ГЛАВА 4

Язык Java

С этой главы начинается наш вводный курс синтаксиса языка Java. Поскольку все читатели отличаются по опыту программирования, нам было нелегко подобрать подходящий уровень для любой аудитории. Мы постарались найти баланс между подробным обзором для начинающих с примерами синтаксиса и предоставлением дополнительной информации для более опытных читателей, чтобы они могли быстро оценить различия между Java и другими языками. Так как синтаксис Java является производным от C, мы иногда приводим сравнения с функциональностью этого языка, но опыт программирования на C от вас не требуется. Далее, в главе 5 рассматривается объектно-ориентированная сторона Java, а также завершается обсуждение базовых возможностей языка. В главе 7 рассматриваются обобщения — механизм, который расширяет возможности типов и позволяет реализовать некоторые виды классов более гибко и безопасно. После этого мы займемся различными API языка Java и покажем, что можно делать с их помощью. Оставшаяся часть книги заполнена примерами решений полезных задач из нескольких областей. Если после вводных глав у вас останутся вопросы, то мы надеемся, что ответы на них вы найдете в примерах кода. И конечно, вы всегда сможете повышать квалификацию! Мы постараемся давать ссылки на другие ресурсы, которые помогут читателям, желающим продолжить изучение Java за рамками рассматриваемых тем.

Для тех, кто только начинает знакомство с программированием, постоянным попутчиком должен стать интернет. Бесчисленные сайты, статьи «Википедии», публикации в блогах и сообщество Stack Overflow помогут разобраться в конкретных темах и найти ответы на возникающие вопросы. Например, в этой книге рассматривается язык Java и объясняется, как начать писать на нем полезные программы, но книга не рассказывает о таких фундаментальных концепциях программирования, как *алгоритмы*. Впрочем, эти концепции естественным образом вошли в обсуждения и примеры кода, а ссылки на сторонние ресурсы позволят вам закрепить в памяти некоторые подробности и заполнить вынужденные пробелы.

Кодирование текста

Java — язык для интернета. Интернет-сообщество говорит и пишет на множестве разных человеческих языков, поэтому они должны поддерживаться и в программах, написанных на Java. Один из способов интернационализации в Java основан на наборе символов Unicode («Юникод») — международном стандарте, поддерживающем символы большинства существующих языков[1]. В современных версиях Java символьные и строковые данные хранятся в соответствии со стандартом Unicode 6.0, в котором для представления каждого символа используются как минимум два байта.

Исходный код Java можно писать в формате Unicode и сохранять во множестве разных кодировок, от простой двоичной формы до символов Unicode, закодированных в ASCII. Это делает язык Java удобным для программистов из разных стран. Они могут использовать свои родные языки в именах классов, методов и переменных, а также в тексте, выводимом в приложениях.

Тип Java `char` и класс `String` обладают встроенной поддержкой Unicode. Во внутреннем представлении текст хранится в формате `char[]` или `byte[]`; но язык Java и API работают с ними прозрачно для вас, поэтому вам, как правило, не придется об этом думать. Unicode хорошо сочетается с ASCII (это самая распространенная кодировка символов для английского языка). Первые 256 символов определены как идентичные первым 256 символам кодировки ISO 8859-1 (Latin-1), так что Unicode фактически обладает обратной совместимостью с самыми распространенными англоязычными кодировками. Кроме того, кодировка UTF-8, одна из самых популярных для Unicode, сохраняет значения ASCII в однобайтовой форме. Она используется по умолчанию в скомпилированных файлах классов Java, так что английский текст в памяти хранится компактно.

Многие платформы не способны отображать все символы Unicode, определенные в настоящее время. Поэтому при написании программ на Java можно использовать специальные служебные последовательности (escape-последовательности) Unicode. Символ Unicode может представляться служебной последовательностью следующего вида:

```
\uxxxx
```

Здесь `xxxx` — это последовательность из 1–4 шестнадцатеричных цифр. Она обозначает символ Unicode в кодировке ASCII. Также эта форма используется в Java для вывода символов Unicode в тех средах, где они не поддерживаются.

---

[1] За дополнительной информацией о Unicode обращайтесь на сайт http://www.unicode.org. По иронии судьбы одним из алфавитов, обозначенных как «устаревшие и архаичные» и в настоящее время не поддерживаемых в Unicode, является яванский — исторический язык жителей острова Ява (Java).

Кроме того, в Java есть классы для чтения и записи символьных потоков Unicode в конкретных кодировках, включая UTF-8.

Unicode, как и многие долговечные стандарты в мире технологий, изначально проектировался с запасом: считалось, что ни в какой мыслимой кодировке не может быть более 64 К (65 536) символов. Но в итоге даже этого оказалось мало. Сейчас получили широкое распространение кодировки UTF-32. В частности, символы «эмодзи», часто встречающиеся в приложениях-мессенджерах, кодируются за пределами стандартного диапазона символов Unicode (например, канонический смайлик кодируется в Unicode значением 1F600). Java поддерживает для таких символов многобайтовые служебные последовательности UTF-16. Не все платформы, поддерживающие Java, совместимы с эмодзи; попробуйте запустить `jshell`, чтобы узнать, поддерживаются ли символы-эмодзи на вашем компьютере (рис. 4.1).

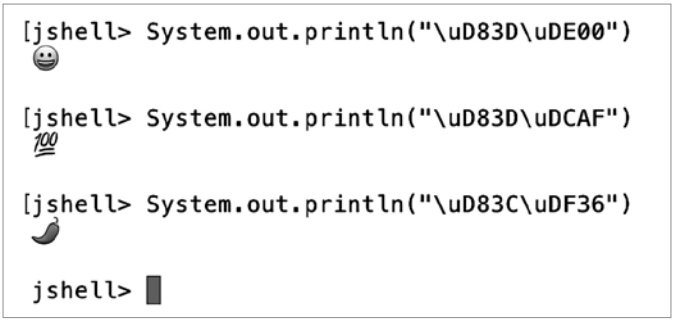

Рис. 4.1. Вывод эмодзи в приложении «Терминал» из macOS

Впрочем, с такими символами необходима осторожность. Нам пришлось показать скриншот, чтобы вы увидели, что эти милые картинки отображаются в `jshell` на Mac. Но если запустить в этой же операционной системе десктопное приложение Java с классами `JFrame` и `JLabel`, о которых мы рассказывали в главе 3, то вы получите результат, показанный на рис. 4.2.

```
jshell> import javax.swing.*

jshell> JFrame f = new JFrame("Emoji Test")
f ==>
javax.swing.JFrame[frame0
,0,23,0x0,invalid,hidden ...
=true]

jshell> JLabel l = new JLabel("Hi \uD83D\uDE00")
l ==> javax.swing.JLabel[,
0,0,0x0,invalid,alignmentX=0. ...
=CENTER]
```

```
jshell> f.add(l)
$12 ==> javax.swing.JLabel[,0,0,0x0,invalid,alignmentX= ...
rticalTextPosition=CENTER]

jshell> f.setSize(300,200)

jshell> f.setVisible(true)
```

Это не означает, что вы не можете использовать или поддерживать эмодзи в своих приложениях, — просто надо знать о различиях в функциональности вывода. Следите за тем, чтобы не создать проблем у пользователей, которые будут запускать ваши приложения.

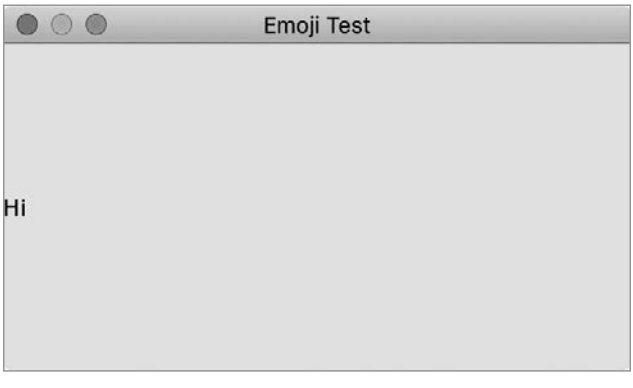

Рис. 4.2. Эмодзи не отображаются в JFrame

Комментарии

Java поддерживает как *блочные комментарии* в стиле C, заключенные в маркеры `/*` и `*/`, так и *строчные комментарии* в стиле C++, обозначаемые маркером `//`:

```
/*  Это
        блочный комментарий
            (в несколько строк)    */

// А это строчный комментарий (в одну строку)

// И это // тоже строчный комментарий
```

Блочные комментарии имеют как начальный, так и конечный маркер; они могут содержать большие объемы текста. Но блочные комментарии не должны быть вложенными, то есть попытка разместить блочный комментарий внутри другого блочного комментария будет ошибкой с точки зрения компилятора. Строчные

комментарии имеют только начальный маркер и завершаются в конце строки; дополнительные маркеры `//` внутри строки ни на что не влияют. Строчные комментарии удобны для включения кратких примечаний в методы; они не конфликтуют с блочными комментариями, так что вы можете закомментировать большие фрагменты кода, в которые они вложены.

Комментарии javadoc

Блочные комментарии, начинающиеся с символов `/**`, представляют собой специальные *doc-комментарии* (документирующие комментарии). Они предназначены для извлечения автоматическими генераторами документации (например, программой `javadoc` из JDK или системами контекстных подсказок во многих IDE). Doc-комментарий завершается конечным маркером `*/`, как и обычный блочный комментарий. Внутри doc-комментария все строки, начинающиеся с символа `@`, интерпретируются как специальные инструкции для генератора документации, передающие ему информацию об исходном коде. По общепринятым соглашениям в начале строк doc-комментария обычно добавляют символ *, как показано в следующем примере, но это не обязательно. Все начальные пробелы и символы * в строках игнорируются:

```
/**
 * Пожалуй, этот класс - самая потрясающая штука, которую вы
 * увидите в своей жизни. Позвольте мне пояснить свое
 * личное видение и причины для его создания.
 * <p>
 * Все началось еще тогда, когда я был ребенком и рос на улицах
 * Айдахо. Спрос на картофель был заоблачным, и жизнь была прекрасна...
 *
 * @see PotatoPeeler
 * @see PotatoMasher
 * @author John 'Spuds' Smith
 * @version 1.00, 19 Nov 2019
 */
class Potato {
```

Программа `javadoc` создает документацию классов в формате HTML, читая исходный код и извлекая встроенные комментарии и теги, начинающиеся с символа `@`. В данном примере теги включают в документацию класса информацию об авторе и версии. Теги `@see` генерируют гипертекстовые ссылки в документации класса.

Компилятор тоже проверяет doc-комментарии. В частности, его интересует тег `@deprecated`, который означает, что метод объявлен устаревшим и лучше не вызывать его в новых программах. Тот факт, что метод является устаревшим, отмечается в скомпилированном файле класса, поэтому при обнаружении в коде устаревших методов будут генерироваться предупреждения (даже если исходный код недоступен).

Doc-комментарии могут располагаться над определениями классов, методов и переменных, но некоторые теги могут быть неприменимы к некоторым из них. Например, тег `@exception` может применяться только к методам. В табл. 4.1 приведена сводка тегов, поддерживаемых в doc-комментариях.

Таблица 4.1. Теги в doc-комментариях

Тег	Описание	Применение
@see	Имя связанного класса	Класс, метод или переменная
@code	Содержимое исходного кода	Класс, метод или переменная
@link	Связанный URL-адрес	Класс, метод или переменная
@author	Имя автора	Класс
@version	Строка с версией	Класс
@param	Имя и описание параметра	Метод
@return	Описание возвращаемого значения	Метод
@exception	Имя и описание исключения	Метод
@deprecated	Объявление элемента как устаревшего	Класс, метод или переменная
@since	Версия API, в которой элемент был добавлен	Переменная

Теги javadoc как метаданные

Теги javadoc в doc-комментариях представляют собой *метаданные*, относящиеся к исходному коду; другими словами, они добавляют описательную информацию о структуре или содержании кода, которая формально не является частью приложения. Некоторые дополнительные инструменты расширяют концепцию тегов в стиле javadoc, позволяя включать в программы Java и другие виды метаданных, которые передаются вместе со скомпилированным кодом и могут легко использоваться приложением, воздействуя на его компиляцию или на логику работы во время выполнения. *Аннотации* Java предоставляют более формальный и расширяемый способ добавления метаданных в классы Java, методы и переменные. Эти метаданные также доступны на стадии выполнения.

Аннотации

Префикс `@` реализует в Java и другую функциональность, которая на первый взгляд напоминает функциональность тегов. *Аннотации* используются в Java для пометки контента, который должен обрабатываться специальным образом. Аннотации применяются к коду **за пределами** комментариев. Информация,

предоставляемая аннотацией, может быть полезна компилятору или IDE. Например, аннотация `@SuppressWarnings` заставляет компилятор (и часто также вашу IDE) скрывать предупреждения о таких потенциальных проблемах, как недоступный код. Когда мы займемся созданием более интересных классов в разделе «Нетривиальное проектирование классов», с. 189, вы увидите, что ваша IDE добавляет в код аннотации `@Overrides`. Они заставляют компилятор выполнять некоторые дополнительные проверки, которые способствуют написанию корректного кода и выявлению ошибок до того, как программа будет запущена вами или другими пользователями.

Вы можете создавать собственные аннотации для работы с другими инструментами и фреймворками. Хотя углубленное изучение аннотаций выходит за рамки этой книги, мы воспользуемся некоторыми очень удобными аннотациями при изучении веб-программирования в главе 12.

Переменные и константы

Итак, вы научились комментировать свой код. Это необходимо для того, чтобы он был наглядным и простым в сопровождении. Теперь нам пора сосредоточиться на самом коде, то есть на том, что компилируется. Все программирование сводится к работе с кодом. Почти во всех языках важная часть кода содержится в *переменных* и *константах*, которые упрощают работу программиста. В Java есть как переменные, так и константы. В переменных хранится та информация, которую вы собираетесь изменять и затем повторно использовать (или информация, неизвестная заранее: например, адрес электронной почты пользователя). В константах хранится информация, которая не должна изменяться. Примеры переменных и констант встречались вам даже в наших маленьких вводных программах. Вспомните простую графическую надпись из раздела «HelloJava» на с. 66:

```
import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame( "Hello, Java!" );
    JLabel label = new JLabel( "Hello, Java!", JLabel.CENTER );
    frame.add(label);
    frame.setSize( 300, 300 );
    frame.setVisible( true );
  }
}
```

В этом фрагменте `frame` — переменная. В строке 5 она заполняется новым экземпляром класса `JFrame`. Затем тот же экземпляр повторно используется

в строке 7 для добавления надписи. В строке 8 эта переменная снова нужна нам, чтобы установить размер окна, а в строке 9 — чтобы перевести окно в видимое состояние. Повторное использование данных — это та область, в которой переменные по-настоящему проявляют себя.

В строке 6 есть константа `JLabel.CENTER`. Константы содержат значения, которые остаются неизменными во время выполнения программы. Казалось бы, зачем хранить таким образом те значения, которые не будут изменяться? Почему бы просто не вписывать их в код каждый раз, когда они понадобятся? Дело в том, что автор кода может выбирать имена для всех констант, что сразу приносит пользу: значения дополняются содержательными описаниями. Возможно, смысл имени `JLabel.CENTER` все еще не совсем очевиден, но слово `CENTER` по крайней мере намекает на то, что здесь происходит.

Кроме того, именованные константы упрощают внесение изменений. Например, если в вашем коде указано предельное количество единиц какого-то используемого ресурса, то скорректировать этот предел будет намного проще, если для этого достаточно изменить инициализированное значение константы. Если бы для этой же цели использовался числовой литерал, например `5`, то вам пришлось бы выискивать во всех файлах Java все вхождения `5` и изменять их каждый раз, когда конкретный экземпляр `5` действительно относится к ограничению соответствующего ресурса. Такой ручной поиск с заменой крайне ненадежен, не говоря уже об однообразии работы.

Типы и исходные значения переменных и констант более подробно описываются далее, в следующем разделе. Как обычно, не стесняйтесь использовать `jshell`, чтобы найти и исследовать некоторые из этих подробностей самостоятельно! Впрочем, помните, что из-за ограничений интерпретатора вы не сможете объявлять собственные константы верхнего уровня в `jshell`. Но вы можете использовать константы, определенные для классов (такие, как упомянутая выше `JLabel.CENTER`), а также определять константы в ваших собственных классах, вводимых в `jshell`. Класс `Math` содержит множество разных математических функций и константу, представляющую число π. Попробуем вычислить и сохранить площадь круга в переменной, а затем продемонстрировать, что повторное присваивание констант не работает.

```
jshell> double radius = 42.0;
radius ==> 42.0

jshell> Math.PI
$2 ==> 3.141592653589793

jshell> Math.PI = 3;
|  Error:
|  cannot assign a value to final variable PI
|  Math.PI = 3;
|  ^-----^
```

```
jshell> double area = Math.PI * radius * radius;
area ==> 5541.769440932396

jshell> radius = 6;
radius ==> 6.0

jshell> area = Math.PI * radius * radius;
area ==> 113.09733552923255

jshell> area
area ==> 113.09733552923255
```

Обратите внимание на ошибку компилятора при попытке присвоить числу π значение 3. Также обратите внимание на то, что значения `radius` и `area` **можно** менять после объявления и инициализации. Тем не менее в любой момент времени переменная может хранить только одно значение. В переменной `area` остается только последний вычисленный результат.

Типы

Система типов в языке программирования описывает, как его элементы данных (только что упоминавшиеся переменные и константы) связываются со своими блоками памяти и как они связываются друг с другом. В языке со статической типизацией (например, C или C++) тип элемента данных представляет собой простой неизменяемый атрибут, который часто соответствует некоторому аппаратному понятию, например регистру или значению указателя. В более динамических языках (таких, как Smalltalk или Lisp) переменные могут обозначать произвольные элементы и даже изменять свои типы на протяжении своего срока жизни. На проверку того, что происходит в этих языках во время выполнения кода, расходуются значительные ресурсы. Языки сценариев, такие как Perl, достигают простоты использования благодаря радикально упрощенной системе типов, в которой переменные могут хранить только некоторые элементы данных, а значения объединяются в общее представление (например, строки).

В Java сочетаются многие лучшие свойства языков как со статической, так и с динамической типизацией. Как и в языках со статической типизацией, каждая переменная и каждый программный элемент в Java имеют тип, известный на стадии компиляции, поэтому исполнительной системе обычно не нужно проверять корректность присваиваний между типами во время выполнения кода. Кроме того, в отличие от традиционных C и C++, Java контролирует информацию об объектах во время выполнения и использует ее для реализации полноценного динамического поведения. Код Java может загружать новые типы во время выполнения и использовать их объектно-ориентированными способами, допуская