваться еще два вида представления – восьмеричное (с основанием 8) и шестнадцатеричное (с основанием 16). В Java восьмеричные значения обозначаются ведущим нулем. Обычные десятичные числа не могут содержать ведущий ноль. Таким образом, внешне вполне допустимое значение 09 приведет к ошибке компиляции, поскольку 9 выходит за пределы диапазона от 0 до 7 допустимых восьмеричных значений. Чаще программисты используют шестнадцатеричное представление чисел, которое явно соответствует словам, размер которых равен 8, 16, 32 и 64 бита, составленным из 8-битных блоков. Шестнадцатеричные значения обозначают ведущим нулем и символом x (0x или 0X). Диапазон допустимых шестнадцатеричных цифр – от 0 до 15, поэтому цифры от 10 до 15 заменяют буквами от A до F (или от a до f).

Целочисленные константы создают значение типа int, которое в Java является 32-битным целочисленным значением. Поскольку Java – строго типизированный язык, может возникать вопрос, каким образом можно присваивать целочисленную константу одному из других целочисленных типов Java, такому как byte или long, не вызывая при этом ошибку несоответствия типа. К счастью, с подобными ситуациями легко справиться. Когда значение константы присваивается переменной типа byte или short, ошибка не генерируется, если значение константы находится в диапазоне допустимых значений целевого типа.

Кроме того, целочисленную константу всегда можно присваивать переменной типа long. Однако чтобы указать константу типа long, придется явно указать компилятору, что значение константы имеет этот тип. Для этого к константе дописывают строчную или прописную букву L. Например, 0x7ffffffffffffffL, или 9223372036854775807L – наибольшая константа типа long. Целочисленное значение можно присваивать типу char, если оно лежит в пределах допустимого диапазона этого типа.

## Константы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробной частью. Они могут быть выражены в стандартной или научной форме записи. Число в стандартной форме записи состоит из целого числа, за которым следуют десятичная точка и дробная часть. Например, 2.0, 3.14159 и 0.6667 представляют допустимые числа с плавающей точкой в стандартной записи. Научная форма записи использует стандартную форму записи числа с плавающей точкой, к которой добавлен суффикс, указывающий степенную функцию числа 10, на которую нужно умножить данное число. Для указания экспоненциальной функции используют символ E или e, за которым следует десятичное число (положительное или отрицательное). Примерами могут служить 6.022E23, 314159E–05 и 2e+100.

По умолчанию в Java константам с плавающей точкой присвоен тип double. Чтобы указать константу типа float, к ней нужно дописать символ F или f. Константу типа double можно также указать явно, дописывая к ней символ D или d. Но, естественно, это излишне. Используемый по умолчанию тип double занимает в памяти 64 бита, в то время как менее точный тип float требует для хранения только 32 битов.

## Булевские константы

Булевские константы очень просты. Значений типа Boolean может существовать только два: true и false. Эти значения не преобразуются ни в какие числовые представления. В Java константа true не равна 1, а константа false не равна 0. В Java эти значения могут быть присвоены только тем переменным, которые объявлены как boolean, или использоваться в выражениях с булевскими операциями.

## Символьные константы

В Java символы представляют собой индексы в наборе символов Unicode. Это 16-битные значения, которые могут быть преобразованы в целые значения, и по отношению к которым можно выполнять целочисленные операции, такие как операции сложения и вычитания. Символьные константы указываются внутри пары одинарных кавычек. Все отображаемые символы ASCII можно вводить непосредственно, указывая их в кавычках, например 'a', 'z' и '@'. Для ввода символов, непосредственный ввод которых невозможен, можно использовать несколько управляющих последовательностей, которые позволяют вводить нужные символы, такие как '\'' для символа одинарной кавычки и '\n' для символа новой строки. Существует также механизм для непосредственного ввода значения символа в восьмеричном или шестнадцатеричном виде. Для ввода значения в восьмеричной форме используют символ обратной косой черты, за которым следует трехзначный номер. Например, последовательность '\141' эквивалентна букве 'a'.

Для ввода шестнадцатеричного значения применяются символы обратной косой черты и u (\u), за которыми следуют четыре шестнадцатеричные цифры.

* \ddd Восьмеричный символ (ddd)
* \uxxxx Шестнадцатеричный символ Unicode (xxxx)
* \' Одинарная кавычка
* \" Двойная кавычка
* \\ Обратная косая черта
* \r Возврат каретки
* \n Новая строка (этот символ называют также символом перевода строки)
* \f Подача страницы
* \t Табуляция
* \b Возврат на одну позицию

## Строковые константы

Указание строковых констант в Java осуществляется так же, как в других языках – путем заключения последовательности символов в пару двойных кавычек. Вот несколько примеров строковых констант:

"Hello World"

"two\nlines"

"\"This is in quotes\""

Управляющие символы и восьмеричная/шестнадцатеричная формы записи, определенные для символьных констант, работают точно так же и внутри строковых констант.

Важно отметить, что в Java строки должны начинаться и заканчиваться в одной строке. В этом языке отсутствует какой-либо управляющий символ продолжения строки, подобный тем, что имеются в ряде других языков.

## Область определения и время существования переменных

До сих пор все использованные в примерах переменные были объявлены в начале метода main(). Однако Java допускает объявление переменных внутри любого блока. Как было сказано ранее, блок начинается открывающей фигурной скобкой и завершается закрывающей фигурной скобкой. Блок задает область определения. Таким образом, при открытии каждого нового блока мы создаем новую область определения. Область определения задает то, какие объекты видимы другим частям программы. Она определяет также время существования этих объектов.

Многие другие языки программирования различают две основных категории областей определения: глобальную и локальную. Однако эти традиционные области определения не очень хорошо вписываются в строгую объектно-ориентированную модель Java.

Хотя глобальную область определения и можно задать, в настоящее время такой подход является скорее исключением, нежели правилом. В Java две основных области определения – это определяемая классом и определяемая методом. Но даже это разделение несколько искусственно. Однако поскольку область определения класса обладает несколькими уникальными свойствами и атрибутами, не применимыми к области определения метода, такое разделение имеет определенный смысл.

Определенная методом область определения начинается с его открывающей фигурной скобки. Однако если данный метод обладает параметрами, они также включаются в область определения метода.

Основное правило, которое следует запомнить: переменные, объявленные внутри области определения, не видны (т.е. недоступны) коду, который находится за пределами этой области. Таким образом, объявление переменной внутри области определения ведет к ее локализации и защите от несанкционированного доступа и/или изменений. Действительно, правила обработки области определения – основа инкапсуляции.

Области определения могут быть вложенными. Например, при каждом создании блока кода мы создаем новую, вложенную область определения. В этих случаях внешняя область определения заключает в себя внутреннюю область. Это означает, что объекты, объявленные во внешней области, будут видны коду, определенному во внутренней области. Тем не менее, обратное не верно. Объекты, которые объявлены во внутренней области определения, не будут видны за ее пределами.

Чтобы понять эффект применения вложенных областей определения рассмотрим следующую программу:

// Демонстрация области определения блока.

class Scope {

public static void main(String args[]) {

int x; // эта переменная известна всему коду внутри метода main

x = 10;

if(x == 10) { // начало новой области определения

int y = 20; // известной только этому блоку

// и x, и y известны в этой области определения.

System.out.println("x и y: " + x + " " + y);

x = y \* 2;

}

// y = 100; // Ошибка! y не известна в этой области определения

// переменная x известна и здесь.

System.out.println("x равна " + x);

}

}

Как видно из комментариев, переменная x объявлена в начале области определения метода main() и доступна всему последующему коду, находящемуся внутри этого метода.

Объявление переменной y осуществляется внутри блока if. Поскольку блок задает область определения, переменная y видна только коду внутри этого блока. Именно поэтому строка y=100;, расположенная вне этого блока, помещена в комментарий. Если удалить символ комментария, это приведет к ошибке времени компиляции, поскольку переменная y не видна за пределами своего блока. Переменную x можно использовать внутри блока if, поскольку код внутри блока (т.е. во вложенной области определения) имеет доступ к переменным, которые объявлены внешней областью.

Внутри блока переменные можно объявлять в любом месте, но они становятся допустимыми только после объявления. Таким образом, если переменная объявлена в начале метода, она доступна всему коду внутри этого метода. И наоборот, если переменная объявлена в конце блока, она, по сути, бесполезна, поскольку никакой код не получит к ней доступ. Например, следующий фрагмент кода неправилен, поскольку переменную count нельзя использовать до ее объявления:

// Этот фрагмент неправилен!

count = 100; // Стоп! Переменную count нельзя использовать до того,

// как она будет объявлена!

int count;

Следует запомнить еще один важный нюанс: переменные создаются при входе в их область определения и уничтожаются при выходе из нее. Это означает, что переменная утратит свое значение сразу по выходу из области определения. Следовательно, переменные, которые объявлены внутри метода, не будут хранить свои значения между обращениями к этому методу. Кроме того, переменная, объявленная внутри блока, будет утрачивать свое значение по выходу из блока. Таким образом, время существования переменной ограничено ее областью определения.

Если объявление переменной содержит инициализацию, инициализация переменной будет повторяться при каждом вхождении в блок, в котором она объявлена. Например, рассмотрим приведенную ниже программу.

// Демонстрация времени существования переменной.

class LifeTime {

public static void main(String args[]) {

int x;

for(x = 0; x < 3; x++) {

int y = -1; // y инициализируется при каждом вхождении в блок

System.out.println("y равна: " + y); // эта строка всегда выводит

// значение -1

y = 100;

System.out.println("y теперь равна: " + y);

}

}

}

Эта программа генерирует следующий вывод:

*y равна: -1*

*y теперь равна: 100*

*y равна: -1*

*y теперь равна: 100*

*y равна: -1*

*y теперь равна: 100*

Как видите, переменная y повторно инициализируется значением –1 при каждом вхождении во внутренний цикл for. Несмотря на то что впоследствии переменной присваивается значение 100, это значение теряется.

И последнее: хотя блоки могут быть вложенными, во внутреннем блоке нельзя объявлять переменные с тем же именем, что и во внешней области. Например, следующая программа ошибочна:

// Компиляция этой программы будет невозможна

class ScopeErr {

public static void main(String args[]) {

int bar = 1;

{ // создание новой области определения

int bar = 2; //Ошибка времени компиляции – переменная bar уже определена!

}

}

}

# Преобразование и приведение типов

Те, кто уже обладает определенным опытом программирования, знают, что достаточно часто программисты присваивают переменной одного типа значение другого. Если оба эти типа совместимы, Java выполнит преобразование автоматически. Например, всегда можно присвоить значение типа int переменной типа long. Однако не все типы совместимы и, следовательно, не все преобразования типов безоговорочно разрешены. Например, не существует никакого определенного автоматического преобразования double в byte.

К счастью, преобразования между несовместимыми типами выполнять все-таки можно. Для этого необходимо использовать приведение, которое выполняет явное преобразование несовместимых типов. Рассмотрим автоматическое преобразование и приведение типов.

## Автоматическое преобразование типов в Java

При присваивании типа данных переменной другого типа автоматическое преобразования типа выполняется в случае удовлетворения следующих двух условий:

* оба типы совместимы;
* длина целевого типа больше длины исходного типа.

При соблюдении этих двух условий выполняется преобразование с расширением. Например, тип int всегда достаточно велик, чтобы хранить все допустимые значения типа byte, поэтому никакие операторы явного приведения типа не требуются.

С точки зрения преобразования с расширением числовые типы, среди которых целочисленный и с плавающей точкой, совместимы друг с другом. Однако не существует автоматических преобразований числовых типов в char или boolean. Типы char и boolean также не совместимы и между собой.

Как уже говорилось ранее, Java выполняет автоматическое преобразование типов при сохранении целочисленной константы в переменных типа byte, short, long или char.

## Приведение несовместимых типов

Хотя автоматическое преобразование типов удобно, оно не в состоянии удовлетворить все потребности. Например, что делать, если нужно присвоить значение типа int переменной типа byte? Это преобразование не будет выполняться автоматически, поскольку тип byte меньше типа int. Иногда этот вид преобразования называют преобразованием с сужением, поскольку значение явно сужается, чтобы оно могло уместиться в целевом типе. Чтобы выполнить преобразование между двумя несовместимыми типами, необходимо использовать приведение типов. Приведение – это всего лишь явное преобразование типов. Общая форма преобразования имеет вид:

(целевой\_тип) значение

Здесь целевой\_тип определяет тип, к которому нужно преобразовать указанное значение. Например, следующий фрагмент кода приводит тип int к типу byte. Если значение целочисленного типа больше допустимого диапазона типа byte, оно будет уменьшено до результата деления по модулю (остатка от целочисленного деления) на диапазон типа byte.

int a;

byte b;

// ...

b = (byte) a;

При присваивании значения с плавающей точкой переменной целочисленного типа выполняется другой вид преобразования типа: усечение. Как вы знаете, целые числа не содержат дробной части. Таким образом, когда значение с плавающей точкой присваивается переменной целочисленного типа, дробная часть отбрасывается. Например, в случае присваивания значения 1,23 целочисленной переменной результирующим значением будет просто 1. Дробная часть – 0,23 – усекается. Конечно, если размер целочисленной части слишком велик, чтобы уместиться в целевом целочисленном типе, значение будет уменьшено до результата деления по модулю на диапазон целевого типа.

Следующая программа демонстрирует ряд преобразований типа, которые требуют приведения:

// Демонстрация приведения типов.

class Conversion {

public static void main(String args[]) {

byte b;

int i = 257;

double d = 323.142;

System.out.println("\nПреобразование int в byte.");

b = (byte) i;

System.out.println("i и b " + i + " " + b);

System.out.println("\nПреобразование double в int.");

i = (int) d;

System.out.println("d и i " + d + " " + i);

System.out.println("\nПреобразование double в byte.");

b = (byte) d;

System.out.println("d и b " + d + " " + b);

}

}

Эта программа генерирует следующий вывод:

*Преобразование int в byte.*

*i и b 257 1*

*Преобразование double в int.*

*d и i 323.142 323*

*Преобразование double в byte.*

*d и b 323.142 67*

Рассмотрим каждое из этих преобразований. Когда значение 257 приводится к типу byte, результатом будет остаток от деления 257 на 256 (диапазон допустимых значений типа byte), который в данном случае равен 1. Когда значение переменной d преобразуется в тип int, его дробная часть отбрасывается. Когда значение переменной d преобразуется в тип byte, его дробная часть отбрасывается, и значение уменьшается до результата деления по модулю на 256, который в данном случае равен 67.

## Автоматическое повышение типа в выражениях

Кроме операций присваивания определенное преобразование типов может выполняться также в выражениях. Для примера рассмотрим следующую ситуацию. Иногда в выражениях в целях обеспечения необходимой точности промежуточное значение может выходить за пределы допустимого диапазона любого из операндов. Например, рассмотрим следующее выражение:

byte a = 40;

byte b = 50;

byte c = 100;

int d = a \* b / c;

Результат вычисления промежуточного члена a\*b вполне может выйти за пределы диапазона допустимых значений его операндов типа byte. Для решения подобных проблем при вычислении выражений Java автоматически повышает тип каждого операнда byte или short до int. То есть вычисление промежуточного выражения a\*b выполняется с применением целочисленных значений, а не байтов. Поэтому результат промежуточного выражения 50 \* 40, равный 2000, оказывается допустимым, несмотря на то, что и для a и для b задан тип byte. Хотя автоматическое повышение типа очень удобно, оно может приводить к досадным ошибкам во время компиляции. Например, следующий внешне абсолютно корректный код приводит к возникновению проблемы:

byte b = 50;

b = b \* 2; // Ошибка! Значение типа int не может быть присвоено

// переменной типа byte!

Код предпринимает попытку повторного сохранения произведения 50 \* 2 – совершенно допустимого значения типа byte – в переменной типа byte. Однако, поскольку во время вычисления выражения тип операндов был автоматически повышен до int, тип результата также был повышен до int. Таким образом, теперь результат выражения имеет тип int, который не может быть присвоен переменной типа byte без приведения типа.

Сказанное справедливо даже тогда, когда, как в данном конкретном случае, значение, которое должно быть присвоено, умещается в переменной целевого типа. В тех случаях, когда последствия переполнения понятны, следует использовать явное приведение типов вроде:

byte b = 50;

b = (byte)(b \* 2);

, которое приводит к правильному значению, равному 100.

## Правила повышения типа

В Java определено несколько правил повышения типа, применяемых к выражениям.

Эти правила следующие: во-первых, тип всех значений byte, short и char повышается до int, как было описано в предыдущем разделе. Во-вторых, если один операнд имеет тип long, тип всего выражения повышается до long. Если один операнд имеет тип float, тип всего выражения повышается до float. Если любой из операндов имеет тип double, типом результата будет double.

Следующая программа демонстрирует повышение типа значения одного из операндов к типу второго в каждой операции с двумя операндами:

class Promote {

public static void main(String args[]) {

byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;

double d = .1234;

double result = (f \* b) + (i / c) – (d \* s);

System.out.println((f \* b) + " + " + (i / c) + " – " + (d \* s));

System.out.println("result = " + result);

}

}

Давайте подробнее рассмотрим повышение типа, выполняемое в следующей строке программы:

double result = (f \* b) + (i / c) – (d \* s);

В первом промежуточном выражении, f\*b, тип переменной b повышается до float, и типом результата вычисления этого промежуточного выражения также является float.

В следующем промежуточном выражении i/c тип c повышается до int и результат вычисления этого выражения – int. Затем в выражении d\*s тип значения s повышается до double, и все промежуточное выражение получает тип double. И, наконец, выполняются операции с этими тремя промежуточными значениями типов float, int и double.

Результат сложения значений типов float и int имеет тип float. Затем тип значения разности результирующего значения типа float и последнего значения типа double повышается до double, который и становится окончательным типом результата выражения.

# Массивы

Массив – это группа однотипных переменных, ссылка на которые выполняется по общему имени. Java допускает создание массивов любого типа, которые могут иметь одно или больше измерений. Доступ к конкретному элементу массива осуществляется по его индексу. Массивы предлагают удобные средства группирования связанной информации.

## Одномерные массивы

Одномерные массивы, по сути, представляют собой список однотипных переменных. Чтобы создать массив, вначале необходимо создать переменную массива требуемого типа.

Общая форма объявления одномерного массива выглядит следующим образом:

тип имя\_переменной[];

Здесь тип задает базовый тип массива. Базовый тип определяет тип данных каждого из элементов, составляющих массив. Таким образом, базовый тип массива определяет тип данных, которые будет содержать массив. Например, следующий оператор объявляет массив month\_days, имеющий тип “массив элементов типа int”:

int month\_days[];

Хотя это объявление утверждает, что month\_days – массив переменных, в действительности никакого массива еще не существует. Фактически значение массива month\_days установлено равным null, которое представляет массив без значений. Чтобы связать month\_days с реальным физическим массивом целочисленных значений, необходимо с помощью операции new распределить память и присвоить ее массиву month\_days.

Подробнее мы рассмотрим эту операцию в следующей главе, однако она нужна сейчас для выделения памяти под массивы. Общая форма операции new применительно к одномерным массивам выглядит следующим образом:

переменная\_массива = new тип[размер];

тип определяет тип данных, для которых резервируется память, размер указывает количество элементов в массиве, а переменная\_массива – переменная массива, связанная с массивом. То есть, чтобы использовать операцию new для распределения памяти, потребуется указать тип и количество элементов, для которых нужно зарезервировать память. Элементы массива, для которых память была выделена операцией new, будут автоматически инициализированы нулевыми значениями. Следующий оператор резервирует память для 12-элементного массива целых значений и связывает их с массивом

month\_days.

month\_days = new int[12];

После выполнения этого оператора month\_days будет ссылаться на массив, состоящий из 12 целочисленных значений. При этом все элементы массива будут инициализированы нулевыми значениями.

Как только массив создан, и память для него распределена, к конкретному элементу массива можно обращаться, указывая его индекс в квадратных скобках. Индексы массива начинаются с нуля. Например, следующий оператор присваивает значение 28 второму

элементу массива month\_days:

month\_days[1] = 28;

Следующая строка кода отображает значение, хранящееся в элементе с индексом, равным 3:

System.out.println(month\_days[3]);

Чтобы продемонстрировать весь процесс в целом, приведем программу, которая создает массив числа дней в каждом месяце.

// Демонстрация использования одномерного массива.

class Array {

public static void main(String args[]) {

int month\_days[];

BookNew\_JAVA-7.indb 84 02.06.2007 1:06:38

Глава 3. Типы данных, переменные и массивы 85

month\_days = new int[12];

month\_days[0] = 31;

month\_days[1] = 28;

month\_days[2] = 31;

month\_days[3] = 30;

month\_days[4] = 31;

month\_days[5] = 30;

month\_days[6] = 31;

month\_days[7] = 31;

month\_days[8] = 30;

month\_days[9] = 31;

month\_days[10] = 30;

month\_days[11] = 31;

System.out.println("В апреле " + month\_days[3] + " дней.");

}

}

Если выполнить эту программу, она выведет количество дней в апреле. Как уже было сказано, в Java индексация элементов массивов начинается с нуля, поэтому количество дней в апреле – month\_days[3], или 30.

Объявление переменной массива можно объединять с распределением самого массива, как показано в следующем примере:

int month\_days[] = new int[12];

Именно так обычно и поступают в профессионально написанных Java-программах.

Массивы можно инициализировать при их объявлении. Этот процесс во многом аналогичен инициализации простых типов. Инициализатор массива – это разделяемый запятыми список выражений, заключенный в фигурные скобки. Запятые разделяют значения элементов массива. Массив будет автоматически создан достаточно большим, чтобы в нем могли уместиться все элементы, указанные в инициализаторе массива. В этом случае использование операции new не требуется. Например, чтобы сохранить количество дней каждого месяца, можно использовать следующий код, который создает и инициализирует массив целых значений:

// Усовершенствованная версия предыдущей программы.

class AutoArray {

public static void main(String args[]) {

int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

System.out.println("В апреле " + month\_days[3] + " дней.");

}

}

При выполнении этой программы она генерирует такой же вывод, как и предыдущая версия.

Система Java выполняет тщательную проверку, чтобы убедиться в том, не была ли случайно предпринята попытка сохранения или ссылки на значения, которые выходят за пределы допустимого диапазона массива. Система времени выполнения Java будет проверять соответствие всех индексов массива допустимому диапазону. Например, система времени выполнения будет проверять соответствие значения каждого индекса допустимому диапазону от 0 до 11 включительно. Попытка обращения к элементам за пределами диапазона массива (указание отрицательных индексов или индексов, которые превышают длину массива) приведет к ошибке времени выполнения.

Приведем еще один пример программы, в которой используется одномерный массив. Эта программа вычисляет среднее значение набора чисел.

// Вычисление среднего значения массива значений.

class Average {

public static void main(String args[]) {

double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};

double result = 0;

int i;

for(i=0; i<5; i++)

result = result + nums[i];

System.out.println("Среднее значение равно " + result / 5);

}

}

## Многомерные массивы

В Java многомерные массивы представляют собой массивы массивов. Они, как можно догадаться, выглядят и действуют подобно обычным многомерным массивам. Однако, как вы увидите, им присущи и несколько незначительных отличий. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок. Например, следующий код объявляет переменную двумерного массива twoD.

int twoD[][] = new int[4][5];

Этот оператор распределяет память для массива размерностью 4×5 и присваивает его переменной twoD. Внутренне эта матрица реализована как массив массивов значений типа int.

Следующая программа нумерует элементы в массиве слева направо, сверху вниз, а затем отображает эти значения.

// Демонстрация двумерного массива.

class TwoDArray {

public static void main(String args[]) {

int twoD[][]= new int[4][5];

int i, j, k = 0;

for(i=0; i<4; i++)

for(j=0; j<5; j++) {

twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) {

for(j=0; j<5; j++)

System.out.print(twoD[i][j] + " ");

System.out.println();

}

}

}

Эта программа генерирует следующий вывод:

*0 1 2 3 4*

*5 6 7 8 9*

*10 11 12 13 14*

*15 16 17 18 19*

При распределении памяти под многомерный массив необходимо указать память только для первого (левого) измерения. Для каждого из остальных измерений память можно распределять отдельно. Например, следующий код резервирует память для первого измерения массива twoD при его объявлении. Распределение памяти для второго измерения массива осуществляется вручную.

int twoD[][] = new int[4][];

twoD[0] = new int[5];

twoD[1] = new int[5];

twoD[2] = new int[5];

twoD[3] = new int[5];

Хотя в данной ситуации отдельное распределение памяти для второго измерения массива и не дает никаких преимуществ, в других ситуациях это может быть полезно. Например, при распределении памяти для измерений массива вручную не нужно распределять одинаковое количество элементов для каждого измерения. Как было отмечено ранее, поскольку в действительность многомерные массивы представляют собой массивы массивов, программист полностью управляет длиной каждого массива. Например, следующая программа создает двумерный массив с различными размерами второго измерения.

// Резервирование памяти вручную для массива с различными

// размерами второго измерения.

class TwoDAgain {

public static void main(String args[]) {

int twoD[][] = new int[4][];

twoD[0] = new int[1];

twoD[1] = new int[2];

twoD[2] = new int[3];

twoD[3] = new int[4];

int i, j, k = 0;

for(i=0; i<4; i++)

for(j=0; j<i+1; j++) {

twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) {

for(j=0; j<i+1; j++)

System.out.print(twoD[i][j] + " ");

System.out.println();

}

}

}

Эта программа генерирует следующий вывод:

*0 1*

*2*

*3 4 5*

*6 7 8 9*

Использование неоднородных (или нерегулярных) массивов может быть не применимо во многих приложениях, поскольку их поведение отличается от обычного поведения многомерных массивов. Однако в некоторых ситуациях нерегулярные массивы могут оказаться весьма эффективными. Например, нерегулярный массив может быть идеальным решением, если требуется очень большой двумерный разреженный массив (т.е. массив, в котором будут использоваться не все элементы).

Многомерные массивы можно инициализировать. Для этого достаточно заключить инициализатор каждого измерения в отдельный набор фигурных скобок. Следующая программа создает матрицу, в которой каждый элемент содержит произведение индексов строки и столбца. Обратите также внимание, что внутри инициализаторов массивов можно применять как значения констант, так и выражения.

// Инициализация двумерного массива.

class Matrix {

public static void main(String args[]) {

double m[][] = {

{ 0\*0, 1\*0, 2\*0, 3\*0 },

{ 0\*1, 1\*1, 2\*1, 3\*1 },

{ 0\*2, 1\*2, 2\*2, 3\*2 },

{ 0\*3, 1\*3, 2\*3, 3\*3 }

};

int i, j;

for(i=0; i<4; i++) {

for(j=0; j<4; j++)

System.out.print(m[i][j] + " ");

System.out.println();

}

}

}

При выполнении эта программа генерирует следующий вывод:

*0.0 0.0 0.0 0.0*

*0.0 1.0 2.0 3.0*

*0.0 2.0 4.0 6.0*

*0.0 3.0 6.0 9.0*

Как видите, каждая строка массива инициализируется в соответствии со значениями, указанными в списках инициализации.

Рассмотрим еще один пример использования многомерного массива. Следующая программа создает трехмерный массив размерности 3×4×5. Затем она загружает каждый элемент произведением его индексов. И, наконец, она отображает эти произведения.

// Демонстрация трехмерного массива.

class ThreeDMatrix {

public static void main(String args[]) {

int threeD[][][] = new int[3][4][5];

int i, j, k;

for(i=0; i<3; i++)

for(j=0; j<4; j++)

for(k=0; k<5; k++)

threeD[i][j][k] = i \* j \* k;

for(i=0; i<3; i++) {

for(j=0; j<4; j++) {

for(k=0; k<5; k++)

System.out.print(threeD[i][j][k] + " ");

System.out.println();

}

System.out.println();

}

}

}

Эта программа генерирует следующий вывод:

*0 0 0 0 0*

*0 0 0 0 0*

*0 0 0 0 0*

*0 0 0 0 0*

*0 0 0 0 0*

*0 1 2 3 4*

*0 2 4 6 8*

*0 3 6 9 12*

*0 0 0 0 0*

*0 2 4 6 8*

*0 4 8 12 16*

*0 6 12 18 24*

## Альтернативный синтаксис объявления массивов

Для объявления массивов можно использовать и вторую форму синтаксиса:

тип[] имя\_переменной;

В этой форме квадратные скобки следуют за указателем типа, а не за именем переменной массива. Например, следующие два объявления эквивалентны:

int al[] = new int[3];

int[] a2 = new int[3];

Приведенные ниже два объявления также эквивалентны:

char twod1[][] = new char[3][4];

char[][] twod2 = new char[3][4];

Описанная вторая форма объявления удобна для одновременного объявления нескольких массивов. Например, объявление

int[] nums, nums2, nums3; // создание трех массивов

, создает три переменных массивов типа int. Оно эквивалентно объявлению

int nums[], nums2[], nums3[]; // создание трех массивов

Альтернативная форма объявления удобна также при указании массива в качестве возвращаемого типа метода. В этой книге используются обе формы объявлений.

## Строки

Как вы, вероятно, заметили, в ходе рассмотрения типов данных и массивов мы не упоминали строки или строковый тип данных. Это связано не с тем, что язык Java не поддерживает этот тип – он его поддерживает. Просто строковый тип данных Java, имеющий имя String, не относится к простым типам. Он не является также и просто массивом символов. Тип String, скорее, определяет объект, и для понимания его полного описания требуется понимание ряда характеристик объектов. Поэтому мы рассмотрим этот тип в последующих главах книги, после рассмотрения объектов. Однако чтобы читатели могли использовать простые строки в примерах программ, мы сейчас кратко опишем этот тип.

Тип String используют для объявления строковых переменных. Можно также объявлять массивы строк. Переменной типа String можно присваивать заключенную в кавычки строковую константу. Переменная типа String может быть присвоена другой переменной типа String. Объект типа String можно применять в качестве аргумента оператора println(). Например, рассмотрим следующий фрагмент кода:

String str = "тестовая строка";

System.out.println(str);

В этом примере str – объект типа String. Ему присвоена строка "тестовая строка", которая отображается оператором println().

Как будет показано в дальнейшем, объекты тип String обладают многими характерными особенностями и атрибутами, которые делают их достаточно мощными и простыми в использовании. Однако в нескольких последующих главах мы будем применять их только в простейшей форме.

# Указатели

Опытные программисты на C/C++ знают, что эти языки поддерживают указатели.

Однако мы не разу их не упоминали. Причина этого проста: Java не поддерживает и не разрешает использование указателей. (Точнее говоря, Java не поддерживает указатели, которые доступны и/или могут быть изменены программистом.) Java не разрешает использование указателей, поскольку это позволило бы Java-программам преодолевать защитный барьер между средой выполнения Java и хост-компьютером. Вспомните, что указателю может быть присвоен любой адрес в памяти – даже те адреса, которые могут находиться вне системы времени выполнения Java.

Поскольку в программах C/C++ указатели используются достаточно интенсивно, их утрата может казаться существенным недостатком Java. В действительности это не так. Среда Java спроектирована так, чтобы до тех пор, пока все действия выполняются в пределах среды выполнения, применение указателей не требовалось, и их использование не дает никаких преимуществ.

# Операции

Язык Java предоставляет среду выполнения множества операций. Большинство их них может быть отнесено к одной из следующих четырех групп: арифметические операции, побитовые операции, операции сравнения и логические операции. В Java также определен ряд дополнительных операций, которые применяются в особых ситуациях. В этой главе мы кратко опишем некоторые операции Java.

## Арифметические операции

Арифметические операции используются в математических выражениях так же, как они применяются в алгебре:

* + Сложение
* – Вычитание (также унарный минус)
* Умножение
* / Деление
* % Деление по модулю
* ++ Инкремент
* += Сложение с присваиванием
* -= Вычитание с присваиванием
* \*= Умножение с присваиванием
* /= Деление с присваиванием
* %= Деление по модулю с присваиванием
* –– Декремент

Операнды арифметических операций должны иметь числовой тип. Арифметические операции нельзя применять к типам boolean, но их можно применять к типам char, поскольку в Java этот тип, по сути, является поднабором типа int.

// Демонстрация основных арифметических операций.

class BasicMath {

public static void main(String args[]) {

// арифметические операции с целочисленными значениями

System.out.println("Целочисленная арифметика");

int a = 1 + 1;

int b = a \* 3;

int c = b / 4;

int d = c – a;

int e = -d;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

System.out.println("e = " + e);

// арифметические операции со значениями типа double

System.out.println("\nАрифметика с плавающей точкой");

double da = 1 + 1;

double db = da \* 3;

double dc = db / 4;

double dd = dc – a;

double de = -dd;

System.out.println("da = " + da);

System.out.println("db = " + db);

System.out.println("dc = " + dc);

System.out.println("dd = " + dd);

System.out.println("de = " + de);

}

}

При выполнении этой программы на экране отобразится следующий вывод:

*Целочисленная арифметика*

*a = 2*

*b = 6*

*c = 1*

*d = -1*

*e = 1*

*Арифметика с плавающей точкой*

*da = 2.0*

*db = 6.0*

*dc = 1.5*

*dd = -0.5*

*de = 0.5*

Следующий пример программы демонстрирует применение операции %:

// Демонстрация использования операции %.

class Modulus {

public static void main(String args[]) {

int x = 42;

double y = 42.25;

System.out.println("x mod 10 = " + x % 10);

System.out.println("y mod 10 = " + y % 10);

}

}

При выполнении эта программа генерирует следующий вывод:

*x mod 10 = 2*

*y mod 10 = 2.25*

Ниже приведен пример программы, который демонстрирует практическое применение нескольких составных операций с присваиванием.

// Демонстрация применения нескольких операций с присваиванием.

class OpEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a += 5;

b \*= 4;

c += a \* b;

c %= 6;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

}

}

Эта программа создает следующий вывод:

*a = 6*

*b = 8*

*c = 3*

Операции ++ и -- – это операции инкремента и декремента. Рассмотрим, что именно делают операции инкремента и декремента.

Операция инкремента увеличивает значение операнда на единицу. Операция декремента уменьшает значение операнда на единицу. Например, следующий оператор:

x = x + 1;

с применением операции инкремента можно записать в таком виде:

x++;

Аналогично, оператор

x = x - 1;

эквивалентен оператору

x--;

Эти операции отличаются тем, что они могут быть записаны как в постфиксной форме, когда символ операции следует за операндом, как в приведенных примерах, так и в префиксной форме, когда он предшествует операнду. В приведенных примерах применение любой из этих форм не имеет никакого значения. Однако, когда операции инкремента/декремента являются частью более сложного выражения, проявляется внешне незначительное, но важное различие между этими двумя формами. В префиксной форме значение операнда увеличивается или уменьшается до извлечения значения для использования в выражении. В постфиксной форме предыдущее значение извлекается для использования в выражении, и лишь после этого значение операнда изменяется. Например:

x = 42;

y = ++x;

В этом случае значение y устанавливается равным 43, как и можно было ожидать, поскольку увеличение значения выполняется перед присваиванием значения x переменной y. Таким образом, строка y=++x эквивалентна следующим двум операторам:

x = x + 1;

y = x;

Однако если операторы записать как

x = 42;

y = x++;

, значение переменной x извлекается до выполнения операции инкремента, и поэтому значение переменной y равно 42. Конечно, в обоих случаях значение переменной x установлено равным 43. Следовательно, строка y=x++; эквивалентна следующим двум операторам:

y = x;

x = x + 1;

Следующая программа демонстрирует применение операции инкремента.

// Демонстрация применения операции ++.

class IncDec {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c;

int d;

c = ++b;

d = a++;

c++;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

}

}

Вывод этой программы выглядит следующим образом:

*a = 2*

*b = 3*

*c = 4*

*d = 1*

## Побитовые операции

Java определяет несколько побитовых операций, которые могут применяться к целочисленным типам: long, int, short, char и byte. Эти операции воздействуют на отдельные биты операндов.

* ~ Побитовая унарная операция NOT (НЕ)
* & Побитовое AND (И)
* | Побитовое OR (ИЛИ)
* ^ Побитовое исключающее OR
* >> Сдвиг вправо
* >>> Сдвиг вправо с заполнением нулями
* << Сдвиг влево
* &= Побитовое AND с присваиванием
* |= Побитовое OR с присваиванием
* ^= Побитовое исключающее OR с присваиванием
* >>= Сдвиг вправо с присваиванием
* >>>= Сдвиг вправо с заполнением нулями с присваиванием
* <<= Сдвиг влево с присваиванием

Поскольку побитовые операции манипулируют битами в целочисленном значении, важно понимать, какое влияние подобные манипуляции могут оказывать на значение.

В частности, важно знать, как среда Java хранит целочисленные значения, и как она представляет отрицательные числа. Поэтому, прежде чем продолжить рассмотрение операций, кратко рассмотрим эти два вопроса.

Все целочисленные типы представляются двоичными числами различной длины. Например, значение типа byte, равное 42, в двоичном представлении имеет вид 00101010, в котором каждая позиция представляет степень двух, начиная с 20 в крайнем справа бите. Бит в следующей позиции будет представлять 21, или 2, следующий – 22, или 4, затем 8, 16, 32 и т.д. Таким образом, двоичное представление числа 42 содержит единичные биты в позициях 1, 3 и 5 (начиная с 0, крайней справа позиции). Следовательно,

42=21+23+25=2+8+32.

Все целочисленные типы (за исключением char) – целочисленные типы со знаком. Это означает, что они могут представлять как положительные, так и отрицательные значения. В Java применяется кодирование, называемое двоичным дополнением, при котором отрицательные числа представляются посредством инвертирования всех битов значения (изменения 1 на 0 и наоборот) и последующего добавления 1 к результату. Например, –42 представляется путем инвертирования все битов в двоичном представлении числа 42, что дает значение 11010101, и добавления 1, что приводит к значению 11010110, или –42.

Чтобы декодировать отрицательное число, необходимо вначале инвертировать все биты, а затем добавить 1 к результату. Например, инвертирование значения –42, или 11010110, приводит к значению 00101001, или 41, после добавления 1 к которому мы получаем 42.

Причина, по которой в Java (и большинстве других компьютерных языков) применяют двоичное дополнение, становится понятной при рассмотрении перехода через нуль.

Если речь идет о значении типа byte, ноль представляется значением 00000000. В случае применения единичного дополнения простое инвертирование всех битов создает значение, равное 11111111, которое представляет отрицательный ноль. Проблема в том, что отрицательный ноль – недопустимое значение в целочисленной математике. Применение двоичного дополнения для представления отрицательных значений позволяет решить эту проблему. При этом к дополнению добавляется 1, что приводит к числу 100000000.

Единичный бит оказывается сдвинутым влево слишком далеко, чтобы умещаться в значении типа byte. Тем самым достигается требуемое поведение, когда –0 эквивалентен 0, а 11111111 – код значения, равного –1. Хотя в приведенном примере мы использовали значение типа byte, тот же базовый принцип применяется и ко всем целочисленным типам Java.

Поскольку в Java для хранения отрицательных значений используется двоичное дополнение – и поскольку в Java все целочисленные значения являются значениями со знаком – применение побитовых операций может легко приводить к неожиданным результатам. Например, установка самого старшего бита равным 1 может привести к тому, что результирующее значение будет интерпретироваться как отрицательное число, независимо от того, к этому результату вы стремились или нет. Во избежание неприятных сюрпризов следует помнить, что независимо от того, как он был установлен, старший бит определяет знак целого числа.

## Побитовые логические операции

Побитовые логические операции – это &, |, ^ и ~. В ходе ознакомления с последующим материалом помните, что побитовые операции применяются к каждому отдельному биту каждого операнда.

Называемая также операцией побитового дополнения, унарная операция NOT (НЕ), ~, инвертирует все биты операнда. Например, число 42, которое имеет следующую последовательность битов:

00101010

в результате применения операции NOT преобразуется в:

11010101

Значение бита, полученное в результате побитовой операции AND, &, равно 1, если соответствующие биты в операндах также равны 1. Во всех остальных случаях значение результирующего бита равно 0. Например:

00101010 42

& 00001111 15

----------------------

00001010 10

Результирующий бит, полученный в результате выполнения операции OR, |, равен 1, если соответствующий бит в любом из операторов равен 1, как показано в следующем примере:

00101010 42

| 00001111 15

---------------------

00101111 47

Результирующий бит, полученный в результате выполнения операции XOR, ^, равен 1, если соответствующий бит только в одном из операндов равен 1. Во всех других случаях результирующий бит равен 0. Обратите внимание на инвертирование последовательности битов числа 42 во всех случаях, когда второй операнд содержит бит, равный 1. Во всех случаях, когда второй операнд содержит бит, равный 0, значение первого операнда остается неизменным. Это свойство пригодится при выполнении некоторых операций с битами.

00101010 42

^ 00001111 15

--------------------

00100101 37

В следующей программе демонстрируется применение побитовых логических операций.

// Демонстрация побитовых логических операций.

class BitLogic {

public static void main(String args[]) {

String binary[] = {

"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",

"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"

};

int a = 3; // 0 + 2 + 1 или 0011 в двоичном представлении

int b = 6; // 4 + 2 + 0 или 0110 в двоичном представлении

int c = a | b;

int d = a & b;

int e = a ^ b;

int f = (~a & b) | (a & ~b);

int g = ~a & 0x0f;

BookNew\_JAVA-7.indb 100 02.06.2007 1:06:41

Глава 4. Операции 101

System.out.println(" a = " + binary[a]);

System.out.println(" b = " + binary[b]);

System.out.println(" a|b = " + binary[c]);

System.out.println(" a&b = " + binary[d]);

System.out.println(" a^b = " + binary[e]);

System.out.println("~a&b|a&~b = " + binary[f]);

System.out.println(" ~a = " + binary[g]);

}

}

В этом примере последовательности битов переменных a и b представляют все четыре возможных комбинации двух двоичных цифр: 0-0, 0-1, 1-0 и 1-1. О действии операций | и & на каждый бит можно судить по результирующим значениям переменных c и d.

Значений, присвоенные переменных e и f, иллюстрируют действие операции ^. Массив строк binary содержит читабельные двоичные представления чисел от 0 до 15. В этом примере массив индексирован, что позволяет увидеть двоичное представление каждого результирующего значения. Массив построен так, чтобы соответствующее строковое представление двоичного значения n хранилось в элементе массива binary[n]. Чтобы его можно было выводить посредством массива binary, значение ~a уменьшается до значения, меньшего 16, путем его объединения со значением 0x0f (0000 1111 в двоичном представлении) операцией AND. Вывод этой программы имеет следующий вид:

*a = 0011*

*b = 0110*

*a|b = 0111*

*a&b = 0010*

*a^b = 0101*

*~a&b|a&~b = 0101*

*~a = 1100*

## Сдвиг влево

Операция сдвига влево, <<, сдвигает все биты значения влево на указанное число позиций. Она имеет следующую общую форму:

значение << число

Здесь число – количество позиций, на которое нужно сдвинуть влево биты в значении значение. То есть операция << смещает влево биты указанного значения на количество позиций, указанных в число. При каждом сдвиге влево самый старший бит сдвигается за пределы допустимого диапазона (и утрачивается), а ноль дописывается справа.

Это означает, что при применении операции сдвига влево к операнду типа int биты утрачиваются, как только они сдвигаются за пределы 31 позиции. Если операнд имеет тип long, биты теряются после сдвига за пределы 63 позиции.

Автоматическое повышение типа, выполняемое в среде Java, приводит к непредвиденным результатам при выполнении сдвига в значениях типа byte и short. Как вы уже знаете, тип значений byte и short повышается до int при вычислении выражений. Более того, результат вычисления такого выражения также имеет тип int. Это означает, что результатом выполнения сдвига влево значения типа byte или short будет значение типа int, и сдвинутые влево биты не будут отброшены до тех пор, пока они не будут сдвинуты за пределы 31 позиции. Более того, при повышении до типа int отрицательное значение типа byte или short получит дополнительный знаковый разряд. Следовательно, старшие биты будут заполнены единицами. Поэтому выполнение операции применительно к значению типа byte или short предполагает необходимость отбрасывания старших байтов результата типа int. Например, при выполнении сдвига влево в значении типа byte вначале будет осуществляться повышение типа значения до int, и лишь затем сдвиг. Это означает, что для получения требуемого сдвинутого значения типа byte необходимо отбросить три старших байта результата. Простейший способ достижения этого – обратное приведение результата к типу byte. Следующая программа демонстрирует эту концепцию.

// Сдвиг влево значения типа byte.

class ByteShift {

public static void main(String args[]) {

byte a = 64, b;

int i;

i = a << 2;

b = (byte) (a << 2);

System.out.println("Первоначальное значение a: " + a);

System.out.println("i and b: " + i + " " + b);

}

}

Эта программа генерирует следующий вывод:

*Первоначальное значение a: 64*

*i and b: 256 0*

Поскольку для выполнения вычислений тип переменной a повышается до int, сдвиг влево на две позиции значения 64 (0100 0000) приводит к значению i, равному 256 (1 0000 0000). Однако переменная b содержит значение, равное 0, поскольку после сдвига младший байт равен 0. Единственный единичный бит оказывается сдвинутым за пределы допустимого диапазона.

Поскольку каждый сдвиг влево на одну позицию, по сути, удваивает исходное значение, программисты часто используют это в качестве эффективной замены умножения на 2. Однако при этом следует соблюдать осторожность. При сдвиге единичного бита в старшую позицию (бит 31 или 63) значение становится отрицательным. Следующая программа демонстрирует это применение операции сдвига влево.

// Применение сдвига влево в качестве быстрого метода умножения на 2.

class MultByTwo {

public static void main(String args[]) {

int i;

int num = 0xFFFFFFE;

for(i=0; i<4; i++) {

num = num << 1;

System.out.println(num);

}

}

}

Программа генерирует следующий вывод:

*536870908*

*1073741816*

*2147483632*

*-32*

Начальное значение было специально выбрано таким, чтобы после сдвига влево на 4 позиции оно стало равным -32. Как видите, после сдвига единичного бита в позицию 31 число интерпретируется как отрицательное.

## Сдвиг вправо

Операция сдвига вправо, >>, сдвигает все биты значения вправо на указанное количество позиций. В общем виде ее можно записать следующим образом:

значение >> число

Здесь число указывает количество позиций, на которое нужно сдвинуть вправо биты в значении значение. То есть операция >> перемещает все биты в указанном значении вправо на количество позиций, указанное в число.

Следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 32, в результате чего значение переменной a становится равным 8:

int a = 32;

a = a >> 2; // теперь a содержит 8

Когда какие-либо биты в значении “сдвигаются прочь”, они теряются. Например, следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 35, что приводит к утрате двух младших битов и повторной установке значения переменной a равным 8:

int a = 35;

a = a >> 2; // a по-прежнему содержит 8

Чтобы лучше понять, как выполняется эта операция, рассмотрим ее применение к двоичным представлениям:

00100011 35

>> 2

00001000 8

При каждом сдвиге вправо выполняется деление значения на два с отбрасыванием любого остатка. Это свойство можно использовать для выполнения высокопроизводительного целочисленного деления на 2. Конечно, при этом нужно быть уверенным, что никакие биты не будут сдвинуты за пределы правой границы.

При выполнении сдвига вправо старшие (расположенные в крайних левых позициях)биты, освобожденные в результате сдвига, заполняются предыдущим содержимым старшего бита. Этот эффект называется дополнительным знаковым разрядом и служит для сохранения знака отрицательных чисел при их сдвиге вправо. Например, результат выполнения операции -8>>1 равен –4, что в двоичном представлении выглядит так:

11111000 –8

>>1

11111100 –4

Интересно отметить, что результат сдвига вправо значения –1 всегда равен –1, поскольку дополнительные знаковые разряды добавляют новые единицы к старшим битам.

Иногда при выполнении сдвига вправо появление дополнительных знаковых разрядов нежелательно. Например, следующая программа преобразует значение byte в соответствующее шестнадцатеричное строковое представление. Обратите внимание, что для обеспечения возможности использования значения в качестве индекса массива шестнадцатеричных символов сдвинутое значение маскируется посредством его объединения со значением 0x0f операцией AND, что приводит к отбрасыванию любых битов дополнительных знаковых разрядов.

// Маскирование дополнительных знаковых разрядов.

class HexByte {

static public void main(String args[]) {

char hex[] = {

'0', '1', '2', '3', '4', '5', '6', '7',

'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'

};

byte b = (byte) 0xf1;

System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);

}

}

Вывод этой программы выглядит следующим образом:

*b = 0xf1*

## Сдвиг вправо без учета знака

Как было показано, при каждом выполнении операция >> автоматически заполняет старший бит его предыдущим содержимым. В результате знак значения сохраняется.

Однако иногда это нежелательно. Например, при выполнении сдвига вправо в какомлибо значении, которое не является числовым, использование дополнительных знаковых разрядов может быть нежелательным. Эта ситуация часто встречается при работе со значениями пикселей и графическими изображениями. Как правило, в этих случаях требуется сдвиг нуля в позицию старшего бита независимо от его первоначального значения. Такое действие называют сдвигом вправо без учета знака. Для его выполнения используют операцию сдвига вправо без учета знака Java, >>>, которая всегда вставляет ноль в позицию старшего бита.

Следующий фрагмент кода демонстрирует применение операции >>>. В этом примере значение переменной a установлено равным –1, все 32 бита двоичного представления которого равны 1. Затем в этом значении выполняется сдвиг вправо на 24 бита с заполнением старших 24 битов нулями и игнорированием обычно используемых дополнительных

знаковых разрядов. В результате значение a становится равным 255.

int a = -1;

a = a >>> 24;

Чтобы происходящее было понятнее, запишем эту же операцию в двоичной форме:

11111111 11111111 11111111 11111111 –1 в двоичном представлении типа int >>>24

00000000 00000000 00000000 11111111 255 в двоичном представлении типа int

Часто операция >>> не столь полезна, как хотелось бы, поскольку она имеет смысл только для 32- и 64-разрядных значений. Помните, что в выражениях тип меньших значений автоматически повышается до int. Это означает применение дополнительных знаковых разрядов и выполнение сдвига по отношению к 32-разрядным, а не 8- или 16-разрядным значениям. То есть программист может подразумевать выполнение сдвига вправо без учета знака применительно к значению типа byte и заполнение нулями, начиная с бита 7.

Однако в действительности это не так, поскольку фактически сдвиг будет выполняться в 32-разрядном значении. Этот эффект демонстрирует следующая программа.

// Сдвиг без учета знака значения типа byte.

class ByteUShift {

static public void main(String args[]) {

char hex[] = {

'0', '1', '2', '3', '4', '5', '6', '7',

'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'

};

byte b = (byte) 0xf1;

byte c = (byte) (b >> 4);

byte d = (byte) (b >>> 4);

byte e = (byte) ((b & 0xff) >> 4);

System.out.println(" b = 0x"

+ hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);

System.out.println(" b >> 4 = 0x"

+ hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);

System.out.println(" b >>> 4 = 0x"

+ hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);

System.out.println("(b & 0xff) >> 4 = 0x"

+ hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);

}

}

Из следующего вывода этой программы видно, что операция >>> не выполняет никаких действий по отношению к значениям типа byte. Для этого примера в качестве значения переменной b было выбрано произвольное отрицательное значение типа byte. Затем переменной c присваивается значение переменной b типа byte, сдвинутое вправо на четыре позиции, которое в связи с применением дополнительных знаковых разрядов равно 0xff. Затем переменной d присваивается значение переменной b типа byte, сдвинутое вправо на четыре позиции без учета знака, которым должно было бы быть значение 0x0f, но в действительности, из-за применения дополнительных знаковых разрядов во время повышения типа b до int перед выполнением сдвига, равное 0xff. Последнее выражение устанавливает значение переменной e равным значению типа byte переменной b, замаскированному до 8 бит с помощью операции AND и затем сдвинутому вправо на четыре позиции, что дает ожидаемое значение, равное 0x0f. Обратите внимание, что операция сдвига вправо без учета знака не применялась к переменной d, поскольку состояние знакового бита после выполнения операции AND было известно.

b = 0xf1

b >> 4 = 0xff

b >>> 4 = 0xff

(b & 0xff) >> 4 = 0x0f

## Побитовые составные операции с присваиванием

Подобно алгебраическим операциям, все двоичные побитовые операции имеют составную форму, которая объединяет побитовую операцию с операцией присваивания.

Например, следующие два оператора, выполняющие сдвиг вправо на четыре позиции в значении переменной a, эквивалентны:

a = a >> 4;

a >>= 4;

Аналогично, эквивалентны и следующие два оператора, которые присваивают переменной a результат выполнения побитовой операции a OR b:

a = a | b;

a |= b;

Следующая программа создает несколько целочисленных переменных, а затем использует составные побитовые операции с присваиванием для манипулирования этими переменными:

class OpBitEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a |= 4;

b >>= 1;

c <<= 1;

a ^= c;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

}

}

Эта программа создает следующий вывод:

*a = 3*

*b = 1*

*c = 6*

## Операции сравнения

Операции сравнения определяют отношение одного операнда с другим. В частности, они определяют равенство и порядок следования.

* == Равно
* != Не равно
* > Больше
* < Меньше
* >= Больше или равно
* <= Меньше или равно

Результат выполнения этих операций – значение типа boolean. Наиболее часто операции сравнения используют в выражениях, которые управляют оператором if и различными операторами цикла.

В Java можно сравнивать значения любых типов, в том числе целые значения, значения с плавающей точкой, символы и булевские значения, используя проверку равенства == и проверку неравенства !=. Обратите внимание, что Java равенство обозначают двумя знаками “равно”, а не одним. (Одиночный знак “равно” – символ операции присваивания.) Сравнение с помощью операций упорядочения применимо только к числовым типам. То есть сравнение для определения того, какой из операндов больше или меньше другого, можно выполнять только для целочисленных операндов, операндов с плавающей точкой или символьных операндов.

Как уже было отмечено, результат выполнения операции сравнения представляет собой значение типа boolean. Например, следующий фрагмент кода вполне допустим:

int a = 4;

int b = 1;

boolean c = a < b;

В данном случае результат выполнения операции a < b (который равен false) сохраняется в переменной c.

Те читатели, которые знакомы с языками C/C++, должны обратить внимание на следующее. В программах на C/C++ следующие типы операторов встречаются очень часто:

int done;

// ...

if(!done) ... // Допустимо в C/C++

if(done) ... // но не в Java.

В Java-программе эти операторы должны быть записаны следующим образом:

if(done == 0) ... // Это стиль Java.

if(done != 0) ...

Это обусловлено тем, что в Java определение значений “истинно” и “ложно” отличается от их определений в языках C/C++. В C/C++ истинным считается любое ненулевое значение, а ложным – ноль. В Java значения true (истинно) и false (ложно) – нечисловые значения, которые никак не сопоставимы с нулевым или ненулевым значением.

Поэтому, чтобы сравнить значение с нулевым или ненулевым значением, необходимо явно использовать одну или несколько операций сравнения.

## Булевские логические операции

Описанные в этом разделе логические операции работают только с операндами типа boolean. Все логические операции с двумя операндами объединяют два значения типа boolean, образуя результирующее значение типа boolean.

* & Логическое AND (И)
* | Логическое OR (ИЛИ)
* ^ Логическое XOR (исключающее OR (ИЛИ))
* || Замыкающее OR
* BookNew\_JAVA-7.indb 107 02.06.2007 1:06:42
* 108 Часть I. Язык Java
* Операция Описание
* && Замыкающее AND
* ! Логическое унарное NOT (НЕ)
* &= AND с присваиванием
* |= OR с присваиванием
* ^= XOR с присваиванием
* == Равно
* != Не равно
* ?: Тернарная операция if-then-else

Логические булевские операции &, | и ^ действуют применительно к значениям типа boolean точно так же, как они действуют по отношению к битам целочисленных значений. Логическая операция ! инвертирует булевское состояние: !true == false и !false == true.

Ниже приведена программа, которая выполняет практически те же действия, что и пример программы BitLogic, представленный ранее, но она работает с логическими значениями типа boolean, а не с двоичными разрядами.

// Демонстрация применения булевских логических операций.

class BoolLogic {

public static void main(String args[]) {

boolean a = true;

boolean b = false;

boolean c = a | b;

boolean d = a & b;

boolean e = a ^ b;

boolean f = (!a & b) | (a & !b);

boolean g = !a;

System.out.println(" a = " + a);

System.out.println(" b = " + b);

System.out.println(" a|b = " + c);

System.out.println(" a&b = " + d);

System.out.println(" a^b = " + e);

System.out.println("!a&b|a&!b = " + f);

System.out.println(" !a = " + g);

}

}

Выполняя эту программу, легко убедиться, что к значениям типа boolean применяются те же логические правила, что и к битам.

## Замыкающие логические операции

Java предоставляет две интересные булевских операции, не встречающиеся во многих других языках программирования. Это вторые версии булевских операций AND и OR, получившие название замыкающих логических операций. Как видно из ранее приведенной таблицы, результат выполнения операции OR равен true, когда значение операнда A равно true, независимо от значения операнда B. Аналогично, результат выполнения операции AND равен false, когда значение операнда A равно false, независимо от значения операнда B. При использовании форм || и && этих операций вместо | и & программа Java не будет вычислять значение правого операнда, если результат выражения можно определить по значению одного левого операнда. Это свойство очень удобно в тех случаях, когда значение правого операнда зависит от значения левого. Например, следующий фрагмент кода демонстрирует преимущество применения замыкающей логической операций для выяснения допустимости операции деления перед вычислением ее результата:

if (denom != 0 && num / denom > 10)

Благодаря применению замыкающей формы операции AND (&&) исключается риск возникновения исключения времени выполнения в случае равенства знаменателя (denom) нулю. Если бы эта строка кода была записана с применением одинарного символа & операции AND, программа вычисляла бы обе части выражения, что приводило бы к исключению времени выполнения при равенстве значения denom нулю.

Замыкающие формы операций AND и OR принято применять в тех случаях, когда требуется использование операций булевской логики, а односимвольные версии – исключительно для побитовых операций. Однако существуют исключения из этого правила. Например, рассмотрим следующий оператор:

if(c==1 & e++ < 100) d = 100;

В данном случае одиночный символ & гарантирует применение операции инкремента к значению e независимо от равенства 1 значения c.

## Операция присваивания

Мы использовали операцию присваивания, начиная с главы 2. Теперь пора рассмотреть эту операцию формально. Символом операции присваивания служит одиночный знак равенства, =. В Java операция присваивания работает аналогично тому, как она работает во многих компьютерных языках. Она имеет следующую общую форму:

переменная = выражение;

В этом операторе тип переменной должен соответствовать типу выражения. Операция присваивания обладает одной интересной особенностью, с которой вы, возможно, еще не знакомы: она позволяет создавать цепочки присваиваний. Например, рассмотрим следующий фрагмент кода:

int x, y, z;

x = y = z = 100;

В этом фрагменте кода единственный оператор устанавливает значения трех переменных x, y и z равными 100. Это обусловлено тем, что = – операция, которая использует значение правого выражения. Таким образом, значение выражения z=100 равно 100, которое затем присваивается переменной y, а затем – переменной x. Использование “цепочки присваивания” – удобный способ установки общего значения группы переменных.

## Операция ?

Синтаксис Java содержит специальную тернарную операцию, которой можно заменять определенные типы операторов if-then-else. Это операция ?. Вначале она может казаться несколько непонятной, но со временем вы убедитесь в ее исключительной эффективности. Эта операция имеет следующую общую форму:

выражение1 ? выражение2 : выражение3

Здесь выражение1 – любое выражение, приводящее к значению типа boolean. Если значение выражения1 – true, программа вычисляет значение выражения2. В противном случае программа вычисляет значение выражения3. Результат выполнения операции ? равен значению вычисленного выражения. И выражение2, и выражение3 должны возвращать значение одного и того же типа, которым не может быть тип void.

Следующий пример программы демонстрирует применение операции ?. Эта программа служит для получения абсолютного значения переменной.

// Демонстрация использования операции ?.

class Ternary {

public static void main(String args[]) {

int i, k;

i = 10;

k = i < 0 ? -i : i; // получение абсолютного значения переменной i

System.out.print("Абсолютное значение ");

System.out.println(i + " равно " + k);

i = -10;

k = i < 0 ? -i : i; // получение абсолютного значения переменной i

System.out.print("Абсолютное значение ");

System.out.println(i + " равно " + k);

}

}

Эта программа генерирует следующий вывод:

*Абсолютное значение 10 равно 10*

*Абсолютное значение -10 равно 10*

# Управляющие операторы

В языках программирования управляющие операторы используются для реализации переходов и ветвлений в потоке выполнения команд программы в соответствии с ее состоянием. Управляющие операторы Java-программы можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода. Операторы выбора позволяют программе выбирать различные ветви выполнения команд в соответствии с результатом выражения или состоянием переменной. Операторы цикла позволяют программе повторять выполнение одного или нескольких операторов (т.е. они образуют циклы). Операторы перехода обеспечивают возможность нелинейного выполнения программы. В этой главе мы рассмотрим все управляющие операторы Java.

## Операторы выбора

В Java поддерживаются два оператора выбора: if и switch. Эти операторы позволяют управлять порядком выполнения команд программы в соответствии с условиями, которые известны только во время выполнения.

Оператор if – оператор условного выбора ветви Java-программы. Его можно использовать для направления выполнения программы по двум различным ветвям. Общая форма этого оператора выглядит следующим образом:

if (условие) оператор1;

else оператор2;

Здесь каждый оператор – это одиночный оператор или составной оператор, заключенный в фигурные скобки (т.е. блок). Условие – это любое выражение, возвращающее значение типа boolean. Выражение else не обязательно. Оператор if работает следующим образом: если условие истинно, программа выполняет оператор1. В противном случае она выполняет оператор2 (если он существует). Ни в одной из ситуаций программа не будет выполнять оба оператора.

Для управления оператором if можно применять и одиночную переменную типа boolean, как показано в следующем фрагменте кода:

boolean dataAvailable;

// ...

if (dataAvailable)

ProcessData();

else

waitForMoreData();

Помните, что только один оператор может следовать непосредственно за ключевым словом if или else. Если нужно включить больше операторов, придется создать код, подобный приведенному в следующем фрагменте:

int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else

waitForMoreData();

В этом примере оба оператора, помещенные внутрь блока, будут выполняться, если значение переменной bytesAvailable больше нуля.

Некоторые программисты предпочитают использовать в операторе if фигурные скобки даже при наличии только одного оператора в каждом выражении. Это упрощает добавление операторов в дальнейшем и избавляет от беспокойства по поводу наличия фигурных скобок. Фактически пропуск определения блока в тех случаях, когда он необходим – достаточно распространенная ошибка. Например, рассмотрим следующий фрагмент кода:

int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else

waitForMoreData();

bytesAvailable = n;

Если судить по уровню отступа в коде, предполагалось, что операторbytesAvailable=n; должен выполняться внутри выражения else. Однако, как вы помните, в Java-программе пробелы значения не имеют, и компилятор никак не может “узнать” намерения программиста. Компиляция этого кода будет выполняться без вывода каких-либо предупреждающих сообщений, но при выполнении программа будет вести себя не так, как было задумано. В следующем фрагменте ошибка предыдущего примера исправлена:

int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else {

waitForMoreData();

bytesAvailable = n;

}

## Вложенные операторы if

Вложенный оператор if – это оператор if, являющийся целью другого оператора if или else. В программах вложенные операторы if встречаются очень часто. При использовании вложенных операторов if важно помнить, что оператор else всегда связан с ближайшим оператором if, расположенным в одном с ним блоке и еще не связанным с другим оператором else. Например:

if(i == 10) {

if(j < 20) a = b;

if(k > 100) c = d; // этот оператор if

else a = c; // связан с этим оператором else

}

else a = d; // этот оператор else связан с оператором if(i == 10)

Как видно из комментариев, последний оператор else связан не с оператором if(j<20), поскольку тот не находится в одном с ним блоке (несмотря на то, что он является ближайшим оператором if, который еще не связан с оператором else). Последний оператор else связан с оператором if(i==10). Внутренний оператор else связан с оператором if(k>100), поскольку тот является ближайшим внутри этого же блока.

## Многозвенная структура if-else-if

Распространенной конструкцией программирования, построенной на основе последовательности вложенных операторов if, является многозвенная структура if-else-if.

Она выглядит следующим образом:

if(условие)

оператор;

else if(условие)

оператор;

else if(условие)

оператор;

...

...

...

else

оператор;

Операторы if выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором if, становится равным true, программа выполняет оператор, связанный с данным оператором if, и пропускает остальную часть многозвенной структуры. Если ни одно из условий не выполняется (не равно true), программа выполнит заключительный оператор else. Этот последний оператор служит условием, используемым по умолчанию. То есть если проверка всех остальных условий дает отрицательный результат, программа выполняет последний оператор else. Если заключительный оператор else не указан, а результат проверки всех остальных условий равен false, программа не будет выполнять никаких действий.

Ниже приведен пример программы, в которой многозвенная структура if-else-if служит для определения времени года, к которому относится конкретный месяц.

// Демонстрация применения операторов if-else-if.

class IfElse {

public static void main(String args[]) {

int month = 4; // Апрель

String season;

if(month == 12 || month == 1 || month == 2)

season = "зиме";

else if(month == 3 || month == 4 || month == 5)

season = "весне";

else if(month == 6 || month == 7 || month == 8)

season = "лету";

else if(month == 9 || month == 10 || month == 11)

season = "осени";

else

season = "вымышленным месяцам";

System.out.println("Апрель относится к " + season + ".");

}

}

Программа создает следующий вывод:

*Апрель относится к весн*е

## Оператор switch

Оператор switch – оператор ветвления в Java. Он предлагает простой способ направления потока выполнения команд по различным ветвям кода в зависимости от значения управляющего выражения. Часто он оказывается эффективнее применения длинных последовательностей операторов if-else-if. Общая форма оператора switch имеет следующий вид:

switch (выражение) {

case значение1:

// последовательность операторов

break;

case значение2:

// последовательность операторов

break;

...

...

...

case значениеN:

// последовательность операторов

break;

default:

// последовательность операторов, выполняемая по умолчанию

}

Выражение должно иметь тип type, short, int или char. Тип каждого значения, указанного в операторах case должен быть совместим с типом выражения. Дублирование значений case не допускается.

Оператор switch работает следующим образом. Значение выражения сравнивается с каждым из значений констант в операторах case. При обнаружении совпадения программа выполняет последовательность кода, следующую за данным оператором case. Если значения ни одной из констант не совпадает со значением выражения, программа выполняет оператор default. Однако этот оператор не обязателен. При отсутствии совпадений со значениями case и отсутствии оператора default программа не выполняет никаких дальнейших действий.

Оператор break внутри последовательности switch служит для прерывания последовательности операторов. Как только программа доходит до оператора break, она продолжает выполнение с первой строки кода, следующей за всем оператором switch. Этот оператор оказывает действие “выхода” из оператора switch.

Ниже представлен простой пример использования оператора switch.

// Простой пример применения оператора switch.

class SampleSwitch {

public static void main(String args[]) {

for(int i=0; i<6; i++)

switch(i) {

case 0:

System.out.println("i равно нулю.");

break;

case 1:

System.out.println("i равно единице.");

break;

case 2:

System.out.println("i равно двум.");

break;

case 3:

System.out.println("i равно трем.");

break;

default:

System.out.println("i больше 3.");

}

}

}

Эта программа генерирует следующий вывод:

*i равно нулю.*

*i равно единице.*

*i равно двум.*

*i равно трем.*

*i больше 3.*

*i больше 3.*

Как видите, на каждой итерации цикла программа выполняет операторы, связанные с константой case, которая соответствует значению переменной i. Все остальные операторы пропускаются. После того, как значение i становится больше 3, оно перестает соответствовать какой-либо константе case, поэтому программа выполняет оператор default.

Оператор break не обязателен. Если его опустить, программа продолжит выполнение со следующего оператора case. В некоторых случаях желательно использовать несколько операторов case без разделяющих их операторов break. Например, рассмотрим следующую программу:

// В последовательности switch операторы break необязательны.

class MissingBreak {

public static void main(String args[]) {

for(int i=0; i<12; i++)

switch(i) {

case 0:

case 1:

case 2:

case 3:

case 4:

System.out.println("i меньше 5");

break;

case 5:

case 6:

case 7:

case 8:

case 9:

System.out.println("i меньше 10");

break;

default:

System.out.println("i равно или больше 10");

}

}

}

Эта программа генерирует следующий вывод:

*i меньше 5*

*i меньше 5*

*i меньше 5*

*i меньше 5*

*i меньше 5*

*i меньше 10*

*i меньше 10*

*i меньше 10*

*i меньше 10*

*i меньше 10*

*i равно или больше 10*

*i равно или больше 10*

Как видите, программа выполняет каждый из операторов case, пока не достигнет оператора break (или конца оператора switch).

Хотя приведенный пример был создан специально в качестве иллюстрации, пропуск операторов break находит множество применений в реальных программах. В качестве более реалистичного примера мы рассмотрим измененную версию приведенной ранее программы со временами года. В этой версии мы использовали оператор switch, что позволило создать более эффективную реализацию.

// Усовершенствованная версия программы с временами года.

class Switch {

public static void main(String args[]) {

int month = 4;

String season;

switch (month) {

case 12:

case 1:

case 2:

season = "зиме";

break;

case 3:

case 4:

case 5:

season = "весне";

break;

case 6:

case 7:

case 8:

season = "лету";

break;

case 9:

case 10:

case 11:

season = "осени";

break;

default:

season = "вымышленным месяцам";

}

System.out.println("Апрель относится к" + season + ".");

}

}

## Вложенные операторы switch

Оператор switch можно использовать в последовательности операторов внешнего оператора switch. Такой оператор называют вложенным оператором switch. Поскольку оператор switch определяет собственный блок, каких-либо конфликтов между константами case внутреннего и внешнего операторов switch не происходит. Например, следующий фрагмент полностью допустим:

switch(count) {

case 1:

switch(target) {// вложенный оператор switch

case 0:

System.out.println("target равна 0");

break;

case 1: // конфликты с внешним оператором switch отсутствуют

System.out.println("target равна 1");

break;

}

break;

case 2: // ...

В данном случае оператор case1: внутреннего оператора switch не конфликтует с оператором case1: внешнего оператора switch. Программа сравнивает значение переменной count только со списком ветвей case внешнего уровня. Если значение count равно 1, программа сравнивает значение переменной target c внутренним списком ветвей case.

Как правило, оператор switch эффективнее набора вложенных операторов if.

Это представляет особый интерес, поскольку позволяет понять работу компилятора Java. Компилируя оператор switch, компилятор Java будет проверять каждую из констант case и создавать “таблицу переходов”, которую будет использовать для выбора ветви программы в зависимости от значения выражения. Поэтому в тех случаях, когда требуется осуществлять выбор в большой группе значений, оператор switch будет выполняться значительно быстрее последовательности операторов if-else. Это обусловлено тем, что компилятору известно, что все константы case имеют один и тот же тип, и их нужно просто проверять на предмет равенства значению выражения switch.

Компилятор не располагает подобными сведениями о длинном списке выражений оператора if.

# Операторы цикла

Операторами цикла Java являются for, while и do-while. Эти операторы образуют конструкции, обычно называемые циклами. Как наверняка известно читателям, циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла. Как вы вскоре убедитесь, Java предлагает средства создания циклов, способные удовлетворить любые потребности программирования.

## Цикл while

Оператор while – наиболее часто используемый оператор цикла Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно. Он имеет следующую общую форму:

while(условие){

//тело цикла

}

Условием может быть любое булевское выражение. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда условие становится ложным, управление передается строке кода, непосредственно следующей за циклом. Фигурные скобки могут быть опущены, только если в цикле повторяется только один оператор.

В качестве примера рассмотрим цикл while, который выполняет обратный отсчет, начиная с 10, вывод ровно 10 строк “тактов”:

// Демонстрация использования цикла while.

class While {

public static void main(String args[]) {

int n = 10;

while(n > 0) {

System.out.println("такт " + n);

n--;

}

}

}

После запуска эта программа выводит десять “тактов”:

*такт 10*

*такт 9*

*такт 8*

*такт 7*

*такт 6*

*такт 5*

*такт 4*

*такт 3*

*такт 2*

*такт 1*

Поскольку цикл while вычисляет свое условное выражение в начале цикла, тело цикла не будет выполнено ни разу, если в самом начале условие оказывается ложным.

Например, в следующем фрагменте кода метод println() никогда не будет вызван:

int a = 10, b = 20;

while(a > b)

System.out.println("Эта строка отображаться не будет");

Тело цикла while (или любого другого цикла Java) может быть пустым. Это обусловлено тем, что синтаксис Java допускает применение нулевого оператора (содержащего только символ точки с запятой). Например, рассмотрим следующую программу:

// Целевая часть цикла может быть пустой.

class NoBody {

public static void main(String args[]) {

int i, j;

i = 100;

j = 200;

// вычисление среднего значения i и j

while(++i < --j) ; // в этом цикле тело цикла отсутствует

System.out.println("Среднее значение равно " + i);

}

}

Эта программа вычисляет среднее значение i и j. Она генерирует следующий вывод:

*Среднее значение равно 150*

Этот цикл while работает следующим образом. Значение i увеличивается, а значение j уменьшается на единицу. Затем программа сравнивает эти два значения. Если новое значение i по-прежнему меньше нового значения j, цикл повторяется. Если значение i равно или больше значения j, выполнение цикла прекращается. По выходу из цикла переменная i будет содержать среднее значение исходных значений i и j. Конечно, эта процедура работает только в том случае, если в самом начале значение i меньше значения j.

Как видите, никакой потребности в наличии тела цикла не существует. Все действия выполняются внутри самого условного выражения. В профессионально написанном Javaкоде короткие циклы часто не содержат тела, если само по себе управляющее выражение может выполнять все необходимые действия.

## Цикл do-while

Как вы видели, если в начальный момент условное выражение, управляющее циклом while, ложно, тело цикла вообще не будет выполняться. Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условное выражение ложно. Иначе говоря, существуют ситуации, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале. К счастью, Java поддерживает именно такой цикл: do-while. Этот цикл всегда выполняет тело цикла хотя бы один раз, поскольку его условное выражение проверяется в конце цикла. Общая форма цикла do-while следующая:

do {

// тело цикла

} while (условие);

При каждом повторении цикла do-while программа вначале выполняет тело цикла, а затем вычисляет условное выражение. Если это выражение истинно, цикл повторяется. В противном случае выполнение цикла прерывается. Как и во всех циклах Java, выражение должно быть булевским.

Ниже приведена измененная программа вывода тактов, которая демонстрирует использование цикла do-while. Она генерирует такой же вывод, что и предыдущая версия.

// Демонстрация использования цикла do-while.

class DoWhile {

public static void main(String args[]) {

int n = 10;

do {

System.out.println("такт " + n);

n--;

} while(n > 0);

}

}

Хотя с технической точки зрения в приведенной программе цикл записан правильно, его можно переписать в более эффективном виде:

do {

System.out.println("такт " + n);

} while(--n > 0);

В этом примере операции декремента переменной n и сравнения результирующего значения с нулем объединены в одном выражении (--n>0). Программа работает следующим образом. Вначале она выполняет оператор --n, уменьшая значение n на единицу и возвращая новое значение переменной n. Затем программа сравнивает это значение с нулем. Если оно больше нуля, выполнение цикла продолжается. В противном случае цикл прерывается.

Цикл do-while особенно удобен при выборе пункта меню, поскольку обычно в этом случае требуется, чтобы тело цикла меню выполнялось, по меньшей мере, один раз.

Рассмотрим следующую программу, которая реализует очень простую систему справки по операторам выбора и цикла Java:

// Использование цикла do-while для выбора пункта меню

class Menu {

public static void main(String args[])

throws java.io.IOException {

char choice;

do {

System.out.println("Справка по:");

System.out.println(" 1. if");

System.out.println(" 2. switch");

System.out.println(" 3. while");

System.out.println(" 4. do-while");

System.out.println(" 5. for\n");

System.out.println("Выберите интересующий пункт:");

choice = (char) System.in.read();

} while( choice < '1' || choice > '5');

System.out.println("\n");

switch(choice) {

case '1':

System.out.println("if:\n");

System.out.println("if(условие) оператор;");

System.out.println("else оператор;");

break;

case '2':

System.out.println("switch:\n");

System.out.println("switch(выражение) {");

System.out.println(" case константа:");

System.out.println(" последовательность операторов");

System.out.println(" break;");

System.out.println(" // ...");

System.out.println("}");

break;

case '3':

System.out.println("while:\n");

System.out.println("while(условие) оператор;");

break;

case '4':

System.out.println("do-while:\n");

System.out.println("do {");

System.out.println(" оператор;");

System.out.println("} while (условие);");

break;

case '5':

System.out.println("for:\n");

System.out.print("for(инициализация; условие; повторение)");

System.out.println(" оператор;");

break;

}

}

}

Пример вывода выполнения этой программы выглядит следующим образом:

Справка *по:*

*1. if*

*2. switch*

*3. while*

*4. do-while*

*5. for*

*Выберите интересующий пункт:*

*4*

*do-while:*

*do {*

*оператор;*

*} while (условие);*

В этой программе в цикле do-while осуществляется проверка допустимости введенного пользователем значения. Если это значение недопустимо, программа предлагает пользователю повторить ввод. Поскольку меню должно отобразиться, по меньшей мере, один раз, цикл do-while является прекрасным средством решения этой задачи.

Отметим еще несколько особенностей приведенного примера. Обратите внимание, что считывание символов с клавиатуры выполняется посредством вызова метода System.in.read(). Это – одна из функций консольного ввода Java. В данном случае метод System.in.read() используется для выяснения выбора, осуществленного пользователем. Этот метод считывает символы из стандартного ввода (возвращаемые в виде целочисленных значений – именно потому тип возвращаемого значения был приведен к char).

Консольный ввод Java может вызывать некоторые затруднения при работе. Более того, большинство реальных Java-программ будут графическими и ориентированным на работу в оконном режиме. Однако в данном случае он удобен. Еще один важный момент.

Поскольку мы используем метод System.in.read(), программа должна содержать выражение throws java.io.IOException. Эта строка необходима для обработки ошибок ввода. Она является составной частью системы обработки исключений Java, которая будет рассмотрена в главе 10.

## Цикл for

Начиная с версии JDK 5, в Java существуют две формы цикла for. Первая – традиционная форма, используемая начиная с исходной версии Java. Вторая – новая форма “for-each”. Мы рассмотрим оба типа цикла for, начиная с традиционной формы.

Общая форма традиционного оператора for выглядит следующим образом:

for(инициализация; условие; повторение) {

// тело

}

Если в цикле будет повторяться только один оператор, фигурные скобки можно опустить.

Цикл for действует следующим образом. При первом запуске цикла программа выполняет инициализационную часть цикла. В общем случае это выражение, устанавливающее значение управляющей переменной цикла, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что выражение инициализации выполняется только один раз. Затем программа вычисляет условие, которое должно быть булевским выражением. Как правило, выражение сравнивает значение управляющей переменной с целевым значением. Если это значение истинно, программа выполняет тело цикла. Если оно ложно, выполнение цикла прерывается. Затем программа выполняет часть повторение цикла. Обычно это выражение, которое увеличивает или уменьшает значение управляющей переменной. Затем программа повторяет цикл, при каждом прохождении вначале вычисляя условное выражение, затем выполняя тело цикла и выполняя выражение повторения. Процесс повторяется до тех пор, пока значение выражения повторения не станет ложным.

Ниже приведена версия программы подсчета “тактов”, в которой использован цикл for.

// Демонстрация использования цикла for.

class ForTick {

public static void main(String args[]) {

int n;

for(n=10; n>0; n--)

System.out.println("такт " + n);

}

}

## Объявление управляющих переменных цикла внутри цикла for

Часто переменная, которая управляет циклом for, требуется только для него и не используется нигде больше. В этом случае переменную можно объявить внутри инициализационной части оператора for. Например, предыдущую программу можно переписать, объявляя управляющую переменную n типа int внутри цикла for:

// Объявление управляющей переменной цикла внутри цикла for.

class ForTick {

public static void main(String args[]) {

// в данном случае переменная n объявляется внутри цикла for

for(int n=10; n>0; n--)

System.out.println("такт " + n);

}

}

При объявлении переменной внутри цикла for необходимо помнить о следующем важном обстоятельстве: область и время существования этой переменной полностью совпадают с областью и временем существования оператора for. (То есть область существования переменной ограничена циклом for.) Вне цикла for переменная прекратит свое существование. Если управляющую переменную цикла нужно использовать в других частях программы, ее нельзя объявлять внутри цикла for. В тех случаях, когда управляющая переменная цикла не требуется нигде больше, большинство программистов Java предпочитают объявлять ее внутри оператора for.

В качестве примера приведем простую программу, которая проверяет, является ли введенное число простым. Обратите внимание, что управляющая переменная цикла i объявлена внутри цикла for, поскольку она нигде больше не требуется.

// Проверка принадлежности к простым числам.

class FindPrime {

public static void main(String args[]) {

int num;

boolean isPrime = true;

num = 14;

for(int i=2; i <= num/i; i++) {

if((num % i) == 0) {

isPrime = false;

break;

}

}

if(isPrime) System.out.println("Простое");

else System.out.println("Не простое");

}

}

В ряде случаев требуется указание нескольких операторов в инициализационной и итерационной частях цикла for. Например, рассмотрим цикл в следующей программе:

class Sample {

public static void main(String args[]) {

int a, b;

b = 4;

for(a=1; a<b; a++) {

System.out.println("a = " + a);

System.out.println("b = " + b);

b--;

}

}

}

Как видите, управление этим циклом осуществляется одновременно двумя переменными. Поскольку цикл управляется двумя переменными, желательно, чтобы их обе можно было бы включить в сам оператор for, а не выполнять обработку переменной b вручную. К счастью, Java предоставляет средства для выполнения этой задачи. Чтобы две или более переменных могли управлять циклом for, Java допускает указывать по несколько операторов как в инициализационной, так и итерационной частях оператора for. Один от другого операторы отделяются запятыми.

Используя запятые, предыдущий цикл for можно записать в более эффективном виде:

// Использование запятой.

class Comma {

public static void main(String args[]) {

int a, b;

for(a=1, b=4; a<b; a++, b--) {

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

}

В этом примере в инициализационной части цикла мы устанавливаем начальные значения обеих управляющих переменных a и b. Оба разделенных запятой оператора в итерационной части выполняются при каждом повторении цикла. Программа генерирует следующий вывод:

*a = 1*

*b = 4*

*a = 2*

*b = 3*

Читатели, знакомые с языками C/C++, знают, что в этих языках запятая – символ операции, который можно использовать в любом допустимом выражении. Однако в Java это не так. В нем запятая служит разделителем.

## Версия “for-each” цикла for

Как вам, возможно, известно, в современной теории языков программирования все большее применение находит концепция циклов “for-each”, которые быстро становится стандартными функциональными возможностями во многих языках. Цикл в стиле “for-each” предназначен для строго последовательного выполнения повторяющихся действий по отношению к коллекции объектов, такой как массив. В отличие некоторых языков, подобных C#, в котором для реализации циклов “for-each” используют ключевое слово foreach, в Java возможность применения цикла “for-each” реализована за счет усовершенствования цикла for. Преимущество этого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово, и никакой ранее существовавший код не разрушается. Цикл for в стиле “for-each” называют также усовершенствованным циклом for. Общая форма версии “for-each” цикла for имеет следующий вид:

for(тип итерационной-переменной : коллекция) блок-операторов

Здесь тип указывает тип, а итер-пер – имя итерационной переменной, которая последовательно будет принимать значения из коллекции, от первого до последнего. Элемент коллекция указывает коллекцию, по которой должен выполняться цикл. С циклом for можно применять различные типы коллекций, но в этой главе мы будем использовать только массивы. (Другие типы коллекций, которые можно применять с циклом for, вроде определенных в каркасе коллекций Collection Framework, рассматриваются в последующих главах книги.) На каждой итерации цикла программа извлекает следующий элемент коллекции и сохраняет его в переменной итер-пер. Цикл выполняется до тех пор, пока не будут получены все элементы коллекции. Поскольку итерационная переменная получает значения из коллекции, тип должен совпадать (или быть совместимым) с типом элементов, хранящихся в коллекции. Таким образом, при выполнении цикла по массивам тип должен быть совместим с базовым типом массива.

Чтобы понять побудительные причины применения циклов в стиле “for-each”, рассмотрим тип цикла for, для замены которого предназначен этот стиль. В следующем фрагменте для вычисления суммы значений элементов массива применяется традиционный цикл for:

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];

Чтобы вычислить сумму, мы последовательно считываем значения каждого из элементов массива nums. Таким образом, чтение всего массива выполняется строго последовательно. Это выполняется путем индексации массива nums вручную по управляющей переменной цикла i.

Цикл for в стиле “for-each” позволяет автоматизировать этот процесс. В частности, применение такого цикла позволяет не устанавливать значение счетчика цикла за счет указания его начального и конечного значений, и исключает необходимость индексации массива вручную. Вместо этого программа автоматически выполняет цикл по всему массиву, последовательно получая значения каждого из его элементов, от первого до последнего. Например, с учетом версии “for-each” цикла for предыдущий фрагмент можно переписать следующим образом:

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int x: nums) sum += x;

При каждом прохождении цикла переменной x автоматически присваивается значение, равное значению следующего элемента массива nums. Таким образом, на первой итерации x содержит 1, на второй – 2 и т.д. При этом не только упрощается синтаксис программы, но и исключается возможность ошибок выхода за пределы массива.

Ниже показан пример полной программы, иллюстрирующей применение описанной версии “for-each” цикла for.

// Использование цикла for в стиле for-each.

class ForEach {

public static void main(String args[]) {

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

// использование стиля for-each для отображения и суммирования значений

for(int x : nums) {

System.out.println("Значение равно: " + x);

sum += x;

}

System.out.println("Сумма равна: " + sum);

}

}

Эта программа генерирует следующий вывод:

*Значение равно: 1*

*Значение равно: 2*

*Значение равно: 3*

*Значение равно: 4*

*Значение равно: 5*

*Значение равно: 6*

*Значение равно: 7*

*Значение равно: 8*

*Значение равно: 9*

*Значение равно: 10*

*Сумма равна: 55*

Как видно из этого вывода, оператор for в стиле “for-each” автоматически выполняет цикл по элементам массива, от наименьшего индекса к наибольшему.

Хотя повторение цикла for в стиле “for-each” выполняется до тех пор, пока не будут обработаны все элементы массива, цикл можно прервать и раньше, используя оператор break. Например, следующая программа суммирует значения пяти первых элементов массива nums.

// Использование оператора break в цикле for в стиле for-each.

class ForEach2 {

public static void main(String args[]) {

int sum = 0;

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// использование цикла for для отображения и суммирования значений

for(int x : nums) {

System.out.println("Значение равно: " + x);

sum += x;

if(x == 5) break; // прекращение цикла после получения 5 значений

}

System.out.println("Сумма пяти первых элементов равна: " + sum);

}

}

Программа генерирует следующий вывод:

*Значение равно: 1*

*Значение равно: 2*

*Значение равно: 3*

*Значение равно: 4*

*Значение равно: 5*

*Сумма пяти первых элементов равна: 15*

Как видите, выполнение цикла прекращается после получения значения пятого элемента. Оператор break можно использовать также и с другими циклами Java. Подробнее этот оператор будет рассмотрен в последующих разделах настоящей главы.

При использовании цикла в стиле “for-each” необходимо помнить о следующем важном обстоятельстве. Его итерационная переменная является переменной “только для чтения”, поскольку она связана только с исходным массивом. Операция присваивания значения итерационной переменной не оказывает никакого влияния на исходный массив.

Иначе говоря, содержимое массива нельзя изменять, присваивая новое значение итерационной переменной. Например, рассмотрим следующую программу:

// Переменная цикла for-each доступна только для чтения.

class NoChange {

public static void main(String args[]) {

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

BookNew\_JAVA-7.indb 130 02.06.2007 1:06:45

Глава 5. Управляющие операторы 131

for(int x : nums) {

System.out.print(x + " ");

x = x \* 10; // этот оператор не оказывает никакого влияния на массив nums

}

System.out.println();

for(int x : nums)

System.out.print(x + " ");

System.out.println();

}

}

Первый цикл for увеличивает значение итерационной переменной на 10. Однако эта операция присваивания не оказывает никакого влияния на исходный массив nums, как видно из результата выполнения второго оператора for. Генерируемый программой вывод подтверждает сказанное:

*1 2 3 4 5 6 7 8 9 10*

*1 2 3 4 5 6 7 8 9 10*

## Итерация в многомерных массивах

Усовершенствованная версия цикла for применима также и к многомерным массивам.

Однако следует помнить, что в Java многомерные массивы состоят из массивов массивов. Это важно при выполнении итерации в многомерном массиве, поскольку результат каждой итерации – следующий массив, а не отдельный элемент. Более того, тип итерационной переменной цикла for должен быть совместим с типом получаемого массива. Например, в случае двумерного массива итерационная переменная должна быть ссылкой на одномерный массив.

В общем случае при использовании цикла “for-each” для выполнения итерации в массиве размерности N получаемые объекты будут массивами размерности N-1. Дабы понять, что из этого следует, рассмотрим следующую программу. В ней вложенные циклы for служат для получения упорядоченных по строкам элементов двумерного массива.

// Использование цикла for в стиле for-each применительно к двумерному массиву.

class ForEach3 {

public static void main(String args[]) {

int sum = 0;

int nums[][] = new int[3][5];

// присвоение значений элементам массива nums

for(int i = 0; i < 3; i++)

for(int j=0; j < 5; j++)

nums[i][j] = (i+1)\*(j+1);

// использование цикла for в стиле for-each для отображения

// и суммирования значений

for(int x[] : nums) {

for(int y : x) {

System.out.println("Значение равно: " + y);

sum += y;

}

}

System.out.println("Сумма: " + sum);

}

}

Эта программа генерирует следующий вывод:

*Значение равно: 1*

*Значение равно: 2*

*Значение равно: 3*

*Значение равно: 4*

*Значение равно: 5*

*Значение равно: 2*

*Значение равно: 4*

*Значение равно: 6*

*Значение равно: 8*

*Значение равно: 10*

*Значение равно: 3*

*Значение равно: 6*

*Значение равно: 9*

*Значение равно: 12*

*Значение равно: 15*

*Сумма: 90*

Следующая строка этой программы заслуживает особого внимания:

for(int x[] : nums) {

Обратите внимание на способ объявления переменной x. Эта переменная – ссылка на одномерный массив целочисленных значений. Это необходимо потому, что результат выполнения каждой итерации цикла for – следующий массив в массиве nums, начиная с массива, указанного элементом nums[0]. Затем внутренний цикл for выполняет итерацию по каждому из этих массивов, отображая значения каждого из элементов.

## Использование усовершенствованного цикла for

Поскольку каждый оператор for в стиле “for-each” может выполнять цикл по элементам массива только последовательно, начиная с первого и заканчивая последним, может показаться, что его применение ограничено. Однако это не так. Множество алгоритмов требуют использования именно этого механизма. Одним из наиболее часто используемых алгоритмов является поиск. Например, следующая программа использует цикл for для поиска значения в неупорядоченном массиве. Поиск прекращается после обнаружения искомого значения.

// Поиск в массиве с применением цикла for в стиле for-each.

class Search {

public static void main(String args[]) {

int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };

int val = 5;

boolean found = false;

// использование цикла for в стиле for-each для в nums значения val

for(int x : nums) {

if(x == val) {

found = true;

break;

}

}

if(found)

System.out.println("Значение найдено!");

}

}

В данном случае выбор стиля “for-each” для цикла for полностью оправдан, поскольку поиск в неупорядоченном массиве предполагает последовательный просмотр каждого из элементов. (Конечно, если бы массив был упорядоченным, можно было бы использовать бинарный поиск, реализация которого требовала бы применения другого стиля цикла.)

К другим типам приложений, которым применение циклов в стиле “for-each” предоставляет преимущества, относятся вычисление среднего значения, отыскание минимального или максимального значения в наборе, поиск дубликатов и т.п.

## Вложенные циклы

Подобно всем другим языкам программирования, Java допускает использование вложенных циклов. То есть один цикл может выполняться внутри другого. Например, в следующей программе использованы вложенные циклы for:

// Циклы могут быть вложенными.

class Nested {

public static void main(String args[]) {

int i, j;

for(i=0; i<10; i++) {

for(j=i; j<10; j++)

System.out.print(".");

System.out.println();

}

}

}

Эта программа генерирует следующий вывод:

*..........*

*.........*

*........*

*.......*

*......*

*.....*

*....*

*...*

*..*

*.*

## Операторы перехода

В Java определены три оператора перехода: break, continue и return. Они передают управление другой части программы. Рассмотрим каждый из них.

Кроме операторов перехода, рассмотренных в этом разделе, Java поддерживает еще один способ изменения порядка выполнения инструкций программы: обработку исключений. Обработка исключений предоставляет структурированный метод, посредством которого программа может обнаруживать и обрабатывать ошибки времени выполнения. Для поддержки этого метода служат ключевые слова try, catch, throws и finally. По сути, механизм обработки ошибок позволяет программе выполнять нелокальные ветви. Поскольку тема обработки исключений очень обширна, она рассмотрена отдельно.

## Использование оператора break

В Java оператор break находит три применения. Во-первых, как уже было показано, он завершает последовательность операторов в операторе switch. Во-вторых, его можно использовать для выхода из цикла. И, в-третьих, его можно использовать в качестве “цивилизованной” формы оператора безусловного перехода (“goto”). Рассмотрим последние два применения.

Используя оператор break, можно вызвать немедленное завершение цикла, пропуская условное выражение и любой остальной код в теле цикла. Когда программа встречает оператор break внутри цикла, она прекращает выполнение цикла, и управление передается оператору, следующему за циклом. Ниже показан простой пример.

// Использование оператора break для выхода из цикла.

class BreakLoop {

public static void main(String args[]) {

for(int i=0; i<100; i++) {

if(i == 10) break; // выход из цикла если i равно 10

System.out.println("i: " + i);

}

System.out.println("Цикл завершен.");

}

}

Эта программа генерирует следующий вывод:

*i: 0*

*i: 1*

*i: 2*

*i: 3*

*i: 4*

*i: 5*

*i: 6*

*i: 7*

*i: 8*

*i: 9*

*Цикл завершен.*

Как видите, хотя цикл for должен был бы выполняться для значений управляющей переменной от 0 до 99, оператор break приводит к более раннему выходу из него, когда значение переменной i становится равным 10.

Оператор break можно использовать в любых циклах Java, в том числе в преднамеренно бесконечных циклах. Например, в предыдущей программе можно было применить цикл while. Эта программа генерирует вывод, совпадающий с предыдущим.

// Использование оператора break для выхода из цикла while.

class BreakLoop2 {

public static void main(String args[]) {

int i = 0;

while(i < 100) {

if(i == 10) break; // выход из цикла, если i равно 10

System.out.println("i: " + i);

i++;

}

System.out.println("Цикл завершен.");

}

}

В случае его использования внутри набора вложенных циклов оператор break осуществляет выход только из самого внутреннего цикла. Например:

// Использование оператора break во вложенных циклах.

class BreakLoop3 {

public static void main(String args[]) {

for(int i=0; i<3; i++) {

System.out.print("Проход " + i + ": ");

for(int j=0; j<100; j++) {

if(j == 10) break; // выход из цикла, если j равно 10

System.out.print(j + " ");

}

System.out.println();

}

System.out.println("Циклы завершены.");

}

}

Эта программа генерирует следующий вывод:

*Проход 0: 0 1 2 3 4 5 6 7 8 9*

*Проход 1: 0 1 2 3 4 5 6 7 8 9*

*Проход 2: 0 1 2 3 4 5 6 7 8 9*

*Циклы завершены.*

Как видите, оператор break во внутреннем цикле может приводить к выходу только из этого цикла. На внешний цикл он не оказывает никакого влияния.

При использовании оператора break необходимо помнить следующее. Во-первых, в цикле можно использовать более одного оператора break. Однако при этом следует соблюдать осторожность. Как правило, применение слишком большого числа операторов break приводит к деструктуризации кода. Во-вторых, оператор break, который завершает последовательность операторов в операторе switch, оказывает влияние только на данный оператор switch, а не на какие-либо содержащие его циклы.

## Использование оператора break в качестве формы оператора безусловного перехода

Кроме его применения с операторами switch и циклами, оператор break можно использовать и сам по себе в качестве “цивилизованной” формы оператора безусловного перехода (“goto”). Язык Java не содержит оператора “goto”, поскольку он позволяет выполнять ветвление программ произвольным и неструктурированным образом. Как правило, код, который управляется оператором “goto”, труден для понимания и поддержки. Кроме того, этот оператор исключает возможность оптимизации кода для определенного компилятора. Однако в некоторых случаях оператор “goto” – ценная и вполне допустимая конструкция управления потоком команд. Например, оператор “goto” может быть полезен при выходе из набора вложенных циклов с большим количеством уровней. Для таких ситуаций Java определяет расширенную форму оператора break. Используя эту форму, можно, например, осуществлять выход из одного или нескольких блоков кода. Эти блоки не обязательно должны быть частью цикла или оператора switch. Они могут быть любым блоком. Более того, можно точно указать оператор, с которого будет продолжено выполнение программы, поскольку эта форма оператора break работает с метками. Как будет показано, оператор break предоставляет все преимущества оператора “goto”, не порождая его проблемы. Общая форма оператора break с меткой имеет следующий вид:

break метка;

Чаще всего, метка – это имя метки, идентифицирующей блок кода. Им может быть как самостоятельный блок кода, так и целевой блок другого оператора. При выполнении этой формы оператора break управление передается названному блоку кода.

Помеченный блок кода должен содержать оператор break, но он не обязательно должен быть непосредственно содержащим его блоком. В частности это означает, что оператор break с меткой можно применять для выхода из набора вложенных блоков. Однако его нельзя использовать для передачи управления внешнему блоку кода, который не содержит данный оператор break.

Чтобы пометить блок, необходимо поместить метку в его начале. Метка – это любой допустимый идентификатор Java, за которым следует двоеточие. Как только блок помечен, его метку можно использовать в качестве цели оператора break. В результате выполнение программы будет продолжено с конца помеченного блока. Например, следующая программа содержит три вложенных блока, каждый из которых помечен своей меткой. Оператор break приводит к переходу к концу блока с меткой second с пропуском двух операторов println().

// Использование оператора break в качестве цивилизованной формы оператора goto.

class Break {

public static void main(String args[]) {

boolean t = true;

first: {

second: {

third: {

System.out.println("Предшествует оператору break.");

if(t) break second; // выход из блока second

BookNew\_JAVA-7.indb 136 02.06.2007 1:06:46

Глава 5. Управляющие операторы 137

System.out.println("Этот оператор не будет выполняться");

}

System.out.println("Этот оператор не будет выполняться");

}

System.out.println("Этот оператор следует за блоком second.");

}

}

}

Эта программа генерирует следующий вывод:

*Предшествует оператору break.*

*Этот оператор следует за блоком second.*

Одно из наиболее распространенных применений оператора break с меткой – его использование для выхода из вложенных циклов. Например, в следующей программе внешний цикл выполняется только один раз:

// Использование оператора break для выхода из вложенных циклов

class BreakLoop4 {

public static void main(String args[]) {

outer: for(int i=0; i<3; i++) {

System.out.print("Проход " + i + ": ");

for(int j=0; j<100; j++) {

if(j == 10) break outer; // выход из обоих циклов

System.out.print(j + " ");

}

System.out.println("Эта строка не будет выводиться");

}

System.out.println("Циклы завершены.");

}

}

Эта программа генерирует следующий вывод:

*Проход 0: 0 1 2 3 4 5 6 7 8 9 Циклы завершены.*

Как видите, когда внутренний цикл выполняет выход во внешний цикл, это приводит к завершению обоих циклов.

Следует иметь в виду, что нельзя выполнить переход к какой-либо метке, которая определена не для содержащего данный оператор break блока. Например, следующая программа содержит ошибку и поэтому ее компиляция будет невозможна:

// Эта программа содержит ошибку.

class BreakErr {

public static void main(String args[]) {

one: for(int i=0; i<3; i++) {

System.out.print("Pass " + i + ": ");

}

for(int j=0; j<100; j++) {

if(j == 10) break one; // ОШИБКА!

System.out.print(j + " ");

}

}

}

Поскольку блок, помеченный меткой one, не содержит оператор break, передача управления этому внешнему блоку невозможна.

## Использование оператора continue

Иногда требуется, чтобы повторение цикла осуществлялось с более раннего оператора его тела. То есть на данной конкретной итерации может требоваться продолжить выполнение цикла без обработки остального кода в его теле. По сути, это означает переход в теле цикла к его окончанию. Для выполнения этого действия служит оператор continue.

В циклах while и do-while оператор continue вызывает передачу управления непосредственно управляющему условному выражению цикла. В цикли for управление передается вначале итерационной части цикла for, а потом условному выражению. Во всех этих трех циклах любой промежуточный код пропускается.

Ниже приведен пример программы, в которой оператор continue используется для вывода двух чисел в каждой строке.

// Демонстрация применения оператора continue.

class Continue {

public static void main(String args[]) {

for(int i=0; i<10; i++) {

System.out.print(i + " ");

if (i%2 == 0) continue;

System.out.println("");

}

}

}

В этом коде операция % служит для проверки четности значения переменной i. Если оно четное, выполнение цикл продолжается без перехода к новой строке. Программа генерирует следующий вывод:

*0 1*

*2 3*

*4 5*

*6 7*

*8 9*

Как и оператор break, оператор continue может содержать метку содержащего его цикла, который нужно продолжить. Ниже показан пример программы, в которой оператор continue применяется для вывода треугольной таблицы умножения чисел от 0 до 9.

// Использование оператора continue с меткой.

class ContinueLabel {

public static void main(String args[]) {

outer: for (int i=0; i<10; i++) {

for(int j=0; j<10; j++) {

if(j > i) {

System.out.println();

continue outer;

}

System.out.print(" " + (i \* j));

}

}

System.out.println();

}

}

В этом примере оператор continue прерывает цикл подсчета значений j и продолжает цикл со следующей итерации цикла подсчета i. Вывод этой программы имеет следующий вид:

*0*

*0 1*

*0 2 4*

*0 3 6 9*

*0 4 8 12 16*

*0 5 10 15 20 25*

*0 6 12 18 24 30 36*

*0 7 14 21 28 35 42 49*

*0 8 16 24 32 40 48 56 64*

*0 9 18 27 36 45 54 63 72 81*

Удачные применения оператора continue встречаются редко. Одна из причин состоит в том, что Java предлагает широкий выбор операторов цикла, удовлетворяющих требованиям большинства приложений. Однако в тех случаях, когда требуется более раннее начало новой итерации, оператор continue предоставляет структурированный метод выполнения этой задачи.

## Оператор return

Последний из управляющих операторов – return. Его используют для выполнения явного возврата из метода. То есть он снова передает управление объекту, который вызвал данный метод. Как таковой этот оператор относится к операторам перехода.

Оператор return можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Таким образом, оператор return немедленно прекращает выполнение метода, в котором он находится. Следующий пример иллюстрирует это. В данном случае оператор return приводит к возврату управления системе времени выполнения Java, поскольку именно она вызывает метод main().

// Демонстрация использования оператора return.

class Return {

public static void main(String args[]) {

boolean t = true;

System.out.println("До выполнения возврата.");

if(t) return; // возврат к вызывающему объекту

System.out.println("Этот оператор выполняться не будет.");

}

}

Вывод этой программы имеет вид:

*До выполнения возврата.*

Как видите, заключительный оператор println() не выполняется. Сразу после выполнения оператора return программа возвращает управление вызывающему объекту.

И последний нюанс: в приведенной программе использование оператора if(t) обязательно. Без него компилятор Java сигнализировал бы об ошибке “unreachable code” (“недостижимый код”), поскольку выяснил бы, что последний оператор println() никогда не будет выполняться. Во избежание этой ошибки в демонстрационном примере пришлось ввести компилятор в заблуждение с помощью оператора if.

# Разработка классов на Java

Мы пользовались классами с самого начала этой книги. Однако до сих пор применялась только наиболее примитивная форма класса. Классы, созданные в предшествующих главах, служили только в качестве контейнеров метода main(), который мы использовали для ознакомления с основами синтаксиса Java. Как вы вскоре убедитесь, классы предоставляют значительно больше возможностей, чем те, которые были представлены до сих пор.

Вероятно, наиболее важное свойство класса то, что он определяет новый тип данных.

После того как он определен, этот новый тип можно применять для создания объектов данного типа. Таким образом, класс – это шаблон объекта, а объект – это экземпляр класса. Поскольку объект является экземпляром класса, термины объект и экземпляр часто используются попеременно.

## Общая форма класса

При определении класса объявляют его конкретную форму и сущность. Это выполняется путем указания данных, которые он содержит, и кода, воздействующего на эти данные. Хотя очень простые классы могут содержать только код или только данные, большинство классов, применяемых в реальных программах, содержит оба эти компонента.

Как будет показано в дальнейшем, код класса определяет интерфейс к его данным.

Для объявления класса служит ключевое слово class. Использованные до сих пор классы в действительности представляли собой очень ограниченные примеры полной формы. Классы могут быть (и обычно являются) значительно более сложными. Упрощенная общая форма определения класса имеет следующий вид:

class имя\_класса {

тип переменная\_экземпляра1;

тип переменная\_экземпляра2;

// ...

тип переменная\_экземпляраN;

тип имя\_метода1(список\_параметров) {

// тело метода

}

тип имя\_метода2(список\_параметров) {

// тело метода

}

// ...

тип имя\_методаN(список\_параметров) {

// тело метода

}

}

Данные, или переменные, определенные внутри класса, называются переменными экземпляра. Код содержится внутри методов. Определенные внутри класса методы и переменные вместе называют членами класса. В большинстве классов действия с переменными экземпляров и доступ к ним выполняются через методы, определенные в этом классе. Таким образом, в общем случае именно методы определяют способ использования данных класса.

Определенные внутри класса переменные называют переменными экземпляра, поскольку каждый экземпляр класса (т.е. каждый объект класса) содержит собственные копии этих переменных. Таким образом, данные одного объекта отделены и отличаются от данных другого объекта. Вскоре мы вернемся к рассмотрению этой концепции, но она слишком важна, чтобы можно было обойтись без хотя бы предварительного ознакомления с нею.

Все методы имеют ту же общую форму, что и метод main(), который мы использовали до сих пор. Однако большинство методов не будут указаны как static или public. Обратите внимание, что общая форма класса не содержит определения метода main(). Классы Java могут и не содержать этот метод. Его обязательно указывать только в тех случаях, когда данный класс служит начальной точкой программы. Более того, аплеты вообще не требуют использования метода main().

Обратите внимание, что объявление класса и реализация методов хранятся в одном месте, а не определены отдельно. Иногда эта особенность приводит к созданию очень больших файлов .java, поскольку любой класс должен быть полностью определен в одном файле исходного кода. Такая архитектура была принята для Java умышленно, поскольку разработчики посчитали, что хранение определения, объявления и реализации в одном файле упрощает сопровождение кода в течение длительного периода его эксплуатации.

## Простой класс

Изучение классов начнем с простого примера. Ниже приведен код класса Box (Параллелепипед), который определяет три переменных экземпляра: width (ширина), height(высота) и depth (глубина). В настоящий момент Box не содержит никаких методов (но вскоре мы добавим в него метод).

class Box {

double width;

double height;

double depth;

}

Как уже было сказано, класс определяет новый тип данных. В данном случае новый тип данных назван Box. Это имя будет использоваться для объявления объектов типа Box. Важно помнить, что объявление class создает только шаблон, но не действительный объект. Таким образом, приведенный код не приводит к появлению никаких объектов типа Box.

Чтобы действительно создать объект Box, нужно использовать оператор, подобный следующему:

Box mybox = new Box(); // создание объекта mybox типа Box

После выполнения этого оператора mybox станет экземпляром класса Box. То есть он обретет “физическое” существование. Но пока можете не задумываться о нюансах этого оператора.

Повторим еще раз: при каждом создании экземпляра класса мы создаем объект, который содержит собственную копию каждой переменной экземпляра, определенной классом. Таким образом, каждый объект Box будет содержать собственные копии переменных экземпляра width, height и depth. Для получения доступа к этим переменным применяется операция точки (.). Эта операция связывает имя объекта с именем переменной экземпляра. Например, чтобы присвоить переменной width объекта mybox значение 100, нужно было бы использовать следующий оператор:

mybox.width = 100;

Этот оператор указывает компилятору, что копии переменной width, хранящейся внутри объекта mybox, нужно присвоить значение, равное 100. В общем случае операцию точки используют для доступа как к переменным экземпляра, так и к методам внутри объекта.

Ниже приведена полная программа, в которой используется класс Box.

/\* Программа, использующая класс Box.

Назовите этот файл BoxDemo.java

\*/

class Box {

double width;

double height;

double depth;

}

// Этот класс объявляет объект типа Box.

class BoxDemo {

public static void main(String args[]) {

Box mybox = new Box();

double vol;

BookNew\_JAVA-7.indb 143 02.06.2007 1:06:46

144 Часть I. Язык Java

// присваивание значений переменным экземпляра mybox

mybox.width = 10;

mybox.height = 20;

mybox.depth = 15;

// вычисление объема параллелепипеда

vol = mybox.width \* mybox.height \* mybox.depth;

System.out.println("Объем равен " + vol);

}

}

Файлу этой программы нужно присвоить имя BoxDemo.java, поскольку метод main() определен в классе, названном BoxDemo, а не Box. Выполнив компиляцию этой программы, вы убедитесь в создании двух файлов .class: одного для Box и одного для BoxDemo. Компилятор Java автоматически помещает каждый класс в отдельный файл с расширением .class. В действительности классы Box и BoxDemo не обязательно должны быть объявлены в одном и том же исходном файле. Каждый класс можно было бы поместить в отдельный файл, названный соответственно Box.java и BoxDemo.java.

Как было сказано ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов Box каждый из них будет содержать собственные копии переменных depth, width и height. Важно понимать, что изменения переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующем программе объявлены два объекта Box:

// Эта программа объявляет два объекта Box.

class Box {

double width;

double height;

double depth;

}

class BoxDemo2 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// присваивание значений переменным экземпляра mybox1

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

//\* присваивание других значений переменным

экземпляра mybox2's \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// вычисление объема первого параллелепипеда

vol = mybox1.width \* mybox1.height \* mybox1.depth;

System.out.println("Объем равен " + vol);

// вычисление объема второго параллелепипеда

vol = mybox2.width \* mybox2.height \* mybox2.depth;

System.out.println("Volume is " + vol);

}

}

Как было сказано ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов Box каждый из них будет содержать собственные копии переменных depth, width и height. Важно понимать, что изменения переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующем программе объявлены два объекта Box:

// Эта программа объявляет два объекта Box.

class Box {

double width;

double height;

double depth;

}

class BoxDemo2 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// присваивание значений переменным экземпляра mybox1

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

//\* присваивание других значений переменным

экземпляра mybox2's \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// вычисление объема первого параллелепипеда

vol = mybox1.width \* mybox1.height \* mybox1.depth;

System.out.println("Объем равен " + vol);

// вычисление объема второго параллелепипеда

vol = mybox2.width \* mybox2.height \* mybox2.depth;

System.out.println("Volume is " + vol);

}

}

Эта программа генерирует следующий вывод:

*Объем равен 3000.0*

*Объем равен 162.0*

Как видите, данные объекта mybox1 полностью изолированы от данных, содержащихся в объекте mybox2.

## Объявление объектов

Как мы уже отмечали, при создании класса вы создаете новый тип данных. Этот тип можно использовать для объявления объектов данного типа. Однако создание объектов класса – двухступенчатый процесс. Вначале необходимо объявить переменную типа класса. Эта переменная не определяет объект. Она представляет собой всего лишь переменную, которая может ссылаться на объект. Затем потребуется получить действительную, физическую копию объекта и присвоить ее этой переменной. Это можно выполнить с помощью операции new. Эта операция динамически (т.е. во время выполнения) распределяет память под объект и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, распределенной операцией new. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты классов должны распределяться динамически. Рассмотрим эту процедуру более подробно.

В приведенном ранее примере программы строка, подобная следующей, используется для объявления объекта типа Box:

Box mybox = new Box();

Этот оператор объединяет только что описанные шаги. Чтобы каждый из шагов был более очевидным, его можно было переписать следующим образом:

Box mybox; // объявление ссылки на объект

mybox = new Box(); // распределение памяти для объекта Box

Первая строка объявляет mybox в качестве ссылки на объект типа Box. После выполнения этой строки переменная mybox содержит значение null, свидетельствующее о том, что она еще не указывает на реальный объект. Любая попытка использования mybox на этом этапе приведет к возникновению ошибки времени компиляции. Следующая строка распределяет память под реальный объект и присваивает переменной mybox ссылку на этот объект. После выполнения второй строки переменную mybox можно использовать, как если бы она была объектом Box. Но в действительности переменная mybox просто содержит адрес памяти реального объекта Box

Ссылки на объекты выглядят подобно указателям. В общих чертах это впечатление верно. Ссылка на объект похожа на указатель памяти. Основное различие между ними – и основное свойство, обеспечивающее безопасность программ Java – в том, что ссылками нельзя манипулировать как настоящими указателями. В частности, ссылка на объект не может указывать на произвольную ячейку памяти, и ею нельзя манипулировать как целочисленным значением.

## Более подробное рассмотрение операции new

Как было сказано, операция new динамически распределяет память для объекта.

Общая форма этой операции имеет следующий вид:

переменная\_класса = new имя\_класса();

Здесь переменная\_класса – переменная создаваемого типа класса. Имя\_класса – это имя класса, конкретизация которого выполняется. Имя класса, за которым следуют круглые скобки, указывает конструктор данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. Конструкторы – важная часть всех классов, и они обладают множеством важных атрибутов. Большинство классов, используемых в реальных программах, явно определяют свои конструкторы внутри своего определения класса. Однако если никакой явный конструктор не указан, Java автоматически предоставит конструктор, используемый по умолчанию. Это же происходит в случае объекта Box.

Пока мы будем пользоваться конструктором, заданным по умолчанию. Вскоре читатели научатся определять собственные конструкторы.

У читателей может возникнуть вопрос, почему не требуется использовать операцию new для таких элементов, как целые числа или символы. Это обусловлено тем, что элементарные типы Java реализованы в не виде объектов, а в виде “обычных” переменных. Это сделано для повышения эффективности. Как вы убедитесь, объекты обладают множеством свойств и атрибутов, которые требуют, чтобы Java-программа обрабатывала их иначе, чем элементарные типы. Отсутствие накладных расходов, связанных с обработкой объектов, при обработке элементарных типов позволяет эффективнее реализовать элементарные типы. Несколько позже мы приведем объектные версии элементарных типов, которые могут пригодиться в ситуациях, когда требуются полноценные объекты этих типов.

Важно понимать, что операция new распределяет память для объекта во время выполнения. Преимущество этого подхода состоит в том, что программа может создавать ровно столько объектов, сколько требуется во время ее выполнения. Однако поскольку объем памяти ограничен, возможна ситуация, когда операция new не сможет выделить память для объекта из-за ее нехватки. В этом случае возникнет исключение времени выполнения. (Обработка этого и других исключений описана в главе 10.) В примерах программ, приведенных в этой книге, можно не беспокоиться по поводу недостатка объема памяти, но в реальных программах эту возможность придется учитывать.

Еще раз рассмотрим различие между классом и объектом. Класс создает новый тип данных, который можно использовать для создания объектов. То есть класс создает логический каркас, определяющий взаимосвязь между его членами. При объявлении объекта класса мы создаем экземпляр этого класса. Таким образом, класс – это логическая конструкция. А объект обладает физической сущностью. (То есть объект занимает область в памяти.) Важно помнить об этом различии.

## Присваивание переменных объектных ссылок

При выполнении присваивания переменные объектных ссылок действуют иначе, чем можно было бы представить. Например, какие действия, по вашему мнению, выполняет следующий фрагмент?

Box b1 = new Box();

Box b2 = b1;

Можно подумать, что переменной b2 присваивается ссылка на копию объекта, на которую ссылается переменная b1. То есть может показаться, что b1 и b2 ссылаются на отдельные и различные объекты. Однако это не так. После выполнения этого фрагмента кода обе переменные b1 и b2 будут ссылаться на один и тот же объект. Присваивание b1 переменной b2 не привело к распределению какой-то памяти или копированию какойлибо части исходного объекта. Эта операция присваивания приводит лишь к тому, что переменная b2 ссылается на тот же объект, что и переменная b1. Таким образом, любые изменения, выполненные в объекте через переменную b2, окажут влияние на объект, на который ссылается переменная b1, поскольку это – один и тот же объект.

Хотя и b1 и b2 ссылаются на один и тот же объект, эти переменные не связаны между собой никаким другим образом. Например, следующая операция присваивания значения переменной b1 просто разорвет связь переменной b1 с исходным объектом, не оказывая влияния на сам объект или на переменную b2:

Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;

В этом примере значение b1 установлено равным null, но переменная b2 по-прежнему указывает на исходный объект. Присваивание ссылочной переменной одного объекта ссылочной переменной другого объекта не ведет к созданию копии объекта, а лишь создает копию ссылки.

## Знакомство с методами

Как было сказано в начале этой главы, обычно классы состоят из двух элементов: переменных экземпляра и методов. Поскольку язык Java предоставляет им столь большие возможности и гибкость, тема методов очень обширна. Фактически многие последующие главы посвящены методам. Однако чтобы можно было приступить к добавлению методов к своим классам, необходимо ознакомиться с рядом их основных характеристик.

Общая форма объявления метода выглядит следующим образом:

тип имя(список\_параметров) {

// тело метода

}

Здесь тип указывает тип данных, возвращаемых методом. Он может быть любым допустимым типом, в том числе типом класса, созданным программистом. Если метод не возвращает значение, типом его возвращаемого значения должен быть void. Имя служит для указания имени метода. Оно может быть любым допустимым идентификатором, кроме тех, которые уже используются другими элементами в текущей области определения.

Список\_параметров – последовательность пар “тип-идентификатор”, разделенных запятыми. По сути, параметры – это переменные, которые принимают значения аргументов, переданных методу во время его вызова. Если метод не имеет параметров, список параметров будет пустым.

Методы, тип возвращаемого значения которых отличается от void, возвращают значение вызывающей процедуре с помощью следующей формы оператора return:

return значение;

Здесь значение – это возвращаемое значение.

Хотя было бы весьма удобно создать класс, который содержит только данные, в реальных программах подобное встречается редко. В большинстве случаев для осуществления доступа к переменным экземпляра, определенным классом, придется использовать методы. Фактически методы определяют интерфейсы большинства классов. Это позволяет программисту, который реализует класс, скрывать конкретную схему внутренних структур данных за более понятными абстракциями метода. Кроме определения методов, которые обеспечивают доступ к данным, можно определять также методы, используемые внутренне самим классом.

// Эта программа содержит метод внутри класса box.

class Box {

double width;

double height;

double depth;

// отображение объема параллелепипеда

void volume() {

System.out.print("Объем равен ");

System.out.println(width \* height \* depth);

}

}

class BoxDemo3 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

// присваивание значений переменным экземпляра mybox1

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* присваивание других значений переменным

экземпляра mybox2 \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// отображение объема первого параллелепипеда

mybox1.volume();

// отображение объема второго параллелепипеда

mybox2.volume();

}

}

Эта программа генерирует следующий вывод, совпадающий с выводом предыдущей версии:

*Объем равен 3000.0*

*Объем равен 162.0*

Внимательно взгляните на следующие две строки кода:

mybox1.volume();

mybox2.volume();

В первой строке присутствует обращение к методу volume(), определенному в mybox1. То есть она вызывает метод volume() по отношению к объекту mybox1, для чего было использовано имя объекта, за которым следует символ операции точки. Таким образом, обращение к mybox1.volume() отображает объем параллелепипеда, определенного объектом mybox1, а обращение к mybox2.volume() – объем параллелепипеда, определенного объектом mybox2. При каждом вызове метода volume() он отображает объем указанного параллелепипеда.

Соображения, приведенные в следующих абзацах, облегчат понимание концепции вызова метода. При вызове метода mybox1.volume() система времени выполнения Java передает управление коду, определенному внутри метода volume(). По завершении выполнения всех операторов внутри метода управление возвращается вызывающей программе и ее выполнение продолжается со строки, которая следует за вызовом метода. В самом общем смысле можно сказать, что метод – способ реализации подпрограмм в Java.

В методе volume() следует обратить внимание на один очень важный нюанс: ссылка на переменные экземпляра width, height и depth выполняется непосредственно, без указания перед ними имени объекта или операции точки. Когда метод использует переменную экземпляра, которая определена его классом, он выполняет это непосредственно, без указания явной ссылки на объект и без применения операции точки. Это становится понятным, если немного подумать. Метод всегда вызывается по отношению к какому-то объекту его класса. Как только этот вызов выполнен, объект известен. Таким образом, внутри метода вторичное указание объекта совершенно излишне. Это означает, что width, height и depth неявно ссылаются на копии этих переменных, хранящиеся в объекте, который вызывает метод volume().

## Возвращение значения

Хотя реализация метода volume() переносит вычисление объема параллелепипеда внутрь класса Box, которому принадлежит этот метод, такой способ вычисления не является наилучшим. Например, что делать, если другой часть программы требуется знание объема параллелепипеда без его отображения? Более рациональный способ реализации метода volume() – вычисление объема параллелепипеда и возврат результата вызывающему объекту. Следующий пример – усовершенствованная версия предыдущей программы – выполняет именно эту задачу.

// Теперь метод volume() возвращает объем параллелепипеда.

class Box {

double width;

double height;

double depth;

// вычисление и возвращение объема

double volume() {

return width \* height \* depth;

}

}

class BoxDemo4 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// присваивание значений переменным экземпляра mybox1

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* присваивание других значений переменным экземпляра mybox2 \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем равен " + vol);

}

}

Как видите, вызов метода volume() выполняется в правой части оператора присваивания. Правой частью этого оператора является переменная, в данном случае vol, которая будет принимать значение, возвращенное методом volume(). Таким образом, после выполнения такого оператора:

vol = mybox1.volume();

, значение mybox1.volume() равно 3000, и этот объем сохраняется в переменной vol.

При работе с возвращаемыми значениями следует учитывать два важных обстоятельства.

Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным методом. Например, если возвращаемым типом какого-либо метода является boolean, нельзя возвращать целочисленное значение. Переменная, принимающая возвращенное методом значение (такая как vol в данном случае), также должна быть совместима с возвращаемым типом, указанным для метода. И еще один нюанс: предыдущую программу можно было бы записать в несколько более эффективной форме, поскольку в действительности переменная vol совершенно не нужна. Обращение к методу volume() можно было бы использовать в операторе println() непосредственно, как в следующей строке кода:

System.out.println("Объем равен " + mybox1.volume());

В этом случае при выполнении оператора println() метод mybox1.volume() будет вызываться автоматически, а возвращаемое им значение будет передаваться методу println().

## Добавление метода, принимающего параметры

Хотя некоторые методы не нуждаются в параметрах, большинство требует их передачи. Параметры позволяют обобщать метод. То есть метод с параметрами может работать с различными данными и/или применяться в ряде несколько различных ситуаций. В качестве иллюстрации рассмотрим очень простой пример. Ниже показан метод, который возвращает квадрат числа 10.

int square()

{

return 10 \* 10;

}

Хотя этот метод действительно возвращает 102, его применение очень ограничено. Однако если его изменить так, чтобы он принимал параметр, как показано в следующем примере, метод square() может стать значительно более полезным.

int square(int i)

{

return i \* i;

}

Теперь метод square() будет возвращать квадрат любого значения, с которым он вызван. То есть теперь метод square() является методом общего назначения, который может вычислять квадрат любого целочисленного значения, а не только числа 10.

Приведем примеры:

int x, y;

x = square(5); // x равно 25

x = square(9); // x равно 81

y = 2;

x = square(y); // x равно 4

В первом обращении к методу square() значение 5 будет передано параметру i. Во втором обращении параметр i примет значение, равное 9. Третий вызов метода передает значение переменной y, которое в этом примере составляет 2. Как видно из этих примеров, метод square() способен возвращать квадрат любых переданных ему данных.

Важно различать два термина: параметр и аргумент. Параметр – это переменная, определенная методом, которая принимает значение при вызове метода. Например, в методе square() параметром является i. Аргумент – это значение, передаваемое методу при его вызове. Например, aquare(100) передает 100 в качестве аргумента. Внутри метода square() параметр i получает это значение.

Методом с параметрами можно воспользоваться для усовершенствования класса Box.

// Эта программа использует метод с параметрами.

class Box {

double width;

double height;

double depth;

// вычисление и возвращение объема

double volume() {

return width \* height \* depth;

}

// установка размеров параллелепипеда

void setDim(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

}

class BoxDemo5 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// инициализация каждого экземпляра Box

mybox1.setDim(10, 20, 15);

mybox2.setDim(3, 6, 9);

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем равен " + vol);

}

}

Как видите, метод setDim() использован для установки размеров каждого параллелепипеда. Например, при выполнении такого оператора:

mybox1.setDim(10, 20, 15);

, значение 10 копируется в параметр w, 20 – в h и 15 – в d. Затем внутри метода setDim() значения w, h и d присваиваются соответственно переменным width, height и depth.

## Конструкторы

Инициализация всех переменных класса при каждом создании его экземпляра может оказаться утомительным процессом. Даже при добавлении функций, предназначенных для увеличения удобства работы, таких как setDim(), было бы проще и удобнее, если бы все действия по установке переменных выполнялись при первом создании объекта.

Поскольку необходимость инициализации возникает столь часто, Java позволяет объектам выполнять собственную инициализацию при их создании. Эта автоматическая инициализация осуществляется с помощью конструктора.

Конструктор инициализирует объект непосредственно во время создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только он определен, конструктор автоматически вызывается непосредственно после создания объекта, перед завершением выполнения операции new. Конструкторы выглядят несколько непривычно, поскольку не имеют ни возвращаемого типа, ни даже типа void. Это обусловлено тем, что неявно заданный возвращаемый тип конструктора класса – тип самого класса. Именно конструктор инициализирует внутреннее состояние объекта так, чтобы код, создающий экземпляр, с самого начала содержал полностью инициализированный, пригодный к использованию объект.

Пример класса Box можно изменить, чтобы значения размеров параллелепипеда присваивались при конструировании объекта. Для этого потребуется заменить метод setDim() конструктором. Вначале определим простой конструктор, который просто устанавливает одинаковые значения размеров для всех параллелепипедов. Эта версия программы имеет вид:

/\* В этом примере класс Box использует конструктор

для инициализации размеров параллелепипеда.

\*/

class Box {

double width;

double height;

double depth;

// Это конструктор класса Box.

Box() {

System.out.println("Конструирование объекта Box");

width = 10;

height = 10;

depth = 10;

}

// вычисление и возвращение объема

double volume() {

return width \* height \* depth;

}

}

class BoxDemo6 {

public static void main(String args[]) {

// объявление, распределение и инициализация объектов Box

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем равен " + vol);

}

}

Эта программа генерирует следующий вывод:

*Конструирование объекта Box*

*Конструирование объекта Box*

*Объем равен 1000.0*

*Объем равен 1000.0*

Как видите, и mybox1, и mybox2 были инициализированы конструктором Box() при их создании. Поскольку конструктор присваивает всем параллелепипедам одинаковые размеры 10×10×10, и mybox1, и mybox2 будут иметь одинаковый объем. Оператор println() внутри конструктора Box() служит исключительно иллюстративным целям.

Большинство конструкторов не выводят никакой информации, а лишь выполняют инициализацию объекта.

Прежде чем продолжить, еще раз рассмотрим операцию new. Как вы уже знаете, при распределении памяти для объекта используют следующую общую форму:

переменная\_класса = new имя\_класса();

Теперь вам должно быть ясно, почему после имени класса требуются круглые скобки. В действительности этот оператор вызывает конструктор класса. Таким образом, в строке:

Box mybox1 = new Box();

, операция new Box() вызывает конструктор Box(). Если конструктор класса не определен явно, Java создает для класса конструктор, который будет использоваться по умолчанию.

Именно поэтому приведенная строка кода работала в предшествующих версиях класса Box, в которых конструктор не был определен. Конструктор, используемый по умолчанию, инициализирует все переменные экземпляра нулевыми значениями. Зачастую конструктора, используемого по умолчанию, вполне достаточно для простых классов, чего обычно нельзя сказать о более сложных. Как только конструктор определен, конструктор, заданный по умолчанию, больше не используется.

## Конструкторы с параметрами

Хотя в предыдущем примере конструктор Box() инициализирует объект Box, он не особенно полезен – все параллелепипеды получают одинаковые размеры. Следовательно, необходим способ конструирования объектов Box с различными размерами. Простейшее решение этой задачи – добавление к конструктору параметров. Как легко догадаться, это делает конструктор значительно более полезным. Например, следующая версия класса Box определяет конструктор с параметрами, который устанавливает размеры параллелепипеда в соответствии со значениями этих параметров. Обратите особое внимание на способ создания объектов Box.

/\* В этой программе класс Box использует конструктор с параметрами

для инициализации размеров параллелепипеда.

\*/

class Box {

double width;

double height;

double depth;

// Это конструктор класса Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// вычисление и возврат объема

double volume() {

return width \* height \* depth;

}

}

class BoxDemo7 {

public static void main(String args[]) {

// объявление, распределение и инициализация объектов Box

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box(3, 6, 9);

double vol;

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем равен " + vol);

}

}

Вывод этой программы имеет следующий вид:

*Объем равен 3000.0*

*Объем равен 162.0*

Как видите, инициализация каждого объекта выполняется в соответствии со значениями, указанными в параметрах его конструктора. Например, в следующей строке:

Box mybox1 = new Box(10, 20, 15);

, значения 10, 20 и 15 передаются конструктору Box() при создании объекта с помощью операции new. Таким образом, копии переменных width, height и depth будут содержать соответственно значения 10, 20 и 15.

## Ключевое слово this

Иногда будет требоваться, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово this. Оно может использоваться внутри любого метода для ссылки на текущий объект. То есть this всегда служит ссылкой на объект, для которого был вызван метод. Ключевое слово this можно использовать везде, где допускается ссылка на объект типа текущего класса.

// Избыточное применение ключевого слова this.

Box(double w, double h, double d) {

this.width = w;

this.height = h;

this.depth = d;

}

Эта версия конструктора Box() действует точно так же, как предыдущая. Применение ключевого слова this избыточно, но совершенно правильно. Внутри метода Box() ключевое слово this всегда будет ссылаться на вызывающий объект. Хотя в данном случае это и излишне, в других случаях, один из которых рассмотрен в следующем разделе, ключевое слово this весьма полезно.

## Сокрытие переменной экземпляра

Как вы знаете, в Java не допускается объявление двух локальных переменных с одним и тем же именем в одной и той же или во включающих одна другую областях определения. Интересно отметить, что могут существовать локальные переменные, в том числе формальные параметры методов, которые перекрываются с именами переменных экземпляра класса. Однако когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная скрывает переменную экземпляра. Именно поэтому внутри класса Box переменные width, height и depth не были использованы в качестве имен параметров конструктора Box(). В противном случае переменная width ссылалась бы на формальный параметр, скрывая переменную экземпляра width. Хотя обычно проще использовать различные имена, существует и другой способ выхода из подобной ситуации. Поскольку ключевое слово this позволяет ссылаться непосредственно на объект, его можно применять для разрешения любых конфликтов пространства имен, которые могут возникать между переменными экземпляра и локальными переменными.

Например, ниже показана еще одна версия метода Box(), в которой имена width, height и depth использованы в качестве имен параметров, а ключевое слово this служит для обращения к переменным экземпляра по этим же именам.

// Этот код служит для разрешения конфликтов пространства имен.

Box(double width, double height, double depth) {

this.width = width;

this.height = height;

this.depth = depth;

}

Небольшое предостережение: иногда подобное применение ключевого слова this может приводить к недоразумениям, и некоторые программисты стараются не применять имена локальных переменных и параметров, скрывающие переменные экземпляров. Конечно, множество программистов придерживаются противоположного мнения и считают целесообразным в целях облегчения понимания программ использовать одни и те же имена, а для предотвращения скрытия переменных экземпляров применяют ключевое слово this. Выбор того или иного подхода – дело личного вкуса.

## Метод finalize()

Иногда при уничтожении объект должен будет выполнять какое-либо действие.

Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может требоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для подобных ситуаций Java предоставляет механизм, называемый финализацией. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора. Чтобы добавить в класс средство выполнения финализации, достаточно определить метод finalize(). Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри метода finalize() нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Сборщик мусора запускается периодически, проверяя наличие объектов, на которые отсутствуют ссылки как со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод finalize() по отношению к объекту.

Общая форма метода finalize() имеет следующий вид:

protected void finalize( )

{

// здесь должен находиться код финализации

}

В этой синтаксической конструкции ключевое слово protected – спецификатор, который предотвращает доступ к методу finalize() со стороны кода, определенного вне его класса.

Важно понимать, что метод finalize() вызывается только непосредственно перед сборкой мусора. Например, он не вызывается при выходе объекта за рамки области определения. Это означает, что неизвестно, когда будет – и, даже будет ли вообще – выполняться метод finalize(). Поэтому программа должна предоставлять другие средства освобождения используемых объектом системных ресурсов и тому подобного. Нормальная работа программы не должна зависеть от метода finalize().

## Перегрузка методов

Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют перегруженными, а процесс – перегрузкой методов. Перегрузка методов – один из способов поддержки полиморфизма в Java. Тем читателям, которые никогда не использовали язык, допускающий перегрузку методов, эта концепция вначале может показаться странной. Но, как вы вскоре убедитесь, перегрузка методов – одна из наиболее впечатляющих и полезных функциональных возможностей Java.

При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа не достаточно для различения двух версий метода. Когда Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове.

Перегрузка методов поддерживает полиморфизм, поскольку это один из способов реализации в Java концепции “один интерфейс, несколько методов”. Перегрузка ценна тем, что она позволяет обращаться к схожим методам по общему имени. Программисту нужно помнить только об общем выполняемом действии. Полиморфизм позволяет свести несколько имен к одному.

## Перегрузка конструкторов

Наряду с перегрузкой обычных методов можно также выполнять перегрузку методов конструкторов. Фактически перегруженные конструкторы станут нормой, а не исключением, для большинства классов, которые вам придется создавать для реальных программ.

Чтобы это утверждение было понятным, вернемся к классу Box, разработанному в предыдущей главе. Его последняя версия имеет следующий вид:

class Box {

double width;

double height;

double depth;

// Это конструктор класса Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// вычисление и возврат значения

double volume() {

return width \* height \* depth;

}

}

Как видите, конструктор Box() требует передачи трех параметров. Это означает, что все объявления объектов Box должны передавать конструктору Box() три аргумента.

Например, следующий оператор недопустим:

Box ob = new Box();

Поскольку конструктор Box() требует передачи трех аргументов, его вызов без аргументов – ошибка. Эта ситуация порождает три важных вопроса. Что если нужно было просто определить параллелепипед и его начальные размеры не имели значения (или не были известны)? Или, нужно иметь возможность инициализировать куб, указывая только один размер, который должен использоваться для всех трех измерений? При текущем определении класса Box все эти дополнительные возможности недоступны. К счастью, решение подобных проблем достаточно просто: достаточно перегрузить конструктор Box, чтобы он учитывал только что описанные ситуации. Ниже приведена программа, которая содержит усовершенствованную версию класса Box, выполняющую эту задачу.

/\* В этом примере класс Box определяет три конструктора для

инициализации размеров параллелепипеда различными способами.

\*/

class Box {

double width;

double height;

double depth;

// конструктор, используемый при указании всех измерений

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// конструктор, используемый, когда ни один из размеров не указан

Box() {

width = -1; // значение -1 используется для указания

height = -1; // неинициализированного

depth = -1; // параллелепипеда

}

// конструктор, используемый при создании куба

Box(double len) {

width = height = depth = len;

}

// вычисление и возврат объема

double volume() {

return width \* height \* depth;

}

}

class OverloadCons {

public static void main(String args[]) {

// создание параллелепипедов с применением различных конструкторов

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box();

Box mycube = new Box(7);

double vol;

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем mybox1 равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем mybox2 равен " + vol);

// получение объема куба

vol = mycube.volume();

System.out.println("Объем mycube равен " + vol);

}

}

Эта программа создает следующий вывод:

*Объем mybox1 равен 3000.0*

*Объем mybox2 равен -1.0*

*Объем mycube равен 343.0*

Как видите, соответствующий перегруженный конструктор вызывается в зависимости от параметров, указанных при выполнении операции new.

## Использование объектов в качестве параметров

До сих пор в качестве параметров методов мы использовали только простые типы. Однако передача методам объектов – и вполне допустима, и достаточно распространена.

Например, рассмотрим следующую короткую программу:

// Методам можно передавать объекты.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// возврат значения true, если параметр o равен вызывающему объекту

boolean equals(Test o) {

if(o.a == a && o.b == b) return true;

else return false;

}

}

class PassOb {

public static void main(String args[]) {

Test ob1 = new Test(100, 22);

Test ob2 = new Test(100, 22);

Test ob3 = new Test(-1, -1);

System.out.println("ob1 == ob2: " + ob1.equals(ob2));

System.out.println("ob1 == ob3: " + ob1.equals(ob3));

}

}

Эта программа создает следующий вывод:

*ob1 == ob2: true*

*ob1 == ob3: false*

Как видите, метод equals() внутри метода Test проверяет равенство двух объектов и возвращает результат этой проверки. То есть он сравнивает вызывающий объект с тем, который был ему передан. Если они содержат одинаковые значения, метод возвращает значение true. В противном случае он возвращает значение false. Обратите внимание, что параметр o в методе equals() указывает Test в качестве типа. Хотя Test – тип класса, созданный программой, он используется совершенно так же, как встроенные типы Java.

Одно из наиболее часто встречающихся применений объектов-параметров – в конструкторах. Часто приходится создавать новый объект так, чтобы вначале он не отличался от какого-то существующего объекта. Для этого потребуется определить конструктор, который в качестве параметра принимает объект его класса. Например, следующая версия класса Box позволяет выполнять инициализацию одного объекта другим:

// В этой версии Box допускает инициализацию одного объекта другим.

class Box {

double width;

double height;

double depth;

// Обратите внимание на этот конструктор. Он использует объект типа Box.

Box(Box ob) { // передача объекта конструктору

width = ob.width;

height = ob.height;

depth = ob.depth;

}

// конструктор, используемый при указании всех измерений

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// конструктор, используемый, если ни одно из изменений не указано

Box() {

width = -1; // значение -1 используется для указания

height = -1; // не инициализированного

depth = -1; // параллелепипеда

}

// конструктор, используемый при создании куба

Box(double len) {

width = height = depth = len;

}

// вычисление и возврат объема

double volume() {

return width \* height \* depth;

}

}

class OverloadCons2 {

public static void main(String args[]) {

// создание параллелепипедов с применением различных конструкторов

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box();

Box mycube = new Box(7);

Box myclone = new Box(mybox1); // создание копии объекта mybox1

double vol;

// получение объема первого параллелепипеда

vol = mybox1.volume();

System.out.println("Объем mybox1 равен " + vol);

// получение объема второго параллелепипеда

vol = mybox2.volume();

System.out.println("Объем mybox2 равен " + vol);

// получение объема куба

vol = mycube.volume();

System.out.println("Объем куба равен " + vol);

// получение объема клона

vol = myclone.volume();

System.out.println("Объем клона равен " + vol);

}

}

Как вы убедитесь, приступив к созданию собственных классов, чтобы объекты можно было конструировать удобным и эффективным образом, нужно располагать множеством форм конструкторов.

В общем случае существует два способа, которыми компьютерный язык может передавать аргументы подпрограмме. Первый способ – вызов по значению. При использовании этого подхода значение аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, выполненные в параметре подпрограммы, не влияют на аргумент. Второй способ передачи аргумента – вызов по ссылке. При использовании этого подхода параметру передается ссылка на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для обращения к реальному аргументу, указанному в вызове. Это означает, что изменения, выполненные в параметре, будут влиять на аргумент, использованный в вызове подпрограммы. Как вы убедитесь, в Java применяются оба подхода, в зависимости от передаваемых данных.

В Java элементарный тип передается методу по значению. Таким образом, все происходящее с параметром, который принимает аргумент, не оказывает влияния вне метода. При передаче объекта методу ситуация изменяется коренным образом, поскольку по существу объекты передаются посредством вызова по ссылке. Следует помнить, что при создании переменной типа класса создается лишь ссылка на объект. Таким образом, при передаче этой ссылки методу, принимающий ее параметр будет ссылаться на тот же объект, на который ссылается аргумент. По сути это означает, что объекты передаются методам посредством вызова по ссылке. Изменения объекта внутри метода влияют на объект, использованный в качестве аргумента. Например, рассмотрим такую программу:

// Объекты передаются по ссылке.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// передача объекта

void meth(Test o) {

o.a \*= 2;

o.b /= 2;

}

}

class CallByRef {

public static void main(String args[]) {

Test ob = new Test(15, 20);

System.out.println("ob.a и ob.b перед вызовом: " +

ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a и ob.b после вызова: " +

ob.a + " " + ob.b);

}

}

Эта программа генерирует следующий вывод:

*ob.a и ob.b перед вызовом: 15 20*

*ob.a и ob.b после вызова: 30 10*

Как видите, в данном случае действия внутри метода meth() влияют на объект, использованный в качестве аргумента. Интересно отметить, что когда ссылка на объект передается методу, сама ссылка передается посредством вызова по значению. Однако поскольку передаваемое значение ссылается на объект, копия этого значения все равно будет ссылаться на тот же объект, что и соответствующий аргумент.

Когда элементарный тип передается методу, это выполняется посредством вызова по значению. Объекты передаются неявно с помощью вызова по ссылке.

## Возврат объектов

Метод может возвращать любой тип данных, в том числе созданные типы классов.

Например, в следующей программе метод icrByTen() возвращает объект, в котором значение переменной a на 10 больше значения этой переменной в вызывающем объекте.

// Возвращение объекта.

class Test {

int a;

Test(int i) {

a = i;

}

Test incrByTen() {

Test temp = new Test(a+10);

return temp;

}

}

class RetOb {

public static void main(String args[]) {

Test ob1 = new Test(2);

Test ob2;

ob2 = ob1.incrByTen();

System.out.println("ob1.a: " + ob1.a);

System.out.println("ob2.a: " + ob2.a);

ob2 = ob2.incrByTen();

System.out.println("ob2.a после второго увеличения значения: "

+ ob2.a);

}

}

Эта программа генерирует следующий вывод:

*ob1.a: 2*

*ob2.a: 12*

*ob2.a после второго увеличения значения: 22*

Как видите, при каждом вызове метода incrByTen() программа создает новый объект и возвращает ссылку на него вызывающей процедуре.

Приведенная программа иллюстрирует еще один важный момент: поскольку все объекты распределяются динамически с помощью операции new, программисту не нужно беспокоиться о том, чтобы объект не вышел за пределы области определения, т.к. выполнение метода, в котором он был создан, прекращается. Объект будет существовать до тех пор, пока где-либо в программе будет существовать ссылка на него. При отсутствии какой-либо ссылки на него объект будет уничтожен во время следующей уборки мусора.

## Введение в управление доступом

Как вы уже знаете, инкапсуляция связывает данные с манипулирующим ими кодом. Однако инкапсуляция предоставляет еще один важный атрибут: управление доступом.

Посредством инкапсуляции можно управлять тем, какие части программы могут получать доступ к членам класса. Управление доступом позволяет предотвращать злоупотребления. Например, предоставляя доступ к данным только посредством четко определенного набора методов, можно предотвратить злоупотребление этими данными. Таким образом, если класс реализован правильно, он создает “черный ящик”, который можно использовать, но внутренний механизм которого защищен от повреждения. Однако представленные ранее классы не полностью соответствуют этой цели. Например, рассмотрим класс Stack, представленный в конце главы 6. Хотя методы push() и pop() действительно предоставляют управляемый интерфейс стека, этот интерфейс не обязателен для использования. То есть другая часть программы может обойти эти методы и обратиться к стеку непосредственно. Понятно, что в “плохих руках” эта возможность может приводить к проблемам. В этом разделе мы представим механизм, с помощью которого можно строго управлять доступом к различным членам класса. Способ доступа к члену класса определяется спецификатором доступа, который изменяет его объявление. В Java определен обширный набор спецификаторов доступа.

Некоторые аспекты управления доступом связаны главным образом с наследованием и пакетами.

Спецификаторами доступа Java являются public (общедоступный), private (приватный) и protected (защищенный). Java определяет также уровень доступа, предоставляемый по умолчанию. Спецификатор protected применяется только при использовании наследования.

Начнем с определения спецификаторов public и private. Когда член класса изменяется спецификатором доступа public, он становится доступным для любого другого кода. Когда член класса указан как private, он доступен только другим членам этого же класса. Теперь вам должно быть понятно, почему методу main() всегда предшествует спецификатор public. Этот метод вызывается кодом, расположенным вне данной программы – т.е. системой времени выполнения Java. При отсутствии спецификатора доступа по умолчанию член класса считается общедоступным внутри своего собственного пакета, но недоступным для кода, расположенного вне этого пакета. В уже разработанных нами классах все члены класса использовали режим доступа, определенный по умолчанию, который, по сути, является общедоступным. Однако, как правило, это не будет соответствовать реальным требованиям. Обычно будет требоваться ограничить доступ к членам данных класса – предлагая доступ только через методы.

Кроме того, в ряде случаев придется определять приватные методы класса.

Спецификатор доступа предшествует остальной спецификации типа члена. То есть оператор объявления члена должен начинаться со спецификатора доступа. Например:

public int i;

private double j;

private int myMethod(int a, char b) { // ...

Чтобы влияние использования общедоступного и приватного доступа было понятно, рассмотрим следующую программу:

/\* Эта программа демонстрирует различие между спецификаторами

public и private.

\*/

class Test {

int a; // доступ, определенный по умолчанию

public int b; // общедоступный доступ

private int c; // приватный доступ

// методы доступа к c

void setc(int i) { // установка значения переменной c

c = i;

}

int getc() { // получение значения переменной c

return c;

}

}

class AccessTest {

public static void main(String args[]) {

Test ob = new Test();

// Эти операторы правильны, a и b доступны непосредственно

ob.a = 10;

ob.b = 20;

// Этот оператор неверен и может вызвать ошибку

// ob.c = 100; // Ошибка!

// Доступ к объекту c должен осуществляться посредством методов его класса

ob.setc(100); // OK

System.out.println("a, b, и c: " + ob.a + " " + ob.b + " " + ob.getc());

}

}

Как видите, внутри класса Test использован метод доступа, заданный по умолчанию, что в данном примере равносильно указанию доступа public. Объект b явно указан как public. Объект c указан как приватный. Это означает, что он недоступен для кода, переделенного вне его класса. Поэтому внутри класса AccessTest объект c не может применяться непосредственно. Доступ к нему должен осуществляться посредством его общедоступных методов setc() и getc(). Удаление символа комментария из начала строки:

// ob.c = 100; // Ошибка!

, сделало бы компиляцию этой программы невозможной из-за нарушений правил доступа.

## Ключевое слово static

В некоторых случаях желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Обычно обращение к члену класса должно выполняться только в сочетании с объектом его класса. Однако можно создать член класса, который может использоваться самостоятельно, без ссылки на конкретный экземпляр. Чтобы создать такой член, в начало его объявления нужно поместить ключевое слово static. Когда член класса объявлен как static (статический), он доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть объявлены как методы, так и переменные. Наиболее распространенный пример статического члена – метод main(). Этот метод объявляют как static, поскольку он должен быть объявлен до создания любых объектов.

Переменные экземпляров, объявленные как static, по существу являются глобальными переменными. При объявлении объектов их класса программа не создает никаких копий переменной static. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

На методы, объявленные как static, накладывается ряд ограничений.

* Они могут вызывать только другие статические методы.
* Они должны осуществлять доступ только к статическим переменным.
* Они ни коим образом не могут ссылаться на члены типа this или super. (Ключевое слово super связано с наследованием и описывается в следующей главе.)

Если для инициализации переменных типа static нужно выполнить вычисления, можно объявить статический блок, который будет выполняться только один раз при первой загрузке класса. В следующем примере показан класс, который содержит статический метод, несколько статических переменных и статический блок инициализации:

// Демонстрация статических переменных, методов и блоков.

class UseStatic {

static int a = 3;

static int b;

static void meth(int x) {

System.out.println("x = " + x);

System.out.println("a = " + a);

System.out.println("b = " + b);

}

static {

System.out.println("Статический блок инициализирован.");

b = a \* 4;

}

public static void main(String args[]) {

meth(42);

}

}

Сразу после загрузки класса UseStatic программа выполняет все операторы static. Вначале значение a устанавливается равным 3, затем программа выполняет блок static, который выводит сообщение, а затем инициализирует переменную b значением a\*4, или 12. Затем программа вызывает метод main(), который обращается к методу meth(), передавая параметру x значение 42. Три оператора println() ссылаются на две статических переменные a и b на локальную переменную x.

Вывод этой программы имеет такой вид:

*Статический блок инициализирован.*

*x = 42*

*a = 3*

*b = 12*

За пределами класса, в котором они определены, статические методы и переменные могут использоваться независимо от какого-либо объекта. Для этого достаточно указать имя их класса, за которым должна следовать операция точки. Например, если метод типа static нужно вызвать извне его класса, это можно выполнить, используя следующую общую форму:

имя\_класса.метод( )

Здесь имя\_класса – имя класса, в котором объявлен метод тип static. Как видите, этот формат аналогичен применяемому для вызова нестатических методов через переменные объектных ссылок. Статическая переменная доступна аналогичным образом – посредством операции точки, следующей за именем класса. Так в Java реализованы управляемые версии глобальных методов и переменных.

## Ключевое слово final

Переменная может быть объявлена как final (окончательная). Это позволяет предотвратить изменение содержимого переменной. Это означает, что переменная типа final должна быть инициализирована во время ее объявления. Например:

final int FILE\_NEW = 1;

final int FILE\_OPEN = 2;

final int FILE\_SAVE = 3;

final int FILE\_SAVEAS = 4;

final int FILE\_QUIT = 5;

Теперь все последующие части программы могут пользоваться переменной FILE\_OPEN и прочими так, как если бы они были константами, без риска изменения их значений.

В практике программирования на Java принято идентификаторы всех переменных типа final записывать прописными буквами. Переменные, объявленные как final, не занимают отдельную область памяти для каждого экземпляра – т.е., по сути, они являются константами.

Ключевое слово final можно применять также к методам, но в этом случае его значение существенно отличается от применяемого к переменным. Это второе применение ключевого слова final описано в следующей главе, посвященной наследованию.

## Массивы как классы

Все массивы реализованы как объекты. В связи с этим существует специальный атрибут массива, который наверняка пригодится. В частности, размер массива – т.е. количество элементов, которые может содержать массив – хранится в его переменной экземпляра length. Все массивы обладают этой переменной, которая всегда будет содержать размер массива. Ниже приведен пример программы, которая демонстрирует это свойство.

// Эта программа демонстрирует член длины массива.

class Length {

public static void main(String args[]) {

int a1[] = new int[10];

int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};

int a3[] = {4, 3, 2, 1};

System.out.println("длина a1 равна " + a1.length);

System.out.println("длина a2 равна " + a2.length);

System.out.println("длина a3 равна " + a3.length);

}

}

Эта программа генерирует следующий вывод:

*длина a1 равна 10*

*длина a2 равна 8*

*длина a3 равна 4*

Как видите, программа отображает размер каждого массива. Имейте в виду, что значение переменной length никак не связано с количеством действительно используемых элементов. Оно отражает лишь то количество элементов, которое может содержать массив.

## Представление вложенных и внутренних классов

Java позволяет определять класс внутри другого класса. Такие классы называют вложенными классами. Область определения вложенного класса ограничена областью определения внешнего класса. Таким образом, если класс B определен внутри класса A, класс B не может существовать независимо от класса A. Вложенный класс имеет доступ к членам, в том числе приватным, класса, в который он вложен. Однако внешний класс не имеет доступ к членам вложенного класса. Вложенный класс, который объявлен непосредственно внутри области определения своего внешнего класса, является его членом.

Можно также объявлять вложенные классы, являющиеся локальными для блока.

Существует два типа вложенных классов: статические и нестатические. Статический вложенный класс – класс, к которому применен модификатор static. Поскольку он является статическим, он должен обращаться к своему внешнему классу посредством объекта. То есть он не может непосредственно ссылаться на члены своего внешнего класса. Из-за этого ограничения статические вложенные классы используются редко.

Наиболее важный тип вложенного класса ­–­­ внутренний класс. Внутренний класс – это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них так же, как это делают остальные нестатические члены внешнего класса.

Следующая программа иллюстрирует определение и использование внутреннего класса. Класс Outer содержит одну переменную экземпляра outer\_x, один метод экземпляра test() и определяет один внутренний класс Inner.

// Демонстрация использования внутреннего класса.

class Outer {

int outer\_x = 100;

void test() {

Inner inner = new Inner();

inner.display();

}

// это внутренний класс

class Inner {

void display() {

System.out.println("вывод: outer\_x = " + outer\_x);

}

}

}

class InnerClassDemo {

public static void main(String args[]) {

Outer outer = new Outer();

outer.test();

}

}

Это приложение генерирует следующий вывод:

*вывод: outer\_x = 100*

В этой программе внутренний класс Inner определен в области определения класса Outer. Поэтому любой кода в классе Inner может непосредственно обращаться к переменной outer\_x. Метод экземпляра display() определен внутри класса Inner. Этот метод отображает значение переменной outer\_x в стандартном выходном потоке. Метод main() экземпляра InnerClassDemo создает экземпляр класса Outer и вызывает его метод test(). Этот метод создает экземпляр класса Inner и вызывает метод display().

Важно понимать, что экземпляр класса Inner может быть создан только внутри области определения класса Outer. Компилятор Java генерирует сообщение об ошибке, если любой код вне класса Outer пытается инициализировать класс Inner. Однако экземпляр класса Inner можно создать снаружи класса Outer, уточняя имя внутреннего класса именем внешнего класса – например Outer.Inner.

Как уже было сказано, внутренний класс имеет доступ ко всем элементам своего внешнего класса, но не наоборот. Члены внутреннего класса известны только внутри области определения внутреннего класса и не могут быть использованы внешним классом.

Например:

// Компиляция этой программы будет не возможна.

class Outer {

int outer\_x = 100;

void test() {

Inner inner = new Inner();

inner.display();

}

// это внутренний класс

class Inner {

int y = 10; // y – локальная переменная класса Inner

void display() {

System.out.println("вывод: outer\_x = " + outer\_x);

}

}

void showy() {

System.out.println(y); // ошибка; здесь переменная y не известна!

}

}

class InnerClassDemo {

public static void main(String args[]) {

Outer outer = new Outer();

outer.test();

}

}

В этом примере переменная y объявлена как переменная экземпляра класса Inner.

Поэтому она не известна за пределами класса и не может использоваться методом showy().

Хотя мы уделили основное внимание внутренним классам, определенным в качестве членов внутри области определения внешнего класса, внутренние классы можно определять внутри области определения любого блока. Например, вложенный класс можно определить внутри блока, определенного методом, или даже внутри тела цикла for, как показано в следующем примере:

// Определение внутреннего класса внутри цикла for.

class Outer {

int outer\_x = 100;

void test() {

for(int i=0; i<10; i++) {

class Inner {

void display() {

System.out.println("вывод: outer\_x = " + outer\_x);

}

}

Inner inner = new Inner();

inner.display();

}

}

}

class InnerClassDemo {

public static void main(String args[]) {

Outer outer = new Outer();

outer.test();

}

}

Вывод, генерируемый этой версией программы, показан ниже.

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

*вывод: outer\_x = 100*

# Работа со строками в Java

Вероятно, String – наиболее часто используемый класс из библиотеки классов Java. Очевидная причина этого в том, что строки – исключительно важный элемент программирования.

Прежде всего, следует уяснить, что любая создаваемая строка в действительности представляет собой объект типа String. Даже строковые константы в действительности являются объектами String. Например, в операторе:

System.out.println("Это – также объект String");

, строка "Это – также объект String" – константа типа String.

Во-вторых, объекты типа String являются неизменными. После того как объект типа String создан, его содержимое не может изменяться. Хотя это может казаться серьезным ограничением, на самом деле это не так по двум причинам. Если нужно изменить строку, всегда можно создать новую строку, содержащую все изменения. В Java определен равноправный класс String, StringBuffer, допускающий изменение строк, что позволяет выполнять в Java все обычные манипуляции со строками. (Класс StringBuffer описан во второй части этой книги.)

Существует множество способов создания строк. Простейший из них – воспользоваться оператором вроде следующего:

String myString = "тестовая строка";

Как только объект String создан, его можно использовать во всех ситуациях, в которых допустимо использование строк. Например, следующий оператор отображает содержимое myString:

System.out.println(myString);

Для объектов типа String в Java определена одна операция: +. Она служит для объединения двух строк. Например, оператор:

String myString = "Мне" + " нравится " + "Java.";

Приводит к тому, что содержимым переменной myString становится строка "Мне нравится Java".

Следующая программа иллюстрирует описанные концепции:

// Демонстрация применения строк.

class StringDemo {

public static void main(String args[]) {

String strOb1 = "Первая строка";

String strOb2 = "Вторая строка";

String strOb3 = strOb1 + " и " + strOb2;

System.out.println(strOb1);

System.out.println(strOb2);

System.out.println(strOb3);

}

}

Эта программа создает следующий вывод:

*Первая строка*

*Вторая строка*

*Первая строка и вторая строка*

Класс String содержит несколько методов, которые можно использовать. Опишем некоторые из них. С помощью метода equals() можно проверять равенство двух строк.

Метод length() позволяет выяснить длину строки. Вызывая метод charAt(), можно получить символ с указанным индексом. Ниже приведены общие формы этих трех методов:

boolean equals(String объект)

int length( )

char charAt(int индекс)

Следующая программа демонстрирует применение этих методов:

// Демонстрация некоторых методов класса String.

class StringDemo2 {

public static void main(String args[]) {

String strOb1 = "Первая строка";

String strOb2 = "Вторая строка";

String strOb3 = strOb1;

System.out.println("Длина strOb1: " +

strOb1.length());

System.out.println("Символ с индексом 3 в strOb1: " +

strOb1.charAt(3));

if(strOb1.equals(strOb2))

System.out.println("strOb1 == strOb2");

else

System.out.println("strOb1 != strOb2");

if(strOb1.equals(strOb3))

System.out.println("strOb1 == strOb3");

else

System.out.println("strOb1 != strOb3");

}

}

Эта программа генерирует следующий вывод:

*Длина strOb1: 12*

*Символ с индексом 3 в strOb1: s*

*strOb1 != strOb2*

*strOb1 == strOb3*

Конечно, подобно тому, как могут существовать массивы любого другого типа объектов, могут существовать и массивы строк. Например:

// Демонстрация использования массивов объектов типа String.

class StringDemo3 {

public static void main(String args[]) {

String str[] = { "один", "два", "три" };

for(int i=0; i<str.length; i++)

System.out.println("str[" + i + "]: " +

str[i]);

}

}

Вывод этой программы имеет вид:

*str[0]: один*

*str[1]: два*

*str[2]: три*

## Использование аргументов командной строки

Иногда будет требоваться передать определенную информацию программе во время ее запуска. Для этого используют аргументы командной строки метода main(). Аргумент командной строки – это информация, которую во время запуска программы задают в командной строке непосредственно после ее имени. Доступ к аргументам командной строки внутри Java-программы не представляет сложности – они хранятся в виде строк в массиве String, переданного методу main(). Первый аргумент командной строки хранится в элементе массива args[0], второй – в элементе args[0] и т.д. Например, следующая программа отображает все аргументы командной строки, с которыми она вызывается.

// Отображение всех аргументов 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]);

}

}

Попытайтесь выполнить эту программу, введя следующую строку:

* java CommandLine this is a test 100 -1

В результате отобразится следующий вывод:

*args[0]: this*

*args[1]: is*

*args[2]: a*

*args[3]: test*

*args[4]: 100*

*args[5]: -1*

Все аргументы командной строки передаются как строки. Численные значения нужно вручную преобразовать в их внутренние представления.

## Аргументы переменной длины

В JDK 5 была добавлена новая функциональная возможность, которая упрощает создание методов, принимающих переменное количество аргументов. Эта функциональная возможность получила название varargs (сокращение термина variable-length arguments – аргументы переменной длины). Метод, который принимает переменное число аргументов, называют методом переменной арности, или просто методом varargs.

Ситуации, в которых методу нужно передавать переменное количество аргументов, встречаются не так уж редко. Например, метод, который открывает подключение к Internet, может принимать имя пользователя, пароль, имя файла, протокол и тому подобное, но применять значения, заданные по умолчанию, если какие-либо из этих сведений опущены. В этой ситуации было бы удобно передавать только те аргументы, для которых заданные по умолчанию значения не применимы. Еще один пример – метод printf(), входящий в состав библиотеки ввода-вывода Java.

// Использование массива для передачи методу переменного

// количества аргументов. Это старый стиль подхода

// к обработке аргументов переменной длины.

class PassArray {

static void vaTest(int v[]) {

System.out.print("Количество аргументов: " + v.length +

" Содержимое: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

// Обратите внимание на способ создания массива

// для хранения аргументов.

int n1[] = { 10 };

int n2[] = { 1, 2, 3 };

int n3[] = { };

vaTest(n1); // 1 аргумент

vaTest(n2); // 3 аргумента

vaTest(n3); // без аргументов

}

}

Эта программа создает следующий вывод:

*Количество аргументов: 1 Содержимое: 10*

*Количество аргументов: 3 Содержимое: 1 2 3*

*Количество аргументов: 0 Содержимое:*

В программе методу vaTest() аргументы передаются через массив v. Этот старый подход к обработке аргументов переменной длины позволяет методу vaTest() принимать любое число аргументов. Однако он требует, чтобы эти аргументы были вручную помещены в массив до вызова метода vaTest(). Создание массива при каждом вызове метода vaTest() не только трудоемкая, но и чреватая ошибками задача. Функциональная возможность использования методов varargs обеспечивает более простой и эффективный подход.

Для указания аргумента переменной длины используют три точки (...). Например, вот как метод vaTest() можно записать с использованием аргумента переменной длины:

static void vaTest(int ... v) {

Эта синтаксическая конструкция указывает компилятору, что метод vaTest() может вызываться с нулем или более аргументов. В результате v неявно объявляется как массив типа int[]. Таким образом, внутри метода vaTest() доступ к v осуществляется с использованием синтаксиса обычного массива. Предыдущая программа с применением метода vararg приобретает следующий вид:

// Демонстрация использования аргументов переменной длины.

class VarArgs {

// теперь vaTest() использует аргументы переменной длины.

static void vaTest(int ... v) {

System.out.print("Количество аргументов: " + v.length +

" Содержимое: ");

For(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

// Обратите внимание на возможные способы вызова

// vaTest() с переменным числом аргументов.

vaTest(10); // 1 аргумент

vaTest(1, 2, 3); // 3 аргумента

vaTest(); // без аргументов

}

}

Вывод этой программы совпадает с выводом исходной версии.

Наряду с параметром переменной длины массив может содержать “нормальные” параметры. Однако параметр переменной длины должен быть последним параметром, объявленным методом. Существует еще одно ограничение, о котором следует знать: метод должен содержать только одни параметр типа varargs.

Метод, который принимает аргумент переменной длины, можно перегружать.

// Параметры vararg и перегрузка.

class VarArgs3 {

static void vaTest(int ... v) {

System.out.print("vaTest(int ...): " +

"Количество аргументов: " + v.length +

" Содержимое: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

static void vaTest(boolean ... v) {

System.out.print("vaTest(boolean ...) " +

"Количество аргументов: " + v.length +

" Содержимое: ");

for(boolean x : v)

System.out.print(x + " ");

System.out.println();

}

static void vaTest(String msg, int ... v) {

System.out.print("vaTest(String, int ...): " +

msg + v.length +

" Содержимое: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

vaTest(1, 2, 3);

vaTest("Проверка: ", 10, 20);

vaTest(true, false, false);

}

}

Эта программа создает следующий вывод:

*vaTest(int ...): Количество аргументов: 3 Содержимое: 1 2 3*

*vaTest(String, int ...): Проверка: 2 Содержимое: 10 20*

*vaTest(boolean ...) Количество аргументов: 3 Содержимое: true false false*

## Параметры переменной длины и неопределенность

При перегрузке метода, принимающего аргумент переменной длины, могут случаться непредвиденные ошибки. Они связаны с неопределенностью, которая может возникать при вызове перегруженного метода с аргументом переменной длины. Например, рассмотрим следующую программу:

// Аргументы переменной длины, перегрузка и неопределенность.

//

// Эта программа содержит ошибку, и ее компиляция

// будет невозможна!

class VarArgs4 {

static void vaTest(int ... v) {

System.out.print("vaTest(int ...): " +

"Количество аргументов: " + v.length +

" Содержимое: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

static void vaTest(boolean ... v) {

System.out.print("vaTest(boolean ...) " +

"Количество аргументов: " + v.length +

" Содержимое: ");

for(boolean x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

vaTest(1, 2, 3); // OK

vaTest(true, false, false); // OK

vaTest(); // Ошибка: неопределенность!

}

}

В этой программе перегрузка метода vaTest() выполняется вполне корректно.

Однако ее компиляция будет невозможна из-за следующего вызова:

vaTest(); // Ошибка: неопределенность!

Поскольку параметр типа vararg может быть пустым, этот вызов может быть преобразован в обращение к vaTest(int ...) или к vaTest(boolean ...). Оба варианта допустимы. Поэтому вызов принципиально неоднозначен.

Рассмотрим еще один пример неопределенности. Следующие перегруженные версии метода vaTest() изначально неоднозначны, несмотря на то, что одна из них принимает обычный параметр:

static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...

Хотя списки параметров метода vaTest() различны, компилятор не имеет возможности разрешения вызова.

# Наследование и полиморфизм

Наследование – одно из фундаментальных понятий объектно-ориентированного программирования, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять свои уникальные характеристики. В терминологии Java наследуемый класс называют суперклассом. Наследующий класс носит название подкласса. Следовательно, подкласс – это специализированная версия суперкласса. Он наследует все переменные экземпляра и методы, определенные суперклассом, и добавляет собственные, уникальные элементы.

## Основы наследования

Чтобы наследовать класс, достаточно просто вставить определение одного класса в другой с использованием ключевого слова extends. В качестве иллюстрации рассмотрим короткий пример. Следующая программа создает суперкласс A и подкласс B. Обратите внимание на использование ключевого слова extends для создания подкласса класса A.

// Простой пример наследования.

// Создание суперкласса.

class A {

int i, j;

void showij() {

System.out.println("i и j: " + i + " " + j);

}

}

// Создание подкласса путем расширения класса A.

class B extends A {

int k;

void showk() {

System.out.println("k: " + k);

}

void sum() {

System.out.println("i+j+k: " + (i+j+k));

}

}

class SimpleInheritance {

public static void main(String args[]) {

A superOb = new A();

B subOb = new B();

// Суперкласс может использоваться самостоятельно.

superOb.i = 10;

superOb.j = 20;

System.out.println("Содержимое superOb: ");

superOb.showij();

System.out.println();

/\* Подкласс имеет доступ ко всем общедоступным членам

своего суперкласса. \*/

subOb.i = 7;

subOb.j = 8;

subOb.k = 9;

System.out.println("Содержимое subOb: ");

subOb.showij();

subOb.showk();

System.out.println();

System.out.println("Сумма i, j и k в subOb:");

subOb.sum();

}

}

Эта программа создает следующий вывод:

*Содержимое superOb:*

*i и j: 10 20*

*Содержимое subOb:*

*i и j: 7 8*

*k: 9*

*Сумма i, j и k в subOb:*

*i+j+k: 24*

Как видите, подкласс B включает в себя все члены своего суперкласса A. Именно поэтому subOb имеет доступ к переменным i и j и может вызывать метод showij(). Кроме того, внутри метода sum() возможна непосредственная ссылка на переменные i и j, как если бы они были частью класса B.

Несмотря на то что A – суперкласс класса B, он также является полностью независимым, самостоятельным классом. То, что класс является суперклассом подкласса, не означает невозможность его самостоятельного использования. Более того, подкласс может быть суперклассом другого подкласса.

Общая форма объявления класса, который наследует от суперкласса, следующая:

class имя\_подкласса extends имя\_суперкласса {

// тело класса

}

Для каждого создаваемого подкласса можно указывать только один суперкласс. Java не поддерживает наследование нескольких суперклассов в одном подклассе. Как было сказано, можно создать иерархию наследования, в которой подкласс становится суперклассом другого подкласса. Однако никакой класс не может быть собственным суперклассом.

## Доступ к членам и наследование

Хотя подкласс включает в себя все члены своего суперкласса, он не может получать доступ к тем членам суперкласса, которые объявлены как private. Например, рассмотрим следующую простую иерархию классов:

/\* В иерархии классов приватные члены остаются приватными

для своего класса.

Эта программа содержит ошибку, и ее компиляция

будет невозможна.

\*/

// Создание суперкласса.

class A {

int i; // общедоступная по умолчанию

private int j; // приватная для A

void setij(int x, int y) {

i = x;

j = y;

}

}

// Переменная j класса A в этом классе недоступна.

class B extends A {

int total;

void sum() {

total = i + j; // ОШИБКА, j в этом классе недоступна

}

}

class Access {

public static void main(String args[]) {

B subOb = new B();

subOb.setij(10, 12);

subOb.sum();

System.out.println("Сумма равна " + subOb.total);

}

}

Компиляция этой программы будет невозможна, поскольку ссылка на переменную j внутри метода sum() класса B приводит к нарушению правил доступа. Поскольку переменная j объявлена как private, она доступна только другим членам ее собственного класса. Подкласс не имеет к ней доступа.

Рассмотрим более реальный пример, который поможет проиллюстрировать возможности наследования. В нем мы расширим последнюю версию класса Box, разработанную в предыдущей главе, добавив в нее четвертый компонент, который назовем weight (вес).

Таким образом, новый класс будет содержать ширину, высоту, глубину и вес параллелепипеда.

// В этой программе наследование используется для расширения класса Box.

class Box {

double width;

double height;

double depth;

// конструирование клона объекта

Box(Box ob) { // передача объекта конструктору

width = ob.width;

height = ob.height;

depth = ob.depth;

}

// конструктор, используемый при указании всех измерений

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// конструктор, используемый, если ни одно из изменений не указано

Box() {

width = -1; // значение -1 используется для указания

height = -1; // неинициализированного

depth = -1; // параллелепипеда

}

// конструктор, используемый при создании куба

Box(double len) {

width = height = depth = len;

}

// вычисление и возврат объема

double volume() {

return width \* height \* depth;

}

}

// Расширение класса Box включением в него веса.

class BoxWeight extends Box {

double weight; // вес параллелепипеда

// конструктор BoxWeight

BoxWeight(double w, double h, double d, double m) {

width = w;

height = h;

depth = d;

weight = m;

}

}

class DemoBoxWeight {

public static void main(String args[]) {

BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);

BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);

double vol;

vol = mybox1.volume();

System.out.println("Объем mybox1 равен " + vol);

System.out.println("Вес mybox1 равен " + mybox1.weight);

System.out.println();

vol = mybox2.volume();

System.out.println("Объем mybox2 равен " + vol);

System.out.println("Вес mybox2 равен " + mybox2.weight);

}

}

Эта программа генерирует следующий вывод:

*Объем mybox1 равен 3000.0*

*Вес mybox1 равен 34.3*

*Объем mybox2 равен 24.0*

*Вес mybox2 равен 0.076*

Класс BoxWeight наследует все характеристики класса Box и добавляет к ним компонент weight. Классу BoxWeight не нужно воссоздавать все характеристики класса Box. Он может просто расширять Box в соответствии с конкретными целями.

Основное преимущество наследования состоит в том, что как только суперкласс, который определяет общие атрибуты набора объектов, создан, его можно использовать для создания любого числа более специализированных классов. Каждый подкласс может точно определять свою собственную классификацию. Например, следующий класс наследует характеристики класса Box и добавляет атрибут цвета.

// Этот код расширяет класс Box, включая в него атрибут цвета.

class ColorBox extends Box {

int color; // цвет параллелепипеда

ColorBox(double w, double h, double d, int c) {

width = w;

height = h;

depth = d;

color = c;

}

}

Помните, как только суперкласс, который определяет общие аспекты объекта, создан, он может наследоваться для создания специализированных классов. Каждый подкласс добавляет собственные уникальные атрибуты. В этом заключается сущность наследования.

## Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной суперкласса может быть присвоена ссылка на любой подкласс, производный от данного суперкласса. Этот аспект наследования будет весьма полезен во множестве ситуаций. Рассмотрим следующий пример:

class RefDemo {

public static void main(String args[]) {

BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);

Box plainbox = new Box();

double vol;

vol = weightbox.volume();

System.out.println("Объем weightbox равен " + vol);

System.out.println("Вес weightbox равен " +

weightbox.weight);

System.out.println();

// присваивание объекту BoxWeight ссылки на ссылку объекта Box

plainbox = weightbox;

vol = plainbox.volume(); // OK, метод volume() определен в Box

System.out.println("Объем plainbox равен " + vol);

BookNew\_JAVA-7.indb 195 02.06.2007 1:06:53

196 Часть I. Язык Java

/\* Следующий оператор ошибочен, поскольку plainbox

не определяет член weight. \*/

// System.out.println("Вес plainbox равен " + plainbox.weight);

}

}

В этом примере weightbox – ссылка на объекты BoxWeight, а painbox – ссылка на объекты Box. Поскольку BoxWeight – подкласс класса Box, ссылке painbox можно присваивать ссылку на объект weightbox.

Важно понимать, что доступные объекты определяются типом ссылочной переменной, а не типом объекта, на который она ссылается. То есть при присваивании ссылочной переменной суперкласса ссылки на объект подкласса доступ предоставляется только к указанным в ней частям объекта, определенного суперклассом. Именно поэтому объект plainbox не имеет доступа к переменной weight даже в том случае, когда он ссылается на объект BoxWeight. Если немного подумать, это становится понятным – суперклассу не известно, что именно подкласс добавляет в него. Поэтому последняя строка кода в предыдущем фрагменте оформлена в виде комментария. Ссылка объекта Box не имеет доступа к полю weight, поскольку оно не определено в классе Box.

## Использование ключевого слова super

В предшествующих примерах классы, производные от класса Box, были реализованы не столь эффективно и надежно, как могли бы. Например, конструктор BoxWeight явно инициализирует поля width, height и depth класса Box. Это не только ведет к дублированию кода суперкласса, что весьма неэффективно, но и предполагает наличие у подкласса доступа к этим членам. Однако в ряде случаев придется создавать суперкласс, подробности реализации которого доступны только для него самого (т.е. с приватными членами данных). В этом случае подкласс никак не сможет самостоятельно непосредственно обращаться или инициализировать эти переменные. Поскольку инкапсуляция – один из главных атрибутов ООП, не удивительно, что Java предлагает решение этой проблемы.

Во всех случаях, когда подклассу нужно сослаться на его непосредственный суперкласс, это можно выполнить с помощью ключевого слова super.

Ключевое слово super имеет две общих формы. Первую используют для вызова конструктора суперкласса, а вторую – для обращения к члену суперкласса, скрытому членом подкласса. Рассмотрим обе формы.

Подкласс может вызывать конструктор, определенный его суперклассом, с помощью следующей формы ключевого слова super:

super(список\_аргументов);

Список\_аргументов определяет любые аргументы, требуемые конструктору в суперклассе. Оператор super() всегда должен быть первым выполняемым внутри конструктора подкласса.

В качестве иллюстрации использования оператора super() рассмотрим следующую усовершенствованную версию класса BoxWeight():

// Теперь класс BoxWeight использует ключевое слово super

// для инициализации своих атрибутов объекта Box.

class BoxWeight extends Box {

double weight; // вес параллелепипеда

// инициализация переменных width, height и depth с помощью super()

BoxWeight(double w, double h, double d, double m) {

super(w, h, d); // вызов конструктора суперкласса

weight = m;

}

}

В этом примере метод BoxWeight вызывает super() с аргументами w, h и d. Это приводит к вызову конструктора Box(), который инициализирует width, height и depth, используя переданные ему значения этих параметров. Теперь класс BoxWeight не инициализирует эти значения самостоятельно. Ему нужно инициализировать только свое уникальное значение – weight. В результате при необходимости эти значения могут оставаться приватными значениями класса Box.

В приведенном примере метод super() был вызван с тремя аргументами. Поскольку конструкторы могут быть перегруженными, super() можно вызывать, используя любую форму, определенную суперклассом. Программа выполнит тот конструктор, который соответствует указанным аргументам.

Рассмотрим основные концепции применения конструктора super(). Когда подкласс вызывает конструктор super(), он вызывает конструктор своего непосредственного суперкласса. Таким образом, super() всегда ссылается на суперкласс, расположенный в иерархии непосредственно над вызывающим классом. Это положение справедливо даже в случае многоуровневой иерархии. Кроме того, оператор super() всегда должен быть первым оператором, выполняемым внутри конструктора подкласса. Вторая форма ключевого слова super действует подобно ключевому слову this, за исключением того, что она всегда ссылается на суперкласс подкласса, в котором она использована.

Вторая форма применения ключевого слова super наиболее подходит в тех ситуациях, когда имена членов подкласса скрывают члены суперкласса с такими же именами.

Рассмотрим следующую простую иерархию классов:

// Использование ключевого слова super для предотвращения скрытия имени.

class A {

int i;

}

// Создание подкласса посредством расширения класса A.

class B extends A {

int i; // эта переменная i скрывает переменную i в классе A

B(int a, int b) {

super.i = a; // i в классе A

i = b; // i в классе B

}

void show() {

System.out.println("i в суперклассе: " + super.i);

System.out.println("i в подклассе: " + i);

}

}

class UseSuper {

public static void main(String args[]) {

B subOb = new B(1, 2);

subOb.show();

}

}

Эта программа отображает следующее:

*i в суперклассе: 1*

*i в подклассе: 2*

Хотя переменная экземпляра i в классе B скрывает переменную i в классе A, ключевое слово super позволяет получить доступ к переменной i, определенной в суперклассе. Как вы увидите, ключевое слово super можно использовать также для вызова методов, которые скрываются подклассом.

## Создание многоуровневой иерархии

До сих пор мы использовали простые иерархии классов, которые состояли только из суперкласса и подкласса. Однако можно строить иерархии, которые содержат любое количество уровней наследования. Как уже отмечалось, вполне допустимо использовать подкласс в качестве суперкласса другого подкласса. Например, класс C может быть подклассом класса B, который, в свою очередь, является подклассом класса A. В подобных ситуациях каждый подкласс наследует все характеристики всех его суперклассов.

## Порядок вызова конструкторов

В каком порядке вызываются конструкторы классов, образующих иерархию, при ее создании? Например, какой конструктор вызывается раньше: A или B, если B – это подкласс, а A – суперкласс? В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса, и заканчивая подклассом. Более того, поскольку super() должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок остается неизменным, независимо от того, используется ли форма super().

Если конструктор super() не применяется, программа использует конструктор каждого суперкласса, заданный по умолчанию или не содержащий параметров. В следующей программе демонстрируется порядок выполнения конструкторов.

// Демонстрация порядка вызова конструкторов.

// Создание суперкласса.

class A {

A() {

System.out.println("Внутри конструктора A.");

}

}

// Создание подкласса посредством расширения класса A.

class B extends A {

B() {

System.out.println("Внутри конструктора B.");

}

}

// Создание еще одного подкласса посредством расширения класса B.

class C extends B {

C() {

System.out.println("Внутри конструктора C.");

}

}

class CallingCons {

public static void main(String args[]) {

C c = new C();

}

}

Эта программа генерирует следующий вывод:

*Внутри конструктора A*

*Внутри конструктора B*

*Внутри конструктора C*

Как видите, конструкторы вызываются в порядке наследования.

Если немного подумать, становится ясно, что выполнение конструкторов в порядке наследования имеет смысл. Поскольку суперкласс ничего не знает о своих подклассах, любая инициализация, которую он должен выполнить, полностью независима и, возможно, обязательна для выполнения любой инициализации, выполняемой подклассом. Поэтому она должна выполняться первой.

## Переопределение методов

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, говорят, что метод подкласса переопределяет метод суперкласса. Когда переопределенный метод вызывается из подкласса, он всегда будет ссылаться на версию этого метода, определенную подклассом. Версия метода, определенная суперклассом, будет сокрыта. Рассмотрим следующий пример:

// Переопределение метода.

class A {

int i, j;

A(int a, int b) {

i = a;

j = b;

}

// отображение i и j

void show() {

System.out.println("i и j: " + i + " " + j);

}

}

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b);

k = c;

}

// отображение k – этот метод переопределяет метод show() класса A

void show() {

System.out.println("k: " + k);

}

}

class Override {

public static void main(String args[]) {

B subOb = new B(1, 2, 3);

subOb.show(); // этот оператор вызывает метод show() класса B

}

}

Эта программа создает следующий вывод:

*k: 3*

Когда программа вызывает метод show() по отношению к объекту типа B, она использует версию этого метода, определенную внутри класса B. То есть версия метода show(), определенная внутри класса B, переопределяет версию, объявленную внутри класса A.

Если нужно получить доступ к версии переопределенного метода, определенного в суперклассе, это можно сделать с помощью ключевого слова super. Например, в следующей версии класса B версия метода show(), объявленная в суперклассе, вызывается внутри версии подкласса. Это позволяет отобразить все переменные экземпляров.

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b);

k = c;

}

void show() {

super.show(); // этот оператор вызывает метод show() класса A

System.out.println("k: " + k);

}

}

Подстановка этой версии класса A в предыдущую программу приведет к следующему выводу:

*i и j: 1 2*

*k: 3*

В этой версии super.show() вызывает версию метода show(), определенную в суперклассе. Переопределение метода выполняется только в том случае, если имена и сигнатуры типов двух методов идентичны. В противном случае два метода являются просто перегруженными. Например, рассмотрим измененную версию предыдущего примера.

// Методы с различающимися сигнатурами являются

// перегруженными, а не переопределенными.

class A {

int i, j;

A(int a, int b) {

i = a;

j = b;

}

// отображение i и j

void show() {

System.out.println("i и j: " + i + " " + j);

}

}

// Создание подкласса посредством расширения класса A.

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b);

k = c;

}

// перегрузка метода show()

void show(String msg) {

System.out.println(msg + k);

}

}

class Override {

public static void main(String args[]) {

B subOb = new B(1, 2, 3);

subOb.show("Это k: "); // вызов метода show() класса B

subOb.show(); // вызов метода show() класса A

}

}

Эта программа создает следующий вывод:

*Это k: 3*

*i и j: 1 2*

Версия метода show(), определенная в классе B, принимает строковый параметр. В результате ее сигнатура типа отличается от сигнатуры метода в классе B, который не принимает никаких параметров. Поэтому никакое переопределение (или сокрытие имени) не происходит. Вместо этого просто выполняется перегрузка версии метода show(), определенной в классе A, версией, определенной в классе B.

## Динамическая диспетчеризация методов

Хотя приведенные в предыдущем разделе примеры демонстрируют механизм переопределения методов, они не показывают всех возможностей. Действительно, если бы переопределение методов служило лишь для удобства работы с пространством имен, оно представляло бы только определенный теоретический интерес, но имело бы очень небольшое практическое значение. Однако это не так. Переопределение методов служит основой для одной из наиболее мощных концепций Java – динамической диспетчеризации методов. Динамическая диспетчеризация методов – механизм, посредством которого разрешение обращения к переопределенному методу осуществляется во время выполнения, а не во время компиляции.

Динамическая диспетчеризация методов важна потому, что именно с ее помощью Java реализует полиморфизм времени выполнения. Рассмотрение этой концепции начнем с повторной формулировки одного важного принципа: ссылочная переменная суперкласса может ссылаться на объект подкласса.

Система Java использует этот факт для разрешения обращений к переопределенным методам во время выполнения. Вот как это происходит. Когда вызов переопределенного метода реализуется посредством ссылки на суперкласс, Java выбирает нужную версию этого метода в зависимости от типа объекта ссылки в момент вызова. Таким образом, этот выбор осуществляется во время выполнения. При ссылке на различные типы объектов программа будет обращаться к различным версиям переопределенного метода. Иначе говоря, выбор для выполнения версии переопределенного метода осуществляется в зависимости от типа объекта ссылки (а не от типа ссылочной переменной). Следовательно, если суперкласс содержит метод, переопределяемый подклассом, то при наличии ссылки на различные типы объектов через ссылочную переменную суперкласса программа будет выполнять различные версии метода.

В следующем примере иллюстрируется динамическая диспетчеризация методов.

// Динамическая диспетчеризация методов

class A {

void callme() {

System.out.println("Внутри метода callme класса A");

}

}

class B extends A {

// переопределение метода callme()

void callme() {

System.out.println("Внутри метода callme класса B");

}

}

class C extends A {

// переопределение метода callme()

void callme() {

System.out.println("Внутри метода callme класса C");

}

}

class Dispatch {

public static void main(String args[]) {

A a = new A(); // объект типа A

B b = new B(); // объект типа B

C c = new C(); // объект типа C

A r; // получение ссылки типа A

r = a; // r ссылается на объект A

r.callme(); // вызов версии метода callme, определенной в A

r = b; // r ссылается на объект B

r.callme(); // вызов версии метода callme, определенной в B

r = c; // r ссылается на объект C

r.callme(); // вызов версии метода callme, определенной в C

}

}

Эта программа генерирует следующий вывод:

*Внутри метода callme класса A*

*Внутри метода callme класса B*

*Внутри метода callme класса C*

Эта программа создает один суперкласс A и два его подкласса: B и C. Подклассы B и C переопределяют метод callme(), объявленные в классе A. Внутри метода main() программа объявляет объекты типов A, B и C. Программа объявляет также ссылку типа A по имени r. Затем программа по очереди присваивает переменной r ссылку на каждый тип объекта и использует эту ссылку для вызова метода callme(). Как видно из вывода, выполняемая версия метода callme() определяется по типу объекта ссылки во время выполнения. Если бы выбор осуществлялся по типу ссылочной переменной, r, вывод отражал бы три обращения к методу callme() класса A.

## Использование переопределения методов

Как уже было сказано, переопределенные методы позволяют Java поддерживать полиморфизм времени выполнения. Большое значение полиморфизма для объектно-ориентированного программирования обусловлено следующей причиной: он позволяет общему классу указывать методы, которые станут общими для всех его производных классов, в то же время позволяя подклассам определять конкретные реализации некоторых или всех этих методов. Переопределенные методы – еще один используемый в Java способ реализации аспекта полиморфизма под названием “один интерфейс, множество методов”.

Одно из основных условий успешного применения полиморфизма – понимание того, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. В случае его правильного применения суперкласс предоставляет все элементы, которые подкласс может использовать непосредственно. Он определяет также те методы, которые производный класс должен реализовать самостоятельно. Это позволяет подклассу определять собственные методы при сохранении единообразия интерфейса. Таким образом, объединяя наследование и переопределенные методы, суперкласс может определять общую форму методов, которые будут использоваться всеми его подклассами.

Динамический, реализуемый во время выполнения полиморфизм – один из наиболее мощных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность существующих библиотек кода вызывать методы применительно к экземплярам новых классов без повторной компиляции при сохранении четкого абстрактного интерфейса – чрезвычайно мощное средство.

Рассмотрим более реальный пример использования переопределения методов.

Следующая программа создает суперкласс Figure, который хранит размеры двумерного объекта. Она определяет также метод area(), который вычисляет площадь объекта. Программа создает два класса, производных от класса Figure: Rectangle и Triangle. Каждый из этих подклассов переопределяет метод area(), чтобы он возвращал соответственно площадь четырехугольника и треугольника.

// Применение полиморфизма времени выполнения.

class Figure {

double dim1;

double dim2;

Figure(double a, double b) {

dim1 = a;

dim2 = b;

}

double area() {

System.out.println("Площадь фигуры не определена.");

return 0;

}

}

class Rectangle extends Figure {

Rectangle(double a, double b) {

super(a, b);

}

// переопределение метода area для четырехугольника

double area() {

System.out.println("В области четырехугольника.");

BookNew\_JAVA-7.indb 208 02.06.2007 1:06:55

Глава 8. Наследование 209

return dim1 \* dim2;

}

}

class Triangle extends Figure {

Triangle(double a, double b) {

super(a, b);

}

// переопределение метода area для прямоугольного треугольника

double area() {

System.out.println("В области треугольника.");

return dim1 \* dim2 / 2;

}

}

class FindAreas {

public static void main(String args[]) {

Figure f = new Figure(10, 10);

Rectangle r = new Rectangle(9, 5);

Triangle t = new Triangle(10, 8);

Figure figref;

figref = r;

System.out.println("Площадь равна " + figref.area());

figref = t;

System.out.println("Площадь равна " + figref.area());

figref = f;

System.out.println("Площадь равна " + figref.area());

}

}

Эта программа создает следующий вывод:

*В области четырехугольника.*

*Площадь равна 45*

*В области треугольника.*

*Площадь равна 40*

*Область фигуры не определена.*

*Площадь равна 0*

Двойственный механизм наследования и полиморфизма времени выполнения позволяет определить единый интерфейс, используемый несколькими различными, но сходными типами объектов. В данном случае, если объект является производным от Figure, его площадь можно вычислять, вызывая метод area(). Интерфейс выполнения этой операции остается неизменным, независимо от типа фигуры.

## Использование абстрактных классов

В ряде ситуаций нужно будет определять суперкласс, который объявляет структуру определенной абстракции без предоставления полной реализации каждого метода. То есть иногда придется создавать суперкласс, определяющий только обобщенную форму, которую будут совместно использовать все его подклассы, добавляя необходимые детали. Такой класс определяет сущность методов, которые должны реализовать подклассы.

Например, такая ситуация может возникать, когда суперкласс не в состоянии создать полноценную реализацию метода. Именно такая ситуация имела место в классе Figure в предыдущем примере. Определение метода area() – просто шаблон. Он не будет вычислять и отображать площадь объекта какого-либо типа.

Как вы убедитесь в процессе создания собственных библиотек классов, отсутствие полного определения метода в контексте суперкласса – не столь уж редкая ситуация.

Эту проблему можно решать двумя способами. Один из них, как было показано в предыдущем примере – просто вывод предупреждающего сообщения. Хотя этот подход и полезен в определенных ситуациях – например, при отладке – обычно он не годится. Могут существовать методы, которые должны быть переопределены подклассом, чтобы подкласс имел какой-либо смысл. Рассмотрим класс Triangle. Он лишен всякого смысла, если метод area() не определен. В этом случае необходим способ убедиться в том, что подкласс действительно переопределяет все необходимые методы. В Java для этого служит абстрактный метод.

Потребовать, чтобы определенные методы переопределялись подклассом, можно посредством указания модификатора типа abstract. Иногда такие методы называют относящимися к компетенции подкласса, поскольку в суперклассе для них никакой реализации не предусмотрено. Таким образом, подкласс должен переопределять эти методы – он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используют следующую общую форму:

abstract тип имя(список\_параметров);

Как видите, в этой форме тело метода отсутствует.

Любой класс, который содержит один или более абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно поместить ключевое слово abstract перед ключевым словом class в начале объявления класса. Абстрактный класс не может содержать какие-то объекты. То есть абстрактный класс не может быть непосредственно конкретизирован с помощью операции new. Такие объекты были бы бесполезны, поскольку абстрактный класс определен не полностью. Нельзя также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо также быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод, и класса, который реализует этот метод.

// Простой пример применения абстракции.

abstract class A {

abstract void callme();

// абстрактные классы все же могут содержать конкретные методы

void callmetoo() {

System.out.println("Это конкретный метод.");

}

}

class B extends A {

void callme() {

System.out.println("Реализация метода callme класса B.");

}

}

class AbstractDemo {

public static void main(String args[]) {

B b = new B();

b.callme();

b.callmetoo();

}

}

Обратите внимание, что в этой программе класс A не содержит объявлений какихлибо объектов. Как уже было сказано, конкретизация абстрактного класса невозможна.

И еще один нюанс: класс A реализует конкретный метод callmetoo(). Это вполне допустимо. Абстрактные классы могут содержать любое необходимое количество конкретных реализаций.

Хотя абстрактные классы не могут быть использованы для конкретизации объектов, их можно применять для создания ссылок на объекты, поскольку в Java полиморфизм времени выполнения реализован посредством ссылок на суперкласс. Поэтому должна существовать возможность создания ссылки на абстрактный класс, которая может использоваться для указания на объект подкласса. Применение этого свойства показано в следующем примере.

Используя абстрактный класс, можно усовершенствовать созданный ранее класс Figure. Поскольку понятие площади неприменимо к неопределенной двумерной фигуре, следующая версия программы объявляет метод area() внутри класса Figure как abstract. Конечно, это означает, что все классы, производные от Figure, должны переопределять метод area().

// Использование абстрактных методов и классов.

abstract class Figure {

double dim1;

double dim2;

Figure(double a, double b) {

dim1 = a;

dim2 = b;

}

// теперь метод area является абстрактным

abstract double area();

}

class Rectangle extends Figure {

Rectangle(double a, double b) {

super(a, b);

}

// переопределение метода area для четырехугольника

double area() {

System.out.println("В области четырехугольника.");

return dim1 \* dim2;

}

}

class Triangle extends Figure {

Triangle(double a, double b) {

super(a, b);

}

// переопределение метода area для четырехугольника

double area() {

System.out.println("В области треугольника.");

return dim1 \* dim2 / 2;

}

}

class AbstractAreas {

public static void main(String args[]) {

// Figure f = new Figure(10, 10); // теперь недопустимо

Rectangle r = new Rectangle(9, 5);

Triangle t = new Triangle(10, 8);

Figure figref; // этот оператор допустим, никакой объект не создается

BookNew\_JAVA-7.indb 211 02.06.2007 1:06:55

212 Часть I. Язык Java

figref = r;

System.out.println("Площадь равна " + figref.area());

figref = t;

System.out.println("Площадь равна " + figref.area());

}

}

Как видно из комментария внутри метода main(), объявление объектов типа Figure более недопустимо, поскольку теперь этот класс является абстрактным. И все подклассы класса Figure должны переопределять метод area(). Чтобы убедиться в этом, попытайтесь создать подкласс, который не переопределяет метод area(). Это приведет к ошибке времени компиляции.

Хотя создание объекта типа Figure недопустимо, можно создать ссылочную переменную типа Figure. Переменная figref объявлена как ссылка на Figure – т.е. ее можно использовать для ссылки на объект любого класса, производного от Figure. Как мы уже поясняли, разрешение переопределенных методов во время выполнения осуществляется путем ссылки на суперкласс.

## Использование ключевого слова final для предотвращения переопределения

Хотя переопределение методов – одно из наиболее мощных средств Java, в некоторых случаях его желательно избежать. Чтобы запретить переопределение метода, в начале его объявления необходимо указать ключевое слово final. Методы, объявленные как final, переопределяться не могут. Следующий фрагмент кода иллюстрирует это применениеключевого слова final.

class A {

final void meth() {

System.out.println("Это метод final.");

}

}

class B extends A {

void meth() { // ОШИБКА! Этот метод не может быть переопределен.

System.out.println("Не допускается!");

}

}

Поскольку метод meth() объявлен как final, он не может быть переопределен в классе B. Попытка выполнить это переопределение приведет к ошибке времени компиляции.

Иногда методы, объявленные как final, могут способствовать увеличению производительности программы. Компилятор вправе вставлять вызовы этих методов непосредственно в строку, поскольку он “знает”, что они не будут переопределены подклассом. Часто при вызове небольшого метода типа final компилятор Java может копировать байт-код подпрограммы непосредственно в строку скомпилированного кода вызывающего метода, тем самым снижая значительные накладные расходы системных ресурсов, связанные с вызовом метода. Помещение методов типа final в строку вызывающего кода – лишь потенциальная возможность. Обычно Java разрешает вызовы методов динамически, во время выполнения. Такой подход называют поздним связыванием. Однако поскольку методы типа final не могут переопределяться, обращение к такому методу может быть разрешено во время компиляции. Этот подход называют ранним связыванием.

## Использование ключевого слова final для предотвращения наследования

Иногда будет требоваться предотвратить наследование класса. Для этого в начале объявления класса необходимо поместить ключевое слово final. Объявление класса как final неявным образом объявляет все его методы также как final. Как легко догадаться, одновременное объявление класса как abstract и как final недопустимо, поскольку абстрактный класс принципиально является незавершенным и только его подклассы предоставляют полную реализацию методов.

Ниже приведен пример класса типа final.

final class A {

// ...

}

// Следующий класс недопустим.

class B extends A { // ОШИБКА! Класс A не может иметь подклассы.

// ...

}

Как видно из комментария, класс B не может наследовать от класса A, поскольку A объявлен как final.

# Класс Object

В Java определен один специальный класс – Object. Все остальные классы являются подклассами этого класса. То есть Object – суперкласс всех остальных классов. Это означает, что ссылочная переменная типа Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная типа Object может ссылаться также на любой массив. Класс Object определяет методы, которые доступны в любом объекте.

Методы getClass(), notify(), notifyAll() и wait() объявлены как final. Остальные методы можно переопределять. Обратите внимание на два метода: equals() и toString(). Метод equals() сравнивает содержимое двух объектов. Если объекты эквивалентны, он возвращает значение true, если нет – false. Точное определение равенства зависит от типа сравниваемых объектов. Метод toString() возвращает строку, которая содержит описание объекта, по отношению к которому он вызван. Кроме того, этот метод автоматически вызывается при выводе объекта с помощью метода println(). Многие классы переопределяют этот метод. Это позволяет им приспосабливать описание специально для создаваемых ими объектных типов.

* Object clone(): Создает новый объект, не отличающийся от клонируемого объекта
* boolean equals(Object object): Определяет, равен ли один объект другому
* void finalize(): Вызывается перед удалением неиспользуемого объекта
* Class getClass(): Получает класс объекта во время выполнения
* int hashCode(): Возвращает хеш-код, связанный с вызывающим объектом
* void notify(): Возобновляет выполнение потока, который ожидает вызывающего объекта
* void notifyAll(): Возобновляет выполнение всех потоков, которые ожидают вызывающего объекта
* String toString(): Возвращает строку, которая описывает объект
* void wait(), void wait(long milliseconds), void wait(long milliseconds  
  int nanoseconds): Ожидает другого потока выполнения

# Интерфейсы и пакеты

Пакеты – это контейнеры классов, которые используются для сохранения изолированности пространства имен класса. Например, пакет позволяет создать класс по имени List, который можно хранить в отдельном пакете, не беспокоясь о возможных конфликтах с другим классом List, хранящимся в каком-то другом месте. Пакеты хранятся в иерархической структуре и явно импортируются в определения новых классов.

В предшествующих главах было описано использование методов для определения интерфейса к данным класса. С помощью ключевого слова interface Java позволяет полностью абстрагировать интерфейс от его реализации. Используя это ключевое слово, можно указать набор методов, которые могут быть реализованы одним или более классов. В действительности сам по себе интерфейс не определяет никакой реализации. Хотя они подобны абстрактным классам, интерфейсы предоставляют дополнительную возможность: один класс может реализовать более одного интерфейса. И наоборот, класс может наследоваться только от одного суперкласса (абстрактного или не абстрактного).

## Пакеты

В предшествующих главах для всех примеров классов мы использовали имена из одного пространства имен. Это означает, что во избежание конфликта имен для каждого класса нужно было указывать уникальное имя. По истечении некоторого времени при отсутствии какого-либо способа управления пространством имен может возникнуть ситуация, когда выбор удобных описательных имен отдельных классов станет затруднительным. Кроме того, требуется также какой-нибудь способ обеспечения того, чтобы выбранное имя класса было достаточно уникальным и не конфликтовало с именами классов, выбранными другими программистами.

Пакет служит одновременно механизмом и присвоения имен, и управления видимостью. Внутри пакета можно определить классы, не доступные коду вне этого пакета. Можно также определить члены класса, которые видны только другим членам этого же пакета. Такой механизм позволяет классам располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

Создание пакета является простой задачей: достаточно включить команду package в качестве первого оператора исходного файла Java. Любые классы, объявленные внутри этого файла, будут принадлежать указанному пакету. Оператор package определяет пространство имен, в котором хранятся классы. Если оператор package опущен, имена классов помещаются в используемый по умолчанию пакет без имени.

Хотя для коротких примеров программ пакет, используемый по умолчанию, вполне подходит, он не годится для реальных приложений. В большинстве случаев для кода придется определять пакет.

Пакет задает имя пакета. Например, показанный ниже оператор создает пакет MyPackage:

package MyPackage;

Для хранения пакетов система Java использует каталоги файловой системы. Например, файлы .class любых классов, объявленных в качестве составной части пакета MyPackage, должны храниться в каталоге MyPackage. Помните, что регистр символов имеет значение, а имя каталога должно в точности совпадать имени пакета.

Один и тот же оператор package может присутствовать в более чем одном файле. Этот оператор просто указывает пакет, к которому принадлежат классы, определенные в данном файле. Он не препятствует тому, чтобы другие классы в других файлах были частью этого же пакета. Большинство пакетов, используемых в реальных программах, распределено по множеству файлов.

Java позволяет создавать иерархию пакетов. Для этого применяется символ точки. Оператор многоуровневого пакета имеет следующую общую форму:

package пакет1[.пакет2[.пакет3]];

Иерархия пакетов должна быть отражена в файловой системе среды разработки Java. Необходимо тщательно проверять правильность выбора имен пакетов. Имя пакета нельзя изменить, не изменяя имя каталога, в котором хранятся классы.

## Защита доступа

В предшествующих главах вы узнали о различных аспектах механизма управления доступом Java и его спецификаторах. Например, вы уже знаете, что доступ к приватному члену класса предоставляется только другим членам этого класса. Пакеты добавляют к управлению доступом еще одно измерение. Как вы вскоре убедитесь, Java предоставляет множество уровней защиты, обеспечивая очень точное управление видимостью переменных и методов внутри классов, подклассов и пакетов.

Классы и пакеты одновременно служат средствами инкапсуляции и хранилищем пространства имен и области определения переменных и методов. Пакеты играют роль контейнеров классов и других подчиненных пакетов. Классы служат контейнерами данных и кода. Класс – наименьшая единица абстракции Java. Вследствие взаимодействия между классами и пакетами Java определяет четыре категории видимости членов класса.

* Подклассы в одном пакете.
* Классы в одном пакете, не являющиеся подклассами.
* Подклассы в различных пакетах.
* Классы, которые не находятся в одном пакете и не являются подклассами.

Три спецификатора доступа: private, public и protected – предоставляют разнообразные способы создания множество уровней доступа, необходимых для этих категорий.

## Импорт пакетов

Ни один из основных классов Java не хранится в неименованном пакете, используемом по умолчанию. Все стандартные классы хранятся в каком-либо именованном пакете. Поскольку внутри пакетов классы должны быть полностью определены именем или именами их пакетов, длинное, разделенное точками имя пути пакета каждого используемого класса может оказаться слишком громоздким. Поэтому, чтобы определенные классы или весь пакет можно было сделать видимыми, в Java включен оператор import.

После того как класс импортирован, на него можно ссылаться непосредственно, используя только его имя. Оператор import служит только для удобства программистов и не является обязательным с технической точки зрения для создания завершенной Java-программы. Однако если в приложении придется ссылаться на несколько десятков классов, оператор import значительно уменьшит объем вводимого кода.

В исходном файле Java-программы операторы import должны следовать непосредственно за оператором package (если таковой имеется) перед любыми определениями классов. Оператор import имеет следующую общую форму:

import пакет1[.пакет2].(имя\_класса|\*);

В этой форме пакет1 – имя пакета верхнего уровня, пакет2 – имя подчиненного пакета внутри внешнего пакета, отделенное символом точки (.). Глубина вложенности пакетов практически не ограничена ничем, кроме файловой системы. И, наконец, имя\_класса может быть задано либо явно, либо с помощью символа звездочки (\*), который указывает компилятору Java о необходимости импорта всего пакета. Следующий фрагмент демонстрирует применение обеих форм оператора:

import java.util.Date;

import java.io.\*;

Использование формы с применением символа звездочки может привести к увеличению времени компиляции – особенно при импорте нескольких больших пакетов. Однако использование формы с применением звездочки никак не влияет на производительность системы времени выполнения или на размеры классов.

Все стандартные классы, поставляемые с системой Java, хранятся в пакете java. Основные функции языка хранятся в пакете java.lang внутри пакета java. Обычно каждый пакет или класс, который нужно использовать, приходится импортировать. Но поскольку система Java бесполезна без многих функций, определенных в пакете java.lang, компилятор неявно импортирует его для всех программ. Это эквивалентно присутствию следующей строки в каждой из программ:

import java.lang.\*;

При наличии в двух различных пакетах, импортируемых с применением формы со звездочкой, классов с одинаковыми именами компилятор никак на это не отреагирует, если только не будет предпринята попытка использования одного из этих классов. В этом случае возникнет ошибка времени компиляции, и имя класса придется указать явно, задавая его пакет.

Полностью определенное имя класса с указанием полной иерархии пакетов можно использовать везде, где можно допускается имя класса. Например, в следующем фрагменте кода присутствует оператор импорта:

import java.util.\*;

class MyDate extends Date {

}

Этот же пример без оператора import выглядит следующим образом:

class MyDate extends java.util.Date {

}

# Интерфейсы

Применение ключевого слова interface позволяет полностью абстрагировать интерфейс класса от его реализации. То есть с использованием ключевого слова interface можно задать действия, которые должен выполнять класс, но не то, как именно он должен это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменных экземпляров, а объявления их методов не содержат тела метода. На практике это означает, что можно объявлять интерфейсы, которые не делают никаких допущений относительно их реализации. Как только интерфейс определен, его может реализовать любое количество классов. Кроме того, один класс может реализовать любое число интерфейсов.

Чтобы реализовать интерфейс, класс должен создать полный набор методов, определенных интерфейсом. Однако каждый класс может определять нюансы своей реализации данного интерфейса. Ключевое слово interface позволяет в полной мере использовать концепцию полиморфизма под названием “один интерфейс, несколько методов”.

Интерфейсы предназначены для поддержки динамического разрешения методов во время выполнения. Обычно чтобы вызов метода мог выполняться из одного класса в другом, оба класса должны присутствовать во время компиляции, дабы компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование создает статическую и нерасширяемую среду обработки классов. В такой системе функциональные возможности неизбежно передаются по иерархии классов все выше и выше, в результате чего механизмы будут становиться доступными все большему числу подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. Поскольку иерархия интерфейсов не совпадает с иерархией классов, классы, никак не связанные между собой в иерархии классов, могут реализовать один и тот же интерфейс. Именно здесь возможности интерфейсов проявляются наиболее полно.

Интерфейсы добавляют большинство функциональных возможностей, требуемых многим приложениям, которым в обычных условиях в языках вроде C++ пришлось бы прибегать к использованию множественного наследования.

## Определение интерфейса

Определение интерфейса во многом подобно определению класса. Общая форма интерфейса имеет следующий вид:

доступ interface имя {

возвращаемый\_тип имя\_метода1(список\_параметров);

возвращаемый\_тип имя\_метода2(список\_параметров);

тип имя\_конечной\_переменной1 = значение;

тип имя\_конечной\_переменной2 = значение;

// ...

Возвращаемый\_тип имя\_методаN(список\_параметров);

тип имя\_конечной\_переменнойN = значение;

}

Если определение не содержит никакого спецификатора доступа, используется доступ по умолчанию, и интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс объявлен как public, он может быть использован любым другим кодом. В этом случае интерфейс должен быть единственным общедоступным интерфейсом, объявленным в файле, и имя файла должно совпадать с именем интерфейса. Имя – имя интерфейса, которым может быть любой допустимый идентификатор.

Обратите внимание, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует символ точки с запятой. По сути, они представляют собой абстрактные методы. Ни один из указанных внутри интерфейса методов не может обладать никакой заданной по умолчанию реализацией. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Переменные могут быть объявлены внутри объявлений интерфейсов. Они неявно объявляются как final и static – т.е. реализующий класс не может их изменять. Кроме того, они должны быть также инициализированы. Все методы и переменные неявно объявляются как public.

Ниже приведен пример определения интерфейса. В нем объявляется простой интерфейс, который содержит один метод callback(), принимающий единственный целочисленный параметр.

interface Callback {

void callback(int param);

}

## Реализация интерфейсов

Как только интерфейс определен, его может реализовать один или более классов. Чтобы реализовать интерфейс, в определение класса потребуется включить конструкцию implements, а затем создать методы, определенные интерфейсом. Общая форма класса, который содержит выражение implements, имеет следующий вид:

доступ class имя\_класса [extends суперкласс]

[implements интерфейс [,интерфейс...]] {

// тело\_класса

}

Если класс реализует более одного интерфейса, имена интерфейсов разделяются запятыми. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то один и тот же метод будет использоваться клиентами любого интерфейса. Методы, которые реализуют интерфейс, должны быть объявлены как public. Кроме того, сигнатура типа реализующего метода должна в точности совпадать с сигнатурой типа, указанной в определении interface.

Рассморим небольшой пример класса, который реализует приведенный ранее интерфейс Callback.

class Client implements Callback {

// Реализует интерфейс Callback

public void callback(int p) {

System.out.println("Метод callback, вызванный со значением " + p);

}

}

Обратите внимание, что метод callback() объявлен с использованием спецификатора доступа public. При реализации метода интерфейса он должен быть объявлен как public.

Вполне допустима и достаточно распространена ситуация, когда классы, которые реализуют интерфейсы, определяют собственные дополнительные члены. Например, следующая версия класса Client реализует метод callback() и добавляет метод nonInfaceMeth():

class Client implements Callback {

// Реализует интерфейс Callback

public void callback(int p) {

System.out.println("Метод callback, вызванный со значением " + p);

}

void nonIfaceMeth() {

System.out.println("Классы, которые реализуют интерфейсы" +

"могут определять также и другие члены.");

}

}

## Доступ к реализациям через ссылки на интерфейсы

Переменные можно объявлять как объектные ссылки, которые используют тип интерфейса, а не тип класса. Посредством такой переменной можно ссылаться на любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода с помощью одной из таких ссылок выбор нужной версии будет производиться в зависимости от конкретного экземпляра интерфейса, на который выполняется ссылка. Это – одна из главных особенностей интерфейсов. Поиск выполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позже, чем код, который вызывает методы по отношению к этим классам. Диспетчеризация кода может выполняться посредством интерфейса без необходимости наличия каких-либо сведений о “вызывающем”. Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса.

Поскольку в системе Java динамический поиск методов во время выполнения сопряжен со значительными накладными расходами по сравнению с обычным вызовом методов, в коде, для которого важна производительность, интерфейсы следует использовать только тогда, когда это действительно необходимо.

В следующем примере метод callback() вызывается через ссылочную переменную интерфейса:

class TestIface {

public static void main(String args[]) {

Callback c = new Client();

c.callback(42);

}

}

Эта программа создает следующий вывод:

*Метод callback, вызванный со значением 42*

Обратите внимание, что хотя переменная c объявлена с типом интерфейса Callback, ей был присвоен экземпляр класса Client. Хотя переменную c можно использовать для доступа к методу callback(), она не имеет доступа к каким-то другим членам класса Client. Ссылочная переменная интерфейса располагает сведениями только о тех методах, которые объявлены ее объявлением interface. Таким образом, переменная c не может применяться для доступа к методу nonIfaceMeth(), поскольку она объявлена классом Client, а не классом Callback.

Хотя приведенный пример формально показывает, как ссылочная переменная интерфейса может получать доступ к объекту реализации, он не демонстрирует полиморфные возможности такой ссылки. Чтобы продемонстрировать пример такого применения, вначале создадим вторую реализацию интерфейса Callback:

// Еще одна реализация интерфейса Callback.

class AnotherClient implements Callback {

// Реализация интерфейса Callback

BookNew\_JAVA-7.indb 226 02.06.2007 1:06:57

Глава 9. Пакеты и интерфейсы 227

public void callback(int p) {

System.out.println("Еще одна версия callback");

System.out.println("p в квадрате равно " + (p\*p));

}

}

Теперь проверим работу следующего класса:

class TestIface2 {

public static void main(String args[]) {

Callback c = new Client();

AnotherClient ob = new AnotherClient();

c.callback(42);

c = ob; // теперь c ссылается на объект AnotherClient

c.callback(42);

}

}

Эта программа создает следующий вывод:

*callback вызванный со значением 42*

*Еще одна версия callback*

*p в квадрате равно 1764*

Как видите, вызываемая версия метода callback() определяется типом объекта, на который переменная c ссылается во время выполнения. Представленный пример очень прост, поэтому вскоре мы приведем еще один, более реальный пример.

## Частичные реализации

Если класс содержит интерфейс, но не полностью реализует определенные им методы, он должен быть объявлен как abstract (абстрактный). Например:

abstract class Incomplete implements Callback {

int a, b;

void show() {

System.out.println(a + " " + b);

}

// ...

}

В этом примере класс Incomplete не реализует метод callback() и должен быть объявлен как абстрактный. Любой класс, который наследует Incomplete, должен реализовать метод callback() либо быть также объявленным как abstract.

## Вложенные интерфейсы

Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется интерфейсом-членом или вложенным интерфейсом. Вложенный интерфейс может быть объявлен как public, private или protected. Это отличает его от интерфейса верхнего уровня, который должен быть либо объявлен как public, либо, как уже было отмечено, должен использовать уровень доступа, заданный по умолчанию.

Когда вложенный интерфейс используется вне содержащей его области определения, он должен определяться именем класса или интерфейса, членом которого он является. То есть вне класса или интерфейса, в котором объявлен вложенный интерфейс, его имя должно быть полностью определено.

## Переменные в интерфейсах

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов посредством простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. При включении интерфейса в класс (т.е. при “реализации” интерфейса) имена всех этих переменных будут помещены в область констант. (Это аналогично использованию в программе C/C++ заголовочного файла для создания большого числа констант типа #define или объявлений const.)

Если интерфейс не содержит никаких методов, любой класс, который включает в себя такой интерфейс, в действительности ничего не реализует. Это равносильно тому, что класс импортировал бы постоянные поля в пространство имен класса в качестве переменных типа final. В следующем примере эта технология применяется для реализации автоматизированной “системы принятия решений”.

import java.util.Random;

interface SharedConstants {

int NO = 0;

int YES = 1;

int MAYBE = 2;

int LATER = 3;

int SOON = 4;

int NEVER = 5;

}

class Question implements SharedConstants {

Random rand = new Random();

int ask() {

int prob = (int) (100 \* rand.nextDouble());

if (prob < 30)

return NO; // 30%

else if (prob < 60)

return YES; // 30%

else if (prob < 75)

return LATER; // 15%

else if (prob < 98)

return SOON; // 13%

else

return NEVER; // 2%

}

}

class AskMe implements SharedConstants {

static void answer(int result) {

switch(result) {

case NO:

System.out.println("Нет");

break;

case YES:

System.out.println("Да");

break;

case MAYBE:

System.out.println("Возможно");

break;

case LATER:

System.out.println("Позднее");

break;

case SOON:

System.out.println("Вскоре");

break;

case NEVER:

System.out.println("Никогда");

break;

}

}

public static void main(String args[]) {

Question q = new Question();

answer(q.ask());

answer(q.ask());

answer(q.ask());

answer(q.ask());

}

}

Обратите внимание, что в этой программе использован один из стандартных Javaклассов – Random. Этот класс генерирует псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в требуемой программой форме. В этом примере применяется метод nextDouble(), который возвращает случайные числа в диапазоне от 0,0 до 1,0.

В приведенном примере программы два класса Question и AskMe реализуют интерфейс SharedConstants, в котором определены константы NO (Нет), YES (Да), MAYBE (Возможно), SOON (Вскоре), LATER (Позднее) и NEVER (Никогда). Код внутри каждого класса ссылается на эти константы так, как если бы каждый класс определял или наследовал их непосредственно.

## Возможность расширения интерфейсов

Ключевое слово extends позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов. Когда класс реализует интерфейс, который наследует другой интерфейс, он должен предоставлять реализации всех методов, определенных внутри цепочки наследования интерфейса.

# Аннотации (метаданные)

Начиная с JDK 5, в Java появилось новое мощное средство, которое позволяет встроить информацию поддержки в исходные файлы. Эта информация, называемая также аннотацией, не меняет действия программы. То есть аннотация сохраняет семантику программ неизменной. Однако эта информация может быть использована различными инструментальными средствами, как во время разработки, так и в период развертывания. Например, аннотация может обрабатываться генераторами исходного кода. Термин метаданные также используется для именования этого средства, но более описательный термин средства аннотирования программ применяется более широко.

## Основы аннотирования

Аннотации создаются посредством механизма, основанного на интерфейсе. Начнем с примера. Вот объявление аннотации под названием MyAnno:

// Простой тип аннотации.

@interface MyAnno {

String str();

int val();

}

Во-первых, обратите внимание на символ @, предшествующий ключевому слову interface. Это говорит компилятору, что объявлен тип аннотации. Далее отметим два метода-члена – str() и val(). Все аннотации состоят только из объявлений методов. Однако вы не определяете тел этих методов. Вместо этого их реализует Java. Более того, методы ведут себя в большей степени подобно полям, как вы вскоре убедитесь.

Аннотация не может включать слова extends. Однако все аннотации автоматически расширяют интерфейс Annotation. То есть Annotation является суперинтерфейсом для всех аннотаций. Он объявлен в пакете java.lang.annotation. В Annotation переопределены методы hashCode(), equals() и toString(), которые определены в Object.

В нем также специфицирован метод annotationType(),возвращающий объект Class, который представляет вызывающую аннотацию.

Объявив аннотацию, вы можете использовать ее для аннотирования объявления.

Любой тип объявления может иметь аннотацию, ассоциированную с ним. Например, аннотироваться могут классы, методы, поля, параметры и константы enum. Аннотированной может быть даже сама аннотация. Во всех случаях аннотация предшествует остальной части объявления.

Когда вы применяете аннотацию, то присваиваете значения ее членам. Например, ниже показан вариант применения MyAnno к методу:

// Аннотирование метода.

@MyAnno(str = "Пример аннотации", val = 100)

public static void myMeth() { // ...

Эта аннотация связана с методом myMeth(). Посмотрите внимательно на синтаксис аннотации. За именем аннотации, которому предшествует @, следует взятый в скобки список инициализаторов членов. Чтобы присвоить члену значение, значение присваивается имени члена. Таким образом, в этом примере строка “Пример аннотации” присваивается члену str MyAnno. Обратите внимание, что в этом присваивании никаких скобок за str не следует. Когда члену аннотации дается значение, используется только его имя. То есть члены аннотации в данном контексте выглядят как поля.

## Спецификация политики удержания

Прежде чем объяснять аннотации дальше, необходимо обсудить политики удержания аннотаций (annotation retention policies). Политика удержания определяет, в какой точке аннотация отбрасывается. Java определяет три таких политики, которые инкапсулированы в перечислении java.lang.annotation.RetentionPolicy. Это SOURCE, CLASS и RUNTIME.

Аннотации с политикой удержания SOURCE удерживаются только в исходном файле и отбрасываются при компиляции.

Аннотации с политикой удержания CLASS сохраняются в файле .class во время компиляции. Однако они недоступны JVM во время выполнения.

Аннотации с политикой удержания RUNTIME сохраняются в файле .class во время компиляции и остаются доступными JVM во время выполнения. То есть политика RUNTIME представляет аннотации наиболее высокой степени постоянства.

Политика удержания для аннотации задается с помощью одной из встроенных аннотаций Java: @Retention. Ее общая форма показана ниже:

@Retention(политика\_удержания)

Здесь политика\_удержания должна быть одной из описанных ранее констант. Если для аннотации не указано никакой политики сохранения, используется CLASS.

В следующей версии MyAnno с помощью @Retention указывается политика RUNTIME. То есть MyAnno будет доступна JVM во время выполнения программы.

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno {

String str();

int val();

}

## Получение аннотаций во время выполнения с использованием рефлексии

Хотя аннотации спроектированы в основном для использования инструментами разработки и развертывания, если они специфицируют политику удержания RUNTIME, то могут быть опрошены во время выполнения любой Java-программой за счет использования рефлексии. Рефлексия – это средство, позволяющее получить информацию о классе во время выполнения программы. Программный интерфейс (API) рефлексии содержится в пакете java.lang.reflect. Существует множество способов применения рефлексии, и мы не будем здесь обсуждать их все. Тем не менее, мы пройдемся по нескольким примерам, имеющим отношение к аннотациям.

Первый шаг в использовании рефлексии – это получение объекта Class, представляющего класс, чью аннотацию нужно получить. Class – это один из встроенных классов Java, определенный в пакете java.lang. Он детально рассматривается во второй части книги. Есть разные способы получения объекта Class. Один из простейших – вызвать метод getClass(),определенный в Object. Его общая форма показана ниже:

final Class getClass()

Он возвращает объект Class, который представляет вызывающий объект. После того, как вы получите объект Class, вы должны использовать его методы для получения информации о различных элементах, объявленных в классе, включая его аннотацию. Если вы хотите получить аннотации, ассоциированные с определенным элементом класса, вы должны сначала получить объект, представляющий этот элемент.

Например, Class представляет (помимо прочих) методы getMethod(), getField() и getConstructor(), которые получают информацию о методе, поле и конструкторе соответственно. Эти методы возвращают объекты типов Method, Field и Constructor.

Чтобы понять этот процесс, рассмотрим пример, который получает аннотации, ассоциированные с методом. Чтобы сделать это, вы сначала получаете объект Class, представляющий класс, затем вызываете метод getMethod() этого объекта, указав имя метода. getMethod() имеет следующую общую форму:

Method getMethod(String methName, Class ... paramTypes)

Имя метода передается в methName. Если метод принимает аргументы, то объекты Class, представляющие их типы, также должны быть указаны в paramTypes. Обратите внимание, что paramTypes – это список аргументов переменной длины (varargs). Это означает, что вы можете специфицировать столько типов параметров, сколько нужно, включая ноль. getMethod() возвращает объект Method, который представляет метод. Если метод не может быть найден, возбуждается исключение NoSuchMethodException.

От объектов Class, Method, Field или Constructor вы можете получить специфические аннотации, ассоциированные с этим объектом, обратившись к getAnnotation(). Его общая форма представлена ниже:

Annotation getAnnotation(Class annoType)

Здесь annoType – это объект Class, представляющий аннотацию, в которой вы заинтересованы. Метод возвращает ссылку на аннотацию. Используя эту ссылку, вы можете получить значения, ассоциированные с членами аннотации.

Метод возвращает null, если аннотация не найдена; это случай, когда аннотация не имеет @Retention, установленную в RUNTIME.

Ниже показана программа, которая собирает все описанные части вместе и использует рефлексию для отображения аннотации, ассоциированной с методом.

import java.lang.annotation.\*;

import java.lang.reflect.\*;

// Объявление типа аннотации.

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno {

String str();

int val();

}

class Meta {

// Аннотировать метод.

@MyAnno(str = "Пример аннотации", val = 100)

public static void myMeth() {

Meta ob = new Meta();

// Получить аннотацию из метода

// и отобразить значения членов.

try {

// Для начала получить Class,

// представляющий класс.

Class c = ob.getClass();

// Теперь получить объект Method,

// представляющий этот метод.

Method m = c.getMethod("myMeth");

// Далее получить аннотацию класса.

MyAnno anno = m.getAnnotation(MyAnno.class);

// Наконец, отобразить аннотацию.

System.out.println(anno.str() + " " + anno.val());

} catch (NoSuchMethodException exc) {

System.out.println("Метод не найден.");

}

}

public static void main(String args[]) {

myMeth();

}}

Вот как выглядит результат работы этой программы:

*Пример аннотации 100*

Эта программа использует рефлексию, как описано, чтобы получить и отобразить значения str и val в аннотации MyAnno, ассоциированной с myMeth() в классе Meta.

Есть две вещи, на которые следует обратить особое внимание. Первая – выражение MyAnno.class в строке:

MyAnno anno = m.getAnnotation(MyAnno.class);

Это выражение вычисляется как объект Class типа MyAnno – аннотация. Это называется литералом класса. Вы можете использовать выражения этого типа всякий раз, когда требуется объект Class известного класса. Например, следующий оператор служит для получения объекта Class для Meta:

Class c = Meta.class;

Конечно, такой подход работает, только когда вы знаете имя класса объекта заранее, что не всегда возможно. Вообще вы можете получать литерал класса для классов, интерфейсов, примитивных типов и массивов.

Второй интересный момент – это способ, которым значения, ассоциированные с str и val, получаются, когда они выводятся в следующей строке:

System.out.println(anno.str() + " " + anno.val());

Обратите внимание, что они вызываются с применением синтаксиса вызова методов. Тот же подход используется всякий раз, когда требуется получить член аннотации.

## Получение всех аннотаций

Вы можете получить сразу все аннотации, имеющие @Retention, равную RUNTIME, которые ассоциированы с позицией, вызвав getAnnotations() для этой позиции. Общая форма этого метода выглядит так:

Annotation[] getAnnotations()

Он возвращает массив аннотаций. getAnnotations() может быть вызван для объектов типа Class, Method, Constructor и Field.

Вот еще один пример с рефлексией, который показывает, как получить все аннотации, ассоциированные с классом и методом. Он объявляет две аннотации. Затем он использует их для аннотирования класса и метода.

// Показать все аннотации для класса и метода.

import java.lang.annotation.\*;

import java.lang.reflect.\*;

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno {

String str();

int val();

}

@Retention(RetentionPolicy.RUNTIME)

@interface What {

String description();

}

@What(description = "Аннотация тестового класса")

@MyAnno(str = "Meta2", val = 99)

class Meta2 {

@What(description = "Аннотация тестового метода")

@MyAnno(str = "Testing", val = 100)

public static void myMeth() {

Meta2 ob = new Meta2();

try {

Annotation annos[] = ob.getClass().getAnnotations();

// Отобразить все аннотации для Meta2.

System.out.println("Все аннотации для Meta2:");

BookNew\_JAVA-7.indb 303 02.06.2007 1:07:06

304 Часть I. Язык Java

for(Annotation a : annos)

System.out.println(a);

System.out.println();

// Отобразить все аннотации для myMeth.

Method m = ob.getClass( ).getMethod("myMeth");

annos = m.getAnnotations();

System.out.println("Все аннотации для myMeth:");

for(Annotation a : annos)

System.out.println(a);

} catch (NoSuchMethodException exc) {

System.out.println("Метод не найден.");

}

}

public static void main(String args[]) {

myMeth();

}}

Ниже показан результат работы этой программы:

*Все аннотации для Meta2:*

*@What(description=Аннотация тестового класса)*

*@MyAnno(str=Meta2, val=99)*

*Все аннотации для myMeth:*

*@What(description=Аннотация тестового метода)*

*@MyAnno(str=Testing, val=100)*

Эта программа использует getAnnotations() для получения массива всех аннотаций, ассоциированных с классом Meta2 и методом myMeth(). Как объяснялось, getAnnotations() возвращает массив объектов Annotation. Вспомните, что Annotation – это суперинтерфейс для всех интерфейсов аннотаций, и что он переопределяет toString() из класса Object. То есть когда выводится ссылка на Annotation, вызывается его метод toString() для генерации строки, описывающей аннотацию, что и демонстрирует предыдущий пример.

## Встроенные аннотации

В Java определено очень много встроенных аннотаций. Большинство из них специализированы, но семь имеют общее назначение. Четыре из них импортируются из java.lang.annotation: @Retention, @Documented, @Target и @Inherited. Три других – @Override, @Deprecated и @SuppressWarnings – включены в java.lang.

@Retention предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания, как было описано в настоящей главе.

@Documented – это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована. Он предназначен для использования только в качестве аннотации к объявлению аннотации.

@Target – аннотация, специфицирующая типы объявлений, к которым может быть применима аннотация. Предназначена для использования только в качестве аннотации к другим аннотациям. @Target принимает один аргумент, который должен быть константой из перечисления ElementType. Этот аргумент специфицирует типы объявлений, к которым может быть применена аннотация.

@Inherited – аннотация-маркер, которая может применяться в другом объявлении аннотации. Более того, она касается только тех аннотаций, что будут использоваться в объявлениях классов. @Inherited позволяет аннотации суперкласса быть унаследованной в подклассе. Таким образом, когда осуществляется запрос к подклассу на предмет специфической аннотации, то если у этой аннотации в подклассе нет, проверяется суперкласс. Если запрошенная аннотация присутствует у суперкласса и она аннотирована как @Inherited, то эта аннотация будет возвращена.

@Override – аннотация-маркер, которая может применяться только в методах. Метод, аннотированный как @Override, должен переопределять метод суперкласса. Если он этого не делает, в результате возникает ошибка времени компиляции. Она используется для обеспечения того, что метод суперкласса будет действительно переопределен, а не просто перегружен.

@Deprecated – аннотация-маркер. Она указывает, что объявление устарело и должно быть заменено более новой формой.

@SuppressWarnings – указывает, что одно или более предупреждений, которые могут быть выданы компилятором, следует подавить. Подавляемые предупреждения специфицируются именами в строковой форме. Эта аннотация может быть применима к объявлениям любого типа.

Существует некоторое количество ограничений, касающихся объявления аннотаций. Во-первых, одна аннотация не может наследовать другую. Во-вторых, все методы, объявленные в аннотации, должны не принимать параметров. Более того, они должны возвращать один из перечисленных ниже типов: примитивный тип, такой как int или double; объект типа String или Class; тип enum; тип другой аннотации; массив одного из предыдущих типов.

Аннотации не могут быть обобщенными. Другими словами, они не могут принимать параметры-типы. И, наконец, в методах аннотации не может быть указана конструкция throws.

# Обработка ошибок в Java

Исключение Java представляет собой объект, который описывает исключительную (то есть, ошибочную) ситуацию, возникающую в части программного кода. Когда такая исключительная ситуация возникает, создается объект, представляющий исключение, который возбуждается в методе, вызвавшем ошибку. Этот метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях, в некоторой точке исключение перехватывается и обрабатывается. Исключения могут генерироваться системой времени выполнения Java, либо они могут быть сгенерированы вручную вашим кодом. Исключения, которые возбуждает Java, имеют отношение к фундаментальным ошибкам, которые нарушают правила языка Java либо ограничения системы выполнения Java. Вручную сгенерированные исключения обычно применяются для того, чтобы сообщить о некоторых ошибочных ситуациях тому, кто вызвал данный метод.

Обработка исключений Java управляется пятью ключевыми словами: try, catch, throw, throws и finally. Если кратко, они работают следующим образом. Операторы программы, которые вы хотите отслеживать на предмет исключений, помещаются в блок try. Если исключение возникает в блоке try, оно возбуждается. Ваш код может перехватить исключение (используя catch) и обработать его некоторым осмысленным способом.

Сгенерированные системой исключения автоматически возбуждаются системой времени выполнения Java. Чтобы вручную возбудить исключение, используется ключевое слово throw. Любое исключение, которое возбуждается внутри метода, должно быть специфицировано в его интерфейсе ключевым словом throws. Любой код, который в обязательном порядке должен быть выполнен после завершения блока try, помещается в блок finally. Ниже показана общая форма блока обработки исключений.

try {

// блок кода, в котором отслеживаются ошибки

}

catch (Тип\_исключения\_1 exOb) {

// обработчик исключений типа ExceptionType1

}

catch (Тип\_исключения\_2 exOb) {

// обработчик исключений типа ExceptionType2

}

// ...

finally {

// блок кода, который должен быть выполнен после завершения блока try

}

Здесь Тип\_исключения – тип исключения, которое возникает. Остаток настоящей главы посвящен описанию применения этой программной структуры.

## Типы исключений

Все типы исключений являются подклассами встроенного класса Trowable. То есть Trowable расположен на вершине иерархии классов исключений. Немедленно под Trowable в ней находятся два подкласса, которые разделяют все исключения на две отдельные ветви. Одну ветвь возглавляет Exception. Этот класс используется для исключительных условий, которые пользовательская программа должна перехватывать. Это также класс, от которого вы будете наследовать свои подклассы при создании ваших собственных типов исключений. У класса Exception имеется важный подкласс по имени RuntimeException. Исключения этого типа автоматически определяются для программ, которые вы пишете, и включают такие вещи, как деление на ноль и ошибочная индексация массивов.

Другая ветвь начинается с класса Error, определяющего исключения, вызов которых не ожидается при нормальном выполнении программы. Исключения типа Error используются системой времени выполнения Java для обозначения ошибок, происходящих внутри самой окружения. Примером такой ошибки может служить переполнение стека. В этой главе не рассматриваются исключения типа Error, поскольку они обычно создаются в ответ на катастрофические сбои, которые не могут быть обработаны вашей программой.

## Необработанные исключения

Прежде чем вы узнаете, как обрабатывать исключения в своей программе, полезно будет посмотреть, что происходит, когда вы не обрабатываете их. Следующая небольшая программа представляет пример, который намеренно вызывает ошибку деления на ноль:

class Exc0 {

public static void main(String args[]) {

int d = 0;

int a = 42 / d;

}}

Когда система времени выполнения Java обнаруживает попытку деления на ноль, она конструирует новый объект исключения и затем возбуждает исключение. Это прерывает выполнение Exc0, поскольку как только исключение возбуждено, оно должно быть перехвачено обработчиком исключений, а тот должен немедленно с ним что-то сделать.

В данном примере мы не применили никакого собственного обработчика исключений, поэтому исключение перехватывается обработчиком по умолчанию, предоставленным системой времени выполнения Java. Любое исключение, которое не перехвачено вашей программой, в конечном итоге будет перехвачено и обработано этим обработчиком по умолчанию. Обработчик по умолчанию отображает строку, описывающую исключение, печатает трассировку стека от точки возникновения исключения и прерывает программу. Трассировка стека всегда покажет последовательность вызовов методов, которая привела к ошибке.

## Использование try и catch

Хотя обработчик исключений по умолчанию, который предоставляет система времени выполнения Java, удобен для отладки, обычно вы захотите обрабатывать исключения самостоятельно. Это дает два существенных преимущества. Во-первых, вы получаете возможность исправить ошибку. Во-вторых, предотвращается автоматическое прерывание выполнения программы. Большинство пользователей будут недовольны (и это как минимум), если ваша программа будет останавливаться и распечатывать трассировку стека всякий раз при возникновении ошибки. К счастью, предотвратить это достаточно просто.

Чтобы противостоять и обрабатывать ошибки времени выполнения, нужно просто поместить код, который вы хотите наблюдать, внутрь блока try. Непосредственно за блоком try следует включить конструкцию catch, которая специфицирует тип перехватываемого исключения. Чтобы проиллюстрировать, насколько это просто делается, в следующую программу включен блок try с конструкцией catch, который обрабатывает исключение ArithmeticException, генерируемое в результате попытки деления на ноль.

class Exc2 {

public static void main(String args[]) {

int d, a;

try { // Мониторинг блока кода.

d = 0;

a = 42 / d;

System.out.println("Это не будет напечатано.");

} catch (ArithmeticException e) { // перехват ошибки деления на ноль

System.out.println("Деление на ноль.");

}

System.out.println("После оператора catch.");

}}

Эта программа создает следующий вывод:

*Деление на ноль.*

*После оператора catch*.

Обратите внимание, что вызов println() внутри блока try никогда не будет выполняться. Как только исключение возбуждено, управление передается из блока try в блок catch. То есть строка “Это не будет напечатано” не отображается. После того, как блок catch будет выполнен, управление передается на строку программы, следующую за всем механизмом try/catch.

Операторы try и catch составляют единый узел. Область действия catch не распространяется на те операторы, которые идут перед оператором try. Оператор catch не может перехватить исключение, возбужденное другим оператором try (кроме случаев вложенных конструкций try, которые будут описаны ниже). Операторы, которые защищены блоком try, должны быть заключены в фигурные скобки (т.е. они должны находиться внутри блока). Вы не можете применить try к отдельному оператору программы.

Целью правильно построенных операторов catch является разрешение исключительных ситуаций и продолжение работы, как если бы ошибки вообще не случались.

Например, в следующей программе каждая итерация цикла for получает два случайных числа. Эти два числа делятся одно на другое, а результат используется для деления значения 12345. Окончательный результат помещается в a. Если какая-либо из операций деления вызывает ошибку деления на ноль, эта ошибка перехватывается, значение a устанавливается равным 0 и выполнение программы продолжается.

// Обработка исключения с продолжением работы.

import java.util.Random;

class HandleError {

BookNew\_JAVA-7.indb 238 02.06.2007 1:06:58

Глава 10. Обработка исключений 239

public static void main(String args[]) {

int a=0, b=0, c=0;

Random r = new Random();

for(int i=0; i<32000; i++) {

try {

b = r.nextInt();

c = r.nextInt();

a = 12345 / (b/c);

} catch (ArithmeticException e) {

System.out.println("Деление на ноль.");

a = 0; // присвоить ноль и продолжить работу

}

System.out.println("a: " + a);

}

}}

## Множественные операторы catch

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Чтобы справиться с такой ситуацией, вы можете специфицировать два или более операторов catch, каждый для перехвата своего типа исключений. Когда возбуждается исключение, каждый оператор catch проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется. После того, как выполнится один из операторов catch, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком try/catch. В следующем примере кода перехватываются два разных типа исключений.

// Демонстрация применения множественных операторов catch.

class MultiCatch {

public static void main(String args[]) {

try {

int a = args.length;

System.out.println("a = " + a);

BookNew\_JAVA-7.indb 239 02.06.2007 1:06:58

240 Часть I. Язык Java

int b = 42 / a;

int c[] = { 1 };

c[42] = 99;

} catch(ArithmeticException e) {

System.out.println("Деление на 0: " + e);

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Ошибка индекса массива: " + e);

}

System.out.println("После блока try/catch.");

}}

Эта программа вызовет исключение деления на ноль, если будет запущена без аргументов командной строки, поскольку в этом случае a будет равно 0. Она выполнит деление, если будут передан аргумент командной строки, устанавливающий a равным значению больше нуля. Но в этом случае будет сгенерировано исключение ArrayIndexOutOfBoundsException, так как длина массива целых чисел c равна 1, в то время как программа пытается присвоить значение элементу массива c[42].

Когда используются множественные операторы catch, важно помнить, что подклассы исключений должны следовать перед любыми их суперклассами. Это потому, что оператор catch, который использует суперкласс, будет перехватывать все исключения этого суперкласса плюс всех его подклассов. То есть подкласс исключения никогда не будет обработан, если вы попытаетесь его перехватить после его суперкласса. Более того, в Java недостижимый код является ошибкой. Например, рассмотрим следующую программу:

/\* Эта программа содержит ошибку.

Подкласс должен идти перед его суперклассом в

последовательности операторов catch. В противном случае

будет создан недоступный код, что приведет к ошибке при компиляции.

\*/

class SuperSubCatch {

public static void main(String args[]) {

try {

int a = 0;

int b = 42 / a;

} catch(Exception e) {

System.out.println("Общий перехват Exception.");

}

/\* Этот catch никогда не будет достигнут, потому что

ArithmeticException – это подкласс Exception. \*/

catch(ArithmeticException e) { // Ошибка – недостижимый код

System.out.println("Это никогда не выполнится.");

}

}}

Если вы попытаетесь скомпилировать эту программу, то получите сообщение об ошибке, говорящее о том, что второй оператор catch недостижим, потому что исключение уже перехвачено. Поскольку ArithmeticException – подкласс Exception, первый оператор catch обработает все ошибки, основанные на Exception, включая ArithmeticException. Это означает, что второй оператор catch не будет никогда выполнен. Чтобы исправить эту проблему, потребуется изменить порядок следования операторов catch.

## Вложенные операторы try

Операторы try могут быть вложенными. То есть оператор try может находиться внутри блока другого try. Всякий раз, когда управление попадает в блок try, контекст этого исключения заталкивается в стек. Если вложенный оператор try не имеет обработчика catch для определенного исключения, стек “раскручивается” и проверяются на соответствие обработчики catch следующего (внешнего) блока try. Это продолжается до тех пор, пока не будет найден подходящий оператор catch либо пока не будут проверены все уровни вложенных try. Если подходящий оператор catch не будет найден, то исключение обработает система времени выполнения Java. Ниже приведен пример, в котором используются вложенные операторы try.

// Пример вложенных операторов try.

class NestTry {

public static void main(String args[]) {

try {

int a = args.length;

/\* Если не указаны параметры командной строки,

следующий оператор сгенерирует

исключение деления на ноль. \*/

int b = 42 / a;

System.out.println("a = " + a);

try { // вложенный блок try

/\* Если используется один аргумент командной строки,

то исключение деления на ноль

будет сгенерировано следующим кодом. \*/

if(a==1) a = a/(a-a); // деление на ноль

/\* Если используется два аргумента командной строки,

то генерируется исключение выхода за пределы массива. \*/

if(a==2) {

int c[] = { 1 };

c[42] = 99; // генерируется исключение выхода за пределы массива

}

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Индекс за пределами массива: " + e);

}

} catch(ArithmeticException e) {

System.out.println("Деление на 0: " + e);

}

}}

## throw

До сих пор мы перехватывали только исключения, которые возбуждала система времени выполнения Java. Однако существует возможность возбуждать исключения из ваших программ явным образом, используя оператор throw. Его общая форма показана ниже:

throw экземпляр\_Throwable;

Здесь экземпляр\_Throwable должен быть объектом типа Throwable либо подклассом Throwable. Примитивные типы – такие, как int или char, как и классы, отличные от Throwable, например String и Object, не могут быть использованы для исключений.

Существуют два способа получить объект Throwable: с использованием параметра в операторе catch либо созданием объекта оператором new. Поток выполнения останавливается непосредственно после оператора throw – любые последующие операторы не выполняются. Обнаруживается ближайший закрытый блок try, имеющий оператор catch соответствующего исключению типа. Если соответствие найдено, управление передается этому оператору. Если же нет, проверяется следующий внешний блок try и т.д. Если не находится подходящего по типу оператора catch, то обработчик исключений по умолчанию прерывает программу и печатает трассировку стека.

Ниже приведен пример программы, создающей и возбуждающей исключение. Обработчик, который перехватывает его, повторно возбуждает его для внешнего обработчика.

// Демонстрация применения throw.

class ThrowDemo {

static void demoproc() {

try {

throw new NullPointerException("demo");

} catch(NullPointerException e) {

System.out.println("Перехвачено внутри demoproc.");

throw e; // повторно возбудить исключение

}

}

public static void main(String args[]) {

try {

demoproc();

} catch(NullPointerException e) {

System.out.println("Повторный перехват: " + e);

}

}}

Эта программа получает два шанса обработки одной и той же ошибки. Во-первых, main() устанавливает контекст исключения, затем вызывает demoproc(). Метод demoproc() устанавливает другой контекст обработки исключения и немедленно возбуждает новый экземпляр исключения NullPointerException, который перехватывается в следующей строке. Затем исключение возбуждается повторно. Ниже показан результирующий вывод.

*Перехвачено внутри demoproc.*

*Повторный перехват: java.lang.NullPointerException: demo*

Эта программа также демонстрирует, как создавать собственные объекты стандартных исключений Java. Обратите внимание на строку:

throw new NullPointerException("demo");

Здесь операция new используется для конструирования экземпляра NullPointerException. Многие их встроенных исключений времени выполнения Java имеют, по меньшей мере, два конструктора: один без параметров и один со строковым параметром.

Когда применяется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект используется в качестве аргумента print() или println(). Она также может быть получена вызовом метода getMessage(), который определен в Throwable.

## throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен специфицировать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Это делается добавлением к объявлению метода конструкции throws.

Конструкция throws перечисляет типы исключений, которые метод может возбуждать. Это необходимо для всех исключений, кроме имеющих тип Error, RuntimeException либо их подклассов. Все остальные исключения, которые может возбуждать метод, должны быть объявлены в конструкции throws. Если этого не сделать, получится ошибка во время компиляции.

Вот общая форма объявления метода, которая включает оператор throws:

тип имя\_метода(список\_параметров) throws список\_исключений

{

// тело метода

}

Здесь список\_исключений – это разделенный запятыми список исключений, которые метод может возбуждать.

## finally

Когда исключение возбуждено, выполнение метода направляется по нелинейному пути, изменяющему нормальный поток управления внутри метода. В зависимости от того, как закодирован метод, существует даже возможность преждевременного возврата управления. В некоторых методах это может служить причиной серьезных проблем. Например, если метод при входе открывает файл и закрывает его при выходе, вероятно, вы не захотите, чтобы выполнение кода, закрывающего файл, было пропущено из-за применения механизма обработки исключений. Ключевое слово finally предназначено для того, чтобы справиться с такой ситуацией. finally создает блок кода, который будет выполнен после завершения блока try/catch, но перед кодом, следующим за try/catch. Блок finally выполняется независимо от того, возбуждено исключение или нет. Если исключение возбуждено, блок finally выполняется, даже если ни один оператор catch этому исключению не соответствует.

В любой момент, когда метод собирается возвратить управление вызывающему коду изнутри блока try/catch – из-за необработанного исключения, или явным применением оператора return – блок finally будет выполнен перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были получены в начале метода и должны быть освобождены перед возвратом. Оператор finally необязателен. Однако каждый оператор try требует наличия, по крайней мере, одного оператора catch или finally. Ниже приведен пример программы, которая показывает три метода, возвращающих управление разными способами, но ни один из них не пропускает выполнения блока finally.

class FinallyDemo {

// Возбуждает исключение из метода.

static void procA() {

try {

System.out.println("внутри procA");

throw new RuntimeException("демо");

} finally {

System.out.println("блок finally procA");

}

}

// Возврат управления в блоке try.

static void procB() {

try {

System.out.println("внутри procB");

return;

} finally {

System.out.println("блок finally procB");

}

}

// Нормальное выполнение блока try.

static void procC() {

try {

System.out.println("внутри procC");

} finally {

System.out.println("блок finally procC");

}

}

public static void main(String args[]) {

try {

procA();

} catch (Exception e) {

System.out.println("Исключение перехвачено");

}

procB();

procC();

}}

В этом примере procA() преждевременно прерывает выполнение в блоке try, возбуждая исключение. Блок finally все равно выполняется. В procB() возврат управления осуществляется в блоке try оператором return. Блок finally выполняется перед возвратом из procB(). В procC() блок try выполняется нормально, без ошибок. Однако блок finally выполняется все равно.

## Встроенные исключения Java

Внутри стандартного пакета java.lang определено несколько классов исключений. Некоторые из них использовались в предшествующих примерах. Наиболее общие из этих исключений являются подклассами стандартного типа RuntimeException. Как уже объяснялось ранее, эти исключения не нужно включать в список throws метода – они называются непроверяемыми исключениями, поскольку компилятор не проверяет факт обработки или возбуждения методом таких исключений.

## Создание собственных подклассов исключений

Хотя встроенные исключения Java обрабатывают большинство частых ошибок, вероятно, вам потребуется создать ваши собственные типы исключений для обработки ситуаций, специфичных для ваших приложений. Это достаточно просто сделать: просто определите подкласс Exception (который, разумеется, является подклассом Throwable).

Ваши подклассы не обязаны реализовывать что-либо – важно само их присутствие в системе типов, которое позволит использовать их как исключения.

Класс Exception не определяет никаких собственных методов. Естественно, он наследует методы, представленные в Throwable. Таким образом, всем исключениям, включая те, что вы создадите сами, доступны методы, определенные в Throwable.

Exception определяет четыре конструктора. Два были добавлены в JDK 1.4 для поддержки цепочек исключений.

## Сцепленные исключения

Начиная с J2SE 1.4, в подсистему исключений было добавлено новое средство: сцепленные исключения (chained exceptions). Это средство позволяет ассоциировать с одним исключением другое исключение. Второе исключение описывает причину появления первого. Например, представьте ситуацию, когда метод возбуждает исключение ArithmeticException, поскольку была предпринята попытка деления на ноль. Однако реальная причина проблемы заключается в ошибке ввода-вывода, что приводит к неправильному делению. И хотя метод должен возбуждать ArithmeticException, так как произошла именно эта ошибка, вы можете также позволить вызывающему коду узнать о том, что в основе лежит ошибка ввода-вывода. Сцепленные исключения позволяют справиться с этой, а также с любой другой ситуацией, в которой присутствуют уровни исключений. Чтобы разрешить сцепленные исключения, были добавлены два конструктора и два метода к Throwable.

// Демонстрация сцепленных исключений.

class ChainExcDemo {

static void demoproc() {

// создать исключение

NullPointerException e = new NullPointerException("верхний уровень");

// добавить причину

e.initCause(new ArithmeticException("причина"));

throw e;

}

public static void main(String args[]) {

try {

demoproc();

} catch(NullPointerException e) {

BookNew\_JAVA-7.indb 250 02.06.2007 1:07:00

Глава 10. Обработка исключений 251

// отобразить исключение верхнего уровня

System.out.println("Перехвачено: " + e);

// отобразить исключение-причину

System.out.println("Исходная причина: " + e.getCause());

}

}}

Эта программа создает следующий вывод:

*Перехвачено: java.lang.NullPointerException: верхний уровень*

*Исходная причина: java.lang.ArithmeticException: причина*

В этом примере исключением верхнего уровня является NullPointerException. К нему добавлено исключение-причина – ArithmeticException. Когда исключение возбуждается из demoproc(), оно перехватывается в main(). Затем исключение верхнего уровня отображается, а за ним следует исключение, лежащее в основе, которое извлекается методом getCause().

Сцепленные исключения могут вкладываться на любую глубину. То есть исключениепричина может иметь собственную причину. Но имейте в виду, что слишком длинные цепочки сцепленных исключений, скорее всего, свидетельствуют о плохом дизайне.

Сцепленные исключения не являются вещью, совершенно необходимой в каждой программе. Однако в случаях, когда информация об исключении-причине таки нужна, они представляют собой элегантное решение.

# Потоки данных в Java

Как вы могли заметить, до сих пор в примерах программ было задействовано не так много операций ввода-вывода. Фактически помимо print() и println(), никаких методов ввода-вывода в общем-то и не применялось. Причина этого проста: большинство реальных Java-приложений не являются текст-ориентированными консольными программами. Вместо этого они являются графически-ориентированными программами, которые основаны на Abstract Window Toolkit (AWT) или Swing для взаимодействия с пользователем. Хотя текст-ориентированные программы великолепны в качестве учебных примеров, они не занимают сколько-нибудь значительную часть в мире реальных программ. К тому же поддержка консольного ввода-вывода в Java ограничена и не слишком удобна в использовании — даже для простейших программ. Текстовый консольный ввод-вывод не является важным при программировании на Java. Тем не менее, Java обеспечивает мощную и гибкую поддержку ввода-вывода, когда это касается файлов и сетей. Система ввода-вывода Java целостна и последовательна. Фактически, если однажды разобраться с ее базовыми принципами, все остальное для профессионала становится простым.

## Потоки

Java-программы создают потоки ввода-вывода. Поток (stream) — это абстракция, которая либо порождает, либо принимает информацию. Поток связан с физическим устройством с помощью системы ввода-вывода Java. Все потоки ведут себя на один манер, даже несмотря на то, что реальные физические устройства, к которым они подключены, отличаются друг от друга. Таким образом, одни и те же классы и методы ввода-вывода применимы к устройствам разного типа. Аналогично выходной поток может ссылаться на консоль, дисковый файл или сетевое подключение. Потоки — это ясный способ обращения с вводом-выводом без необходимости для вашего кода разбираться с разницей, например, между клавиатурой и сетью. Java реализует потоки внутри иерархии классов, определенных в пакете java.io.

## Байтовые и символьные потоки

Java определяет два типа потоков: байтовые и символьные. Байтовые потоки предоставляют удобные средства для управления вводом и выводом байтов. Байтовые потоки используются, например, при чтении и записи бинарных данных. Символьные потоки предлагают удобные возможности управления вводом и выводом символов. Они используют кодировку Unicode и, таким образом, могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.

## Классы байтовых потоков

Байтовые потоки определены в двух иерархиях классов. На вершине находятся абстрактные классы InputStream и OutputStream. Каждый из этих абстрактных классов имеет несколько реальных подклассов, которые управляют различиями между различными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти.

Абстрактные классы InputStream и OutputStream определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных – это read() и write(), которые, соответственно, читают и пишут байты данных. Оба метода объявлены как абстрактные внутри InputStream и OutputStream. В классах-наследниках они переопределяются.

## Классы символьных потоков

Символьные потоки также определены в двух иерархиях классов. На их вершине находятся два абстрактных класса: Reader и Writer. Эти абстрактные классы управляют потоками символов Unicode. В Java предусмотрено несколько конкретных подклассов для каждого из них.

Абстрактные классы Reader и Writer определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных — это read() и write(), которые, соответственно читают и пишут символьные данные. Эти методы переопределяются в потоковых классах-наследниках.

* BufferedInputStream Буферизированный входной поток.
* BufferedOutputStream Буферизированный выходной поток.
* ByteArrayInputStream Входной поток, читающий из массива байт.
* ByteArrayOutputStream Выходной поток, записывающий в массив байт.
* DataInputStream Входной поток, включающий методы для чтения стандартных типов
* данных Java.
* DataOutputStream Выходной поток, включающий методы для записи стандартных типов данных Java.
* FileInputStream Входной поток, читающий из файла.
* FileOutputStream Выходной поток, записывающий в файл.
* FilterInputStream Реализация InputStream.
* FilterOutputStream Реализация OutputStream.
* InputStream Абстрактный класс, описывающий поток ввода.
* ObjectInputStream Входной поток для объектов.
* ObjectOutputStream Выходной поток для объектов.
* OutputStream Абстрактный класс, описывающий поток вывода.
* PipedInputStream Входной канал (например, межпрограммный).
* PipedOutputStream Выходной канал.
* PrintStream Выходной поток, включающий print() и println().
* PushbackInputStream Входной поток, поддерживающий однобайтовый возврат во входной поток.
* RandomAccessFile Поддерживает файловый ввод-вывод с произвольным доступом.
* SequenceInputStream Входной поток, представляющий собой комбинацию двух и более входных потоков, которые читаются совместно — один после другого.
* BufferedReader Буферизованный входной символьный поток.
* BufferedWriter Буферизованный выходной символьный поток.
* CharArrayReader Входной поток, который читает из символьного массива.
* CharArrayWriter Выходной поток, который пишет в символьный массив.
* FileReader Входной поток, читающий файл.
* FileWriter Выходной поток, пишущий в файл.
* FilterReader Фильтрующий читатель.
* FilterWriter Фильтрующий писатель.
* InputStreamReader Входной поток, транслирующий байты в символы.
* LineNumberReader Входной поток, подсчитывающий строки.
* OutputStreamWriter Выходной поток, транслирующий байты в символы.
* PipedReader Входной канал.
* PipedWriter Выходной канал.
* PrintWriter Выходной поток, включающий print() и println().
* PushbackReader Входной поток, позволяющий возвращать символы обратно в поток.
* Reader Абстрактный класс, описывающий символьный ввод.
* StringReader Входной поток, читающий из строки.
* StringWriter Выходной поток, пишущий в строку.
* Writer Абстрактный класс, описывающий символьный вывод.

## Предопределенные потоки

Как вы знаете, все Java-программы автоматически импортируют пакет java.lang.

В этом пакете определен класс System, инкапсулирующий некоторые аспекты среды времени выполнения. Например, используя некоторые из его методов, можно получить текущее время и настройки различных параметров, ассоциированных с системой. System также содержит три предопределенных потоковых переменных: in, out и err. Эти переменные объявлены как public, static и final в классе System. Это значит, что они могут быть использованы любой другой частью вашей программы без обращения к специфическому объекту System.

* System.out ссылается на стандартный выходной поток. По умолчанию это консоль.
* System.in ссылается на стандартный входной поток, который также по умолчанию является консолью. System.err ссылается на стандартный поток ошибок, который также по умолчанию связан с консолью. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

System.in — это объект типа InputStream, System.out и System.err — объекты типа PrintStream. Это байтовые потоки, хотя обычно они используются для чтения и записи символов с консоли и на консоль.

## Чтение консольного ввода

В Java 1.0 единственным способом выполнения консольного ввода было использование байтового потока, и существует большой объем старого кода, в котором применяется этот подход. Сегодня применение байтового потока для чтения консольного ввода попрежнему технически возможно, но поступать так не рекомендуется. Предпочтительный метод чтения консольного ввода — это использовать символ-ориентированный поток, что значительно упрощает возможности интернационализации и поддержки разрабатываемых программ.

В Java консольный ввод выполняется чтением System.in. Чтобы получить символьный поток, присоединенный к консоли, вы должны поместить System.in в оболочку объекта BufferedReader. BufferedReader поддерживает буферизованный входной поток. Наиболее часто используемый его конструктор выглядит так:

BufferedReader(Reader inputReader)

Здесь inputReader — это поток, который связывается с создаваемым экземпляром BufferedReader. Reader — абстрактный класс. Одним из его конкретных наследников является InputStreamReader, который преобразует байты в символы. Для получения объекта InputStreamReader, который присоединен к System.in, служит следующий конструктор:

InputStreamReader(InputStream inputStream)

Поскольку System.in ссылается на объект типа InputStream, он должен быть использован как параметр inputStream. Собрав все вместе, получим следующую строку кода, которая создает BufferedReader, соединенный с клавиатурой:

BufferedReader br = new BufferedReader(new

InputStreamReader(System.in));

После выполнения этого оператора br представляет собой основанный на символах поток, подключенный к консоли через System.in.

Т.к. наша основная цель – изучение базового синтаксиса и ключевых понятий языка Java мы не будем подробно разбирать работу с System.in. Далее приведен простой пример работы с чтением строк с консоли:

// Чтение строк с консоли с применением BufferedReader.

import java.io.\*;

class BRReadLines {

public static void main(String args[]) throws IOException

{

// Создать BufferedReader с использованием System.in

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String str;

System.out.println("Вводите строки текста.");

BookNew\_JAVA-7.indb 316 02.06.2007 1:07:08

Глава 13. Ввод-вывод, аплеты и другие темы 317

System.out.println("Введите 'стоп' для завершения.");

do {

str = br.readLine();

System.out.println(str);

} while(!str.equals("стоп"));

}}

## Класс PrintWriter

Хотя применение System.out для вывода на консоль допустимо, он рекомендуется в основном для целей отладки или для примеров программ вроде тех, что приводятся в настоящей книге. Для реальных программ рекомендуемым способом записи на консоль при использовании Java является поток PrintWriter. PrintWriter — это один из классов, основанных на символах. Применение такого класса для консольного вывода упрощает интернационализацию ваших программ.

PrintWriter определяет несколько конструкторов. Один из тех, которые мы будем использовать, показан ниже:

PrintWriter(OutputStream outputStream, boolean flushOnNewline)

Здесь outputStream — объект типа OutputStream, а flushOnNewline управляет тем, будет ли Java сбрасывать буфер в выходной поток каждый раз при вызове метода

println(). Если flushOnNewline равно true, то происходит автоматический сброс буфера, если же false, то это автоматически не делается.

PrintWriter поддерживает методы print() и println() для всех типов, включая Object. То есть вы можете использовать эти методы таким же способом, как они применяются в System.out. Если аргумент не простого типа, то PrintWriter вызывает метод toString() и затем печатает результат.

Чтобы писать на консоль с помощью PrintWriter, специфицируйте System.out в качестве выходного потока и сбрасывайте поток после каждого символа новой строки.

Например, следующая строка кода создает PrintWriter, который подключен к консольному выводу:

PrintWriter pw = new PrintWriter(System.out, true);

Показанное ниже приложение иллюстрирует применение PrintWriter для управления консольным выводом:

// Демонстрация PrintWriter

import java.io.\*;

public class PrintWriterDemo {

public static void main(String args[]) {

PrintWriter pw = new PrintWriter(System.out, true);

pw.println("Это строка");

int i = -7;

pw.println(i);

double d = 4.5e-7;

pw.println(d);

}

}

Вывод этой программы будет выглядеть следующим образом:

*Это строка*

*-7*

*4.5E-7*

Помните, что нет ничего неправильного в применении System.out для простого текстового вывода на консоль, когда вы изучаете Java или занимаетесь отладкой своих программ. Однако PrintWriter обеспечит возможность простой интернационализации для реальных программ. Поскольку никаких выгод от использования PrintWriter в простых программах нет, мы продолжим пользоваться System.out для вывода на консоль.

## Чтение и запись файлов

Java предоставляет множество классов и методов, которые позволяют вам читать и записывать файлы. В Java все файлы байт-ориентированы, и Java предоставляет методы для чтения и записи байтов в файл. Однако Java позволяет также поместить байт-ориентированные файловые потоки в оболочки символ-ориентированных объектов. Такая техника описана во второй части книги. В настоящей же главе рассматриваются только основы файлового ввода-вывода.

Два из наиболее часто используемых потоковых класса — это FileInputStream и FileOutputStream, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, вы просто создаете объект одного из этих классов, указав имя файла в качестве аргумента конструктора. Хотя оба класса имеют и дополнительные переопределенные конструкторы, мы будем использовать только следующие из них:

FileInputStream(String fileName) throws FileNotFoundException

FileOutputStream(String fileName) throws FileNotFoundException

Здесь filename — имя файла, который вы хотите открыть. Когда вы создаете входной поток, то если файл не существовал, возбуждается исключение FileNotFoundException. Для выходных потоков, если файл не может быть создан, также возбуждается исключение FileNotFoundException. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

Когда вы завершаете работу с файлом, вы должны закрыть его вызовом метода close(). Этот метод определен и в FileInputStream и в FileOutputStream, как показано ниже:

void close() throws IOException

Чтобы читать файл, вы можете применять версию метода read(), который определен в FileInputStream. Та, что мы будем использовать, выглядит так:

int read() throws IOException

Всякий раз, когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. read() возвращает -1, когда достигнут конец файла. Метод может возбуждать исключение IOException.

В следующей программе read() используется для ввода и отображения содержимого текстового файла, имя которого специфицировано в аргументе командной строки. Обратите внимание на блок try/catch, обрабатывающий две ошибки, которые могут возникнуть при работе программы — когда указанный файл не найден, либо когда пользователь забыл указать имя файла. Вы можете применять тот же подход всякий раз при использовании аргументов командной строки. Другие возможные исключения ввода-вывода просто передаются в main(), что вполне приемлемо для такого простого примера.

Однако, работая с файлами, часто вы пожелаете обработать самостоятельно все исключения ввода-вывода.

/\* Отображение текстового файла.

Чтобы использовать эту программу, укажите

имя файла, который хотите просмотреть.

Например, чтобы просмотреть файл TEST.TXT,

используйте следующую командную строку:

java ShowFile TEST.TXT

\*/

import java.io.\*;

class ShowFile {

public static void main(String args[])

throws IOException

{

int i;

FileInputStream fin;

try {

fin = new FileInputStream(args[0]);

} catch(FileNotFoundException e) {

System.out.println("Файл не найден");

return;

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Использование: ShowFile Файл");

return;

}

BookNew\_JAVA-7.indb 320 02.06.2007 1:07:08

Глава 13. Ввод-вывод, аплеты и другие темы 321

// читать символы до получения символа EOF (конец файла)

do {

i = fin.read();

if(i != -1) System.out.print((char) i);

} while(i != -1);

fin.close();

}}

Для записи в файл вы будете использовать метод write(), определенный в FileOutputStream. Его простейшая форма выглядит так:

void write(int byteval) throws IOException

Этот метод пишет в файл байт, переданный в byteval. Хотя byteval объявлен как целочисленный, в файл записываются только его младшие восемь бит. Если при записи произойдет ошибка, возбуждается исключение IOException. В следующем примере write() используется для копирования текстового файла:

/\* Копирование текстового файла.

Для использования этой программы укажите

имена исходного и целевого файлов.

Например, чтобы скопировать файл FIRST.TXT в файл

SECOND.TXT, используйте следующую командную строку:

java CopyFile FIRST.TXT SECOND.TXT

\*/

import java.io.\*;

class CopyFile {

public static void main(String args[]) throws IOException

{

int i;

FileInputStream fin;

FileOutputStream fout;

try {

// открыть входной файл

try {

fin = new FileInputStream(args[0]);

} catch(FileNotFoundException e) {

System.out.println("Входной файл не найден");

return;

}

// открыть выходной файл

try {

fout = new FileOutputStream(args[1]);

} catch(FileNotFoundException e) {

System.out.println("Ошибка открытия выходного файла");

return;

}

} catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Использование: CopyFile Исходный Целевой");

return;

}

// Копировать файл

try {

do {

i = fin.read();

BookNew\_JAVA-7.indb 321 02.06.2007 1:07:09

322 Часть I. Язык Java

if(i != -1) fout.write(i);

} while(i != -1);

} catch(IOException e) {

System.out.println("Ошибка файла");

}

fin.close();

fout.close();

}}

Обратите внимание на способ обработки потенциальных ошибок ввода-вывода в этой программе. В отличие от некоторых других языков программирования, включая C и C++, которые используют коды ошибок для обнаружения файловых ошибок, Java применяет свой механизм исключений. Это не только делает управление файлами понятнее, но также позволяет Java просто отличать условие достижения конца файла от файловых ошибок во время ввода. В C/C++ многие функции ввода возвращают одно и то же значение, когда происходит ошибка и когда достигается конец файла.

# Generic типы: параметризованный или обобщенный код

После выхода в 1995 г. первоначальной версии 1.0 в язык Java было добавлено множество новых средств. Одним из наиболее значительных и влиятельных новшеств стали обобщения (generics). Во-первых, их появление означало добавление новых синтаксических элементов в язык. Во-вторых, они повлекли за собой изменения во многих классах и методах самого API ядра.

Вероятно, одно из средств Java, которое в наибольшей степени испытало влияние обобщений – это каркас коллекций (Collections Framework).

Коллекция – это группа объектов. Каркас коллекций определяет несколько классов, таких как списки и карты, которые управляют коллекциями. Классы коллекций всегда готовы работать с объектами любых типов. Выгода от добавления в язык обобщений состоит в том, что классы коллекций теперь могут использоваться с полным обеспечением безопасности типов. То есть, помимо предоставления мощного нового элемента языка, обобщения также значительно усовершенствовали существующие средства. Вот почему обобщения представляют собой столь значимое дополнение к Java.

По сути дела, обобщения — это параметризованные типы. Параметризованные типы важны, поскольку позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя обобщения, можно создать единственный класс, например, который будет автоматически работать с разными типами данных. Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются обобщениями, обобщенными классами или обобщенными методами.

Важно понимать, что Java всегда предлагала возможность создавать в определенной мере обобщенные классы, интерфейсы и методы, оперирующие ссылками на тип Object.

Поскольку Object — это суперкласс для всех остальных классов, ссылка на Object может обращаться к объекту любого типа. То есть в старом коде обобщенные классы, интерфейсы и методы использовали ссылки на Object для того, чтобы оперировать объектами различного типа. Проблема была в том, что они не могли обеспечить безопасность типов.

Обобщения добавили в язык безопасность типов, которой так не хватало. Они также упростили процесс, поскольку теперь нет необходимости применять явные приведения для транслирования объектов Object в реальные типы данных, с которыми выполняются действия. Благодаря обобщениям, все приведения выполняются автоматически и неявно.

То есть обобщения расширили ваши возможности повторного использования кода и позволили вам делать это легко и безопасно.

Предупреждение для программистов C++: хотя обобщения похожи на шаблоны в C++, это не одно и то же. Существует ряд фундаментальных отличий между двумя подходами к обобщенным типам. Если у вас имеется опыт применения C++, важно не делать поспешных выводов о том, как обобщения работают в Java.

## Простой пример обобщения

Давайте начнем с простого примера обобщенного класса. В следующей программе определены два класса. Первый — это обобщенный класс Gen, а второй — GenDemo, использующий Gen.

// Простой обобщенный класс.

// Здесь T — это параметр типа,

// который будет заменен реальным типом

// при создании объекта типа Gen.

class Gen<T> {

T ob; // объявление объекта типа T

// Передать конструктору ссылку

// на объект типа T.

Gen(T o) {

ob = o;

}

BookNew\_JAVA-7.indb 340 02.06.2007 1:07:11

Глава 14. Обобщения 341

// Вернуть ob.

T getob() {

return ob;

}

// Показать тип T.

void showType() {

System.out.println("Типом T является " + ob.getClass().getName());

}

}

// Демонстрация обобщенного класса.

class GenDemo {

public static void main(String args[]) {

// Создать Gen-ссылку для Integers.

Gen<Integer> iOb;

// Создать объект Gen<Integer> и присвоить

// ссылку на iOb. Отметьте применение автоупаковки

// для инкапсуляции значения 88 в объект Integer.

iOb = new Gen<Integer>(88);

// Показать тип данных, используемый iOb.

iOb.showType();

// Получить значение iOb. Обратите внимание,

// что никакого приведения не нужно.

int v = iOb.getob();

System.out.println("значение: " + v);

System.out.println();

// Создать объект Gen для String.

Gen<String> strOb = new Gen<String>("Обобщенный тест");

// Показать тип данных, используемый strOb.

strOb.showType();

// Получить значение strOb. Опять же

// приведение не требуется.

String str = strOb.getob();

System.out.println("Значение: " + str);

}

}

Результат работы этой программы:

*Типом T является java.lang.Integer*

*Значение: 88*

*Типом T является java.lang.String*

*Значение: Обобщенный тест*

Давайте внимательно исследуем эту программу.

Во-первых, обратите внимание на объявление Gen в следующей строке:

class Gen<T> {

Здесь T — имя типа-параметра. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного Gen при создании реальных типов.

То есть T применяется в Gen всякий раз, когда требуется тип-параметр. Обратите внимание, что T заключен в <>. Этот синтаксис может быть обобщен. Всякий раз, когда объявляется тип-параметр, он указывается в угловых скобках. Поскольку Gen применяет тип-параметр, Gen является обобщенным классом, который называется также параметризованным типом.

Далее T используется для объявления объекта по имени ob, как показано ниже:

T ob; // объявляет объект типа T

Как объяснялось, T — это место подстановки реального типа, который будет указан при создании объекта Gen. То есть ob будет объектом типа, переданного в T. Например, если T передан тип String, то экземпляр ob будет типа String.

Теперь рассмотрим конструктор Gen:

Gen(T o) {

ob = o;

}

Как видите, параметр o имеет тип T. Это значит, что реальный тип o определяется типом, переданным T при создании объекта Gen. К тому же, поскольку и переменнаяпараметр o, и переменная-член ob имеют тип T, они обе получают одинаковый реальный тип при создании Gen.

Параметр типа T также может быть использован для спецификации типа возврата для метода, как в случае метода getob(), показанного здесь:

T getob() {

return ob;

}

Так как ob тоже имеет тип T, его тип совместим с типом, возвращаемым getob().

Метод showType() отображает тип T вызовом getName() на объекте Class, возвращенным вызовом getClass() на ob. Метод getClass() определен в Object, и потому является членом всех классов. Он возвращает объект Class, соответствующий типу класса объекта, на котором он вызван. Class определяет метод getName(), который возвращает строковое представление имени класса.

Класс GenDemo демонстрирует обобщенный класс Gen. Сначала он создает версию Gen для целых, как показано ниже:

Gen<Integer> iOb;

Посмотрим на это объявление внимательней. Во-первых, отметим, что тип Integer специфицирован в угловых скобках после Gen. В этом случае Integer — это тип-аргумент, который передается в параметре типа Gen, T. Это эффективно создает версию Gen, в которой все ссылки на T транслируются в ссылки на Integer. То есть в данном объявлении ob имеет тип Integer и тип возврата getob() также имеет тип Integer.

Прежде чем двигаться дальше, необходимо заявить, что компилятор Java на самом деле не создает различные версии Gen или любого другого обобщенного класса. Хотя было бы удобно думать в таких терминах, на самом деле подобное не происходит. Вместо этого компилятор удаляет всю обобщенную информацию о типах, выполняя необходимые приведения, чтобы сделать поведение вашего кода таким, будто создана специфическая версия Gen. То есть имеется только одна версия Gen, которая существует в вашей программе. Процесс удаления обобщенной информации о типе называется очисткой (erasure), и мы вернемся к этой теме чуть позднее в настоящей главе. Следующая строка присваивает iOb ссылку на экземпляр Integer-версию класса Gen:

iOb = new Gen<Integer>(88);

Отметим, что когда вызывается конструктор Gen, аргумент типа Integer также указывается. Это необходимо, потому что типом объекта (в данном случае iOb), которому присваивается ссылка, является Gen<Integer>. То есть ссылка, возвращаемая new, также должна иметь тип Gen<Integer>. Если это не так, получается ошибка времени компиляции. Например, следующее присваивание вызовет ошибку компиляции:

iOb = new Gen<Double>(88.0); // Ошибка!

Поскольку iOb имеет тип Gen<Integer>, он не может быть использован для присваивания ссылки типа Gen<Double>. Эта проверка типа является одним из основных преимуществ обобщений, потому что обеспечивает безопасность типов.

Как указано в комментарии к программе, присваивание

iOb = new Gen<Integer>(88);

, использует автоупаковку для инкапсуляции значения 88, имеющего тип int, в Integer. Это работает, потому что Gen<Integer> создает конструктор, принимающий аргумент Integer. Поскольку ожидается Integer, Java автоматически упаковывает 88 внутрь него. Конечно, присваивание также может быть написано явно, как здесь:

iOb = new Gen<Integer>(new Integer(88));

Однако с этой версией не связано никаких преимуществ.

Программа затем отображает тип ob внутри iOb, который есть Integer. Далее программа получает значение ob в следующей строке:

int v = iOb.getob();

Поскольку возвращаемым типом getob() будет T, который заменяется на Integer при объявлении iOb, то возвращаемым типом getob() также будет Integer, который автоматически распаковывается в int и присваивается переменной v, имеющей тип int. То есть нет никакой необходимости приводить тип возвращаемого значения getob() к Integer. Конечно, использовать автоупаковку не обязательно. Предыдущая строка может быть написана так:

int v = iOb.getob().intValue();

Однако автоупаковка позволяет сделать код более компактным.

Далее в GenDemo объявляется объект типа Gen<String>:

Gen<String> strOb = new Gen<String>(" Обобщенный тест");

Поскольку типом-аргументом является String, String подставляется вместо T внутри Gen. Это создает (концептуально) String-версию Gen, что и демонстрируют остальные строки программы.

Когда объявляется экземпляр обобщенного типа, аргумент, переданный в качестве параметра типа, должен быть типом класса. Вы не можете использовать примитивный тип вроде int или char. Например, Gen можно передать любой тип класса в T, но нельзя передать примитивный тип в качестве параметра типа. Таким образом, следующее объявление недопустимо:

Gen<int> strOb = new Gen<int>(53); // Ошибка, нельзя использовать

// примитивные типы

Конечно, невозможность использовать примитивный тип не является серьезным ограничением, так как вы можете применять оболочки типов (как это и делается в предыдущем примере) для инкапсуляции примитивных типов. Более того, механизм автоупаковки и автораспаковки Java делает использование оболочек типов прозрачным.

Ключевой момент в понимании обобщенных типов в том, что ссылка на одну специфическую версию обобщенного типа не совместима с другой версией того же обобщенного типа. Например, следующая строка, если ее добавить к предыдущей программе, вызовет ошибку и программа не скомпилируется:

iOb = strOb; // Не верно!

Даже несмотря на то, что iOb и strOb имеют тип Gen<T>, они являются ссылками на разные типы, потому что типы их параметров отличаются. Это часть того способа, благодаря которому обобщения добавляют безопасность типов и предотвращают ошибки.

Теперь вы можете задать себе следующий вопрос: если та же функциональность, которую мы обнаружили в обобщенном классе Gen, может быть достигнута без обобщений, т.е. простым указанием Object в качестве типа данных и применением правильных приведений, в чем же выгода от того, что класс Gen параметризован? Ответ: в том, что обобщения автоматически гарантируют безопасность типов во всех операциях, где задействован Gen. В процессе работы с ним исключается необходимость явного приведения и ручной проверки типов в коде.

Чтобы понять выгоды от обобщений, для начала рассмотрим следующую программу, которая создает необобщенный эквивалент Gen:

// NonGen — функциональный эквивалент Gen

// не использующий обобщений.

class NonGen {

Object ob; // ob теперь имеет тип Object

// Передать конструктору ссылку на объект типа Object

NonGen(Object o) {

ob = o;

}

// Вернуть тип Object.

Object getob() {

return ob;

}

// Показать тип ob.

void showType() {

System.out.println("Типом ob является " +

ob.getClass().getName());

}}

// Демонстрация необобщенного класса.

class NonGenDemo {

public static void main(String args[]) {

NonGen iOb;

// Создать объект NonGen и сохранить

// Integer в нем. Автоупаковка используется.

iOb = new NonGen(88);

BookNew\_JAVA-7.indb 344 02.06.2007 1:07:12

Глава 14. Обобщения 345

// Показать тип данных, используемый iOb.

iOb.showType();

// Получить значение iOb.

// На этот раз приведение необходимо.

int v = (Integer) iOb.getob();

System.out.println("значение: " + v);

System.out.println();

// Создать другой объект NonGen и

// сохранить в нем String.

NonGen strOb = new NonGen("Тест без обобщений");

// Показать тип данных, используемый strOb.

strOb.showType();

// Получить значение strOb.

// Опять же — приведение необходимо.

String str = (String) strOb.getob();

System.out.println("Значение: " + str);

// Это компилируется, но концептуально неверно!

iOb = strOb;

v = (Integer) iOb.getob(); // ошибка времени выполнения!

}}

В этой версии программы присутствует несколько интересных моментов. Для начала NonGen заменяет все обращения к типу T на Object. Это дает NonGen возможность хранить объекты любого типа, как это делает и обобщенная версия. Однако это не дает возможности Java-компилятору иметь какую-то реальную информацию о типе данных, в действительности сохраняемых в NonGen, что плохо по двум причинам. Во-первых, для извлечения сохраненных данных требуется явное приведение. Во-вторых, многие ошибки несоответствия типов не могут быть обнаружены до времени выполнения. Рассмотрим каждую из этих проблем поближе.

Обратите внимание на строку:

int v = (Integer) iOb.getob();

Поскольку типом возврата getob() является Object, необходимо привести его к Integer, чтобы позволить выполнить автораспаковку и сохранить значение в v. Если убрать приведение, программа не скомпилируется. В версии с обобщением приведение происходит неявно. В версии без обобщения приведение должно быть явным. Это не только неудобство, но также потенциальный источник ошибок.

Теперь рассмотрим следующую кодовую последовательность в конце программы:

// Это компилируется, но концептуально неверно!

iOb = strOb;

v = (Integer) iOb.getob(); // ошибка времени выполнения!

Здесь strOb присваивается iOb. Однако strOb ссылается на объект, содержащий строку, а не целое число. Это присваивание синтаксически корректно, потому что все ссылки NonGen одинаковы, и любая ссылка NonGen может указывать на любой другой объект типа NonGen. Однако этот оператор семантически неверен, что и отражено в следующей строке. Здесь тип возврата getob() приводится к Integer, и затем делается попытка присвоить это значение v. Проблема в том, что iOb теперь ссылается на объект, который хранит String, а не Integer. К несчастью, без использования обобщений компилятор Java не имеет возможности обнаружить это. Вместо этого возбуждается исключение времени выполнения. Возможность создавать безопасный в отношении типов код, в котором ошибки несоответствия типов перехватываются компилятором — это главная выгода от обобщений. Хотя использование ссылок на Object для создания “псевдо-обобщенного” кода всегда возможно, нужно помнить, что такой код не является безопасным в отношении типов, и злоупотребление им приводит к исключениям времени выполнения.

Обобщения предотвращают такие вещи. По сути, благодаря обобщениям, то, что было ошибками времени выполнения, становится ошибками времени компиляции. Это и есть главное преимущество.

## Общая форма обобщенного класса

Синтаксис, показанный в предыдущих примерах, может быть обобщен. Так выглядит синтаксис объявления обобщенного класса:

class имя\_класса<список\_параметров\_типов> { // ...

Ниже показан синтаксис объявления ссылки на обобщенный класс:

имя\_класса<список\_аргументов\_типов> имя\_переменной =

new имя\_класса<список\_аргументов\_типов>(список\_аргументов\_констант);

В предыдущих примерах параметры типов могли быть заменены любыми типами классов. Это подходит ко многим случаям, но иногда удобно ограничить перечень типов, передаваемых в параметрах. Например, предположим, что вы хотите создать обобщенный класс, который содержит метод, возвращающий среднее значение массива чисел. Более того, вы хотите использовать этот класс для получения среднего значения чисел, включая целые и числа с плавающей точкой одинарной и двойной точности. То есть вы хотите специфицировать тип числовых данных обобщенно, используя параметр типа. Чтобы создать такой класс, можно попробовать что-то вроде этого:

// Stats пытается (безуспешно) создать

// обобщенный класс, который вычисляет

// среднее значение массива чисел

// заданного типа.

//

// Класс содержит ошибку!

class Stats<T> {

T[] nums; // nums — это массив элементов типа T

// Передать конструктору ссылку

// на массив значений типа T.

Stats(T[] o) {

nums = o;

}

// Возвращает double во всех случаях.

double average() {

double sum = 0.0;

for(int i=0; i < nums.length; i++)

sum += nums[i].doubleValue(); // Ошибка!!!

return sum / nums.length;

}}

Метод average() в Stats пытается получить double-версию каждого числа в массиве nums, вызывая doubleValue(). Поскольку все числовые классы, такие как Integer и Double, являются подклассами Number, и Number определяет метод doubleValue(), этот метод доступен всем числовым классам-оболочкам. Проблема в том, что компилятор не имеет возможности узнать, что вы намерены создавать объекты Stats, используя только числовые типы. То есть, когда вы компилируете Stats, выдается сообщение об ошибке, говорящее о том, что метод doubleValue() не известен. Чтобы решить эту проблему, вам нужен какой-то способ сообщить компилятору, что вы собираетесь передавать в параметре T только числовые типы. Более того, необходим еще некоторый способ гарантии того, что будут передаваться только числовые типы.

Чтобы справиться с этой ситуацией, Java предлагает ограниченные типы. Когда указывается параметр типа, вы можете создать ограничение сверху, которое объявляет суперкласс, от которого все типы-аргументы должны быть унаследованы. Это достигается применением слова extends при указании типа параметра, как показано ниже:

<T extends superclass>

Это означает, что T может быть заменено только классом superclass, либо его подклассами. То есть superclass объявляет включающую верхнюю границу.

Вы можете использовать ограничение сверху, чтобы исправить класс Stats, показанный выше, задав верхнюю границу используемого параметра типа Number:

// В этой версии Stats тип-аргумент

// T должен быть либо Number, либо классом,

// унаследованным от Number.

class Stats<T extends Number> {

T[] nums; // массив Number или подклассов

// Передать конструктору ссылку на массив

// элементов Number или его подклассов.

Stats(T[] o) {

nums = o;

}

// Возвратить double во всех случаях.

double average() {

double sum = 0.0;

for(int i=0; i < nums.length; i++)

sum += nums[i].doubleValue();

return sum / nums.length;

}}

BookNew\_JAVA-7.indb 348 02.06.2007 1:07:12

Глава 14. Обобщения 349

// Демонстрация Stats.

class BoundsDemo {

public static void main(String args[]) {

Integer inums[] = { 1, 2, 3, 4, 5 };

Stats<Integer> iob = new Stats<Integer>(inums);

double v = iob.average();

System.out.println("Среднее значение iob равно " + v);

Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Double> dob = new Stats<Double>(dnums);

double w = dob.average();

System.out.println("Среднее значение dob равно " + w);

// Это не скомпилируется, потому что String не является

// подклассом Number.

// String strs[] = { "1", "2", "3", "4", "5" };

// Stats<String> strob = new Stats<String>(strs);

// double x = strob.average();

// System.out.println("Среднее значение strob равно " + v);

}}

Результат работы этой программы выглядит следующим образом:

*Среднее значение iob равно 3.0*

*Среднее значение dob равно 3.3*

Обратите внимание, что Stats теперь объявлен так:

class Stats<T extends Number> {

Поскольку тип T теперь ограничен классом Number, компилятор Java знает, что все объекты типа T могут вызывать doubleValue(), так как это метод класса Number. Это уже серьезное преимущество. Однако в качестве дополнительного бонуса ограничение T также предотвращает создание нечисловых объектов Stats. Например, если вы попытаетесь убрать комментарии в строках, находящихся в конце программы, и перекомпилировать ее, то получите ошибку времени компиляции, потому что String не является подклассом Number.

В дополнение к использованию типа класса как ограничения вы также можете также применять тип интерфейса. Фактически вы можете специфицировать в качестве ограничений множество интерфейсов. Более того, такое ограничение может включать как тип класса, так и один или более интерфейсов. В этом случае тип класса должен быть задан первым. Когда ограничение включает тип интерфейса, допустимы только типы-аргументы, реализующие этот интерфейс. Указывая ограничение, имеющее класс и интерфейс либо множество интерфейсов, применяйте операцию & для их объединения, например:

class Gen<T extends MyClass & MyInterface> { // ...

Здесь T ограничено классом по имени MyClass и интерфейсом MyInteface. То есть любой тип, переданный в T, должен быть подклассом MyClass и иметь реализацию MyInteface.

## Создание обобщенного метода

Как было показано в предыдущих примерах, методы внутри обобщенного класса могут использовать параметр-класс, и, следовательно, обобщения касаются также параметров методов. Однако можно объявить обобщенный метод, который сам по себе использует один или более параметров типов. Более того, можно объявить обобщенный метод, который включен в не параметризованный (необобщенный) класс.

Начнем с примера. В следующей программе объявлен необобщенный класс по имени GenMethDemo и статический обобщенный метод внутри класса по имени isIn(). Метод isIn() определяет, является ли объект членом массива. Он может быть использован с любым типом объектов и массивов до тех пор, пока массив содержит объекты, совместимые с типом искомого объекта.

// Демонстрация простого обобщенного метода.

class GenMethDemo {

// Определение, содержится ли объект в массиве.

static <T, V extends T> boolean isIn(T x, V[] y) {

for(int i=0; i < y.length; i++)

if(x.equals(y[i])) return true;

return false;

}

BookNew\_JAVA-7.indb 356 02.06.2007 1:07:13

Глава 14. Обобщения 357

public static void main(String args[]) {

// Применение isIn() для Integer.

Integer nums[] = { 1, 2, 3, 4, 5 };

if(isIn(2, nums))

System.out.println("2 содержится в nums");

if(!isIn(7, nums))

System.out.println("7 не содержится в nums");

System.out.println();

// Применение isIn() для String.

String strs[] = { "один", "два", "три",

"четыре", "пять" };

if(isIn("два", strs))

System.out.println("два содержится в strs");

if(!isIn("семь", strs))

System.out.println("семь содержится в strs");

// Не скомпилируется! Типы должны быть совместимыми.

// if(isIn("два", nums))

// System.out.println("два содержится в strs ");

}}

Результат работы этой программы показан ниже:

*2 содержится в nums*

*7 не содержится в nums*

*два содержится в strs*

*семь не содержится в strs*

Рассмотрим isIn() поближе. Для начала посмотрите, как объявлен метод в следующей строке:

static <T, V extends T> boolean isIn(T x, V[] y) {

Параметр типа объявлен перед типом возврата метода. Второе — тип V ограничен сверху типом T. То есть V либо должен тем же типом, что и T, либо типом его подклассов.

Это отношение указывает, что isIn() может быть вызван только с аргументами, совместимыми между собой. Также обратите внимание, что метод isIn() статический, что позволяет вызывать его независимо от какого-либо объекта. Однако ясно, что обобщенные методы могут быть как статическими, так и нестатическими. Нет никаких ограничений на этот счет.

Теперь обратите внимание на то, как isIn() вызывается внутри main() — с нормальным синтаксисом вызовов, без необходимости специфицировать аргументы типа. Это потому, что типы аргументов подразумеваются автоматически, и типы T и V определяются соответственно. Например, в следующем вызове:

if(isIn(2, nums))

, тип первого аргумента — Integer (благодаря автоупаковке), что подставляет Integer вместо T. Базовый тип второго аргумента — также Integer, что подставляет его и вместо V.

Во втором вызове используются типы String, и вместо типов T и V подставляются String.

Теперь обратите внимание на закомментированный код:

// if(isIn("два", nums))

// System.out.println("два содержится в strs ");

Если вы уберете комментарии с этих строк и затем попытаетесь скомпилировать программу, то получите ошибку. Причина в том, что тип параметра V ограничен T конструкцией extends в объявлении V. Это значит, что V должно иметь либо тип T, либо тип его подкласса. А в этом случае первый аргумент имеет тип String, но второй — Integer, который не является подклассом String. Это вызовет ошибку несоответствия типов во время компиляции. Такая способность обеспечивать безопасность типов — одно из наиболее существенных преимуществ обобщенных методов.

Синтаксис, использованный для создания isIn(), можно обобщить. Вот синтаксис обобщенного метода:

<type-param-list> ret-type meth-name(param-list) { // ...

Во всех случаях type-param-list — это разделенный запятыми список параметров типа. Обратите внимание, что для обобщенного метода список параметров типа предшествует типу возврата.

## Обобщенные конструкторы

Конструкторы также могут быть обобщенными, даже если их классы таковыми не являются. Например, рассмотрим следующую короткую программу:

// Использование обобщенного конструктора.

class GenCons {

private double val;

<T extends Number> GenCons(T arg) {

val = arg.doubleValue();

}

void showval() {

System.out.println("val: " + val);

}}

class GenConsDemo {

public static void main(String args[]) {

GenCons test = new GenCons(100);

GenCons test2 = new GenCons(123.5F);

test.showval();

test2.showval();

}}

Вывод этой программы:

*val: 100.0*

*val: 123.5*

Поскольку GenCons() специфицирует параметр обобщенного типа, который может быть подклассом Number, GenCons() можно вызывать с любым числовым типом, включая Integer, Float или Double. Таким образом, даже несмотря на то, что GenCons — необобщенный класс, его конструктор обобщен.

## Обобщенные интерфейсы

В дополнение к обобщенным классам и методам вы можете объявлять обобщенные интерфейсы. Обобщенные интерфейсы специфицируются так же, как и обобщенные классы. Ниже показан пример. В нем создается интерфейс MinMax, объявляющий методы min() и max(), которые, как ожидается, должны возвращать минимальное и максимальное значения из некоторого множества объектов.

// Пример обобщенного интерфейса.

// Интерфейс Min/Max.

interface MinMax<T extends Comparable<T>> {

T min();

T max();

}

// Реализация MinMax

class MyClass<T extends Comparable<T>> implements MinMax<T> {

T[] vals;

MyClass(T[] o) { vals = o; }

// Возвращает минимальное значение из vals.

public T min() {

T v = vals[0];

for(int i=1; i < vals.length; i++)

if(vals[i].compareTo(v) < 0) v = vals[i];

return v;

}

// Возвращает максимальное значение из vals.

public T max() {

T v = vals[0];

for(int i=1; i < vals.length; i++)

if(vals[i].compareTo(v) > 0) v = vals[i];

return v;

}}

class GenIFDemo {

public static void main(String args[]) {

Integer inums[] = {3, 6, 2, 8, 6 };

Character chs[] = {'b', 'r', 'p', 'w' };

MyClass<Integer> iob = new MyClass<Integer>(inums);

MyClass<Character> cob = new MyClass<Character>(chs);

System.out.println("Максимальное значение в inums: " + iob.max());

System.out.println("Минимальное значение в inums: " + iob.min());

System.out.println("Максимальное значение в chs: " + cob.max());

System.out.println("Минимальное значение в chs: " + cob.min());

}}

Результат работы этой программы:

*Максимальное значение в inums: 8*

*Минимальное значение в inums: 2*

*Максимальное значение в chs: w*

*Минимальное значение в chs: b*

Хотя большинство аспектов этой программы просты для понимания, некоторые ключевые моменты следует пояснить. Во-первых, обратите внимание на то, как объявлен MinMax:

interface MinMax<T extends Comparable<T>> {

В общем случае, обобщенный интерфейс объявляется так же, как и обобщенный класс. В этом случае тип параметра T имеет верхнюю границу Comparable — интерфейс, определенный в java.lang. Класс, который реализует Comparable, определяет объекты, которые могут быть упорядочены. То есть требование, чтобы T был ограничен сверху Comparable, гарантирует, что MinMax может быть использован только с объектами, которые могут сравниваться между собой. (Более подробную информацию о Comparable можно найти в главе 16.) Отметим, что Comparable сам по себе также является обобщенным интерфейсом (он был преобразован в обобщенную форму в JDK 5). Он принимает параметр типа объектов, которые должны сравниваться.

Далее MinMax реализует класс MyClass. Вот как выглядит объявление класса MyClass:

class MyClass<T extends Comparable<T>> implements MinMax<T> {

Обратите особое внимание на способ, которым параметр типа T объявлен в MyClass и затем передан MinMax. Поскольку MinMax требует типа, который реализует Comparable, реализующий класс (в данном случае — MinMax) должен специфицировать то же ограничение. Более того, однажды установленное, это ограничение уже не нужно повторять в операторе implements. Фактически так поступать некорректно. Например, следующая строка неверна и не может быть скомпилирована:

// Это не правильно!

class MyClass<T extends Comparable<T>>

implements MinMax<T extends Comparable<T>> {

Однажды установленный, параметр типа просто передается интерфейсу без последующих модификаций. Вообще если класс реализует обобщенный интерфейс, то такой класс также должен быть обобщенным, по крайней мере, в тех пределах, где он принимает параметр типа, переданный интерфейсу. Например, следующая попытка объявления MyClass вызовет ошибку:

class MyClass implements MinMax<T> { // Неверно!

Поскольку MyClass не объявляет параметра, нет способа передать его MinMax. В этом случае идентификатор T просто неизвестен, и компилятор выдаст ошибку. Конечно, если класс реализует специфический тип обобщенного интерфейса, как показано ниже:

class MyClass implements MinMax<Integer> { // OK

, то реализующий класс не обязан быть обобщенным.

Обобщенный интерфейс представляет два преимущества. Во-первых, он может быть реализован для разных типов данных. Во-вторых, он позволяет включить ограничения на типы данных, для которых он может быть реализован. В примере MinMax вместо T могут быть подставлены только типы, совместимые с Comparable.

Вот общий синтаксис обобщенного интерфейса:

interface имя\_интерфейса<список\_параметров\_типов> { // ...

Здесь список\_параметров\_типов — разделенный запятыми список параметров типов. Когда реализуется обобщенный интерфейс, вы должны специфицировать аргументы типов, как показано ниже:

class имя\_класса<список\_параметров\_типов>

implements имя\_интерфейса<список\_аргументов\_типов> {

## Иерархии обобщенных классов

Обобщенные классы могут быть частью иерархии классов — так же, как и любые другие необобщенные классы. То есть обобщенный класс может выступать в качестве суперкласса или подкласса. Ключевое отличие между обобщенными и необобщенными иерархиями состоит в том, что в обобщенной иерархии любые аргументы типов, необходимые обобщенному суперклассу, всеми подклассами должны передаваться по иерархии вверх.

Это похоже на способ, которым аргументы конструкторов передаются по иерархии.

Ниже приведен пример иерархии, которая использует обобщенный суперкласс:

// Простая иерархия обобщенных классов.

class Gen<T> {

T ob;

Gen(T o) {

ob = o;

}

// Возвращает ob.

T getob() {

return ob;

}

}

// Подкласс Gen.

class Gen2<T> extends Gen<T> {

Gen2(T o) {

super(o);

}

}

В этой иерархии Gen2 расширяет обобщенный класс Gen. Обратите внимание на то, как в следующей строке объявлен Gen2:

class Gen2<T> extends Gen<T> {

Параметр типа T специфицирован в Gen2 и также передается Gen в операторе extends. Это значит, что тип, переданный Gen2, будет также передан Gen. Например, следующее объявление

Gen2<Integer> num = new Gen2<Integer>(100);

, передает Integer как параметр типа классу Gen. То есть ob внутри части Gen класса Gen2 будет иметь тип Integer. Отметим также, что Gen2 никак не использует параметр типа T, кроме того, что передает его суперклассу Gen. Даже если подкласс обобщенного суперкласса никак не нуждается в том, чтобы быть обобщенным, несмотря на это он все же должен специфицировать параметры типа, необходимые его обобщенному суперклассу.

Конечно, при необходимости подкласс может добавлять свои собственные параметры типа. Например, ниже показан вариант предыдущей иерархии, в котором Gen2 добавляет собственный параметр типа:

// Подкласс может добавлять свои собственные параметры типа.

class Gen<T> {

T ob; // Объявление объекта типа T

// Передача конструктору

// ссылки на объект типа T.

Gen(T o) {

ob = o;

}

// Возвращает ob.

T getob() {

return ob;

}}

// Подкласс Gen, который определяет

// второй параметр типа по имени V.

class Gen2<T, V> extends Gen<T> {

V ob2;

Gen2(T o, V o2) {

super(o);

ob2 = o2;

}

V getob2() {

return ob2;

}}

// Создание объекта типа Gen2.

class HierDemo {

public static void main(String args[]) {

// Создание объектов Gen2 для String и Integer.

Gen2<String, Integer> x =

new Gen2<String, Integer>("Значение равно: ", 99);

System.out.print(x.getob());

System.out.println(x.getob2());

}}

Обратите внимание на объявление Gen2, показанное в следующей строке:

class Gen2<T, V> extends Gen<T> {

Здесь T — тип, переданный Gen, а V — тип, специфичный для Gen2. V используется для объявления объекта, названного ob2, а также в качестве типа возврата метода getob2(). В main() создается объект Gen2 с типом String для параметра T и типом Integer для параметра V.

## Обобщенный подкласс

Абсолютно приемлемо, когда суперклассом для обобщенного класса выступает класс необобщенный. Например, рассмотрим программу:

// Необобщенный класс может быть суперклассом

// для обобщенного подкласса.

// Необобщенный класс.

class NonGen {

int num;

NonGen(int i) {

num = i;

}

int getnum() {

return num;

}}

// Обобщенный подкласс.

class Gen<T> extends NonGen {

T ob; // Объявление объекта типа T

// Передать конструктору объект

// типа T.

Gen(T o, int i) {

super(i);

ob = o;

}

// Возвращает ob.

T getob() {

return ob;

}}

// Создать объект Gen.

class HierDemo2 {

public static void main(String args[]) {

// Создать объект Gen для String.

Gen<String> w = new Gen<String>("Добро пожаловать", 47);

System.out.print(w.getob() + " ");

System.out.println(w.getnum());

}}

Результат работы этой программы показан ниже:

*Добро пожаловать 47*

Обратите внимание на то, как в этой программе Gen наследуется от NonGen:

class Gen<T> extends NonGen {

Поскольку NonGen — необобщенный класс, никакие аргументы типа не указываются. То есть даже если Gen объявляет тип-параметр T, он не требуется (и не может быть использован) NonGen. То есть Gen наследуется от NonGen обычным способом. Никаких специальных условий не требуется.

## Ошибки неоднозначности

Включение в язык обобщений породило новый тип ошибок, от которых вам нужно защищаться: неоднозначность (ambiquity). Ошибки неоднозначности случаются, когда очистка порождает два внешне разных обобщенных объявления, разрешаемых в виде одного очищенного типа, что вызывает конфликт. Вот пример, который включает перегрузку методов:

// Неоднозначность порождается очисткой перегруженных методов.

class MyGenClass<T, V> {

T ob1;

V ob2;

// ...

// Эти два перегруженных метода неоднозначны

// и не скомпилируются.

void set(T o) {

ob1 = o;

}

void set(V o) {

ob2 = o;

}

}

Обратите внимание, что MyGenClass объявляет два обобщенных типа: T и V. Внутри MyGenClass предпринимается попытка перегрузить set() на основе параметров T и V.

Это выглядит резонным, потому что кажется, что T и V — разные типы. Однако здесь возникают две проблемы неоднозначности.

Первая — судя по тому, как написан MyGenClass, нет требования, чтобы T и V были разными типами. Например, в принципе совершенно корректно сконструировать объект MyGenClass следующим образом:

MyGenClass<String, String> obj = new MyGenClass<String, String>()

В этом случае T и V будут заменены String. Это делает обе версии set() идентичными, что, конечно же, представляет собой ошибку.

Вторая, более фундаментальная проблема состоит в том, что очистка типов приводит обе версии метода к следующему виду:

void set(Object o) { // ...

То есть перегрузка set(), как пытается сделать MyGenClass, в действительности неоднозначна.

Ошибки неоднозначности бывает трудно исправить. Например, если вы знаете, что V всегда будет неким подтипом String, то можете попытаться исправить MyGenClass, переписав его объявление следующим образом:

class MyGenClass<T, V extends String> { // почти OK!

Это позволит скомпилировать MyGenClass, и вы даже сможете создавать экземпляры объектов этого класса примерно так:

MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();

Это работает, потому что Java может аккуратно определить, когда какой метод должен быть вызван. Однако неоднозначность возникнет, когда вы попытаетесь выполнить строку:

MyGenClass<String, String> x = new MyGenClass<String, String>();

В этом случае, поскольку и T, и V являются String, то какую версию set() нужно вызвать?

Честно говоря, в предыдущем примере было бы лучше использовать два метода с разными именами, вместо того чтобы перегружать set(). Часто разрешение неоднозначности требует реструктуризации кода, потому что неоднозначность свидетельствует о концептуальной ошибке в дизайне.

## Ограничения на статические члены

Никакой static-член не может использовать тип параметра, объявленный в его классе. Например, все static-члены этого класса являются недопустимыми:

class Wrong<T> {

// Неверно, нельзя создать статические переменные типа T.

static T ob;

// Неверно, ни один статический метод не может использовать T.

static T getob() {

return ob;

}

// Неверно, ни один статический метод не может иметь доступ

// к объекту типа T.

static void showob() {

System.out.println(ob);

}}

Несмотря на то что вы не можете объявить static-члены, которые используют тип параметра, объявленный в окружающем классе, вы можете объявлять обобщенные static методы, определяющие их собственные параметры типа, как это делалось ранее в настоящей главе.

## Ограничения обобщенных массивов

Существуют два важных ограничения обобщений, касающиеся массивов. Во-первых, вы не можете создать экземпляр массива, чей базовый тип — параметр типа. Во-вторых, вы не можете создать массив специфичных для типа обобщенных ссылок.

## Ограничения обобщенных исключений

Обобщенный класс не может расширять Throwable. Это значит, что вы не сможете создать обобщенных классов исключений.