ГЛАВА 4

Объекты и классы

В этой главе...

- Введение в объектно-ориентированное программирование
- Применение предопределенных классов
- Определение собственных классов
- Статические поля и методы
- Параметры методов
- Конструирование объектов
- Пакеты
- Путь к классам
- Комментарии и документирование
- Рекомендации по разработке классов

В этой главе рассматриваются следующие вопросы.

- Введение в объектно-ориентированное программирование.
- Создание объектов классов из стандартной библиотеки Java.
- Создание собственных классов.

Если вы недостаточно хорошо ориентируетесь в вопросах объектно-ориентированного программирования, внимательно прочитайте эту главу. Для объектно-ориентированного программирования требуется совершенно иной образ мышления по сравнению с подходом, типичным для процедурных языков. Освоить новые принципы создания программ не всегда просто, но сделать это необходимо. Для овладения языком Java нужно хорошо знать основные понятия объектно-ориентированного программирования.

Тем, у кого имеется достаточный опыт программирования на C++, материал этой и предыдущей глав покажется хорошо знакомым. Но у Java и C++ имеются

существенные отличия, поэтому последний раздел этой главы следует прочесть очень внимательно. Примечания к C++ помогут вам плавно перейти от C++ к Java.

4.1. Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) в настоящее время стало доминирующей методикой программирования, вытеснив "структурные" или процедурные подходы, разработанные в 1970-х годах. Java — это полностью объектно-ориентированный язык, и для продуктивного программирования на нем необходимо знать основные принципы ООП.

Объектно-ориентированная программа состоит из объектов. Каждый объект обладает определенными функциональными возможностями, предоставляемыми в распоряжение пользователей, а также скрытой реализацией. Одни объекты для своих программ вы можете взять в готовом виде из библиотеки, другие вам придется спроектировать самостоятельно. Строить ли свои объекты или приобретать готовые — зависит от вашего бюджета или времени. Но, как правило, до тех пор, пока объекты удовлетворяют вашим требованиям, вам не нужно особенно беспокоиться о том, каким образом реализованы их функциональные возможности.

Традиционное структурное программирование заключается в разработке набора процедур (или алгоритмов) для решения поставленной задачи. Определив эти процедуры, программист должен найти подходящий способ хранения данных. Вот почему создатель языка Pascal Никлаус Вирт (Niklaus Wirth) назвал свою известную книгу по программированию *Алгоритмы + Структуры данных = Программы* (Algorithms + Data Structures = Programs; издательство Prentice Hall, 1975 г). Обратите внимание на то, что в названии этой книги алгоритмы стоят на первом месте, а структуры данных — на втором. Это отражает образ мышления программистов того времени. Сначала они решали, как манипулировать данными, а затем — какую структуру применить для организации этих данных, чтобы с ними было легче работать. Подход ООП в корне изменил ситуацию, поставив на первое место данные и лишь на второе — алгоритмы, предназначенные для их обработки.

Для решения небольших задач процедурный подход оказывается вполне пригодным. Но объекты более приспособлены для решения более крупных задач. Рассмотрим в качестве примера небольшой веб-браузер. Его реализация может потребовать 2000 процедур, каждая из которых манипулирует набором глобальных данных. В стиле ООП та же самая программа может быть составлена всего из 100 классов, в каждом из которых в среднем определено по 20 методов (рис. 4.1). Такая структура программы гораздо удобнее для программирования. В ней легче находить ошибки. Допустим, что данные некоторого объекта находятся в неверном состоянии. Очевидно, что намного легче найти причину неполадок среди 20 методов, имеющих доступ к данным, чем среди 2000 процедур.

4.1.1. Классы

Для дальнейшей работы вам нужно усвоить основные понятия и терминологию ООП. Наиболее важным понятием является класс, применение которого уже демонстрировалось в примерах программ из предыдущих глав. *Класс* — это шаблон или образец, по которому будет создан объект. Обычно класс сравнивают с формой для выпечки печенья, а объект — это само печенье. Конструирование объекта на основе некоторого класса называется получением *экземпляра* этого класса.

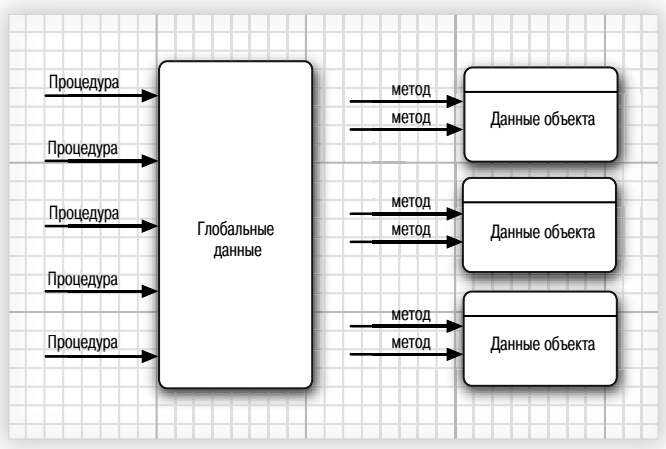

Рис. 4.1. Сравнение процедурного и объектно-ориентированного программирования

Как следует из примеров программ в предыдущих главах, весь код, написанный на Java, находится в классах. Стандартная библиотека Java содержит несколько тысяч классов, предназначенных для решения самых разных задач, например, для построения пользовательского интерфейса, календарей, установления сетевых соединений и т.д. Несмотря на это, программирующие на Java продолжают создавать свои собственные классы, чтобы формировать объекты, характерные для разрабатываемого приложения, а также приспосабливать классы из стандартной библиотеки под свои нужды.

Инкапсуляция (иногда называемая *сокрытием информации*) — это ключевое понятие для работы с объектами. Формально инкапсуляцией считается обычное объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются *полями экземпляра*, а функции и процедуры, выполняющие операции над данными, — его *методами*. В конкретном объекте, т.е. экземпляре класса, поля экземпляра имеют определенные значения. Множество этих значений называется текущим *состоянием* объекта. Вызов любого метода для объекта может изменить его состояние.

Следует еще раз подчеркнуть, что основной принцип инкапсуляции заключается в запрещении прямого доступа к полям экземпляра данного класса из других классов. Программы должны взаимодействовать с данными объекта только через методы этого объекта. Инкапсуляция обеспечивает внутреннее поведение объектов, что имеет решающее значение для повторного их использования и надежности работы программ. Это означает, что в классе можно полностью изменить способ хранения данных. Но поскольку для манипулирования данными используются одни и те же методы, то об этом ничего не известно, да и не особенно важно другим объектам.

Еще один принцип ООП облегчает разработку собственных классов в Java: один класс можно построить на основе других классов. В этом случае говорят, что новый класс *расширяет* тот класс, на основе которого он создан. Язык Java, по существу, создан на основе "глобального суперкласса", называемого `Object`. Все остальные объекты расширяют его. В следующей главе мы рассмотрим этот вопрос более подробно.

Если класс разрабатывается на основе уже существующего, то новый класс содержит все свойства и методы расширяемого класса. Кроме того, в него добавляются новые методы и поля данных. Расширение класса и получение на его основе нового называется *наследованием*. Более подробно принцип наследования будет рассмотрен в следующей главе.

4.1.2. Объекты

В ООП определены следующие ключевые свойства объектов.

- *Поведение* объекта — что с ним можно делать и какие методы к нему можно применять.
- *Состояние* объекта — как этот объект реагирует на применение методов.
- *Идентичность* объекта — чем данный объект отличается от других, характеризующихся таким же поведением и состоянием.

Все объекты, являющиеся экземплярами одного и того же класса, ведут себя одинаково. *Поведение* объекта определяется методами, которые можно вызвать. Каждый объект сохраняет информацию о своем *состоянии*. Со временем состояние объекта может измениться, но спонтанно это произойти не может. Состояние объекта может изменяться только в результате вызовов методов. (Если состояние объекта изменилось вследствие иных причин, значит, принцип инкапсуляции не соблюден.)

Состояние объекта не полностью описывает его, поскольку каждый объект имеет свою собственную *идентичность*. Например, в системе обработки заказов два заказа могут отличаться друг от друга, даже если они относятся к одним и тем же товарам. Заметим, что индивидуальные объекты, представляющие собой экземпляры класса, *всегда* отличаются своей идентичностью и, как правило, — своим состоянием.

Эти основные свойства объектов могут оказывать взаимное влияние. Например, состояние объекта может оказывать влияние на его поведение. (Если заказ выполнен или оплачен, объект может отказаться вызвать метод, требующий добавить или удалить товар. И наоборот, если заказ пуст, т.е. ни одна единица товара не была заказана, он не может быть выполнен.)

4.1.3. Идентификация классов

В традиционной процедурной программе выполнение начинается сверху, т.е. с функции `main()`. При проектировании объектно-ориентированной системы понятия "верха" как такового не существует, и поэтому начинающие осваивать ООП часто интересуются, с чего же следует начинать. Ответ таков: сначала нужно идентифицировать классы, а затем добавить методы в каждый класс.

Простое эмпирическое правило для идентификации классов состоит в том, чтобы выделить для них имена существительные при анализе проблемной области. С другой стороны, методы соответствуют глаголам, обозначающим действие. Например, при описании системы обработки заказов используются следующие имена существительные.

- Товар.
- Заказ.
- Адрес доставки.
- Оплата.
- Счет.

Этим именам соответствуют классы `Item`, `Order` и т.д.

Далее выбираются глаголы. Изделия *вводятся* в заказы. Заказы *выполняются* или *отменяются*. Оплата заказа *осуществляется*. Используя эти глаголы, можно определить объект, выполняющий такие действия. Так, если поступил новый заказ, ответственность за его обработку должен нести объект `Order` (Заказ), поскольку именно в нем содержится информация о способе хранения и сортировке заказываемых товаров. Следовательно, в классе `Order` должен существовать метод `add()` — добавить, получающий объект `Item` (Товар) в качестве параметра.

Разумеется, упомянутое выше правило выбора имен существительных и глаголов является не более чем рекомендацией. И только опыт может помочь программисту решить, какие именно существительные и глаголы следует выбрать при создании класса и его методов.

4.1.4. Отношения между классами

Между классами существуют три общих вида отношений.

- Зависимость ("использует — что-то").
- Агрегирование ("содержит — что-то").
- Наследование ("является — чем-то").

Отношение *зависимости* наиболее очевидное и распространенное. Например, в классе `Order` используется класс `Account`, поскольку объекты типа `Order` должны иметь доступ к объектам типа `Account`, чтобы проверить кредитоспособность заказчика. Но класс `Item` не зависит от класса `Account`, потому что объекты типа `Item` вообще не интересует состояние счета заказчика. Следовательно, один класс зависит от другого класса, если его методы выполняют какие-нибудь действия над экземплярами этого класса.

Старайтесь свести к минимуму количество взаимозависимых классов. Если класс `A` не знает о существовании класса `B`, то он тем более ничего не знает о любых изменениях в нем! (Это означает, что любые изменения в классе `B` не повлияют на поведение объектов класса `A`.)

Отношение *агрегирования* понять нетрудно, потому что оно конкретно. Например, объект типа `Order` может содержать объекты типа `Item`. Агрегирование означает, что объект класса `A` содержит объекты класса `B`.

НА ЗАМЕТКУ! Некоторые специалисты не признают понятие агрегирования и предпочитают использовать более общее отношение ассоциации или связи между классами. С точки зрения моделирования это разумно. Но для программистов гораздо удобнее отношение, при котором один объект содержит другой. Пользоваться понятием агрегирования удобнее по еще одной причине: его обозначение проще для восприятия, чем обозначение отношения ассоциации (табл. 4.1).

Наследование выражает отношение между конкретным и более общим классом. Например, класс `RushOrder` наследует от класса `Order`. Специализированный класс `RushOrder` содержит особые методы для обработки приоритетов и разные методы для вычисления стоимости доставки товаров, в то время как другие его методы, например, для заказа товаров и выписывания счетов, унаследованы от класса `Order`. Вообще говоря, если класс `A` расширяет класс `B`, то класс `A` наследует методы класса `B` и, кроме них, имеет дополнительные возможности. (Более подробно наследование рассматривается в следующей главе.)

Многие программисты пользуются средствами UML (Unified Modeling Language — унифицированный язык моделирования) для составления диаграмм классов, описывающих отношения между классами. Пример такой диаграммы приведен на рис. 4.2, где классы обозначены прямоугольниками, а отношения между ними — различными стрелками. В табл. 4.1 приведены основные обозначения, принятые в языке UML.

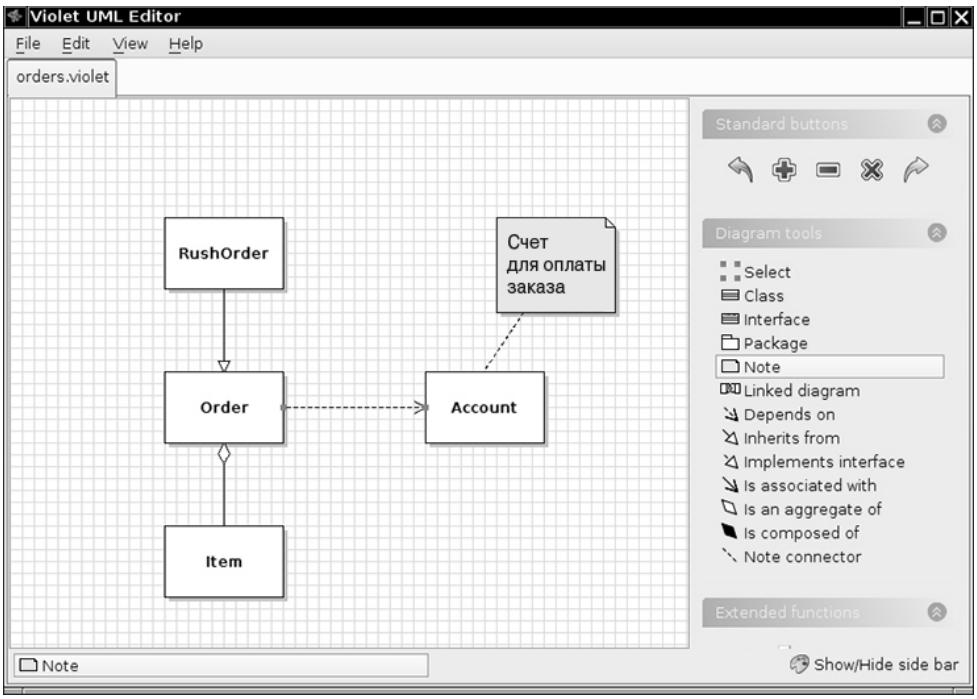

Рис. 4.2. Диаграмма классов

Таблица 4.1. Обозначение отношений между классами в UML

Отношение	Обозначение в UML
Наследование	
Реализация интерфейса	
Зависимость	
Агрегирование	
Связь	
Направленная связь	

4.2. Применение предопределенных классов

В языке Java ничего нельзя сделать без классов, поэтому мы вкратце обсудили в предыдущих разделах, каким образом действуют некоторые из них. Но в Java имеются также классы, к которым не совсем подходят приведенные выше рассуждения.

Характерным тому примером служит класс Math. Как упоминалось в предыдущей главе, методы из класса Math, например метод Math.random(), можно вызывать, вообще ничего не зная об их реализации. Для обращения к методу достаточно знать его имя и параметры (если они предусмотрены). Это признак инкапсуляции, который, безусловно, справедлив для всех классов. Но в классе Math отсутствуют данные; он инкапсулирует *только* функциональные возможности, не требуя ни данных, ни их сокрытия. Это означает, что можно и не заботиться о создании объектов и инициализации их полей, поскольку в классе Math ничего подобного нет!

В следующем разделе будет рассмотрен класс Date. На примере этого класса будет показано, каким образом создаются экземпляры и вызываются методы из класса.

4.2.1. Объекты и объектные переменные

Чтобы работать с объектами, их нужно сначала создать и задать их исходное состояние. Затем к этим объектам можно применять методы. Для создания новых экземпляров в Java служат *конструкторы*. Конструктор — это специальный метод, предназначенный для создания и инициализации экземпляра класса. В качестве примера можно привести класс Date, входящий в состав стандартной библиотеки Java. С помощью объектов этого класса можно описать текущий или любой другой момент времени, например "December 31, 1999, 23:59:59 GMT".

НА ЗАМЕТКУ! У вас может возникнуть вопрос: почему для представления даты и времени применяются классы, а не встроенные типы данных, как в ряде других языков программирования? Такой подход применяется, например, в Visual Basic, где дата задается в формате **#6/1/1995#**. На первый взгляд это удобно — программист может использовать встроенный тип и не заботиться о классах. Но не кажущееся ли такое удобство? В одних странах даты записываются в формате месяц/день/год, а в других — год/месяц/день. Могут ли создатели языка предусмотреть все возможные варианты? Даже если это и удастся сделать, соответствующие средства будут слишком сложны, причем программисты будут вынуждены применять их. Использование классов позволяет переложить ответственность за решение этих проблем с создателей языка на разработчиков библиотек. Если системный класс не годится, то разработчики всегда могут написать свой собственный класс. В качестве аргумента в пользу такого подхода отметим, что библиотека Java для дат довольно запутана и уже переделывалась дважды.

Имя конструктора всегда совпадает с именем класса. Следовательно, конструктор класса Date называется Date(). Чтобы создать объект типа Date, конструктор следует объединить с операцией new, как показано ниже, где создается новый объект, который инициализируется текущими датой и временем.

```
new Date()
```

При желании объект можно передать методу, как показано ниже.

```
System.out.println(new Date());
```

И наоборот, можно вызвать метод для вновь созданного объекта. Среди методов класса Date имеется метод toString(), позволяющий представить дату в виде символьной строки. Он вызывается следующим образом:

```
String = new Date().toString();
```

В этих двух примерах созданный объект использовался только один раз. Но, как правило, объектом приходится пользоваться неоднократно. Чтобы это стало

возможным, необходимо связать объект с некоторым идентификатором, иными словами, присвоить объект переменной, как показано ниже.

```
Date birthday = new Date();
```

На рис. 4.3 наглядно демонстрируется, каким образом переменная `birthday` ссылается на вновь созданный объект.

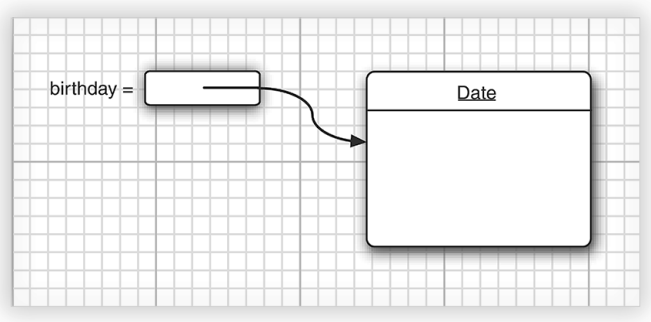

Рис. 4.3. Создание нового объекта

У объектов и объектных переменных имеется существенное отличие. Например, в приведенной ниже строке кода определяется объектная переменная `deadline`, которая может ссылаться на объекты типа `Date`.

```
Date deadline; // переменная deadline не ссылается ни на один из объектов
```

Важно понимать, что на данном этапе сама переменная `deadline` объектом не является и даже не ссылается ни на один из объектов. Поэтому ни один из методов класса `Date` пока еще нельзя вызывать по этой переменной. Попытка сделать это приведет к появлению сообщения об ошибке:

```
s = deadline.toString(); // вызывать метод еще рано!
```

Сначала нужно инициализировать переменную `deadline`. Для этого имеются две возможности. Прежде всего, переменную можно инициализировать вновь созданным объектом:

```
deadline = new Date();
```

Кроме того, переменной можно присвоить ссылку на существующий объект, как показано ниже.

```
deadline = birthday;
```

Теперь переменные `deadline` и `birthday` ссылаются на *один и тот же* объект (рис. 4.4).

Важно понять, что объектная переменная фактически не содержит никакого объекта. Она лишь *ссылается* на него. Значение любой объектной переменной в Java представляет собой ссылку на объект, размещенный в другом месте. Операция `new` также возвращает ссылку. Например, приведенная ниже строка кода состоит из двух частей: в операции `new Date()` создается объект типа `Date`, а переменной `deadline` присваивается ссылка на вновь созданный объект.

```
Date deadline = new Date();
```

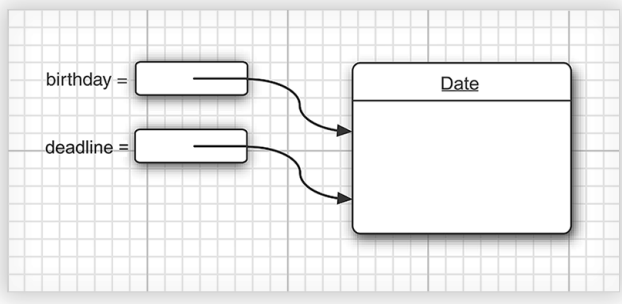

Рис. 4.4. Объектные переменные, ссылающиеся на один и тот же объект

Объектной переменной можно явно присвоить пустое значение `null`, чтобы указать на то, что она пока не ссылается ни на один из объектов:

```
deadline = null;
...
if(deadline != null)
  System.out.println(deadline);
```

Если попытаться вызвать метод по ссылке на переменную с пустым значением `null`, то при выполнении программы возникнет ошибка, как показано ниже.

```
birthday = null;
String s = birthday.toString(); // Ошибка при выполнении!
```

Локальные переменные не инициализируются автоматически пустым значением `null`. Программирующий должен сам инициализировать переменную, выполнив операцию `new` или присвоив пустое значение `null`.

НА ЗАМЕТКУ C++! Многие ошибочно полагают, что объектные переменные в Java похожи на ссылки в C++. Но в C++ пустые ссылки не допускаются и не присваиваются. Объектные переменные в Java следует считать аналогами указателей на объекты. Например, следующая строка кода Java:

```
Date birthday; // Java
```

почти эквивалентна такой строке кода C++:

```
Date* birthday; // C++
```

Такая аналогия все расставляет на свои места. Разумеется, указатель **`Date*`** не инициализируется до тех пор, пока не будет выполнена операция **`new`**. Синтаксис подобных выражений в C++ и Java почти совпадает.

```
Date* birthday = new Date(); // C++
```

При копировании одной переменной в другую в обеих переменных оказывается ссылка на один и тот же объект. Указатель **`NULL`** в C++ служит эквивалентом пустой ссылки **`null`** в Java.

Все объекты в Java располагаются в динамической области памяти, иначе называемой "кучей". Если объект содержит другую объектную переменную, она представляет собой всего лишь указатель на другой объект, расположенный в этой области памяти.

В языке C++ указатели доставляют немало хлопот, поскольку они часто приводят к ошибкам. Очень легко создать неверный указатель или потерять управление памятью. А в Java подобные сложности вообще не возникают. Если вы пользуетесь неинициализированным указателем, то исполняющая система обязательно сообщит об ошибке, а не продолжит выполнение некорректной программы, выдавая случайные результаты. Вам не нужно беспокоиться об управлении

памятью, поскольку механизм сборки "мусора" выполняет все необходимые операции с памятью автоматически.

В языке C++ большое внимание уделяется автоматическому копированию объектов с помощью копирующих конструкторов и операций присваивания. Например, копией связного списка является новый связный список, который, имея старое содержимое, содержит совершенно другие связи. Иными словами, копирование объектов осуществляется так же, как и копирование встроенных типов. А в Java для получения полной копии объекта служит метод **`clone()`**.

4.2.2. Класс `LocalDate` из библиотеки Java

В предыдущих примерах использовался класс `Date`, входящий в состав стандартной библиотеки Java. Экземпляр класса `Date` находится в состоянии, которое отражает *конкретный момент времени*.

Пользуясь классом `Date`, совсем не обязательно знать формат даты. Тем не менее время в этом классе представлено количеством миллисекунд (положительным или отрицательным), отсчитанным от фиксированного момента времени, так называемого начала *эпохи*, т.е. от момента времени 00:00:00 UTC, 1 января 1970 г. Сокращение UTC означает Universal Coordinated Time (Универсальное скоординированное время) — научный стандарт времени. Стандарт UTC применяется наряду с более известным стандартом GMT (Greenwich Mean Time — среднее время по Гринвичу).

Но класс `Date` не очень удобен для манипулирования датами. Разработчики библиотеки Java посчитали, что представление даты, например "`December 31, 1999, 23:59:59`", является совершенно произвольным и должно зависеть от календаря. Данное конкретное представление подчиняется григорианскому календарю, самому распространенному в мире. Но тот же самый момент времени совершенно иначе представляется в китайском или еврейском лунном календаре, не говоря уже о календаре, которым будут пользоваться потенциальные потребители с Марса.

НА ЗАМЕТКУ! Вся история человечества сопровождалась созданием календарей — систем именования различных моментов времени. Как правило, основой для календарей служил солнечный или лунный цикл. Если вас интересуют подобные вопросы, обратитесь за справкой, например, к книге Наума Дершовица (Nachum Dershowitz) и Эдварда М. Рейнгольда (Edward M. Reingold) *Calendrical Calculations* (издательство Cambridge University Press, 2nd ed., 2001 г.). Там вы найдете исчерпывающие сведения о календаре французской революции, календаре Майя и других экзотических системах отсчета времени.

Разработчики библиотеки Java решили отделить вопросы, связанные с отслеживанием моментов времени, от вопросов их представления. Таким образом, стандартная библиотека Java содержит два отдельных класса: класс `Date`, представляющий момент времени, и класс `LocalDate` , выражающий даты в привычном календарном представлении. В версии Java SE 8 внедрено немало других классов для манипулирования различными характеристиками даты и времени, как поясняется в главе 6 второго тома настоящего издания.

Отделение измерения времени от календарей является грамотным решением, вполне отвечающим принципам ООП.

Для построения объектов класса `LocalDate` нужно пользоваться не его конструктором, а статическими *фабричными методами*, автоматически вызывающими соответствующие конструкторы. Так, в следующем выражении:

```
LocalDate.now()
```

создается новый объект, представляющий дату построения этого объекта.

Безусловно, построенный объект желательно сохранить в объектной переменной, как показано ниже.

```
LocalDate newYearsEve = LocalDate.of(1999, 12, 31);
```

Имея в своем распоряжении объект типа `LocalDate`, можно определить год, месяц и день с помощью методов `getYear()`, `getMonthValue()` и `getDayOfMonth()`, как демонстрируется в следующем примере кода:

```
int year = newYearsEve.getYear(); // 1999 г.
int month = newYearsEve.getMonthValue(); // 12-й месяц
int day = newYearsEve.getDayOfMonth(); // 31-е число
```

На первый взгляд, в этом нет никакого смысла, поскольку те же самые значения даты использовались для построения объекта. Но иногда можно воспользоваться датой, вычисленной иным способом, чтобы вызвать упомянутые выше методы и получить дополнительные сведения об этой дате. Например, с помощью метода `plusDays()` можно получить новый объект типа `LocalDate`, отстоящий во времени на заданное количество дней от объекта, к которому этот метод применяется:

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
year = aThousandDaysLater.getYear(); // 2002 г.
month = aThousandDaysLater.getMonthValue(); // 9-й месяц
day = aThousandDaysLater.getDayOfMonth(); // 26-е число
```

В классе `LocalDate` инкапсулированы поля экземпляра для хранения заданной даты. Не заглянув в исходный код этого класса, нельзя узнать, каким образом в нем представлена дата. Но ведь назначение такой инкапсуляции в том и состоит, что пользователю класса `LocalDate` вообще не нужно об этом знать. Ему важнее знать о тех методах, которые доступны в этом классе.

НА ЗАМЕТКУ! На самом деле в классе **`Date`** имеются такие методы, как **`getDay()`**, **`getMonth()`** и **`getYear()`**, но пользоваться ими без крайней необходимости *не* рекомендуется. Метод объявляется не рекомендованным к применению, когда разработчики библиотеки решают, что его не стоит больше применять в новых программах.

Эти методы были частью класса **`Date`** еще до того, как разработчики библиотеки Java поняли, что классы, реализующие разные календари, разумнее было бы отделить друг от друга. Внедрив такие классы еще в версии Java 1.1, они пометили методы из класса **`Date`** как не рекомендованные к применению. Вы вольны и дальше пользоваться ими в своих программах, получая при этом предупреждения от компилятора, но лучше вообще отказаться от их применения, поскольку они могут быть удалены из последующих версий библиотеки.

4.2.3. Модифицирующие методы и методы доступа

Рассмотрим еще раз следующий вызов метода `plusDays()`, упоминавшегося в предыдущем разделе:

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
```

Что произойдет с объектом `newYearsEve` после этого вызова? Будет ли он отодвинут во времени на 1000 дней назад? Оказывается, нет. В результате вызова метода `plusDays()` получается новый объект типа `LocalDate`, который затем присваивается переменной `aThousandDaysLater`, а исходный объект остается без изменения. В таком случае говорят, что метод `plusDays()` *не* модифицирует объект, для которого он вызывается. (Аналогичным образом действует метод `toUpperCase()` из класса `String`, упоминавшийся

в главе 3. Когда этот метод вызывается для символьной строки, которая остается без изменения, в итоге возвращается новая строка символов в верхнем регистре.)

В более ранней версии стандартной библиотеки Java имелся другой класс `GregorianCalendar`, предназначенный для обращения с календарями. Ниже показано, как добавить тысячу дней к дате, представленной этим классом.

```
GregorianCalendar someDay = new GregorianCalendar(1999, 11, 31);
    // В этом класса месяцы нумеруются от 0 до 11
someDay.add(Calendar.DAY_OF_MONTH, 1000);
```

В отличие от метода `LocalDate.plusDays()`, метод `GregorianCalendar.add()` является *модифицирующим*. После его вызова состояние объекта `someDay` изменяется. Ниже показано, как определить это новое состояние. Переменная была названа `someDay`, а не `newYearsEve` потому, что она больше не содержит канун нового года после вызова модифицирующего метода.

```
year = someDay.get(Calendar.YEAR); // 2002 г.
month = someDay.get(Calendar.MONTH) + 1; // 9-й месяц
day = someDay.get(Calendar.DAY_OF_MONTH); // 26-е число
```

С другой стороны, методы, получающие только доступ к объектам, не модифицируя их, называют *методами доступа*. К их числу относятся, например, методы `LocalDate.getYear()` и `GregorianCalendar.get()`.

НА ЗАМЕТКУ C++! Для обозначения метода доступа в C++ служит суффикс `const`. Метод, не объявленный с помощью ключевого слова **`const`**, считается модифицирующим. Но в Java нет специальных синтаксических конструкций, позволяющих отличать модифицирующие методы от методов доступа.

И в завершение рассмотрим пример программы, в которой демонстрируется применение класса `LocalDate`. Эта программа выводит на экран календарь текущего месяца в следующем формате:

```
Sun Mon Tue Wed Thu Fri Sat
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19  20  21  22
23  24  25  26* 27  28  29
30
```

Текущий день помечен в календаре звездочкой. Как видите, программа должна знать, как вычисляется длина месяца и текущий день недели. Рассмотрим основные стадии выполнения данной программы. Сначала в ней создается объект типа `LocalDate`, инициализированный текущей датой:

```
LocalDate date = LocalDate.now();
```

Затем определяется текущий день и месяц:

```
int month = date.getMonthValue();
int today = date.getDayOfMonth();
```

После этого переменной `date` присваивается первый день месяца и из этой даты получается день недели.

```
date = date.minusDays(today - 1); // задать 1-й день месяца
DayOfWeek weekday = date.getDayOfWeek();
int value = weekday.getValue();
    // 1 = понедельник, ... 7 = воскресенье
```

Сначала переменной weekday присваивается объект типа DayOfWeek, а затем для этого объекта вызывается метод getValue(), чтобы получить числовое значение дня недели. Это числовое значение соответствует международному соглашению о том, что воскресенье приходится на конец недели. Следовательно, понедельнику соответствует возвращаемое данным методом числовое значение **1**, вторнику — **2** и так далее до воскресенья, которому соответствует числовое значение **7**.

Обратите внимание на то, что первая строка после заголовка календаря выведена с отступом, чтобы первый день месяца выпадал на соответствующий день недели. Ниже приведен фрагмент кода для вывода заголовка и отступа в первой строке календаря.

```
System.out.println("Mon Tue Wed Thu Fri Sat Sun");
for (int i = 1; i < value; i++)
  System.out.print("  ");
```

Теперь все готово для вывода самого календаря. С этой целью организуется цикл, где дата в переменной date перебирается по всем дням месяца. На каждом шаге цикла выводится числовое значение даты. Если дата в переменной date приходится на текущий день месяца, этот день помечается знаком *****. Затем дата в переменной date устанавливается на следующий день, как показано ниже. Когда в цикле достигается начало новой недели, выводится знак перевода строки.

```
while (date.getMonthValue() == month)
{
  System.out.printf("%3d", date.getDayOfMonth());
  if (date.getDayOfMonth() == today)
    System.out.print("*");
  else
    System.out.print(" ");
  date = date.plusDays(1);
  if (date.getDayOfWeek().getValue() == 1) System.out.println();
}
```

Когда же следует остановиться? Заранее неизвестно, сколько в месяце дней: 31, 30, 29 или 28. Поэтому цикл продолжается до тех пор, пока дата в переменной date остается в пределах текущего месяца. Исходный код данной программы полностью приведен в листинге 4.1.

Как видите, класс LocalDate позволяет легко создавать программы для работы с календарем, выполняя такие сложные действия, как отслеживание дней недели и учет продолжительности месяцев. Программирующему не нужно ничего знать, каким образом в классе LocalDate вычисляются месяцы и дни недели. Ему достаточно пользоваться *интерфейсом* данного класса, включая методы plusDays() и getDayOfWeek(). Основное назначение рассмотренной здесь программы — показать, как пользоваться интерфейсом класса для решения сложных задач, не вникая в подробности реализации.

Листинг 4.1. Исходный код из файла CalendarTest/CalendarTest.java

```
1  import java.time.*;
2
3  /**
4   * @version 1.5 2015-05-08
5   * @author Cay Horstmann
6  */
```

```
7
8  public class CalendarTest
9  {
10   public static void main(String[] args)
11   {
12     LocalDate date = LocalDate.now();
13     int month = date.getMonthValue();
14     int today = date.getDayOfMonth();
15
16     date = date.minusDays(today - 1); // задать 1-й день месяца
17     DayOfWeek weekday = date.getDayOfWeek();
18     int value = weekday.getValue();
19         // 1 = понедельник, ... 7 = воскресенье
20     System.out.println("Mon Tue Wed Thu Fri Sat Sun");
21     for (int i = 1; i < value; i++)
22       System.out.print("  ");
23     while (date.getMonthValue() == month)
24     {
25       System.out.printf("%3d", date.getDayOfMonth());
26       if (date.getDayOfMonth() == today)
27         System.out.print("*");
28       else
29         System.out.print(" ");
30       date = date.plusDays(1);
31       if (date.getDayOfWeek().getValue() == 1)
32         System.out.println();
33     }
34     if (date.getDayOfWeek().getValue() != 1) System.out.println();
35   }
36 }
```

java.util.LocalDate 8

- **static LocalTime now()**
  Строит объект, представляющий текущую дату.
- **static LocalTime of(int *year*, int *month*, int *day*)**
  Строит объект, представляющий заданную дату.
- **int getYear()**
- **int getMonthValue()**
- **int getDayOfMonth()**
  Получают год, месяц и день из текущей даты.
- **DayOfWeek getDayOfWeek()**
  Получает день недели из текущей даты в виде экземпляра класса **DayOfWeek**. Для получения дня недели в пределах от **1** (понедельник) до **7** (воскресенье) следует вызвать метод **getValue()**.
- **LocalDate plusDays(int *n*)**
- **LocalDate minusDays(int *n*)**
  Выдают дату на ***n*** дней после или до текущей даты.

4.3. Определение собственных классов

В примерах кода из главы 3 уже предпринималась попытка создавать простые классы. Но все они состояли из единственного метода `main()`. Теперь настало время показать, как создаются "рабочие" классы для более сложных приложений. Как правило, в этих классах метод `main()` отсутствует. Вместо этого они содержат другие методы и поля. Чтобы написать полностью завершенную программу, нужно объединить несколько классов, один из которых содержит метод `main()`.

4.3.1. Класс `Employee`

Простейшая форма определения класса в Java выглядит следующим образом:

```
class ИмяКласса
{
  поле_1
  поле_2

  конструктор_1
  конструктор_2
...
  метод_1
  метод_2
...
}
```

Рассмотрим следующую, весьма упрощенную версию класса `Employee`, который можно использовать для составления платежной ведомости:

```
class Employee
{
  // поля экземпляра
  private String name;
  private double salary;
  private Date hireDay;

  // конструктор
  public Employee(String n, double s, int year, int month, int day)
  {
    name = n;
    salary = s;
    hireDay = LocalDate.of(year, month, day);
  }

  // метод
  public String getName()
  {
    return name;
  }

  // другие методы
  . . .
}
```

Более подробно реализация этого класса будет проанализирована в последующих разделах, а сейчас рассмотрим код, приведенный в листинге 4.2 и демонстрирующий практическое применение класса `Employee`.

Листинг 4.2. Исходный код из файла EmployeeTest/EmployeeTest.java

```
1  import java.time.*;
2
3  /**
4   * В этой программе проверяется класс Employee
5   * @version 1.12 2015-05-08
6   * @author Cay Horstmann
7  */
8  public class EmployeeTest
9  {
10   public static void main(String[] args)
11   {
12     // заполнить массив staff тремя объектами типа Employee
13     Employee[] staff = new Employee[3];
14
15     staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16     staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17     staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
18
19     // поднять всем работникам зарплату на 5%
20     for (Employee e : staff)
21       e.raiseSalary(5);
22
23     // вывести данные обо всех объектах типа Employee
24     for (Employee e : staff)
25       System.out.println("name=" + e.getName() + ",salary="
26             + e.getSalary() + ",hireDay=" 26 + e.getHireDay());
27   }
28 }
29
30 class Employee
31 {
32   private String name;
33   private double salary;
34   private LocalDate hireDay;
35
36   public Employee(String n, double s, int year, int month, int day)
37   {
38     name = n;
39     salary = s;
40     hireDay = LocalDate.of(year, month, day);
41   }
42
43   public String getName()
44   {
45     return name;
46   }
47
48   public double getSalary()
49   {
50     return salary;
51   }
52
53   public LocalDate getHireDay()
54   {
55     return hireDay;
56   }
57
```

```
58   public void raiseSalary(double byPercent)
59   {
60     double raise = salary * byPercent / 100;
61     salary += raise;
62   }
63 }
```

В данной программе сначала создается массив типа `Employee`, в который заносятся три объекта работников:

```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Затем вызывается метод `raiseSalary()` из класса `Employee`, чтобы поднять зарплату каждого работника на 5%, как показано ниже.

```
for (Employee e : staff)
  e.raiseSalary(5);
```

И наконец, с помощью методов `getName()`, `getSalary()` и `getHireDay()` выводятся данные о каждом работнике.

```
for (Employee e : staff)
  System.out.println("name=" + e.getName()
      + ",salary=" + e.getSalary()
      + ",hireDay=" + e.getHireDay());
```

Следует заметить, что данный пример программы состоит из двух классов: `Employee` и `EmployeeTest`, причем последний объявлен открытым с модификатором доступа `public`. Метод `main()` с описанными выше операторами содержится в классе `EmployeeTest`. Исходный код данной программы содержится в файле `EmployeeTest.java`, поскольку его имя должно совпадать с именем открытого класса. В исходном файле может быть только один класс, объявленный как `public`, а также любое количество классов, в объявлении которых данное ключевое слово отсутствует.

При компиляции исходного кода данной программы создаются два файла классов: `EmployeeTest.class` и `Employee.class`. Затем начинается выполнение программы, для чего интерпретатору байт-кода указывается имя класса, содержащего основной метод `main()` данной программы:

```
java EmployeeTest
```

Интерпретатор начинает обрабатывать метод `main()` из класса `EmployeeTest`. В результате выполнения кода создаются три новых объекта типа `Employee` и отображается их состояние.

4.3.2. Использование нескольких исходных файлов

Рассмотренная выше программа из листинга 4.2 состоит из двух классов в одном исходном файле. Многие программирующие на Java предпочитают размещать каждый класс в отдельном файле. Например, класс `Employee` можно разместить в файле `Employee.java`, а класс `EmployeeTest` — в файле `EmployeeTest.java`.

Имеются разные способы компиляции программы, код которой содержится в двух исходных файлах. Например, при вызове компилятора можно использовать шаблонный символ подстановки следующим образом:

```
javac Employee*.java
```

В результате все исходные файлы, имена которых совпадают с указанным шаблоном, будут скомпилированы в файлы классов. С другой стороны, можно также ограничиться приведенной ниже командой.

```
javac EmployeeTest.java
```

Как ни странно, файл `Employee.java` также будет скомпилирован. Обнаружив, что в файле `EmployeeTest.java` используется класс `Employee`, компилятор Java станет искать файл `Employee.class`. Если компилятор его не найдет, то автоматически будет скомпилирован файл `Employee.java`. Более того, если файл `Employee.java` создан позже, чем существующий файл `Employee.class`, компилятор языка Java *автоматически* выполнит повторную компиляцию и создаст исходный файл данного класса.

НА ЗАМЕТКУ! Если вы знакомы с утилитой **`make`**, доступной в Unix и других операционных системах, то такое поведение компилятора не станет для вас неожиданностью. Дело в том, что в компиляторе Java реализованы функциональные возможности этой утилиты.

4.3.3. Анализ класса `Employee`

Проанализируем класс `Employee`, начав с его методов. Изучая исходный код, нетрудно заметить, что в классе `Employee` реализованы один конструктор и четыре метода, перечисляемые ниже.

```
public Employee(String n, double s,int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

Все методы этого класса объявлены как `public`, т.е. обращение к этим методам может осуществляться из любого класса. (Существуют четыре возможных уровня доступа. Все они рассматриваются в этой и следующей главах.)

Далее следует заметить, что в классе имеются три поля экземпляра для хранения данных, обрабатываемых в объекте типа `Employee`, как показано ниже.

```
private String name;
private double salary;
private Date hireDay;
```

Ключевое слово `private` означает, что к данным полям имеют доступ только методы самого класса `Employee`. Ни один внешний метод не может читать или записывать данные в эти поля.

НА ЗАМЕТКУ! Поля экземпляра могут быть объявлены как **`public`**, однако делать этого не следует. Ведь в этом случае любые компоненты программы (классы и методы) могут обратиться к открытым полям и видоизменить их содержимое, и, как показывает опыт, всегда найдется какой-нибудь код, который непременно воспользуется этими правами доступа в самый неподходящий момент. Поэтому мы настоятельно рекомендуем всегда закрывать доступ к полям экземпляра с помощью ключевого слова **`private`**.

И наконец, следует обратить внимание на то, что два из трех полей экземпляра сами являются объектами. В частности, поля `name` и `hireDay` являются ссылками на экземпляры классов `String` и `Date`. В ООП это довольно распространенное явление: одни классы часто содержат поля с экземплярами других классов.

4.3.4. Первые действия с конструкторами

Рассмотрим следующий конструктор класса Employee:

```
public Employee(String n, double s, int year, int month, int day)
{
  name = n;
  salary = s;
  GregorianCalendar calendar =
           new GregorianCalendar(year, month - 1, day);
  hireDay = calendar.getTime();
}
```

Как видите, имя конструктора совпадает с именем класса. Этот конструктор выполняется при создании объекта типа Employee, заполняя поля экземпляра заданными значениями. Например, при создании экземпляра класса Employee с помощью оператора

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

поля экземпляра заполняются следующими значениями:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

У конструкторов имеется существенное отличие от других методов: конструктор можно вызывать только в сочетании с операцией new. Конструктор нельзя применить к существующему объекту, чтобы изменить информацию в его полях. Например, приведенный ниже вызов приведет к ошибке во время компиляции.

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ОШИБКА!
```

Мы еще вернемся в этой главе к конструкторам. А до тех пор запомните следующее.

- Имя конструктора совпадает с именем класса.
- Класс может иметь несколько конструкторов.
- Конструктор может иметь один или несколько параметров или же вообще их не иметь.
- Конструктор не возвращает никакого значения.
- Конструктор всегда вызывается совместно с операцией new.

НА ЗАМЕТКУ C++! Конструкторы в Java и C++ действуют одинаково. Но учтите, что все объекты в Java размещаются в динамической памяти и конструкторы вызываются только вместе с операцией **new**. Те, у кого имеется опыт программирования на C++, часто допускают следующую ошибку:

```
Employee number007("James Bond", 10000, 1950, 1, 1)
    // допустимо в C++, но не в Java!
```

Это выражение в C++ допустимо, а в Java — нет.

ВНИМАНИЕ! Будьте осмотрительны, чтобы не присваивать локальным переменным такие же имена, как и полям экземпляра. Например, приведенный ниже конструктор не сможет установить зарплату работника.

```
public Employee(String n, double s, ...)
{
  String name = n; // ОШИБКА!
```

```
    double salary = s; // ОШИБКА!
  }
```

В конструкторе объявляются *локальные* переменные **name** и **salary**. Доступ к этим переменным возможен только внутри конструктора. Они *скрывают* поля экземпляра с аналогичными именами. Некоторые программисты могут написать такой код автоматически. Подобные ошибки очень трудно обнаружить. Поэтому нужно быть внимательными, чтобы не присваивать переменным имена полей экземпляра.

4.3.5. Явные и неявные параметры

Методы объекта имеют доступ ко всем его полям. Рассмотрим следующий метод:

```
public void raiseSalary(double byPercent)
{
  double raise = salary * byPercent / 100;
  salary += raise;
}
```

В этом методе устанавливается новое значение в поле `salary` того объекта, для которого вызывается этот метод. Например, следующий вызов данного метода:

```
number007.raiseSalary(5);
```

приведет к увеличению на 5% значения в поле `number007.salary` объекта `number007`. Строго говоря, вызов данного метода приводит к выполнению следующих двух операторов:

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

У метода `raiseSalary()` имеются два параметра. Первый параметр, называемый *неявным*, представляет собой ссылку на объект типа `Employee`, который указывается перед именем метода. А второй параметр называется *явным* и указывается как число в скобках после имени данного метода.

Нетрудно заметить, что явные параметры перечисляются в объявлении метода, например `double byPercent`. Неявный параметр в объявлении метода не приводится. В каждом методе ключевое слово `this` обозначает неявный параметр. По желанию метод `raiseSalary()` можно было бы переписать следующим образом:

```
public void raiseSalary(double byPercent)
{
  double raise = this.salary * byPercent / 100;
  this.salary += raise;
}
```

Некоторые предпочитают именно такой стиль программирования, поскольку в нем более отчетливо различаются поля экземпляра и локальные переменные.

НА ЗАМЕТКУ C++! Методы обычно определяются в C++ за пределами класса, как показано ниже.

```
void Employee::raiseSalary(double byPercent) // В C++, но не в Java
{
  ...
}
```

Если определить метод в классе, он автоматически станет встраиваемым.

```
class Employee
{
  ...
```

```
  int getName() { return name; } // встраиваемая функция в C++
}
```

А в Java все методы определяются в пределах класса, но это не делает их встраиваемыми. Виртуальная машина Java анализирует, как часто производится обращение к методу, и принимает решение, должен ли метод быть встраиваемым. Динамический компилятор находит краткие, частые вызовы методов, которые не являются переопределенными, и оптимизирует их соответствующим образом.

4.3.6. Преимущества инкапсуляции

Рассмотрим далее очень простые методы `getName()`, `getSalary()` и `getHireDay()` из класса `Employee`. Их исходный код приведен ниже.

```
public String getName()
{
  return name;
}

public double getSalary()
{
  return salary;
}

public Date getHireDay()
{
  return hireDay;
}
```

Они служат характерным примером *методов доступа*. А поскольку они лишь возвращают значения полей экземпляра, то иногда их еще называют *методами доступа к полям*. Но не проще ли было сделать поля `name`, `salary` и `hireDay` открытыми для доступа (т.е. объявить их как `public`) и не создавать отдельные методы доступа к ним?

Дело в том, что поле `name` доступно лишь для чтения. После того как значение этого поля будет установлено конструктором, ни один метод не сможет его изменить. А это дает гарантию, что данные, хранящиеся в этом поле, не будут искажены.

Поле `salary` доступно не только для чтения, но и для записи, но изменить значение в нем способен только метод `raiseSalary()`. И если окажется, что в поле записано неверное значение, то отладить нужно будет только один метод. Если бы поле `salary` было открытым, причина ошибки могла бы находиться где угодно.

Иногда требуется иметь возможность читать и видоизменять содержимое поля. Для этого придется реализовать в составе класса следующие три компонента.

- Закрытое (`private`) поле данных.
- Открытый (`public`) метод доступа.
- Открытый (`public`) модифицирующий метод.

Конечно, сделать это намного труднее, чем просто объявить открытым единственное поле данных. Но такой подход дает немалые преимущества. Во-первых, внутреннюю реализацию класса можно изменять совершенно независимо от других классов. Допустим, что имя и фамилия работника хранятся отдельно:

```
String firstName;
String lastName;
```

Тогда в методе `getName()` возвращаемое значение должно быть сформировано следующим образом:

```
firstName + " " + lastName
```

И такое изменение оказывается совершенно незаметным для остальной части программы. Разумеется, методы доступа и модифицирующие методы должны быть переработаны, чтобы учесть новое представление данных. Но это дает еще одно преимущество: модифицирующие методы могут выполнять проверку ошибок, тогда как при непосредственном присваивании открытому полю некоторого значения ошибки не выявляются. Например, в методе `setSalary()` можно проверить, не стала ли зарплата отрицательной величиной.

ВНИМАНИЕ! Будьте осмотрительны при создании методов доступа, возвращающих ссылки на изменяемый объект. Создавая класс **Emloyee**, авторы книги нарушили это правило в предыдущем издании: метод **getHireDay()** возвращает объект класса **Date**, как показано ниже.

```
class Employee
{
   private Date hireDay;
   ...
   public Date getHire();
  {
    return hireDay; // Неудачно!
  }

  ...
}
```

В отличие от класса **LocalDate**, где отсутствуют модифицирующие методы, в классе **Date** имеется модифицирующий метод **setTime()** для установки времени в миллисекундах. Но из-за того, что объекты типа **Date** оказываются изменяемыми, нарушается принцип инкапсуляции! Рассмотрим следующий пример неверного кода:

```
Employee harry = ...;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);
    // значение в объекте изменено
```

Причина ошибки в этом коде проста: обе ссылки, **d** и **harry.hireDay**, делаются на один и тот же объект (рис. 4.5). В результате применения модифицирующих методов к объекту **d** автоматически изменяется открытое состояние объекта работника типа **Employee**!

Чтобы вернуть ссылку на изменяемый объект, его нужно сначала клонировать. *Клон* — это точная копия объекта, находящаяся в другом месте памяти. Подробнее о клонировании речь пойдет в главе 6. Ниже приведен исправленный код.

```
class Employee
{
  . . .
  public Date getHireDay()
  {
    return hireDay.clone();
  }
  . . .
}
```

В качестве эмпирического правила пользуйтесь методом **clone()**, если вам нужно скопировать изменяемое поле данных.

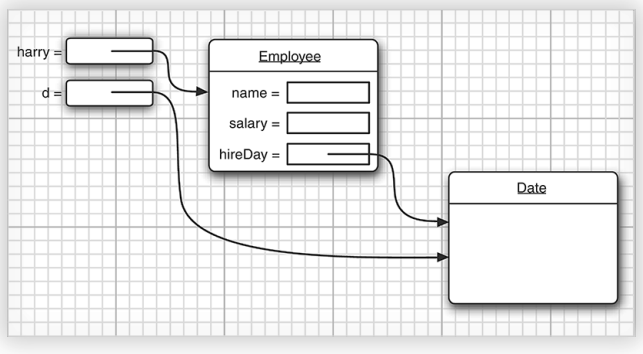

Рис. 4.5. Возврат ссылки на изменяемое поле данных

4.3.7. Привилегии доступа к данным в классе

Как вам должно быть уже известно, метод имеет доступ к закрытым данным того объекта, для которого он вызывается. Но он также может обращаться к закрытым данным *всех* объектов *своего* класса! Рассмотрим в качестве примера метод `equals()`, сравнивающий два объекта типа `Employee`.

```
class Employee
{
  . . .
  public boolean equals(Employee other)
  {
    return name.equals(other.name);
  }
}
```

Типичный вызов этого метода выглядит следующим образом:

```
if (harry.equals(boss)) ...
```

Этот метод имеет доступ к закрытым полям объекта `harry`, что и не удивительно. Но он также имеет доступ к полям объекта `boss`. И это вполне объяснимо, поскольку `boss` — объект типа `Employee`, а методы, принадлежащие классу `Employee`, могут обращаться к закрытым полям *любого* объекта этого класса.

НА ЗАМЕТКУ C++! В языке C++ действует такое же правило. Метод имеет доступ к переменным и функциям любого объекта своего класса.

4.3.8. Закрытые методы

При реализации класса все поля данных делаются закрытыми, поскольку предоставлять к ним доступ из других классов весьма рискованно. А как поступить с методами? Для взаимодействия с другими объектами требуются открытые методы. Но в ряде случаев для вычислений нужны вспомогательные методы. Как правило, эти вспомогательные методы не являются частью интерфейса, поэтому указывать при их объявлении ключевое слово `public` нет необходимости. И чаще всего они объявляются как `private`, т.е. как закрытые. Чтобы сделать метод закрытым, достаточно изменить ключевое слово `public` на `private` в его объявлении.

Сделав метод закрытым, совсем не обязательно сохранять его при переходе к другой реализации. Такой метод труднее реализовать, а возможно, он окажется вообще ненужным, если изменится представление данных, что, в общем, несущественно. Важнее другое: до тех пор, пока метод является закрытым (`private`), разработчики класса могут быть уверены в том, что он никогда не будет использован в операциях, выполняемых за пределами класса, а следовательно, они могут просто удалить его. Если же метод является открытым (`public`), его нельзя просто так опустить, поскольку от него может зависеть другой код.

4.3.9. Неизменяемые поля экземпляра

Поля экземпляра можно объявить с помощью ключевого слова `final`. Такое поле должно инициализироваться при создании объекта, т.е. необходимо гарантировать, что значение поля будет установлено по завершении каждого конструктора. После этого его значение изменить уже нельзя. Например, поле `name` из класса `Employee` можно объявить неизменяемым, поскольку после создания объекта оно уже не изменяется, а метода `setName()` для этого не существует.

```
class Employee
{
  ...
  private final String name;
}
```

Модификатор `final` удобно применять при объявлении полей простых типов или полей, типы которых задаются *неизменяемыми классами*. Неизменяемым называется такой класс, методы которого не позволяют изменить состояние объекта. Например, неизменяемым является класс `String`. Если класс допускает изменения, то ключевое слово `final` может стать источником недоразумений. Рассмотрим поле

```
private final StringBuilder evaluations;
```

которое инициализируется в конструкторе класса `Employee` таким образом:

```
evaluations = new StringBuilder();
```

Ключевое слово `final` означает, что ссылка на объект, хранящаяся в переменной `evaluations`, вообще не будет делаться на другой объект типа `StringBuilder`. Но в то же время объект может быть изменен следующим образом:

```
public void giveGoldStar()
{
  evaluations.append(LocalDate.now() + ": Gold star!\n");
}
```

4.4. Статические поля и методы

При объявлении метода `main()` во всех рассматривавшихся до сих пор примерах программ использовался модификатор `static`. Рассмотрим назначение этого модификатора доступа.

4.4.1. Статические поля

Поле с модификатором доступа `static` существует в одном экземпляре для всего класса. Но если поле не статическое, то каждый объект содержит его копию. Допустим, требуется присвоить уникальный идентификационный номер каждому

работнику. Для этого достаточно добавить в класс Employee поле id и статическое поле nextId, как показано ниже.

```
class Employee
{
  ...
  private int id;

  private static int nextId = 1;
}
```

Теперь у каждого объекта типа Employee имеется свое поле id, а также поле nextId, которое одновременно принадлежит всем экземплярам данного класса. Иными словами, если существует тысяча объектов типа Employee, то в них есть тысяча полей id: по одному на каждый объект. В то же время существует только один экземпляр статического поля nextId. Даже если не создано ни одного объекта типа Employee, статическое поле nextId все равно существует. Оно принадлежит классу, а не конкретному объекту.

НА ЗАМЕТКУ! В большинстве объектно-ориентированных языков статические поля называются *полями класса*. Термин *статический* унаследован как малозначащий пережиток от языка C++.

Реализуем следующий простой метод:

```
public void setId()
{
  id = nextId;
  nextId++;
}
```

Допустим, требуется задать идентификационный номер объекта harry следующим образом:

```
harry.setId();
```

Затем устанавливается текущее значение в поле id объекта harry, а значение статического поля nextId увеличивается на единицу, как показано ниже.

```
harry.id = Employee.nextId;
Employee.nextId++;
```

4.4.2. Статические константы

Статические переменные используются довольно редко. В то же время статические константы применяются намного чаще. Например, статическая константа, задающая число π, определяется в классе Math следующим образом:

```
public class Math
{
  ...
  public static final double PI = 3.14159265358979323846;
  ...
}
```

Обратиться к этой константе в программе можно с помощью выражения Math.PI. Если бы ключевое слово static было пропущено, константа PI была бы обычным полем экземпляра класса Math. Это означает, что для доступа к такой константе нужно было бы создать объект типа Math, причем каждый такой объект имел бы свою копию константы PI.

Еще одной часто употребляемой является статическая константа `System.out`. Она объявляется в классе `System` следующим образом:

```
public class System
{
  ...
  public static final PrintStream out = ...;
  ...
}
```

Как уже упоминалось не раз, делать поля открытыми в коде не рекомендуется, поскольку любой объект сможет изменить их значения. Но открытыми константами (т.е. полями, объявленными с ключевым словом `final`) можно пользоваться смело. Так, если поле `out` объявлено как `final`, ему нельзя присвоить другой поток вывода:

```
System.out = new PrintStream(...); // ОШИБКА: поле out изменить нельзя!
```

НА ЗАМЕТКУ! Анализируя исходный код класса **`System`**, можно обнаружить в нем метод **`setOut()`**, позволяющий присвоить полю **`System.out`** другой поток. Как же этот метод может изменить переменную, объявленную как **`final`**? Дело в том, что метод **`setOut()`** является платформенно-ориентированным, т.е. он реализован средствами конкретной платформы, а не языка Java. Платформенно-ориентированные методы способны обходить механизмы контроля, предусмотренные в Java. Это очень необычный обходной прием, который ни в коем случае не следует применять в своих программах.

4.4.3. Статические методы

Статическими называют методы, которые не оперируют объектами. Например, метод `pow()` из класса `Math` является статическим. При вызове метода `Math.pow(x, a)` вычисляется степень числа `xa`. При выполнении этого метода не используется ни один из экземпляров класса `Math`. Иными словами, у него нет неявного параметра `this`. Это означает, что в статических методах не используется текущий объект по ссылке `this`. (А в нестатических методах неявный параметр `this` ссылается на текущий объект; см. раздел 4.3.5 ранее в этой главе.)

Статическому методу из класса `Employee` недоступно поле экземпляра `id`, поскольку он не оперирует объектом. Но статические методы имеют доступ к статическим полям класса. Ниже приведен пример статического метода.

```
public static int getNextId()
{
  return nextId; // возвратить статическое поле
}
```

Чтобы вызвать этот метод, нужно указать имя класса следующим образом:

```
int n = Employee.getNextId();
```

Можно ли пропустить ключевое слово `static` при обращении к этому методу? Можно, но тогда для его вызова потребуется ссылка на объект типа `Employee`.

НА ЗАМЕТКУ! Для вызова статического метода можно использовать и объекты. Так, если **`harry`** — это объект типа **`Employee`**, то вместо вызова **`Employee.getNextId()`** можно сделать вызов **`harry.getNextId()`**. Но такое обозначение усложняет восприятие программы, поскольку для вычисления результата метод **`getNextId()`** не обращается к объекту **`harry`**. Поэтому для вызова статических методов рекомендуется использовать имена их классов, а не объекты.

Статические методы следует применять в двух случаях.

- Когда методу не требуется доступ к данным о состоянии объекта, поскольку все необходимые параметры задаются явно (например, в методе `Math.pow()`).
- Когда методу требуется доступ лишь к статическим полям класса (например, при вызове метода `Employee.getNextId()`).

НА ЗАМЕТКУ C++! Статические поля и методы в Java и C++, по существу, отличаются только синтаксически. Для доступа к статическому полю или методу, находящемуся вне области действия, в C++ можно воспользоваться операцией `::`, например **`Math::PI`**. Любопытно происхождение термина *статический*. Сначала ключевое слово **`static`** было внедрено в C для обозначения локальных переменных, которые не уничтожались при выходе из блока. В этом контексте термин *статический* имеет смысл: переменная продолжает существовать после выхода из блока, а также при повторном входе в него. Затем термин *статический* приобрел в C второе значение для глобальных переменных и функций, к которым нельзя получить доступ из других файлов. Ключевое слово **`static`** было просто использовано повторно, чтобы не вводить новое. И наконец, в C++ это ключевое слово было применено в третий раз, получив совершенно новую интерпретацию. Оно обозначает переменные и методы, принадлежащие классу, но ни одному из объектов этого класса. Именно этот смысл ключевое слово **`static`** имеет и в Java.

4.4.4. Фабричные методы

Рассмотрим еще одно применение статических методов. В таких классах, как, например, `LocalDate` и `NumberFormat`, для построения объектов применяются статические фабричные методы. Ранее в этой главе уже демонстрировалось применение фабричных методов `LocalDate.now()` и `LocalDate.of()`. А в следующем примере кода показано, каким образом в классе `NumberFormat` получаются форматирующие объекты для разных стилей вывода результатов:

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // выводит $0.10
System.out.println(percentFormatter.format(x));  // выводит 10%
```

Почему же не использовать для этой цели конструктор? На то есть две причины.

- Конструктору нельзя присвоить произвольное имя. Его имя всегда должно совпадать с именем класса. Так, в классе `NumberFormat` имеет смысл применять разные имена для разных типов форматирования.
- При использовании конструктора тип объекта фиксирован. Если же применяются фабричные методы, они возвращают объект типа `DecimalFormat`, наследующий свойства из класса `NumberFormat`. (Подробнее вопросы наследования будут обсуждаться в главе 5.)

4.4.5. Метод `main()`

Отметим, что статические методы можно вызывать, не имея ни одного объекта данного класса. Например, для того чтобы вызвать метод `Math.pow()`, объекты типа `Math` не нужны. По той же причине метод `main()` объявляется как статический:

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

```
  {
    // здесь создаются объекты
  }
}
```

Метод `main()` не оперирует никакими объектами. На самом деле при запуске программы еще нет никаких объектов. Статический метод `main()` выполняется и конструирует объекты, необходимые программе.

СОВЕТ. Каждый класс может содержать метод **`main()`**. С его помощью удобно организовать модульное тестирование классов. Например, метод **`main()`** можно добавить в класс **`Employee`** следующим образом:

```
class Employee
{
  public Employee(String n, double s, int year, int month, int day)
  {
    name = n;
    salary = s;
    hireDay = LocalDate.of(year, month, day);
  }
. . .
  public static void main(String[] args) // модульный тест
  {
    Employee e = new Employee ("Romeo", 50000, 2003, 3, 31);
    e.raiseSalary(10);
    System.out.println(e.getName() + " " + e.getSalary());
  }
  . . .
}
```

Если требуется протестировать только класс **`Employee`**, то для этого достаточно ввести команду `java Employee`. А если класс **`Employee`** является частью крупного приложения, то последнее можно запустить на выполнение по команде `java Application`. В этом случае метод **`main()`** из класса **`Employee`** вообще не будет выполнен.

Программа, приведенная в листинге 4.3, содержит простую версию класса `Employee` со статическим полем `nextId` и статическим методом `getNextId()`. В этой программе массив заполняется тремя объектами типа `Employee`, а затем выводятся данные о работниках. И наконец, для демонстрации статического метода на экран выводится очередной доступный идентификационный номер.

Следует заметить, что в классе `Employee` имеется также статический метод `main()` для модульного тестирования. Попробуйте выполнить метод `main()` по командам `java Employee` и `java StaticTest`.

Листинг 4.3. Исходный код из файла `StaticTest/StaticTest.java`

```
1  /**
2   * В этой программе демонстрируются статические методы
3   * @version 1.01 2004-02-19
4   * @author Cay Horstmann
5   */
6  public class StaticTest
7  {
8    public static void main(String[] args)
9    {
```

```
10     // заполнить массив staff тремя объектами типа Employee
11     Employee[] staff = new Employee[3];
12
13     staff[0] = new Employee("Tom", 40000);
14     staff[1] = new Employee("Dick", 60000);
15     staff[2] = new Employee("Harry", 65000);
16
17     // вывести данные обо всех объектах типа Employee
18     for (Employee e : staff)
19     {
20        e.setId();
21        System.out.println("name=" + e.getName() + ",id="
22              + e.getId() + ",salary=" + e.getSalary());
23     }
24
25     int n = Employee.getNextId(); // вызвать статический метод
26     System.out.println("Next available id=" + n);
27   }
28 }
29
30 class Employee
31 {
32   private static int nextId = 1;
33
34   private String name;
35   private double salary;
36   private int id;
37
38   public Employee(String n, double s)
39   {
40     name = n;
41     salary = s;
42     id = 0;
43   }
44   public String getName()
45   {
46     return name;
47   }
48
49   public double getSalary()
50   {
51     return salary;
52   }
53
54   public int getId()
55   {
56     return id;
57   }
58
59   public void setId()
60   {
61     id = nextId; // установить следующий доступный идентификатор
62     nextId++;
63   }
64
65   public static int getNextId()
66   {
67      return nextId; // возвратить статическое поле
68   }
69
```

```
70   public static void main(String[] args) // выполнить модульный тест
71   {
72     Employee e = new Employee("Harry", 50000);
73     System.out.println(e.getName() + " " + e.getSalary());
74   }
75 }
```

4.5. Параметры методов

Рассмотрим термины, которые употребляются для описания способа передачи параметров методам (или функциям) в языках программирования. Термин *вызов по значению* означает, что метод получает значение, переданное ему из вызывающей части программы. *Вызов по ссылке* означает, что метод получает из вызывающей части программы *местоположение* переменной. Таким образом, метод может *модифицировать* (т.е. видоизменить) значение переменной, передаваемой по ссылке, но не переменной, передаваемой по значению. Фраза "вызов по..." относится к стандартной компьютерной терминологии, описывающей способ передачи параметров в различных языках программирования, а не только в Java. (На самом деле существует еще и третий способ передачи параметров — *вызов по имени*, представляющий в основном исторический интерес, поскольку он был применен в языке Algol, который относится к числу самых старых языков программирования высокого уровня.)

В языке Java всегда используется *только* вызов по значению. Это означает, что метод получает копии значений всех своих параметров. По этой причине метод не может видоизменить содержимое ни одной из переменных, передаваемых ему в качестве параметров.

Рассмотрим для примера следующий вызов:

```
double percent = 10;
harry.raiseSalary(percent);
```

Каким бы образом ни был реализован метод, после его вызова значение переменной `percent` все равно останется равным **10**.

Проанализируем эту ситуацию подробнее. Допустим, в методе предпринимается попытка утроить значение параметра, как показано ниже.

```
public static void tripleValue(double x); // не сработает!
{
   x = 3 * x;
}
```

Если вызвать этот метод следующим образом:

```
double percent = 10;
tripleValue(percent);
```

такой прием не сработает. После вызова метода значение переменной `percent` по-прежнему остается равным **10**. В данном случае происходит следующее.

1. Переменная `x` инициализируется копией значения параметра `percent` (т.е. числом `10`).
2. Значение переменной `x` утраивается, и теперь оно равно **30**. Но значение переменной `percent` по-прежнему остается равным **10** (рис. 4.6).
3. Метод завершает свою работу, и его переменный параметр `x` больше не используется.

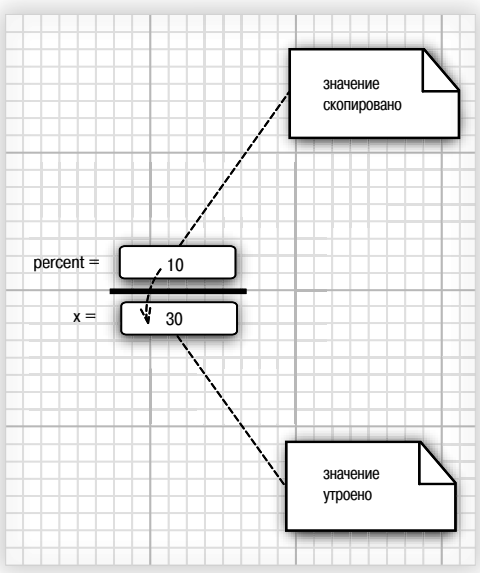

Рис. 4.6. Видоизменение значения в вызываемом методе не оказывает никакого влияния на параметр, передаваемый из вызывающей части программы

Но существуют два следующих типа параметров методов.

- Примитивные типы (т.е. числовые и логические значения).
- Ссылки на объекты.

Как было показано выше, методы не могут видоизменить параметры примитивных типов. Совсем иначе дело обстоит с объектами. Нетрудно реализовать метод, утраивающий зарплату работников, следующим образом:

```
public static void tripleSalary(Employee x) // сработает!
{
  x.raiseSalary(200);
}
```

При выполнении следующего фрагмента кода происходят перечисленные ниже действия.

```
harry = new Employee(...);
tripleSalary(harry);
```

1. Переменная **x** инициализируется копией значения переменной `harry`, т.е. ссылкой на объект.
2. Метод `raiseSalary()` применяется к объекту по этой ссылке. В частности, объект типа `Employee`, доступный по ссылкам `x` и `harry`, получает сумму зарплаты работников, увеличенную на **200%**.
3. Метод завершает свою работу, и его параметр `x` больше не используется. Разумеется, переменная `harry` продолжает ссылаться на объект, где зарплата увеличена втрое (рис. 4.7).

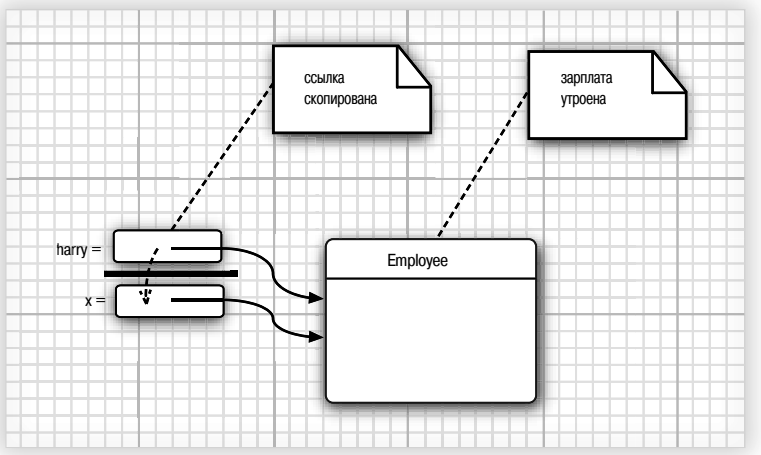

Рис. 4.7. Если параметр ссылается на объект, последний может быть видоизменен

Как видите, реализовать метод, изменяющий состояние объекта, передаваемого как параметр, совсем не трудно. В действительности такие изменения вносятся очень часто по следующей простой причине: метод получает копию ссылки на объект, поэтому копия и оригинал ссылки указывают на один и тот же объект.

Во многих языках программирования (в частности, C++ и Pascal) предусмотрены два способа передачи параметров: вызов по значению и вызов по ссылке. Некоторые программисты (и, к сожалению, даже авторы некоторых книг) утверждают, что в Java при передаче объектов используется вызов по ссылке. Но это совсем не так. Для того чтобы развеять это бытующее заблуждение, обратимся к конкретному примеру. Ниже приведен метод, выполняющий обмен двух объектов типа `Employee`.

```
public static void swap(Employee x, Employee y) // не сработает!
{
  Employee temp = x;
  x = y;
  y = temp;
}
```

Если бы в Java для передачи объектов в качестве параметров использовался вызов по ссылке, этот метод действовал бы следующим образом:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// ссылается ли теперь переменная a на Bob, а переменная b — на Alice?
```

Но на самом деле этот метод не меняет местами ссылки на объекты, хранящиеся в переменных `a` и `b`. Сначала параметры `x` и `y` метода `swap()` инициализируются *копиями* этих ссылок, а затем эти копии меняются местами в данном методе, как показано ниже.

```
// переменная x ссылается на Alice, а переменная y — на Bob
Employee temp = x;
x = y;
```

```
y = temp;
// теперь переменная x ссылается на Bob, а переменная y — на Alice
```

В конце концов, следует признать, что все было напрасно. По завершении работы данного метода переменные x и y уничтожаются, а исходные переменные a и b продолжают ссылаться на прежние объекты (рис. 4.8).

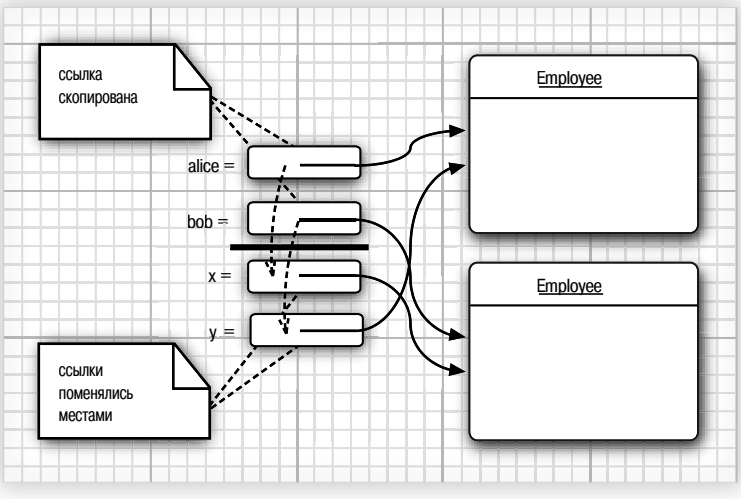

Рис. 4.8. Перестановка ссылок в вызываемом методе не имеет никаких последствий для вызывающей части программы

Таким образом, в Java для передачи объектов не применяется вызов по ссылке. Вместо этого ссылки на объекты передаются *по значению*. Ниже поясняется, что может и чего он не может метод делать со своими параметрами.

- Метод не может изменять параметры примитивных типов (т.е. числовые и логические значения).
- Метод может изменять *состояние* объекта, передаваемого в качестве параметра.
- Метод не может делать в своих параметрах ссылки на новые объекты.

Все эти положения демонстрируются в примере программы из листинга 4.4. Сначала в ней предпринимается безуспешная попытка утроить значение числового параметра.

```
Testing tripleValue:
(Тестирование метода tripleValue())
Before: percent=10.0
(До выполнения)
End of method: x=30.0
(В конце метода)
After: percent=10.0
(После выполнения)
```

Затем в программе успешно утраивается зарплата работника.

```
Testing tripleSalary:
(Тестирование метода tripleSalary())
```

```
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

После выполнения метода состояние объекта, на который ссылается переменная harry, изменяется. Ничего невероятного здесь нет, поскольку в данном методе состояние объекта было модифицировано по копии ссылки на него. И наконец, в программе демонстрируется безрезультатность работы метода swap().

```
Testing swap:
(Тестирование метода swap())
Before: a=Alice
Before: b=Bob
End of method: x=Bob
End of method: y=Alice
After: a=Alice
After: b=Bob
```

Как видите, переменные параметры x и y меняются местами, но исходные переменные a и b остаются без изменения.

Листинг 4.4. Исходный код из файла ParamTest/ParamTest.java

```
1  /**
2   * В этой программе демонстрируется передача параметров в Java
3   * @version 1.00 2000-01-27
4   * @author Cay Horstmann
5  */
6  public class ParamTest
7  {
8    public static void main(String[] args)
9    {
10      /*
11        * Тест 1: методы не могут видоизменять числовые параметры
12      */
13      System.out.println("Testing tripleValue:");
14      double percent = 10;
15      System.out.println("Before: percent=" + percent);
16      tripleValue(percent);
17      System.out.println("After: percent=" + percent);
18
19      /*
20        * Тест 2: методы могут изменять состояние объектов,
21        * передаваемых в качестве параметров
22      */
23      System.out.println("\nTesting tripleSalary:");
24      Employee harry = new Employee("Harry", 50000);
25      System.out.println("Before: salary=" + harry.getSalary());
26      tripleSalary(harry);
27      System.out.println("After: salary=" + harry.getSalary());
28
29      /*
30        * Тест 3: методы не могут присоединять новые объекты
31        * к объектным параметрам
32      */
33      System.out.println("\nTesting swap:");
34      Employee a = new Employee("Alice", 70000);
```

```
35       Employee b = new Employee("Bob", 60000);
36       System.out.println("Before: a=" + a.getName());
37       System.out.println("Before: b=" + b.getName());
38       swap(a, b);
39       System.out.println("After: a=" + a.getName());
40       System.out.println("After: b=" + b.getName());
41    }
42
43    public static void tripleValue(double x) // не сработает!
44    {
45       x = 3 * x;
46       System.out.println("End of method: x=" + x);
47    }
48
49    public static void tripleSalary(Employee x) // сработает!
50    {
51       x.raiseSalary(200);
52       System.out.println("End of method: salary=" + x.getSalary());
53    }
54
55    public static void swap(Employee x, Employee y)
56    {
57       Employee temp = x;
58       x = y;
59       y = temp;
60       System.out.println("End of method: x=" + x.getName());
61       System.out.println("End of method: y=" + y.getName());
62    }
63 }
64 class Employee // упрощенный класс Employee
65 {
66    private String name;
67    private double salary;
68
69    public Employee(String n, double s)
70    {
71       name = n;
72       salary = s;
73    }
74
75    public String getName()
76    {
77       return name;
78    }
79
80    public double getSalary()
81    {
82       return salary;
83    }
84
85    public void raiseSalary(double byPercent)
86   {
87      double raise = salary * byPercent / 100;
88      salary += raise;
89   }
90 }
```

НА ЗАМЕТКУ C++! В языке C++ применяется как вызов по значению, так и вызов по ссылке. Например, можно без особого труда реализовать методы **`void tripleValue(double& x)`** или **`void swap(Employee& x, Emloyee& y)`**, которые видоизменяют свои ссылочные параметры.

4.6. Конструирование объектов

Ранее было показано, как писать простые конструкторы, определяющие начальные состояния объектов. Но конструирование объектов — очень важная операция, и поэтому в Java предусмотрены самые разные механизмы написания конструкторов. Все эти механизмы рассматриваются ниже.

4.6.1. Перегрузка

У некоторых классов имеется не один конструктор. Например, пустой объект типа `StringBuilder` можно сконструировать (или, проще говоря, построить) следующим образом:

```
StringBuilder messages = new StringBuilder();
```

С другой стороны, исходную символьную строку можно указать таким образом:

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

Оба способа конструирования объектов носят общее название *перегрузки*. О перегрузке говорят в том случае, если у нескольких методов (в данном случае нескольких конструкторов) имеются одинаковые имена, но разные параметры. Компилятор должен сам решить, какой метод вызвать, сравнивая типы параметров, определяемых при объявлении методов, с типами значений, указанных при вызове методов. Если ни один из методов не соответствует вызову или же если одному вызову одновременно соответствует несколько вариантов, возникает ошибка компиляции. (Этот процесс называется *разрешением перегрузки*.)

НА ЗАМЕТКУ! В языке Java можно перегрузить любой метод, а не только конструкторы. Следовательно, для того чтобы полностью описать метод, нужно указать его имя и типы параметров. Подобное написание называется *сигнатурой* метода. Например, в классе **`String`** имеются четыре открытых метода под названием **`indexOf()`**. Они имеют следующие сигнатуры:

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Тип возвращаемого методом значения не входит в его сигнатуру. Следовательно, нельзя создать два метода, имеющих одинаковые имена и типы параметров и отличающихся лишь типом возвращаемого значения.

4.6.2. Инициализация полей по умолчанию

Если значение поля в конструкторе явно не задано, то ему автоматически присваивается значение по умолчанию: числам — нули; логическим переменным — логическое значение `false`; ссылкам на объект — пустое значение `null`. Но полагаться на действия по умолчанию не следует. Если поля инициализируются неявно, программа становится менее понятной.

НА ЗАМЕТКУ! Между полями и локальными переменными имеется существенное отличие. Локальные переменные всегда должны явно инициализироваться в методе. Но если поле не инициализируется в классе явно, то ему автоматически присваивается значение, задаваемое по умолчанию (0, **`false`** или **`null`**).

Рассмотрим в качестве примера класс `Employee`. Допустим, в конструкторе не задана инициализация значений некоторых полей. По умолчанию поле `salary` должно инициализироваться нулем, а поля `name` и `hireDay` — пустым значением `null`. Но при вызове метода `getName()` или `getHireDay()` пустая ссылка `null` может оказаться совершенно нежелательной, как показано ниже.

```
Date h = harry.getHireDay();
calendar.setTime(h); // Если h = null, генерируется исключение
```

4.6.3. Конструктор без аргументов

Многие классы содержат конструктор без аргументов, создающий объект, состояние которого устанавливается соответствующим образом по умолчанию. В качестве примера ниже приведен конструктор без аргументов для класса `Employee`.

```
public Employee()
{
  name = "";
  salary = 0;
  hireDay = new Date();
}
```

Если в классе совсем не определены конструкторы, то автоматически создается конструктор без аргументов. В этом конструкторе *всем* полям экземпляра присваиваются их значения, предусмотренные по умолчанию. Так, все числовые значения, содержащиеся в полях экземпляра, окажутся равными нулю, логические переменные — `false`, объектные переменные — `null`.

Если же в классе есть хотя бы один конструктор и явно не определен конструктор без аргументов, то создавать объекты, не предоставляя аргументы, нельзя. Например, у класса `Employee` из листинга 4.2 имеется один следующий конструктор:

```
Employee(String name, double salary, int y, int m, int d)
```

В этой версии данного класса нельзя создать объект, поля которого принимали бы значения по умолчанию. Иными словами, следующий вызов приведет к ошибке:

```
e = new Employee();
```

ВНИМАНИЕ! Следует иметь в виду, что конструктор без аргументов вызывается только в том случае, если в классе не определены другие конструкторы. Если же в классе имеется хотя бы один конструктор с параметрами и требуется создать экземпляр класса с помощью приведенного ниже выражения, следует явно определить конструктор без аргументов.

`new ИмяКласса()`

Разумеется, если значения по умолчанию во всех полях вполне устраивают, можно создать следующий конструктор без аргументов:

```
public ИмяКласса()
{
}
```

4.6.4. Явная инициализация полей

Конструкторы можно перегружать в классе, как и любые другие методы, а следовательно, задать начальное состояние полей его экземпляров можно несколькими способами. Каждое поле экземпляра следует всегда снабжать осмысленными значениями независимо от вызова конструктора.

В определении класса имеется возможность присвоить каждому полю соответствующее значение, как показано ниже.

```
class Employee
{
  ...
  private String name = "";
}
```

Это присваивание выполняется до вызова конструктора. Такой подход оказывается особенно полезным в тех случаях, когда требуется, чтобы поле имело конкретное значение независимо от вызова конструктора класса. При инициализации поля совсем не обязательно использовать константу. Ниже приведен пример, в котором поле инициализируется с помощью вызова метода.

```
class Employee
{
  private static int nextId;
  private int id = assignId();
  . . .
  private static int assignId()
  {
    int r = nextId;
    nextId++;
    return r;
  }
  . . .
}
```

НА ЗАМЕТКУ C++! В языке C++ нельзя инициализировать поля экземпляра непосредственно в описании класса. Значения всех полей должны задаваться в конструкторе. Но в C++ имеется синтаксическая конструкция, называемая *списком инициализации*, как показано ниже.

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
: name(n),
  salary(s),
  hireDay(y, m, d)
{
}
```

Эта специальная синтаксическая конструкция служит в C++ для вызова конструкторов полей. А в Java поступать подобным образом нет никакой необходимости, поскольку объекты не могут содержать подобъекты, но разрешается иметь только ссылки на них.

4.6.5. Имена параметров

Создавая даже элементарный конструктор (а большинство из них таковыми и являются), трудно выбрать подходящие имена для его параметров. Обычно в качестве имен параметров служат отдельные буквы, как показано ниже.

```
public Employee(String n, double s)
{
```

```
  name = n;
  salary = s;
}
```

Но недостаток такого подхода заключается в том, что, читая программу, невозможно понять, что же означают параметры `n` и `s`. Некоторые программисты добавляют к осмысленным именам параметров префикс **`"a"`**.

```
public Employee(String aName, double aSalary)
{
  name = aName;
  salary = aSalary
}
```

Такой код вполне понятен. Любой читатающий его может сразу определить, в чем заключается смысл параметра. Имеется еще один широко распространенный прием. Чтобы воспользоваться им, следует знать, что параметры *скрывают* поля экземпляра с такими же именами. Так, если вызвать метод с параметром `salary`, то ссылка `salary` будет делаться на параметр, а не на поле экземпляра. Доступ к полю экземпляра осуществляется с помощью выражения `this.salary`. Напомним, что ключевое слово `this` обозначает неявный параметр, т.е. конструируемый объект, как демонстрируется в следующем примере:

```
public Employee(String name, double salary)
{
  this.name = name;
  this.salary = salary;
}
```

НА ЗАМЕТКУ C++! В языке C++ к именам полей экземпляра обычно добавляются префиксы, обозначаемые знаком подчеркивания или буквой (нередко для этой цели служит буква **`m`** или **`x`**). Например, поле, в котором хранится сумма зарплаты, может называться **`_salary`**, **`mSalary`** или **`xSalary`**. Программирующие на Java, как правило, так не поступают.

4.6.6. Вызов одного конструктора из другого

Ключевым словом `this` обозначается неявный параметр метода. Но у этого слова имеется еще одно назначение. Если *первый оператор* конструктора имеет вид `this(...)`, то вызывается другой конструктор этого же класса. Ниже приведен характерный тому пример.

```
public Employee(double s)
{
  // вызвать конструктор Employee(String, double)
  this("Employee " + nextId, s);
  nextId++;
}
```

Если выполняется операция `new Employee(60000)`, то конструктор `Employee (double)` вызывает конструктор `Employee(String, double)`. Применять ключевое слово `this` для вызова другого конструктора очень удобно — нужно лишь один раз написать общий код для конструирования объекта.

НА ЗАМЕТКУ C++! Ссылка **`this`** в Java сродни указателю **`this`** в C++. Но в C++ нельзя вызвать один конструктор из другого. Для того чтобы реализовать общий код инициализации объекта в C++, нужно создать отдельный метод.

4.6.7. Блоки инициализации

Ранее мы рассмотрели два способа инициализации поля:

- установка его значения в конструкторе;
- присваивание значения при объявлении.

На самом деле в Java существует еще и третий механизм: использование блока инициализации. Такой блок выполняется всякий раз, когда создается объект данного класса. Рассмотрим следующий пример кода:

```
class Employee
{
  private static int nextId;

  private int id;
  private String name;
  private double salary;

  // блок инициализации
  {
    id = nextId;
    nextId++;
  }

  public Employee(String n, double s)
  {
    name = n;
    salary = s;
  }

  public Employee()
  {
    name = "";
    salary = 0;
  }

  ...
}
```

В этом примере начальное значение поля `id` задается в блоке инициализации объекта, причем неважно, какой именно конструктор используется для создания экземпляра класса. Блок инициализации выполняется первым, а вслед за ним — тело конструктора. Этот механизм совершенно не обязателен и обычно не применяется. Намного чаще применяются более понятные способы задания начальных значений полей.

НА ЗАМЕТКУ! В блоке инициализации допускается обращение к полям, определения которых находятся после данного блока. Несмотря на то что инициализация полей, определяемых после блока, формально допустима, поступать так не рекомендуется во избежание циклических определений. Конкретные правила изложены в разделе 8.3.2.3 спецификации Java (**http://docs.oracle.com/javase/specs**). Эти правила достаточно сложны, и учесть их в реализации компилятора крайне трудно. Так, в ранних версиях компилятора они были реализованы не без ошибок. Поэтому в исходном коде блоки инициализации рекомендуется размещать после определений полей.

При таком многообразии способов инициализации полей данных довольно трудно отследить все возможные пути процесса конструирования объектов. Поэтому рассмотрим подробнее те действия, которые происходят при вызове конструктора.

1. Все поля инициализируются значениями, предусмотренными по умолчанию (**0**, **false** или **null**).
2. Инициализаторы всех полей и блоки инициализации выполняются в порядке их следования в объявлении класса.
3. Если в первой строке кода одного конструктора вызывается другой конструктор, то выполняется вызываемый конструктор.
4. Выполняется тело конструктора.

Естественно, что код, отвечающий за инициализацию полей, нужно организовать так, чтобы в нем было легко разобраться. Например, было бы странным, если бы вызов конструкторов класса зависел от порядка объявления полей. Такой подход чреват ошибками.

Инициализировать статическое поле следует, задавая его начальное значение или используя статический блок инициализации. Первый механизм уже рассматривался ранее, а его пример приведен ниже.

```
static int nextId = 1;
```

Если для инициализации статических полей класса требуется сложный код, то удобнее использовать статический блок инициализации. Для этого следует разместить код в блоке и пометить его ключевым словом `static`. Допустим, идентификационные номера работников должны начинаться со случайного числа, не превышающего **10000**. Соответствующий блок инициализации будет выглядеть следующим образом:

```
// Статический блок инициализации
static
{
  Random generator = new Random();
  nextId = generator.nextInt(10000);
}
```

Статическая инициализация выполняется в том случае, если класс загружается впервые. Аналогично полям экземпляра, статические поля принимают значения **0**, **false** или **null**, если не задать другие значения явным образом. Все операторы, задающие начальные значения статических полей, а также статические блоки инициализации выполняются в порядке их перечисления в объявлении класса.

НА ЗАМЕТКУ! Оказывается, что до версии JDK 6 элементарную программу, выводящую символьную строку **"Hello, World"**, можно было написать и без метода **main()**, как показано ниже.

```
public class Hello
{
  static
  {
    System.out.println("Hello, World!");
  }
}
```

При выполнении команды **java Hello** загружался класс **Hello**, статический блок инициализации выводил строку **"Hello, World!"** и лишь затем появлялось сообщение о том, что метод **main()** не определен. Но начиная с версии Java SE 7, сначала выполняется проверка наличия в программе метода **main()**.

В программе, приведенной в листинге 4.5, наглядно демонстрируются многие языковые средства Java, обсуждавшиеся в этом разделе, включая следующие.

- Перегрузка конструкторов.
- Вызов одного конструктора из другого по ссылке `this(...)`.
- Применение конструктора без аргументов.
- Применение блока инициализации.
- Применение статической инициализации.
- Инициализация полей экземпляра.

Листинг 4.5. Исходный код из файла `ConstructorTest/ConstructorTest.java`

```
1  import java.util.*;
2  /**
3   * В этой программе демонстрируется конструирование объектов
4   * @version 1.01 2004-02-19
5   * @author Cay Horstmann
6  */
7  public class ConstructorTest
8  {
9    public static void main(String[] args)
10   {
11     // заполнить массив staff тремя объектами типа Employee
12
13     Employee[] staff = new Employee[3];
14
15     staff[0] = new Employee("Harry", 40000);
16     staff[1] = new Employee(60000);
17     staff[2] = new Employee();
18
19     // вывести данные обо всех объектах типа Employee
20
21     for (Employee e : staff)
22     System.out.println("name=" + e.getName() + ",id="
23            + e.getId() + ",salary="+ e.getSalary());
24   }
25 }
26 class Employee
27 {
28   private static int nextId;
29
30   private int id;
31   private String name = ""; // инициализация поля экземпляра
32   private double salary;
33
34   // статический блок инициализации
35
36   static
37   {
38     Random generator = new Random();
39     // задать произвольное число 0-999 в поле nextId
40     nextId = generator.nextInt(10000);
41   }
42
43   // блок инициализации объекта
```

```
44
45    {
46      id = nextId;
47      nextId++;
48    }
49
50    // три перегружаемых конструктора
51
52    public Employee(String n, double s)
53    {
54      name = n;
55      salary = s;
56    }
57
58    public Employee(double s)
59    {
60      // вызвать конструктор Employee(String, double)
61      this("Employee #" + nextId, s);
62    }
63
64    // конструктор без аргументов
65
66    public Employee()
67    {
68      // поле name инициализируется пустой строкой "" – см. ниже
69      // поле salary не устанавливается явно, а инициализируется нулем
70      // поле id инициализируется в блоке инициализации
71    }
72    public String getName()
73    {
74      return name;
75    }
76
77    public double getSalary()
78    {
79      return salary;
80    }
81
82    public int getId()
83    {
84      return id;
85    }
86  }
```

java.util.Random 1.0

- **Random()**
  Создает новый генератор случайных чисел.
- **int nextInt(int *n*) 1.2**
  Возвращает случайное число в пределах от **0** до ***n* - 1**.

4.6.8. Уничтожение объектов и метод finalize()

В некоторых объектно-ориентированных языках программирования и, в частности, в C++ имеются явные деструкторы, предназначенные для уничтожения объектов.

Их основное назначение — освободить память, занятую объектами. А в Java реализован механизм автоматической сборки "мусора", освобождать память вручную нет никакой необходимости, и поэтому в этом языке деструкторы отсутствуют.

Разумеется, некоторые объекты используют кроме памяти и другие ресурсы, например файлы, или оперируют другими объектами, которые, в свою очередь, обращаются к системным ресурсам. В этом случае очень важно, чтобы занимаемые ресурсы освобождались, когда они больше не нужны.

С этой целью в любой класс можно ввести метод `finalize()`, который будет вызван перед тем, как система сборки "мусора" уничтожит объект. Но на практике, если требуется возобновить ресурсы и сразу использовать их повторно, *нельзя* полагаться только на метод `finalize()`, поскольку заранее неизвестно, когда именно этот метод будет вызван.

НА ЗАМЕТКУ! Вызов метода **`System.runFinalizerOnExit(true)`** гарантирует, что метод **`finalize()`** будет вызван до того, как программа завершит свою работу. Но и этот метод крайне ненадежен и не рекомендован к применению. В качестве альтернативы можно воспользоваться методом **`Runtime.addShutdownHook()`**. Дополнительные сведения о нем можно найти в документации на прикладной программный интерфейс API.

Если ресурс должен быть освобожден сразу после его использования, в таком случае придется написать соответствующий код самостоятельно. С этой целью следует предоставить метод **`close()`**, выполняющий необходимые операции по очистке памяти, вызвав его, когда соответствующий объект больше не нужен. В разделе 7.2.5 главы 7 будет показано, каким образом обеспечивается автоматический вызов такого метода.

4.7. Пакеты

Язык Java позволяет объединять классы в наборы, называемые *пакетами*. Пакеты облегчают организацию работы и позволяют отделить классы, созданные одним разработчиком, от классов, разработанных другими. Стандартная библиотека Java содержит большое количество пакетов, в том числе `java.lang`, `java.util`, `java.net` и т.д. Стандартные пакеты Java представляют собой иерархические структуры. Подобно каталогам на диске компьютера, пакеты могут быть вложены один в другой. Все стандартные пакеты относятся к иерархиям пакетов `java` и `javax`.

Пакеты служат в основном для обеспечения однозначности имен классов. Допустим, двух программистов осенила блестящая идея создать класс `Employee`. Если оба класса будут находиться в разных пакетах, конфликт имен не возникнет. Чтобы обеспечить абсолютную однозначность имени пакета, рекомендуется использовать доменное имя компании в Интернете, записанное в обратном порядке (оно по определению единственное в своем роде). В составе пакета можно создавать подпакеты и использовать их в разных проектах. Рассмотрим в качестве примера домен `horstmann.com`, зарегистрированный автором данной книги. Записав это имя в обратном порядке, можно использовать его как название пакета — `com.horstmann`. В дальнейшем в этом пакете можно создать подпакет, например `com.horstmann.corejava`.

Единственная цель вложенных пакетов — гарантировать однозначность имен. С точки зрения компилятора между вложенными пакетами отсутствует какая-либо связь. Например, пакеты `java.util` и `java.util.jar` вообще не связаны друг с другом. Каждый из них представляет собой независимую коллекцию классов.

4.7.1. Импорт классов

В классе могут использоваться все классы из собственного пакета и все *открытые* классы из других пакетов. Доступ к классам из других пакетов можно получить двумя способами. Во-первых, перед именем каждого класса можно указать полное имя пакета, как показано ниже.

```
java.time.LocalDate today = java.time.LocalDate.now();
```

Очевидно, что этот способ не совсем удобен. Второй, более простой и распространенный способ предусматривает применение ключевого слова `import`. В этом случае имя пакета указывать перед именем класса необязательно.

Импортировать можно как один конкретный класс, так и пакет в целом. Операторы `import` следует разместить в начале исходного файла (после всех операторов `package`). Например, все классы из пакета `java.time` можно импортировать следующим образом:

```
import java.time.*;
```

После этого имя пакета не указывается, как показано ниже.

```
LocalDate today = LocalDate.now();
```

Отдельный класс можно также импортировать из пакета следующим образом:

```
import java.time.LocalDate;
```

Проще импортировать все классы, например, из пакета `java.time.*`. На объем кода это не оказывает никакого влияния. Но если указать импортируемый класс явным образом, то читающему исходный код программы станет ясно, какие именно классы будут в ней использоваться.

СОВЕТ. Работая в ИСР Eclipse, можно выбрать команду меню Source⇨Organize Imports (Исходный код⇨Организовать импорт). В результате выражения типа **`import java.util.*;`** будут автоматически преобразованы в последовательности строк, предназначенных для импорта отдельных классов:

```
import java.util.ArrayList;
import java.util.Date;
```

Такая возможность очень удобна при написании кода.

Следует, однако, иметь в виду, что оператор `import` со звездочкой можно применять для импорта только одного пакета. Но нельзя использовать оператор `import java.*` или `import java.*.*`, чтобы импортировать все пакеты, имена которых содержат префикс **`java`**. В большинстве случаев импортируется весь пакет, независимо от его размера. Единственный случай, когда следует обратить особое внимание на пакет, возникает при конфликте имен. Например, оба пакета, `java.util` и `java.sql`, содержат класс `Date`. Допустим, разрабатывается программа, в которой оба эти пакета импортируются следующим образом:

```
import java.util.*;
import java.sql.*;
```

Если теперь попытаться воспользоваться классом `Date`, возникнет ошибка компиляции:

```
Date today; // ОШИБКА: неясно, какой именно выбрать пакет:
            // java.util.Date или java.sql.date?
```

Компилятор не в состоянии определить, какой именно класс `Date` требуется в программе. Разрешить это затруднение можно, добавив уточняющий оператор `import` следующим образом:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

А что, если на самом деле нужны оба класса `Date`? В этом случае нужно указать полное имя пакета перед именем каждого класса, как показано ниже.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

Обнаружение классов в пакетах входит в обязанности *компилятора*. В байт-кодах, находящихся в файлах классов, всегда содержатся полные имена пакетов.

НА ЗАМЕТКУ C++! Программирующие на C++ считают, что оператор **`import`** является аналогом директивы **`#include`**. Но у них нет ничего общего. В языке C++ директиву **`#include`** приходится применять в объявлениях внешних ресурсов потому, что компилятор C++ не просматривает файлы, кроме компилируемого, а также файлы, указанные в самой директиве **`#include`**. А компилятор Java просматривает содержимое всех файлов при условии, если известно, где их искать. В языке Java можно и не применять механизм импорта, явно называя все пакеты, например **`java.util.Date`**. А в C++ избежать использования директивы **`#include`** нельзя.

Единственное преимущество оператора **`import`** заключается в его удобстве. Он позволяет использовать более короткие имена классов, не указывая полное имя пакета. Например, после оператора **`import java.util.*`** (или **`import java.util.Date`**) к классу **`java.util.Date`** можно обращаться по имени **`Date`**.

Аналогичный механизм работы с пакетами в C++ реализован в виде директивы **`namespace`**. Операторы **`package`** и **`import`** в Java можно считать аналогами директив **`namespace`** и **`using`** в C++.

4.7.2. Статический импорт

Имеется форма оператора `import`, позволяющая импортировать не только классы, но и статические методы и поля. Допустим, в начале исходного файла введена следующая строка кода:

```
import static java.lang.System.*;
```

Это позволит использовать статические методы и поля, определенные в классе `System`, не указывая имени класса:

```
out.println("Goodbye, World!"); // вместо System.out
exit(0); // вместо System.exit
```

Статические методы или поля можно также импортировать явным образом:

```
import static java.lang.System.out;
```

Но в практике программирования на Java такие выражения, как `System.out` или `System.exit`, обычно не сокращаются. Ведь в этом случае исходный код станет более трудным для восприятия. С другой стороны, следующая строка кода:

```
sqrt(pow(x, 2) + pow(y, 2))
```

выглядит более ясной, чем такая строка:

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

4.7.3. Ввод классов в пакеты

Чтобы ввести класс в пакет, следует указать имя пакета в начале исходного файла *перед* определением класса. Например, исходный файл `Employee.java` из листинга 4.7 начинается следующими строками кода:

```
package com.horstmann.corejava;

public class Employee
{
  ...
}
```

Если оператор `package` в исходном файле не указан, то классы, описанные в этом файле, вводятся в *пакет по умолчанию*. У пакета по умолчанию нет имени. Все рассмотренные до сих пор классы принадлежали пакету по умолчанию.

Пакеты следует размещать в подкаталоге, путь к которому соответствует полному имени пакета. Например, все файлы классов из пакета `com.horstmann.corejava` должны находиться в подкаталоге `com/horstmann/corejava` (или `com\horstmann\corejava` в Windows). Компилятор размещает файлы классов в той же самой структуре каталогов.

Программы из листингов 4.6 и 4.7 распределены по двум пакетам: класс `PackageTest` принадлежит пакету по умолчанию, а класс `Employee` — пакету `com.horstmann.corejava`. Следовательно, файл `Employee.class` должен находиться в подкаталоге `com/horstmann/corejava`. Иными словами, структура каталогов должна выглядеть следующим образом:

```
.(базовый каталог)
    |__ PackageTest.java
    |__ PackageTest.class
    |__ com/
         |__ horstmann/
              |__ corejava/
                   |__ Employee.java
                   |__ mployee.class
```

Чтобы скомпилировать программу из листинга 4.6, перейдите в каталог, содержащий файл `PackageTest.java`, и выполните следующую команду:

```
javac Package.java
```

Компилятор автоматически найдет файл **`com/horstmann/corejava/Employee.java`** и скомпилирует его.

Рассмотрим более практический пример. В данном примере пакет по умолчанию не используется. Вместо этого классы распределены по разным пакетам (`com.horstmann.corejava` и `com.mycompany`), как показано ниже.

```
. (базовый каталог)
    |__ com/
         |__ horstmann/
         |     |__ corejava/
         |           |__ Employee.java
         |           |___ Employee.class
         |___ mycompany/
               |__ PayrollApp.java
               |__ PayrollApp.class
```

И в этом случае классы следует компилировать и запускать из *базового каталога*, т.е. того каталога, в котором содержится подкаталог `com`, как показано ниже.

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Не следует также забывать, что компилятор работает с *файлами* (при указании имени файла задаются путь и расширение **`.java`**). А интерпретатор Java оперирует *классами* (имя каждого класса указывается в пакете через точку).

Листинг 4.6. Исходный код из файла `PackageTest/PackageTest.java`

```
1  import com.horstmann.corejava.*;
2  // В этом пакете определен класс Employee
3
4  import static java.lang.System.*;
5  /**
6   * В этой программе демонстрируется применение пакетов
7   * @version 1.11 2004-02-19
8   * @author Cay Horstmann
9  */
10 public class PackageTest
11 {
12   public static void main(String[] args)
13   {
14     // здесь не нужно указывать полное имя
15     // класса com.horstmann.corejava.Employee
16     // поскольку используется оператор import
17     Employee harry =
18               new Employee("Harry Hacker", 50000, 1989, 10, 1);
19
20     harry.raiseSalary(5);
21
22     // здесь не нужно указывать полное имя System.out,
23     // поскольку используется оператор static import
24     out.println("name=" + harry.getName() + ",salary="
25                 + harry.getSalary());
26   }
27 }
```

Листинг 4.7. Исходный код из файла `PackageTest/com/horstmann/corejava/Employee.java`

```
1  package com.horstmann.corejava;
2
3  // классы из этого файла входят в указанный пакет
4
5  import java.time.*;
6
7  // операторы import следуют после оператора package
8
9  /**
10  * @version 1.11 2015-05-08
11  * @author Cay Horstmann
12 */
13 public class Employee
14 {
```

```
15   private String name;
16   private double salary;
17   private LocalDate hireDay;
18
19   public Employee(
20          String name, double salary, int year, int month, int day)
21   {
22     this.name = name;
23     this.salary = salary;
24     hireDay = LocalDate.of(year, month, day);
25   }
26
27   public String getName()
28   {
29     return name;
30   }
31
32   public double getSalary()
33   {
34     return salary;
35   }
36
37   public LocalDate getHireDay()
38   {
39     return hireDay;
40   }
41
42   public void raiseSalary(double byPercent)
43   {
44     double raise = salary * byPercent / 100;
45     salary += raise;
46   }
47 }
```

СОВЕТ. Начиная со следующей главы в исходном коде приводимых примеров программ будут использоваться пакеты. Это даст возможность создавать проекты в ИСР по отдельным главам, а не по разделам.

ВНИМАНИЕ! Компилятор не проверяет структуру каталогов. Допустим, исходный файл начинается со следующей директивы:

```
package com.mycompany;
```

Этот файл можно скомпилировать, даже если он не находится в каталоге **com/mycompany**. Исходный файл будет скомпилирован без ошибок, если он *не зависит* от других пакетов. Но при попытке выполнить скомпилированную программу виртуальная машина *не* найдет нужные классы, если пакеты не соответствуют указанным каталогам, а все файлы классов не размещены в нужном месте.

4.7.4. Область действия пакетов

В приведенных ранее примерах кода уже встречались модификаторы доступа `public` и `private`. Открытые компоненты, помеченные ключевым словом `public`, могут использоваться любым классом. А закрытые компоненты, в объявлении которых указано ключевое слово `private`, могут использоваться только тем классом, в котором они были определены. Если же ни один из модификаторов доступа не указан, то компонент программы (класс, метод или переменная) доступен всем методам в том же самом *пакете*.

Обратимся снова к примеру программы из листинга 4.2. Класс `Employee` не определен в ней как открытый. Следовательно, любой другой класс из того же самого пакета (в данном случае — пакета по умолчанию), например класс `EmployeeTest`, может получить к нему доступ. Для классов такой подход следует признать вполне разумным. Но для переменных подобный способ доступа не годится. Переменные должны быть явно обозначены как `private`, иначе их область действия будет по умолчанию расширена до пределов пакета, что, безусловно, нарушает принцип инкапсуляции. В процессе работы над программой разработчики часто забывают указать ключевое слово `private`. Ниже приведен пример из класса `Window`, принадлежащего пакету `java.awt`, который входит в состав JDK.

```
public class Window extends Container
{
  String warningString;
  . . .
}
```

Обратите внимание на то, что у переменной `warningString` отсутствует модификатор доступа `private`! Это означает, что методы всех классов из пакета `java.awt` могут обращаться к ней и изменить ее значение, например, присвоить ей строку `"Trust me!"` (Доверьтесь мне!). Фактически все методы, имеющие доступ к переменной `warningString`, принадлежат классу `Window`, поэтому эту переменную можно было бы смело объявить как закрытую. Вероятнее всего, те, кто писал этот код, торопились и просто забыли указать ключевое слово `private`. (Не станем упоминать имени автора этого кода, чтобы не искать виноватого, — вы сами можете заглянуть в исходный код.)

НА ЗАМЕТКУ! Как ни странно, этот недостаток не был устранен даже после того, как на него было указано в девяти предыдущих изданиях данной книги. Очевидно, что разработчики упомянутого выше пакета не читают эту книгу. Более того, со временем в класс **`Window`** были добавлены новые поля, и снова половина из них не была объявлена как **`private`**.

А может быть, это совсем и не оплошность? Однозначно ответить на этот вопрос нельзя. По умолчанию пакеты не являются открытыми, а это означает, что всякий может добавлять в пакеты свои классы. Разумеется, злонамеренные или невежественные программисты могут написать код, модифицирующий переменные, область действия которых ограничивается пакетом. Например, в ранних версиях Java можно было легко проникнуть в другой класс пакета `java.awt`. Для этого достаточно было начать определение нового класса со следующей строки:

```
package java.awt;
```

а затем разместить полученный файл класса в подкаталоге `java/awt`. В итоге содержимое пакета `java.awt` становилось открытым для доступа. Подобным ловким способом можно было изменить строку предупреждения, отображаемую на экране (рис. 4.9).

В версии 1.2 создатели JDK запретили загрузку классов, определенных пользователем, если их имя начиналось с `"java."`! Разумеется, эта защита не распространяется на ваши собственные классы. Вместо этого вы можете воспользоваться другим механизмом, который называется *герметизацией пакета* и ограничивает доступ к пакету. Произведя герметизацию пакета, вы запрещаете добавлять в него классы. В главе 9 будет показано, как создать архивный JAR-файл, содержащий герметичные пакеты.

Рис. 4.9. Изменение строки предупреждения в окне аплета

4.8. Путь к классам

Как вам должно быть уже известно, классы хранятся в подкаталогах файловой системы. Путь к классу должен совпадать с именем пакета. Кроме того, можно воспользоваться утилитой **`jar`**, чтобы разместить классы в архивном файле формата JAR (архиве Java; в дальнейшем — просто JAR-файле). В одном архивном файле многие файлы классов и подкаталогов находятся в сжатом виде, что позволяет экономить память и сокращать время доступа к ним. Когда вы пользуетесь библиотекой от независимых разработчиков, то обычно получаете в свое распоряжение один JAR-файл или более. В состав JDK также входит целый ряд JAR-файлов, как, например, файл `jre/lib/rt.jar`, содержащий тысячи библиотечных классов. Из главы 9 вы узнаете о том, как создавать собственные JAR-файлы.

СОВЕТ. Для организации файлов и подкаталогов в архивных JAR-файлах используется формат ZIP. Обращаться с файлом **`rt.jar`** и другими JAR-файлами можно с помощью любой утилиты, поддерживающей формат ZIP.

Чтобы обеспечить совместный доступ программ к классам, выполните следующие действия.

1. Разместите файлы классов в одном или нескольких специальных каталогах, например `/home/user/classdir`. Следует иметь в виду, что этот каталог является *базовым* по отношению к дереву пакета. Если потребуется добавить класс `com.horstmann.corejava.Employee`, то файл класса следует разместить в подкаталоге `/home/user/classdir/com/horstmann/corejava`.
2. Разместите все JAR-файлы в одном каталоге, например `/home/user/archives`.
3. Задайте путь к классу. *Путь к классу* — это совокупность всех базовых каталогов, которые могут содержать файлы классов.

В Unix составляющие пути к классу отделяются друг от друга двоеточиями:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

А в Windows они разделяются точками с запятой:

```
c:\clasdir;.;c:\archives\archive.jar
```

В обоих случаях точка обозначает текущий каталог.
Путь к классу содержит:

- имя базового каталога `/home/user/classdir` или `c:\classes`;
- обозначение текущего каталога (`.`);
- имя JAR-файла `/home/user/archives/archive.jar` или `c:\archives\archive.jar`.

Начиная с версии Java SE 6 для обозначения каталога с JAR-файлами можно указывать шаблонные символы подстановки одним из следующих способов:

```
/home/user/classdir:.:/home/user/archives/'*'
```

или

```
c:\classdir;.;c:\archives\*
```

В UNIX шаблонный символ подстановки **`*`** должен быть экранирован, чтобы исключить его интерпретацию в командной оболочке. Все JAR-файлы (но не файлы с расширением **`.class`**), находящиеся в каталоге `archives`, включаются в путь к классам. Файлы из библиотеки рабочих программ (`rt.jar` и другие файлы формата JAR в каталогах `jre/lib` и `jre/lib/ext`) всегда участвуют в поиске классов, поэтому их не нужно включать явно в путь к классам.

ВНИМАНИЕ! Компилятор **`javac`** всегда ищет файлы в текущем каталоге, а загрузчик виртуальной машины **`java`** обращается к текущему каталогу только в том случае, если в пути к классам указана точка (`.`). Если путь к классам не указан, никаких осложнений не возникает — по умолчанию в него включается текущий каталог (`.`). Если же вы задали путь к классам и забыли указать точку, ваша программа будет скомпилирована без ошибок, но выполняться не будет.

В пути к классам перечисляются все каталоги и архивные файлы, которые служат исходными точками для поиска классов. Рассмотрим следующий простой путь к классам:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

Допустим, интерпретатор осуществляет поиск файла, содержащего класс `com.horstmann.corejava.Employee`. Сначала он ищет его в системных файлах, которые хранятся в архивах, находящихся в каталогах `jre/lib` и `jre/lib/ext`. В этих файлах искомый класс отсутствует, поэтому интерпретатор анализирует пути к классам, по которым он осуществляет поиск следующих файлов:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`;
- `com.horstmann/corejava/Employee.class`, начиная с текущего каталога;
- `com.horstmann/corejava/Employee.class` в каталоге `/home/user/archives/archive.jar`.

На поиск файлов компилятор затрачивает больше времени, чем виртуальная машина. Если вы указали класс, не назвав пакета, которому он принадлежит, компилятор сначала должен сам определить, какой именно пакет содержит данный класс. В качестве возможных источников для классов рассматриваются пакеты, указанные в операторах `import`. Допустим, исходный файл содержит приведенные ниже директивы, а также код, в котором происходит обращение к классу `Employee`.

```
import java.util.*
import com.horstmann.corejava.*;
```

Компилятор попытается найти классы `java.lang.Employee` (поскольку пакет `java.lang` всегда импортируется по умолчанию), `java.util.Employee`, `com.horstmann.corejava.Employee` и `Employee` в текущем пакете. Он ищет каждый из этих файлов во всех каталогах, указанных в пути к классам. Если найдено несколько таких классов, возникает ошибка компиляции. (Классы должны быть определены однозначно, и поэтому порядок следования операторов `import` особого значения не имеет.)

Компилятор делает еще один шаг, просматривая *исходные* файлы, чтобы проверить, были ли они созданы позже, чем файлы классов. Если проверка дает положительный результат, исходный файл автоматически перекомпилируется. Напомним, что из других пакетов можно импортировать лишь открытые классы. Исходный файл может содержать только один открытый класс, а кроме того, имена файла и класса должны совпадать. Следовательно, компилятор может легко определить, где находятся исходные файлы, содержащие открытые классы. Из текущего пакета можно импортировать также классы, не определенные как открытые (т.е. `public`). Исходные коды классов могут содержаться в файлах с разными именами. При импорте класса из текущего пакета компилятор проверяет *все* исходные файлы, чтобы выяснить, в каком из них определен требуемый класс.

4.8.1. Указание пути к классам

Путь к классам лучше всего указать с помощью параметра **`-classpath`** (или **`-cp`**) следующим оразом:

```
java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg.java
```

или

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

Вся команда должна быть набрана в одной строке. Лучше всего ввести такую длинную команду в сценарий командной оболочки или командный файл. Применение параметра **`-classpath`** считается более предпочтительным способом установки путей к классам. Альтернативой этому способу служит установка переменной окружения `CLASSPATH`. А далее все зависит от конкретной командной оболочки. Так, в командной оболочке Bourne Again (bash) для установки путей к классам используется следующая команда:

```
export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
```

В командной оболочке C shell для этой цели применяется следующая команда:

```
setenv CLASSPATH /home/user/classdir:.:/home/user/archives/archive.jar
```

А в командной строке Windows — такая команда:

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

Путь к классам действителен до завершения работы командной оболочки.

ВНИМАНИЕ! Некоторые советуют устанавливать переменную окружения **`CLASSPATH`** на постоянной основе. В целом это неудачная идея. Сначала разработчики забывают о глобальных установках, а затем удивляются, когда классы не загружаются правильно. Характерным тому примером служит программа установки приложения QuickTime от компании Apple в Windows. Эта программа глобально устанавливает переменную окружения **`CLASSPATH`** таким образом, чтобы она указывала на нужный ей JAR-файл, не включая в путь текущий каталог. В итоге очень многие программирующие на Java попадают в тупиковую ситуацию, когда их программы сначала компилируются, а затем не запускаются.

ВНИМАНИЕ! Некоторые советуют вообще не указывать путь к классам, сбрасывая все архивные JAR-файлы в каталог **jre/lib/ext**. Это очень неудачное решение по двум причинам. Архивы, из которых другие классы загружаются вручную, работают неправильно, когда они размещены в каталоге расширений (подробнее о загрузчиках классов речь пойдет в главе 9 второго тома данной книги). Более того, программистам свойственно забывать о файлах, которые они разместили там три месяца назад. А затем они недоумевают, почему загрузчик файлов игнорирует их тщательно продуманный путь к классам, загружая на самом деле давно забытые классы из каталога расширений.

4.9. Документирующие комментарии

В состав JDK входит полезное инструментальное средство — утилита **javadoc**, составляющая документацию в формате HTML из исходных файлов. По существу, интерактивная документация на прикладной программный интерфейс API является результатом применения утилиты **javadoc** к исходному коду стандартной библиотеки Java.

Добавив в исходный код комментарии, начинающиеся с последовательности знаков **/****, нетрудно составить документацию, имеющую профессиональный вид. Это очень удобный способ, поскольку он позволяет совместно хранить как код, так и документацию к нему. Если же поместить документацию в отдельный файл, то со временем она перестанет соответствовать исходному коду. В то же время документацию нетрудно обновить, повторно запустив на выполнение утилиту **javadoc**, поскольку комментарии являются неотъемлемой частью исходного файла.

4.9.1. Вставка комментариев

Утилита **javadoc** извлекает сведения о следующих компонентах программы.

- Пакеты.
- Классы и интерфейсы, объявленные как `public`.
- Методы, объявленные как `public` или `protected`.
- Поля, объявленные как `public` или `protected`.

Защищенные компоненты программы, для объявления которых используется ключевое слово `protected`, будут рассмотрены в главе 5, а интерфейсы — в главе 6. Разрабатывая программу, можно (и даже нужно) комментировать каждый из перечисленных выше компонентов. Комментарии размещаются непосредственно перед тем компонентом, к которому они относятся. Комментарии начинаются знаками **/**** и оканчиваются знаками ***/**. Комментарии вида **/** ... */** содержат произвольный текст, после которого следует дескриптор. Дескриптор начинается со знака `@`, например **@author** или `@param`.

Первое предложение в тексте комментариев должно быть кратким *описанием*. Утилита **javadoc** автоматически формирует страницы, состоящие из кратких описаний. В самом тексте можно использовать элементы HTML-разметки, например, `<em>...</em>` — для выделения текста курсивом, `<code>...</code>` — для форматирования текста моноширинного шрифта, `<strong>...</strong>` — для выделения текста полужирным и даже `<img ...>` — для вставки рисунков. Следует, однако, избегать применения заголовков (`<h1>` – `<h6>`) и горизонтальных линий (`<hr>`), поскольку они могут помешать нормальному форматированию документа.

НА ЗАМЕТКУ! Если комментарии содержат ссылки на другие файлы, например рисунки (диаграммы или изображения компонентов пользовательского интерфейса), разместите эти файлы в каталоге **doc-files**, содержащем исходный файл. Утилита **javadoc** скопирует эти каталоги вместе с находящимися в них файлами из исходного каталога в данный каталог, выделяемый для документации. Поэтому в своих ссылках вы должны непременно указать каталог **doc-files**, например ****.

4.9.2. Комментарии к классам

Комментарии к классу должны размещаться после операторов `import`, непосредственно перед определением класса. Ниже приведен пример подобных комментариев.

```
/**
  *Объект класса {@code Card} имитирует игральную карту,
  *например даму червей. Карта имеет масть и ранг
  *(1=туз, 2...10, 11=валет, 12=дама, 13=король).
*/
public class Card
{
...
}
```

НА ЗАМЕТКУ! Начинать каждую строку документирующего комментария звездочкой нет никакой нужды. Например, следующий комментарий вполне корректен:

```
/**
  Объект класса <code>Card</code> имитирует игральную карту,
  например, даму червей. Карта имеет масть и ранг
  (  1=туз, 2...10, 11=валет, 12=дама, 13=король).
*/
```

Но в большинстве ИСР звездочки в начале строки документирующего комментария устанавливаются автоматически и перестраиваются при изменении расположения переносов строк.

4.9.3. Комментарии к методам

Комментарии должны непосредственно предшествовать методу, который они описывают. Кроме дескрипторов общего назначения, можно использовать перечисленные ниже специальные дескрипторы.

- **@param** *описание переменной*

  Добавляет в описание метода раздел параметров. Раздел параметров можно развернуть на несколько строк. Кроме того, можно использовать элементы HTML-разметки. Все дескрипторы `@param`, относящиеся к одному методу, должны быть сгруппированы вместе.

- **@return** *описание*

  Добавляет в описание метода раздел возвращаемого значения. Этот раздел также может занимать несколько строк и допускает форматирование с помощью дескрипторов HTML-разметки.

- **@throws** *описание класса*

  Указывает на то, что метод способен генерировать исключение. Исключения будут рассмотрены в главе 10.

Рассмотрим следующий пример комментариев к методу:

```
/**
 * Увеличивает зарплату работников
 * @param Переменная byPercent содержит величину
 * в процентах, на которую повышается зарплата
 * (например, 10 = 10%).
 * @return Величина, на которую повышается зарплата
*/
public double raiseSalary(double byPercent)
{
  double raise = salary &* byPercent / 100;
  salary += raise;
  return raise;
}
```

4.9.4. Комментарии к полям

Документировать нужно лишь открытые поля. Они, как правило, являются статическими константами. Ниже приведен пример комментария к полю.

```
/**
 * Масть черви
*/
public static final int HEARTS = 1;
```

4.9.5. Комментарии общего характера

В комментариях к классам можно использовать следующие дескрипторы.

- **`@author`** *имя*

  Создает раздел автора программы. В комментариях может быть несколько таких дескрипторов — по одному на каждого автора.

- **`@version`** *текст*

  Создает раздел версии программы. В данном случае *текст* означает произвольное описание текущей версии программы.

При написании любых комментариев, предназначенных для составления документации, можно также использовать следующие дескрипторы.

- **`@since`** *текст*

  Создает раздел начальной точки отсчета. Здесь *текст* означает описание версии программы, в которой впервые был внедрен данный компонент. Например, `@since version 1.7.1`.

- **`@deprecated`** *текст*

  Добавляет сообщение о том, что класс, метод или переменная не рекомендуется к применению. Подразумевается, что после дескриптора **`@deprecated`** следует некоторое выражение. Например:

  ```
  @deprecated Use <code>setVisible(true)</code> instead
  ```

С помощью дескрипторов **`@see`** и **`@link`** можно указывать гипертекстовые ссылки на соответствующие внешние документы или на отдельную часть того же документа, сформированного с помощью утилиты **`javadoc`**.

- **@see** *ссылка*

  Добавляет ссылку в раздел "См. также". Этот дескриптор можно употреблять в комментариях к классам и методам. Здесь *ссылка* означает одну из следующих конструкций:

  ```
  пакет.класс#метка_компонента
  <a href="...">метка</a>
  "текст"
  ```

  Первый вариант чаще всего встречается в документирующих комментариях. Здесь нужно указать имя класса, метода или переменной, а утилита **javadoc** вставит в документацию соответствующую гипертекстовую ссылку. Например, в приведенной ниже строке кода создается ссылка на метод `raiseSalary(double)` из класса `com.horstmann.corejava.Employee`. Если опустить имя пакета или же имя пакета и класса, то комментарии к данному компоненту будут размещены в текущем пакете или классе.

  ```
  @see com.horstmann.corejava.Employee#raiseSalary(double)
  ```

  Обратите внимание на то, что для отделения имени класса от имени метода или переменной служит знак **#**, а не точка. Компилятор **javac** корректно обрабатывает точки, выступающие в роли разделителей имен пакетов, подпакетов, классов, внутренних классов, методов и переменных. Но утилита **javadoc** не настолько развита логически, и поэтому возникает потребность в специальном синтаксисе.

  Если после дескриптора **@see** следует знак **<**, то необходимо указать гипертекстовую ссылку. Ссылаться можно на любой веб-адрес в формате URL, как показано ниже.

  ```
  @see <a href=
      "www.horstmann.com/corejava.html">Web-страница книги Core Java</a>
  ```

  В каждом из рассматриваемых здесь вариантов составления гипертекстовых ссылок можно указать необязательную *метку*, которая играет роль точки привязки для ссылки. Если не указать метку, точкой привязки для ссылки будет считаться имя программы или URL.

  Если после дескриптора **@see** следует знак **"**, то текст отображается в разделе "См. также", как показано ниже.

  ```
  @see "Core Java volume 2"
  ```

  Для комментирования одного и того же элемента можно использовать несколько дескрипторов **@ see**, но их следует сгруппировать вместе.

- По желанию в любом комментарии можно разместить гипертекстовые ссылки на другие классы или методы. Для этого в любом месте документации достаточно ввести следующий специальный дескриптор:

  ```
  {link пакет.класс#метка_элемента}
  ```

  Описание элемента подчиняется тем же правилам, что и для дескриптора **@see**.

4.9.6. Комментарии к пакетам и обзорные

Комментарии к классам, методам и переменным, как правило, размещаются непосредственно в исходных файлах и выделяются знаками `/** ... */`. Но

для формирования комментариев к *пакетам* следует добавить отдельный файл в каталог каждого пакета. Для этого имеются два варианта выбора.

1. Применить HTML-файл под именем `package.html`. Весь текст, содержащийся между дескрипторами `<body> ... </body>`, извлекается утилитой **`javadoc`**.
2. Подготовить файл под именем `package-info.java`. Этот файл должен содержать начальный документирующий комментарий, отделенный символами **`/**`** и **`*/`**, за которым следует оператор `package`, но больше никакого кода или комментариев.

Кроме того, все исходные файлы можно снабдить обзорными комментариями. Они размещаются в файле `overview.html`, расположенном в родительском каталоге со всеми исходными файлами. Весь текст, находящийся между дескрипторами `<body> ... </body>`, извлекается утилитой **`javadoc`**. Эти комментарии отображаются на экране, когда пользователь выбирает вариант Overview (Обзор) на панели навигации.

4.9.7. Извлечение комментариев в каталог

Допустим, что `docDirectory` — это имя каталога, в котором должны храниться HTML-файлы. Для извлечения комментариев в каталог выполните описанные ниже действия.

1. Перейдите в каталог с исходными файлами, которые подлежат документированию. Если документированию подлежат вложенные пакеты, например `com.horstmann.corejava`, перейдите в каталог, содержащий подкаталог `com`. (Именно в этом каталоге должен храниться файл `overview.html`.)
2. Чтобы извлечь документирующие комментарии в указанный каталог из одного пакета, выполните следующую команду:

   ```
   javadoc -d docDirectory имя_пакета
   ```

3. А для того чтобы извлечь документирующие комментарии в указанный каталог из нескольких пакетов, выполните такую команду:

   ```
   javadoc -d docDirectory имя_пакета_1 имя_пакета_2 ...
   ```

4. Если файлы находятся в пакете по умолчанию, вместо приведенных выше команд выполните следующую команду:

   ```
   javadoc -d docDirectory *.java
   ```

Если опустить параметр **`-d docDirectory`**, HTML-файлы документации будут извлечены в текущий каталог, что вызовет путаницу. Поэтому делать этого не рекомендуется.

При вызове утилиты **`javadoc`** можно указывать различные параметры. Например, чтобы включить в документацию дескрипторы `@author` и `@version`, можно выбрать параметры **`-author`** и **`-version`** (по умолчанию они опущены). Параметр **`-link`** позволяет включать в документацию ссылки на стандартные классы. Например, приведенная ниже команда автоматически создаст ссылку на документацию, находящуюся на веб-сайте компании Oracle.

```
javadoc -link http://docs.oracle.com/javase/8/docs/api *.java
```

Если же выбрать параметр **`-linksource`**, то каждый исходный файл будет преобразован в формат HTML (без цветового кодирования, но с номерами строк), а имя

каждого класса и метода превратится в гиперссылку на исходный код. Другие параметры описаны в документации на **javadoc**, доступной по следующему адресу:

```
http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/
```

НА ЗАМЕТКУ! Если требуется выполнить более тонкую настройку, например, составить документацию в формате, отличающемся от HTML, можно разработать свой собственный *доклет* — приложение, позволяющее формировать документацию произвольного вида. Подробнее о таком способе можно узнать на веб-странице по следующему адресу:

http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/doclet/overview.html

4.10. Рекомендации по разработке классов

В завершение этой главы приведем некоторые рекомендации, которые призваны помочь вам в разработке классов в стиле ООП.

1. *Всегда храните данные в переменных, объявленных как* **private**.

   Первое и главное требование: всеми средствами избегайте нарушения инкапсуляции. Иногда приходится писать методы доступа к полю или модифицирующие методы, но предоставлять доступ к полям не следует. Как показывает горький опыт, способ представления данных может изменяться, но порядок их использования изменяется намного реже. Если данные закрыты, их представление не влияет на использующий их класс, что упрощает выявление ошибок.

2. *Всегда инициализируйте данные.*

   В языке Java локальные переменные не инициализируются, но поля в объектах инициализируются. Не полагайтесь на действия по умолчанию, инициализируйте переменные явным образом с помощью конструкторов.

3. *Не употребляйте в классе слишком много простых типов.*

   Несколько связанных между собой полей простых типов следует объединять в новый класс. Такие классы проще для понимания, а кроме того, их легче видоизменить. Например, следующие четыре поля из класса `Customer` нужно объединить в новый класс `Address`:

   ```
   private String street;
   private String city;
   private String state;
   private int zip;
   ```

   Представленный таким образом адрес легче изменить, как, например, в том случае, если требуется указать интернациональные адреса.

4. *Не для всех полей нужно создавать методы доступа и модификации.*

   Очевидно, что при выполнении программы расчета зарплаты требуется получать сведения о зарплате работника, а кроме того, ее приходится время от времени изменять. Но вряд ли придется менять дату его приема на работу после того, как объект сконструирован. Иными словами, существуют поля, которые после создания объекта совсем не изменяются. К их числу относится, в частности, массив сокращенных названий штатов США в классе `Address`.

5. *Разбивайте на части слишком крупные классы.*

   Это, конечно, слишком общая рекомендация: то, что кажется "слишком крупным" одному программисту, представляется нормальным другому. Но если есть очевидная возможность разделить один сложный класс на два класса попроще, то воспользуйтесь ею. (Опасайтесь, однако, другой крайности. Вряд ли оправданы десять классов, в каждом из которых имеется только один метод.) Ниже приведен пример неудачного составления класса.

```
public class CardDeck // неудачная конструкция
{
  private int[] value;
  private int[] suit;

  public CardDeck() { . . . }
  public void shuffle() { . . . }
  public int getTopValue() { . . . }
  public int getTopSuit() { . . . }
  public void draw() { . . . }
}
```

   На самом деле в этом классе реализованы два разных понятия: во-первых, *карточный стол* с методами `shuffle()` (тасование) и `draw()` (раздача) и, во-вторых, *игральная карта* с методами для проверки ранга и масти карты. Разумнее было бы ввести класс `Card`, представляющий отдельную игральную карту. В итоге имелись бы два класса, каждый из которых отвечал бы за свое, как показано ниже.

```
public class CardDeck
{
  private Card[] cards;
  public CardDeck() { . . . }
  public void shuffle() { . . . }
  public Card getTop() { . . . }
  public void draw() { . . . }
}

public class Card
{
  private int value;
  private int suit;

  public Card(int aValue, int aSuit) { . . . }
  public int getValue() { . . . }
  public int getSuit() { . . . }
}
```

6. *Выбирайте для классов и методов осмысленные имена, ясно указывающие на их назначение.*

   Классы, как и переменные, следует называть именами, отражающими их назначение. (В стандартной библиотеке имеются примеры, где это правило нарушается. Например, класс `Date` описывает время, а не дату.)

   Удобно принять следующие условные обозначения: имя класса должно быть именем существительным (`Order`) или именем существительным, которому предшествует имя прилагательное (`RushOrder`) или деепричастие (`BillingAddress`). Как правило, методы доступа должны начинаться словом

get, представленным строчными буквами (getSalary), а модифицирующие методы — словом set, также представленным строчными буквами (setSalary).

7. *Отдавайте предпочтение неизменяемым классам.*

   Класс LocalDate и прочие классы из пакета java.time являются неизменяемыми. Это означает, что они не содержат методы, способные видоизменить (т.е. модифицировать) состояние объекта. Вместо модификации объектов такие методы, как, например, plusDays(), возвращают новые объекты с видоизмененным состоянием.

   Трудность модификации состоит в том, что она может происходить параллельно, когда в нескольких потоках исполнения предпринимается одновременная попытка обновить объект. Результаты такого обновления непредсказуемы. Если же классы являются неизменяемыми, то их объекты можно благополучно разделять среди нескольких потоков исполнения.

   Таким образом, классы рекомендуется делать неизменяемыми при всякой удобной возможности. Это особенно просто сделать с классами, представляющими значения вроде символьной строки или момента времени. А в результате вычислений просто получаются новые значения, а не обновляются уже существующие.

   Разумеется, не все классы должны быть неизменяемыми. Было бы, например, нелепо, если бы метод raiseSalary() возвращал новый объект типа Employee, когда поднимается заплата работника.

В этой главе были рассмотрены основы создания объектов и классов, которые делают Java объектным языком. Но для того чтобы быть действительно объектно-ориентированным, язык программирования должен также поддерживать наследование и полиморфизм. О реализации этих принципов ООП в Java речь пойдет в следующей главе.