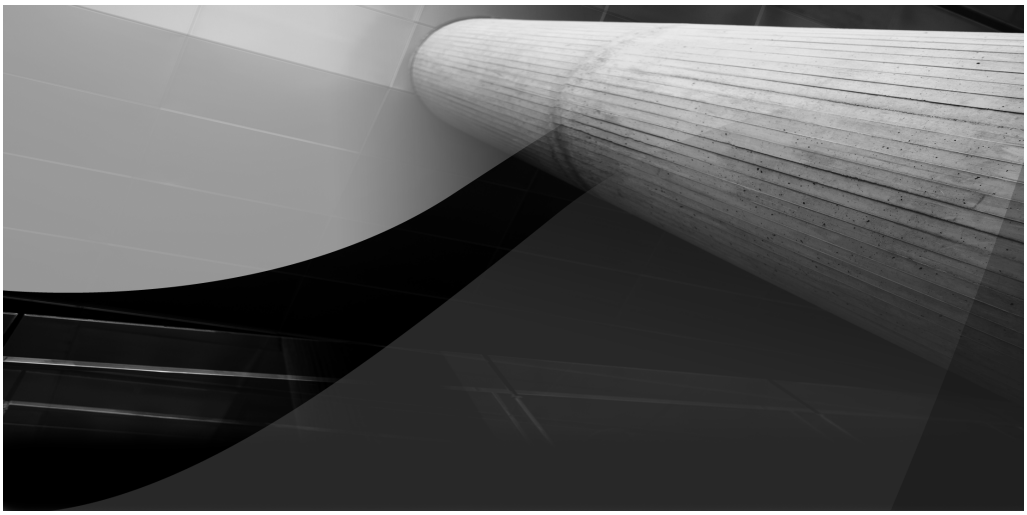

Глава 3

Управляющие операторы

Основные навыки и понятия

- Ввод символов с клавиатуры
- Полная форма условного оператора `if`
- Применение оператора `switch`
- Полная форма цикла `for`
- Применение цикла `while`
- Применение цикла `do-while`
- Применение оператора `break` для выхода из цикла
- Использование оператора `break` в качестве оператора `goto`
- Применение оператора `continue`
- Вложенные циклы

В этой главе вы ознакомитесь с операторами, управляющими ходом выполнения программы. Существуют три категории управляющих операторов: операторы *выбора*, к числу которых относятся операторы `if` и `switch`, *итерационные* операторы, в том числе операторы цикла `for`, `while`, `do-while`, а также операторы *перехода*, включая `break`, `continue` и `return`. Все эти управляющие операторы, кроме оператора `return`, обсуждаемого далее в книге, подробно рассматриваются в этой главе, в начале которой будет показано, каким образом организуется простой ввод данных с клавиатуры.

Ввод символов с клавиатуры

Прежде чем приступать к рассмотрению управляющих операторов в Java, уделим немного внимания средствам, которые позволяют писать интерактивные программы. В рассмотренных до сих пор примерах программ данные *выводились* на экран, но у пользователя не было возможности *вводить* данные. В этих программах, в частности, применялся консольный вывод, но не консольный ввод (с клавиатуры). И объясняется это тем, что возможности ввода данных с клавиатуры в Java опираются на языковые средства, рассматриваемые далее в этой книге. Кроме того, большинство реальных программ на Java и апплетов имеют графический и оконный, а не консольный интерфейс. Именно по этим причинам консольный ввод нечасто применяется в примерах программ, представленных в данной книге. Но имеется один вид консольного ввода, который реализуется очень просто. Это чтение символов с клавиатуры. А поскольку ввод символов применяется в ряде примеров, представленных в этой главе, мы и начнем ее с обсуждения данного вопроса.

Для чтения символа с клавиатуры достаточно вызвать метод `System.in.read()`, где `System.in` — объект ввода (с клавиатуры), дополняющий объект вывода `System.out`. Метод `read()` ожидает нажатия пользователем клавиш, после чего возвращает результат. Возвращаемый им символ представлен целочисленным значением, и поэтому, прежде чем присвоить его символьной переменной, следует выполнить явное его приведе-

ние к типу char. По умолчанию данные, вводимые с консоли, буферизуются *построчно*. Под термином *буфер* здесь подразумевается небольшая область памяти, выделяемая для хранения символов перед тем, как они будут прочитаны программой. В данном случае в буфере хранится целая текстовая строка, и поэтому для передачи программе любого введенного с клавиатуры символа следует нажать клавишу <Enter>. Ниже приведен пример программы, читающей символы, вводимые с клавиатуры.

```
// Чтение символа с клавиатуры.
class KbIn {
  public static void main(String args[])
    throws java.io.IOException {

      char ch;

      System.out.print("Press a key followed by ENTER: ");
      // Ввод символа с клавиатуры.
      ch = (char) System.in.read(); // получить значение типа char

      System.out.println("Your key is: " + ch);
  }
}
```

Выполнение этой программы может дать, например, следующий результат:

```
Press a key followed by ENTER: t
Your key is: t
```

Обратите внимание на то, что метод main() начинается со следующих строк кода:

```
public static void main(String args[])
  throws java.io.IOException {
```

В рассматриваемой здесь программе применяется метод System.in.read(), и поэтому в ее код следует ввести оператор throws java.io.IOException. Этот оператор требуется для обработки ошибок, которые могут возникнуть в процессе ввода данных. Он является частью механизма обработки исключений в Java, более подробно рассматриваемого в главе 9. А до тех пор не обращайте особого внимания на этот оператор, принимая во внимание лишь его назначение.

Построчная буферизация вводимых данных средствами System.in часто приводит к недоразумениям. При нажатии клавиши <Enter> в поток ввода записывается последовательность, состоящая из символов возврата каретки и перевода строки. Эти символы ожидают чтения из буфера ввода. Поэтому в некоторых приложениях, возможно, потребуется удалить символы возврата каретки и перевода строки, прежде чем переходить к следующей операции ввода. Для этого достаточно прочитать их из буфера ввода. Соответствующий пример реализации подобного решения на практике будет представлен далее в главе.

Условный оператор if

Этот условный оператор уже был представлен в главе 1, а здесь он будет рассмотрен более подробно. Ниже приведена полная форма условного оператора if.

```
if(условие) оператор;
else оператор;
```

где *условие* — это некоторое условное выражение, а *оператор* — адресат операторов `if` и `else`. Оператор `else` не является обязательным. Адресатами обоих операторов, `if` и `else`, могут также служить блоки операторов. Ниже приведена общая форма условного оператора `if`, в котором используются блоки операторов.

```
if(условие)
{
  последовательность операторов
}
else
{
  последовательность операторов
}
```

Если условное выражение оказывается истинным, то выполняется адресат оператора `if`. В противном случае выполняется адресат оператора `else`, если таковой существует. Но одновременно не может выполняться и то и другое. Условное выражение, управляющее оператором `if`, должно давать результат типа `bool`.

Для того чтобы продемонстрировать применение оператора `if` (и ряда других управляющих операторов) на практике, разработаем простую игру, основанную на угадывании. Возможно, она понравится вашим детям. В первой версии этой игры программа предложит пользователю угадать задуманную букву от **`A`** до **`Z`**. Если пользователь правильно угадает букву и нажмет на клавиатуре соответствующую клавишу, программа выведет сообщение `** Right **` (Правильно). Ниже приведен исходный код программы, реализующей эту игру.

```
// Игра в угадывание букв.
class Guess {
  public static void main(String args[])
    throws java.io.IOException {
      char ch, answer = 'S';

       System.out.println("I'm thinking of a letter between A and Z.");
       System.out.print("Can you guess it: ");

       ch = (char) System.in.read(); // прочитать символ с клавиатуры

       if(ch == answer) System.out.println("** Right **");
    }
}
```

Эта программа выводит на экран сообщение с предложением угадать букву, а затем читает символ с клавиатуры. Используя условный оператор `if`, она сравнивает введенный символ с правильным ответом (в данном случае это буква **`S`**). Если введена буква **`S`**, то отображается сообщение об угадывании буквы. Опробуя эту программу, не забывайте, что угадываемую букву следует вводить в верхнем регистре.

В следующей версии программы оператор `else` используется для вывода сообщения о том, что буква не была угадана.

```
// Игра в угадывание букв, вторая версия
class Guess2 {
  public static void main(String args[])
    throws java.io.IOException {
```

```
    char ch, answer = 'S';

    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");

    ch = (char) System.in.read(); // ввести символ с клавиатуры

    if(ch == answer) System.out.println("** Right **");
    else System.out.println("...Sorry, you're wrong.");
  }
}
```

Вложенные условные операторы `if`

Вложенные операторы `if` представляют собой условные операторы `if`, являющиеся адресатами оператора `else`. Подобные условные операторы очень часто встречаются в программах. Но, пользуясь ими, следует помнить, что в `Java` оператор `else` всегда связан с ближайшим к нему оператором `if`, находящимся в том же кодовом блоке и не связанном с другим оператором `else`. Рассмотрим следующий пример:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d;
  else a = c; // этот оператор else относится к оператору if(k > 100)
}
else a = d; // а этот оператор else относится к оператору if(i == 10)
```

Как следует из комментариев к приведенному выше фрагменту кода, последний оператор `else` не имеет отношения к оператору `if(j < 20)`, поскольку он не находится с ним в одном кодовом блоке, несмотря на то, что это ближайший оператор `if`, не имеющий себе в пару оператор `else`. Следовательно, последний оператор `else` относится к оператору `if(i == 10)`. А в кодовом блоке оператор `else` связан с оператором `if(k > 100)`, поскольку это самый близкий из всех находящихся к нему операторов `if` в том же самом блоке.

Используя вложенные условные операторы `if`, можно усовершенствовать игру, рассматриваемую здесь в качестве примера. Теперь при неудачной попытке угадать букву пользователю предоставляется дополнительная информация, подсказывающая, насколько он далек от правильного ответа.

```
// Игра в угадывание букв, третья версия.
class Guess3 {
  public static void main(String args[])
    throws java.io.IOException {

    char ch, answer = 'S';

    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");

    ch = (char) System.in.read(); // ввести символ с клавиатуры

    if(ch == answer) System.out.println("** Right **");
    else {
```

```
        System.out.print("...Sorry, you're ");
        // вложенный оператор if
        if(ch < answer) System.out.println("too low");
        else System.out.println("too high");
      }
    }
}
```

Выполнение этой программы может дать, например, следующий результат:

```
I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
```

Многоступенчатая конструкция if-else-if

В программировании часто применяется *многоступенчатая конструкция* `if-else-if`, состоящая из вложенных уловных операторов `if`. Ниже приведена ее общая форма.

```
if(условие)
  оператор;
else if (условие)
  оператор;
else if (условие)
  оператор;
.
.
.
else
  оператор;
```

Условные выражения в такой конструкции вычисляются сверху вниз. Как только обнаружится истинное условие, выполняется связанный с ним оператор, а все остальные операторы в многоступенчатой конструкции опускаются. Если ни одно из условий не является истинным, то выполняется последний оператор `else`, который зачастую служит в качестве условия, устанавливаемого по умолчанию. Когда же последний оператор `else` отсутствует, а все остальные проверки по условию дают ложный результат, никаких действий вообще не выполняется.

Ниже приведен пример программы, демонстрирующий применение многоступенчатой конструкции `if-else-if`.

```
// Демонстрация многоступенчатой конструкции if-else-if.
class Ladder {
  public static void main(String args[]) {
   int x;

    for(x=0; x<6; x++) {
      if(x==1)
        System.out.println("x is one");
      else if(x==2)
        System.out.println("x is two");
      else if(x==3)
        System.out.println("x is three");
      else if(x==4)
        System.out.println("x is four");
```

```
      else
      // Условие, выполняемое по умолчанию.
      System.out.println("x is not between 1 and 4");
    }
  }
}
```

Выполнение этой программы дает следующий результат:

```
x is not between 1 and 4
x is one
x is two
x is three
x is four
x is not between 1 and 4
```

Как видите, устанавливаемый по умолчанию условный оператор `else` выполняется лишь в том случае, если проверки по условию всех предыдущих операторов `if` дают ложный результат.

Оператор switch

Вторым оператором выбора в Java является оператор `switch`, который обеспечивает многонаправленное ветвление программы. Следовательно, этот оператор позволяет сделать выбор среди нескольких альтернативных вариантов дальнейшего выполнения программы. Несмотря на то что многонаправленная проверка может быть организована с помощью последовательного ряда вложенных условных операторов `if`, во многих случаях более эффективным оказывается применение оператора `switch`. Этот оператор действует следующим образом. Значение выражения последовательно сравнивается с константами выбора из заданного списка. Как только будет обнаружено совпадение с одним из условий выбора, выполняется последовательность операторов, связанных с этим условием. Ниже приведена общая форма оператора `switch`.

```
switch(выражение) {
  case константа1:
    последовательность операторов
    break;
case константа2:
    последовательность операторов
    break;
  case константа3:
    последовательность операторов
    break;
.
.
.
  default:
    последовательность операторов
}
```

В версиях Java, предшествующих JDK 7, *выражение*, управляющее оператором `switch`, должно быть типа `byte`, `short`, `int`, `char` или перечислением. (Подробнее о перечислениях речь пойдет в главе 12.)

Начиная с версии JDK 7, *выражение* может относиться к типу `String`. Это означает, что в современных версиях Java для управления оператором `switch` можно пользоваться символьной строкой. (Этот прием программирования демонстрируется в главе 5 при рассмотрении класса `String`.) Но зачастую в качестве выражения, управляющего оператором `switch`, вместо сложного выражения употребляется простая переменная.

Последовательность операторов из ветви `default` выполняется в том случае, если ни одна из констант выбора не совпадает с заданным выражением. Ветвь `default` не является обязательной. Если же она отсутствует и выражение не совпадает ни с одним из условий выбора, то никаких действий вообще не выполняется. Если же происходит совпадение с одним из условий выбора, то выполняются операторы, связанные с этим условием, вплоть до оператора `break`.

Ниже приведен пример программы, демонстрирующий применение оператора `switch`.

```
// Демонстрация оператора switch.
class SwitchDemo {
  public static void main(String args[]) {
    int i;

    for(i=0; i<10; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero");
          break;
        case 1:
          System.out.println("i is one");
          break;
        case 2:
          System.out.println("i is two");
          break;
        case 3:
          System.out.println("i is three");
          break;
        case 4:
          System.out.println("i is four");
          break;
        default:
          System.out.println("i is five or more");
      }
  }
}
```

Результат выполнения данной программы выглядит следующим образом:

```
i is zero
i is one
i is two
i is three
i is four
i is five or more
i is five or more
i is five or more
```

```
i is five or more
i is five or more
```

Как видите, на каждом шаге цикла выполняются операторы, связанные с совпадающей константой выбора в одной из ветвей `case`, в обход всех остальных ветвей. Когда же значение переменной `i` становится равным или больше пяти, оно не совпадает ни с одной из констант выбора, и поэтому управление получает выражение, следующее за оператором `default`.

Формально оператор `break` может отсутствовать, но, как правило, в реальных приложениях он применяется. При выполнении оператора `break` оператор `switch` завершает работу и управление передается следующему за ним оператору. Если же в последовательности операторов, связанных с совпадающей константой выбора в одной из ветвей `case`, не содержится оператор `break`, то сначала выполняются все операторы в *этой* ветви, а затем операторы, *совпадающие* с константой выбора в следующей ветви `case`. Этот процесс продолжается до тех пор, пока не встретится оператор `break` или же будет достигнут конец оператора `switch`.

В качестве упражнения проанализируйте исходный код приведенной ниже программы. Сможете ли вы предсказать, как будет выглядеть результат ее выполнения?

```
// Демонстрация оператора switch без оператора break.
class NoBreak {
  public static void main(String args[]) {
    int i;

    for(i=0; i<=5; i++) {
      switch(i) {
        case 0:
          // Далее следует "провал" в ветвях case оператора switch.
          System.out.println("i is less than one");
        case 1:
          System.out.println("i is less than two");
        case 2:
          System.out.println("i is less than three");
        case 3:
          System.out.println("i is less than four");
        case 4:
          System.out.println("i is less than five");
      }
      System.out.println();
    }
  }
}
```

Выполнение этой программы дает следующий результат:

```
i is less than one
i is less than two
i is less than three
i is less than four
i is less than five

i is less than two
i is less than three
i is less than four
```

```
i is less than five

i is less than three
i is less than four
i is less than five

i is less than four
i is less than five

i is less than five
```

Как демонстрирует приведенный выше пример, выполнение программы будет продолжено в следующей ветви `case` в отсутствие оператора `break`. А в следующем примере кода показано, что в операторе `switch` могут присутствовать пустые ветви `case`:

```
switch(i) {
  case 1:
  case 2:
  case 3: System.out.println("i is 1, 2 or 3");
    break;
  case 4: System.out.println("i is 4");
    break;
}
```

Если в приведенном выше фрагменте кода переменная `i` имеет значение **`1`**, **`2`** или **`3`**, то вызывается первый метод `println()`. А если ее значение равно **`4`**, вызывается второй метод **`println()`**. Такое расположение нескольких пустых ветвей **`case`** подряд нередко используется в тех случаях, когда нескольким ветвям должен соответствовать один и тот же общий код.

Вложенные операторы `switch`

Один оператор `switch` может быть частью последовательности операторов другого, внешнего оператора `switch`. И такой оператор `switch` называется *вложенным*. Константы выбора внутреннего и внешнего операторов `switch` могут содержать общие значения, не вызывая никаких конфликтов. Например, следующий фрагмент кода является вполне допустимым:

```
switch(ch1) {
  case 'A': System.out.println("This A is part of outer switch.");
    switch(ch2) {
      case 'A':
        System.out.println("This A is part of inner switch");
        break;
      case 'B': // ...
    }  // конец внутреннего оператора switch
    break;
  case 'B': // ...
```

Пример для опробования 3.1.

Начало построения справочной системы Java

Исходный файл: `Help.java`

В этом проекте предстоит создать простую справочную систему, предоставляющую сведения о синтаксисе

управляющих операторов Java. Программа, реализующая эту справочную систему, отображает меню с названиями операторов и ожидает выбора одного из них. Как только пользователь выберет один из пунктов меню, на экран будут выведены сведения о синтаксисе соответствующего оператора. В первой версии данной программы предоставляются сведения только об операторах `if` и `switch`. А в последующих проектах будут добавлены справочные данные об остальных управляющих операторах.

Последовательность действий

1. Создайте новый файл `Help.java`.
2. В начале работы программы отображается следующее меню:

```
Help on:
1. if
2. switch
Choose one:
```

Для этой цели потребуется приведенная ниже последовательность операторов.

```
System.out.println("Help on:");
System.out.println(" 1. if");
System.out.println(" 2. switch");
System.out.print("Choose one: ");
```

3. Далее программа получает данные о выборе пользователя. С этой целью вызывается метод `System.in.read()`, как показано ниже.

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

4. После этого в программе используется оператор `switch` для отображения сведений о синтаксисе выбранного оператора.

```
switch(choice) {
  case '1':
    System.out.println("The if:\n");
    System.out.println("if(condition) statement;");
    System.out.println("else statement;");
    break;
  case '2':
    System.out.println("The switch:\n");
    System.out.println("switch(expression) {");
    System.out.println(" case constant:");
    System.out.println(" statement sequence");
    System.out.println(" break;");
    System.out.println(" // ...");
    System.out.println("}");
    break;
  default:
    System.out.print("Selection not found.");
}
```

Обратите внимание на то, как в ветви `default` перехватываются сведения о неправильно сделанном выборе. Так, если пользователь введет значение **3**, оно не совпадет ни с одной из констант в ветвях `case` оператора `switch`, и тогда управление будет передано коду в ветви `default`.

5. Ниже приведен весь исходный код программы из файла `Help.java`.

```
/*
  Пример для опробования 3.1. Простая справочная система.
*/
class Help {
  public static void main(String args[])
    throws java.io.IOException {
      char choice;

      System.out.println("Help on:");
      System.out.println(" 1. if");
      System.out.println(" 2. switch");
      System.out.print("Choose one: ");
      choice = (char) System.in.read();

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

      switch(choice) {
        case '1':
          System.out.println("The if:\n");
          System.out.println("if(condition) statement;");
          System.out.println("else statement;");
          break;
        case '2':
          System.out.println("The switch:\n");
          System.out.println("switch(expression) {");
          System.out.println(" case constant:");
          System.out.println(" statement sequence");
          System.out.println(" break;");
          System.out.println(" // ...");
         System.out.println("}");
          break;
        default:
          System.out.print("Selection not found.");
      }
    }
}
```

6. Выполнение этой программы дает следующий результат:

```
Help on:
1. if
2. switch
Choose one: 1

The if:

if(condition) statement;
else statement;
```

Обращение к знатоку

ВОПРОС. В каких случаях вместо оператора `switch` следует использовать многоступенчатую конструкцию `if-else-if`?

ОТВЕТ. Вообще говоря, многоступенчатая конструкция `if-else-if` уместна в тех случаях, когда проверка условий не сводится к выяснению совпадения или несовпадения выражения с одиночным значением. Рассмотрим для примера следующую последовательность условных операторов:

```
if(x < 10) // ...
else if(y != 0) // ...
else if(!done) // ...
```

Данную последовательность нельзя заменить оператором `switch`, поскольку в трех ее условиях используются разные переменные и разные виды сравнения. Многоступенчатую конструкцию `if-else-if` целесообразно также применять для проверки значений с плавающей точкой и других объектов, типы которых отличаются от типов, предусмотренных для ветвей оператора `switch`.

Цикл `for`

Цикл `for` уже был представлен в главе 1, а здесь он рассматривается более подробно. Вас должны приятно удивить эффективность и гибкость этого цикла. Прежде всего обратимся к самым основным и традиционным формам цикла `for`.

Ниже приведена общая форма цикла `for` для повторного выполнения единственного оператора.

```
for(инициализация; условие; итерация) оператор;
```

А вот как выглядит его форма для повторного выполнения кодового блока.

```
for(инициализация; условие; итерация)
{
  последовательность операторов;
}
```

где *инициализация*, как правило, представлена оператором присваивания, задающим первоначальное значение переменной, которая выполняет роль счетчика и управляет циклом; *условие* — это логическое выражение, определяющее необходимость повторения цикла; а *итерация* — выражение, определяющее величину, на которую должно изменяться значение переменной, управляющей циклом, на каждом шаге цикла. Обратите внимание на то, что эти три основные части оператора цикла `for` должны быть разделены точкой с запятой. Выполнение цикла `for` будет продолжаться до тех пор, пока проверка условия дает истинный результат. Как только эта проверка даст ложный результат, цикл завершится, а выполнение программы будет продолжено с оператора, следующего после цикла `for`.

Ниже приведен пример программы, где цикл `for` служит для вывода на экран значений квадратного корня чисел в пределах от **1** до **99**. В данной программе отображается также ошибка округления, допущенная при вычислении квадратного корня.

```
// Вывод квадратных корней чисел от 1 до 99 вместе с ошибкой округления.
class SqrRoot {
  public static void main(String args[]) {
```

```
    double num, sroot, rerr;

    for(num = 1.0; num < 100.0; num++) {
      sroot = Math.sqrt(num);
      System.out.println("Square root of " + num +
                         " is " + sroot);

      // вычислить ошибку округления
      rerr = num - (sroot * sroot);
      System.out.println("Rounding error is " + rerr);
      System.out.println();
    }
  }
}
```

Обратите внимание на то, что ошибка округления вычисляется путем возведения в квадрат квадратного корня числа. Полученное значение отнимается от исходного числа.

Переменная цикла может как увеличиваться, так и уменьшаться, а величина приращения может выбираться произвольно. Например, в приведенном ниже фрагменте кода выводятся числа в пределах от **100** до **-95**, и на каждом шаге переменная цикла уменьшается на **5**.

```
// Цикл for, выполняющийся с отрицательным приращением переменной.
class DecrFor {
  public static void main(String args[]) {
    int x;

    // На каждом шаге цикла управляющая им переменная уменьшается на 5.
    for(x = 100; x > -100; x -= 5)
      System.out.println(x);
  }
}
```

В отношении циклов `for` следует особо подчеркнуть, что условное выражение всегда проверяется в самом начале цикла. Это означает, что код в цикле может вообще не выполняться, если проверяемое условие с самого начала оказывается ложным. Рассмотрим следующий пример:

```
for(count=10; count < 5; count++)
  x += count; // этот оператор не будет выполнен
```

Этот цикл вообще не будет выполняться, поскольку первоначальное значение переменной `count`, которая им управляет, сразу же оказывается больше **5**. А это означает, что условное выражение `count < 5` оказывается ложным с самого начала, т.е. еще до выполнения первого шага цикла.

Некоторые разновидности цикла `for`

Цикл `for` относится к наиболее универсальным операторам языка Java, поскольку он допускает самые разные варианты применения. Например, для управления циклом можно использовать несколько переменных. Рассмотрим следующий пример программы:

```
// Применение запятых в операторе цикла for.
class Comma {
```

```
  public static void main(String args[]) {
    int i, j;

    // Для управления этим циклом используются две переменные.
    for(i=0, j=10; i < j; i++, j--)
      System.out.println("i and j: " + i + " " + j);
  }
}
```

Выполнение этой программы дает следующий результат:

```
i and j: 0 10
i and j: 1 9
i and j: 2 8
i and j: 3 7
i and j: 4 6
```

В данном примере запятыми разделяются два оператора инициализации и еще два итерационных выражения. Когда цикл начинается, инициализируются обе переменные, `i` и `j`. Всякий раз, когда цикл повторяется, переменная `i` инкрементируется, а переменная `j` декрементируется. Применение нескольких переменных управления циклом нередко оказывается удобным и упрощает некоторые алгоритмы. Теоретически в цикле `for` может быть указано любое количество операторов инициализации и итерации, но на практике такой цикл получается слишком громоздким, если применяется больше двух подобных операторов.

Условным выражением, управляющим циклом `for`, может быть любое действительное выражение, дающее результат типа `bool`. В него не обязательно должна входить переменная управления циклом. В следующем примере программы цикл будет выполняться до тех пор, пока пользователь не введет с клавиатуры букву **S**.

```
// Выполнение цикла до тех пор, пока с клавиатуры
// не будет введена буква S.
class ForTest {
  public static void main(String args[])
    throws java.io.IOException {

    int i;

    System.out.println("Press S to stop.");
    for(i = 0; (char) System.in.read() != 'S'; i++)
      System.out.println("Pass #" + i);
  }
}
```

Пропуск отдельных частей в определении цикла `for`

Ряд интересных разновидностей цикла `for` получается в том случае, если оставить пустыми отдельные части в определении цикла. В Java допускается оставлять пустыми любые или же все части инициализации, условия и итерации в операторе цикла `for`. В качестве примера рассмотрим следующую программу:

```
// Пропуск отдельных частей в определении цикла for.
class Empty {
```

```
  public static void main(String args[]) {
    int i;

    // В определении этого цикла отсутствует итерационное выражение.
    for(i = 0; i < 10; ) {
      System.out.println("Pass #" + i);
        i++; // инкрементировать переменную управления циклом
    }
  }
}
```

В данном примере итерационное выражение в определении цикла `for` оказывается пустым, т.е. вообще отсутствует. Вместо этого переменная `i`, управляющая циклом, инкрементируется в теле самого цикла. Это означает, что всякий раз, когда цикл повторяется, значение переменной `i` проверяется на равенство числу **10**, но никаких других действий при этом не происходит. А поскольку переменная `i` инкрементируется в теле цикла, то сам цикл выполняется обычным образом, выводя приведенный ниже результат.

```
Pass #0
Pass #1
Pass #2
Pass #3
Pass #4
Pass #5
Pass #6
Pass #7
Pass #8
Pass #9
```

В следующем примере программы из определения цикла `for` исключена инициализирующая часть.

```
// Пропуск дополнительных частей в определении цикла for.
class Empty2 {
  public static void main(String args[]) {
    int i;

    // Из определения этого цикла исключено не только
    // итерационное, но и инициализирующее выражение.
    i = 0;
    for(; i < 10; ) {
      System.out.println("Pass #" + i);
      i++; // инкрементировать переменную управления циклом
    }
  }
}
```

В данном примере переменная `i` инициализируется перед началом цикла, а не в самом цикле `for`. Как правило, переменная управления циклом инициализируется в цикле `for`. Выведение инициализирующей части за пределы цикла обычно делается лишь в том случае, если первоначальное значение управляющей им переменной получается в результате сложного процесса, который нецелесообразно вводить в само определение цикла `for`.

Бесконечный цикл

Если оставить пустым выражение условия в определении цикла `for`, то получится *бесконечный цикл*, т.е. такой цикл, который никогда не завершается. В качестве примера в следующем фрагменте кода показано, каким образом в Java обычно создается бесконечный цикл:

```
for(;;) // цикл, намеренно сделанный бесконечным
{
  //...
}
```

Этот цикл будет выполняться бесконечно. Несмотря на то что бесконечные циклы требуются для решения некоторых задач программирования, например при разработке командных процессоров операционных систем, большинство так называемых "бесконечных" циклов на самом деле представляют собой циклы со специальными требованиями к завершению. (Подробнее об этом речь пойдет ближе к концу главы, но, как правило, выход из бесконечного цикла осуществляется с помощью оператора `break`.)

Циклы без тела

В Java допускается оставлять пустым тело цикла `for` или любого другого цикла, поскольку *пустой оператор* с точки зрения синтаксиса этого языка считается действительным. Циклы без тела нередко оказываются полезными. Например, в следующей программе цикл без тела служит для получения суммы чисел от **1** до **5**:

```
// Тело цикла for может быть пустым.
class Empty3 {
  public static void main(String args[]) {
    int i;

    int sum = 0;
    // Несмотря на отсутствие тела, в этом цикле
    // производится суммирование чисел от 1 до 5!
    for(i = 1; i <= 5; sum += i++) ;
      System.out.println("Sum is " + sum);
  }
}
```

Выполнение этой программы дает следующий результат:

```
Sum is 15
```

Обратите внимание на то, что процесс суммирования чисел выполняется полностью в операторе цикла `for`, и для этого тело цикла не требуется. В этом цикле особое внимание обращает на себя итерационное выражение.

```
sum += i++
```

Подобные операторы не должны вас смущать. Они часто встречаются в программах, профессионально написанных на Java, и становятся вполне понятными, если разобрать их по частям. Дословно приведенный выше оператор означает следующее: сложить со значением переменной `sum` результат суммирования значений переменных `sum` и `i`, а затем инкрементировать значение переменной `i`. Следовательно, данный оператор равнозначен следующей последовательности операторов:

```
sum = sum + i;
i++;
```

Объявление управляющих переменных в цикле for

Нередко переменная, управляющая циклом for, требуется только для выполнения самого цикла и нигде больше не используется. В таком случае управляющую переменную можно объявить в инициализирующей части оператора цикла for. Например, в приведенной ниже программе вычисляется сумма и факториал чисел от 1 до 5, а переменная i, управляющая циклом for, объявляется в этом цикле.

```
// Объявление переменной управления циклом в самом цикле for.
class ForVar {
  public static void main(String args[]) {
    int sum = 0;
    int fact = 1;

    // Вычисление факториала чисел от 1 до 5.
    // Переменная управления объявляется в этом цикле for.
    for(int i = 1; i <= 5; i++) {
      sum += i; // Она доступна во всем цикле.
      fact *= i;
    }

    // Но недоступна за пределами цикла.

    System.out.println("Sum is " + sum);
    System.out.println("Factorial is " + fact);
  }
}
```

Объявляя переменную в цикле for, не следует забывать о том, что область действия этой переменной ограничивается пределами оператора цикла for. Это означает, что за пределами цикла действие данной переменной прекращается. Так, в приведенном выше примере переменная i оказывается недоступной за пределами цикла for. Для того чтобы использовать переменную управления циклом в каком-нибудь другом месте программы, ее нельзя объявлять в цикле for.

Прежде чем переходить к чтению последующих разделов этой главы, поэкспериментируйте с собственными разновидностями цикла for. В ходе эксперимента вы непременно обнаружите замечательные свойства этого цикла.

Расширенный цикл for

С недавних пор в распоряжение программирующих на Java предоставлен так называемый *расширенный* цикл for, обеспечивающий специальные средства для перебора объектов из коллекции, например из массива. Расширенный цикл for будет представлен в главе 5 при рассмотрении массивов.

Цикл while

Еще одной разновидностью циклов в Java является `while`. Ниже приведена общая форма этого цикла.

```
while (условие) оператор;
```

где *оператор* — это единственный оператор или блок операторов, а *условие* означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле *оператор* выполняется до тех пор, пока *условие* истинно. Как только *условие* становится ложным, управление программой передается строке кода, следующей непосредственно после цикла.

Ниже приведен простой пример использования цикла `while` для вывода на экран букв английского алфавита.

```
// Демонстрация цикла while.
class WhileDemo {
  public static void main(String args[]) {
    char ch;

    // вывести буквы английского алфавита, используя цикл while
    ch = 'a';
    while(ch <= 'z') {
      System.out.print(ch);
      ch++;
    }
  }
}
```

В данном примере переменная `ch` инициализируется кодом буквы **a**. На каждом шаге цикла на экран сначала выводится значение переменной `ch`, а затем это значение увеличивается на единицу. Процесс продолжается до тех пор, пока значение переменной `ch` не станет больше кода буквы **z**.

Как и в цикле `for`, в цикле `while` проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может вообще не выполняться, а кроме того, избавляет от необходимости производить отдельную проверку перед самим циклом. Данное свойство цикла `while` демонстрируется в следующем примере программы, где вычисляются целые степени числа **2** от **0** до **9**.

```
// Вычисление целых степеней числа 2.
class Power {
  public static void main(String args[]) {
    int e;
    int result;

    for(int i=0; i < 10; i++) {
      result = 1;
      e = i;
      while(e > 0) {
        result *= 2;
        e--;
      }

      System.out.println("2 to the " + i +
                         " power is " + result);
```

```
    }
  }
}
```

Ниже приведен результат выполнения данной программы.

```
2 to the 0 power is 1
2 to the 1 power is 2
2 to the 2 power is 4
2 to the 3 power is 8
2 to the 4 power is 16
2 to the 5 power is 32
2 to the 6 power is 64
2 to the 7 power is 128
2 to the 8 power is 256
2 to the 9 power is 512
```

Обратите внимание на то, что цикл `while` выполняется только в том случае, если значение переменной `e` больше нуля. А когда оно равно нулю, как это имеет место на первом шаге цикла `for`, цикл `while` пропускается.

Обращение к знатоку

ВОПРОС. В языке Java циклы обладают большой гибкостью. Как же в таком случае выбрать цикл, наиболее подходящий для решения конкретной задачи?

ОТВЕТ. Если число итераций известно заранее, то лучше выбрать цикл `for`. Цикл `do-while` подходит в тех случаях, если требуется, чтобы была выполнена как минимум одна итерация. А цикл `while` оказывается наиболее удобным тогда, когда число повторений цикла заранее неизвестно.

Цикл do-while

Третьей и последней разновидностью циклов в Java является `do-while`. В отличие от циклов `for` и `while`, в которых условие проверялось в самом начале, в цикле `do-while` условие выполнения проверяется в самом конце. Это означает, что цикл `do-while` всегда выполняется хотя бы один раз. Ниже приведена общая форма цикла `do-while`.

```
do {
   операторы;
} while (условие);
```

При наличии лишь одного оператора фигурные скобки в данной форме записи необязательны. Тем не менее они зачастую используются для того, чтобы сделать конструкцию `do-while` более удобочитаемой и не путать ее с конструкцией цикла `while`. Цикл `do-while` выполняется до тех пор, пока условное выражение истинно.

```
// Демонстрация цикла do-while.
class DWDemo {
  public static void main(String args[])
    throws java.io.IOException {

    char ch;

    do {
```

```
      System.out.print("Press a key followed by ENTER: ");
      ch = (char) System.in.read(); // ввести символ с клавиатуры
    } while(ch != 'q');
  }
}
```

Используя цикл `do-while`, мы можем усовершенствовать игру в угадывание букв, созданную в начале этой главы. На этот раз выполнение цикла будет продолжаться до тех пор, пока пользователь не угадает букву.

```
// Игра в угадывание букв, четвертая версия.
class Guess4 {
  public static void main(String args[])
    throws java.io.IOException {

      char ch, ignore, answer = 'S';

      do {
        System.out.println("I'm thinking of a letter between A and Z.");
        System.out.print("Can you guess it: ");

       // ввести символ с клавиатуры
       ch = (char) System.in.read();
       // отвергнуть все остальные символы во входном буфере
      do {
        ignore = (char) System.in.read();
      } while(ignore != '\n');

      if(ch == answer) System.out.println("** Right **");
      else {
        System.out.print("...Sorry, you're ");
        if(ch < answer) System.out.println("too low");
        else System.out.println("too high");
        System.out.println("Try again!\n");
      }
    } while(answer != ch);
  }
}
```

Ниже приведен один из возможных вариантов выполнения данной программы в интерактивном режиме.

```
I'm thinking of a letter between A and Z.
Can you guess it: A
...Sorry, you're too low
Try again!

I'm thinking of a letter between A and Z.
Can you guess it: Z
...Sorry, you're too high
Try again!

I'm thinking of a letter between A and Z.
Can you guess it: S
** Right **
```

Обратите внимание на еще одну интересную особенность данной программы: в ней применяются два цикла `do-while`. Первый цикл выполняется до тех пор, пока пользователе не введет правильную букву. А второй цикл приведен ниже и требует дополнительных пояснений.

```
// отвергнуть все остальные символы во входном буфере
      do {
        ignore = (char) System.in.read();
      } while(ignore != '\n');
```

Как пояснялось ранее, консольный ввод буферизуется построчно. Это означает, что для передачи символов, вводимых с клавиатуры, приходится нажимать клавишу <Enter>, что приводит к формированию последовательности символов перевода строки и возврата каретки. Эти символы сохраняются во входном буфере вместе с введенными с клавиатуры. Кроме того, если ввести с клавиатуры несколько символов подряд, не нажав клавишу <Enter>, они так и останутся во входном буфере.

В рассматриваемом здесь цикле эти символы отвергаются до тех пор, пока не будет достигнут конец строки. Если не сделать этого, лишние символы будут передаваться программе как отгадываемые, что не соответствует правилам игры в отгадывание. (Для того чтобы убедиться в этом, попробуйте закомментировать внутренний цикл `do-while` в исходном коде программы.) После представления ряда других языковых средств Java в главе 10 рассматриваются некоторые более совершенные способы обработки консольного ввода. Но применение метода `read()` в данной программе дает элементарное представление о принципе действия системы ввода-вывода в Java. А кроме того, в данной программе демонстрируется еще один пример применения циклов в практике программирования на Java.

Пример для опробования 3.2. Расширение справочной системы Java

Исходный файл: `Help2.java`

В этом проекте предстоит расширить справочную систему Java, созданную в примере для опробования 3.1. В эту версию программы будут добавлены сведения о синтаксисе циклов `for`, `while` и `do-while`. Кроме того, будет реализована проверка действий пользователя, работающего с меню. Цикл будет повторяться до тех пор, пока пользователь не введет допустимое значение.

Последовательность действий

1. Скопируйте файл `Help.java` в новый файл `Help2.java`.
2. Измените часть программы, ответственную за отображение вариантов, предлагаемых пользователю на выбор. Реализуйте ее с помощью циклов так, как показано ниже.

```
public static void main(String args[])
  throws java.io.IOException {
    char choice, ignore;

    do {
      System.out.println("Help on:");
      System.out.println(" 1. if");
      System.out.println(" 2. switch");
      System.out.println(" 3. for");
      System.out.println(" 4. while");
```

```
      System.out.println(" 5. do-while\n");
      System.out.print("Choose one: ");

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

      do {
        ignore = (char) System.in.read();
      } while(ignore != '\n');
    } while( choice < '1' | choice > '5');
```

3. Обратите внимание на вложенные циклы `do-while`, используемые с целью избавиться от нежелательных символов, оставшихся во входном буфере. После внесения приведенных выше изменений программа будет отображать меню в цикле до тех пор, пока пользователь не введет числовое значение в пределах от **1** до **5**.
4. Дополните оператор `switch` выводом на экран сведений о синтаксисе циклов `for`, `while` и `do-while`, как показано ниже.

```
switch(choice) {
  case '1':
    System.out.println("The if:\n");
    System.out.println("if(condition) statement;");
    System.out.println("else statement;");
    break;
  case '2':
    System.out.println("The switch:\n");
    System.out.println("switch(expression) {");
    System.out.println(" case constant:");
    System.out.println(" statement sequence");
    System.out.println(" break;");
    System.out.println(" // ...");
    System.out.println("}");
    break;
  case '3':
    System.out.println("The for:\n");
    System.out.print("for(init; condition; iteration)");
    System.out.println(" statement;");
    break;
  case '4':
    System.out.println("The while:\n");
    System.out.println("while(condition) statement;");
    break;
  case '5':
    System.out.println("The do-while:\n");
    System.out.println("do {");
    System.out.println(" statement;");
    System.out.println("} while (condition);");
    break;
}
```

5. Обратите внимание на то, что в данном варианте оператора `switch` отсутствует ветвь `default`. А поскольку цикл отображения меню будет выполняться до тех пор, пока пользователь не введет допустимое значение, необходимость в обработке неправильных значений отпадает.
6. Ниже приведен весь исходный код программы из файла `Help2.java`.

```
/*
  Пример для опробования 3.2.

  Расширенная справочная система, в которой для обработки
  результатов выбора из меню используется цикл do-while.
*/
class Help2 {
  public static void main(String args[])
    throws java.io.IOException {
    char choice, ignore;

    do {
      System.out.println("Help on:");
      System.out.println(" 1. if");
      System.out.println(" 2. switch");
      System.out.println(" 3. for");
      System.out.println(" 4. while");
      System.out.println(" 5. do-while\n");
      System.out.print("Choose one: ");

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

      do {
        ignore = (char) System.in.read();
      } while(ignore != '\n');
    } while( choice < '1' | choice > '5');

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

    switch(choice) {
      case '1':
        System.out.println("The if:\n");
        System.out.println("if(condition) statement;");
        System.out.println("else statement;");
        break;
      case '2':
        System.out.println("The switch:\n");
        System.out.println("switch(expression) {");
        System.out.println(" case constant:");
        System.out.println(" statement sequence");
        System.out.println(" break;");
        System.out.println(" // ...");
        System.out.println("}");
        break;
      case '3':
        System.out.println("The for:\n");
        System.out.print("for(init; condition; iteration)");
        System.out.println(" statement;");
        break;
      case '4':
        System.out.println("The while:\n");
        System.out.println("while(condition) statement;");
        break;
      case '5':
```

```
            System.out.println("The do-while:\n");
            System.out.println("do {");
            System.out.println(" statement;");
            System.out.println("} while (condition);");
            break;
        }
      }
    }
```

Применение оператора break для выхода из цикла

С помощью оператора `break` можно специально организовать немедленный выход из цикла в обход любого кода, оставшегося в теле цикла, а также минуя проверку условия цикла. Когда в теле цикла встречается оператор `break`, цикл завершается, а выполнение программы возобновляется с оператора, следующего после этого цикла. Рассмотрим следующий краткий пример программы:

```
// Применение оператора break для выхода из цикла.
class BreakDemo {
  public static void main(String args[]) {
    int num;

    num = 100;

    // выполнять цикл до тех пор, пока квадрат значения
    // переменной i меньше значения переменной num
    for(int i=0; i < num; i++) {
      if(i*i >= num) break; // прекратить цикл, если i*i >= 100
        System.out.print(i + " ");
    }
    System.out.println("Loop complete.");
  }
}
```

Выполнение этой программы дает следующий результат:

```
0 1 2 3 4 5 6 7 8 9 Loop complete.
```

Как видите, цикл `for` организован для выполнения в пределах значений переменной **num** от **0** до **100**. Но, несмотря на это, оператор `break` прерывает этот цикл раньше, когда квадрат значения переменной `i` становится больше значения переменной **num**.

Оператор `break` можно применять в любых циклах, предусмотренных в Java, включая и те, что намеренно организованы бесконечными. В качестве примера ниже приведен простой пример программы, в которой вводимые данные читаются до тех пор, пока пользователь не введет с клавиатуры букву **q**.

```
// Чтение вводимых данных до тех пор, пока не будет получена буква q.
class Break2 {
  public static void main(String args[])
    throws java.io.IOException {
      char ch;
```

```
      // "Бесконечный" цикл, завершаемый с помощью оператора break.
      for( ; ; ) {
        ch = (char) System.in.read(); // ввести символ с клавиатуры
        if(ch == 'q') break;
      }
      System.out.println("You pressed q!");
    }
}
```

Если оператор `break` применяется в целом ряде вложенных циклов, то он прерывает выполнение только самого внутреннего цикла. В качестве примера рассмотрим следующую программу:

```
// Применение оператора break во вложенных циклах.
class Break3 {
  public static void main(String args[]) {

    for(int i=0; i<3; i++) {
      System.out.println("Outer loop count: " + i);
      System.out.print(" Inner loop count: ");

      int t = 0;
      while(t < 100) {
        if(t == 10) break; // прервать цикл, если t = 10
          System.out.print(t + " ");
          t++;
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

Выполнение этой программы дает следующий результат:

```
Outer loop count: 0
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 1
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Outer loop count: 2
    Inner loop count: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

Как видите, оператор `break` из внутреннего цикла вызывает прерывание только этого цикла. А на выполнение внешнего цикла он не оказывает никакого влияния.

В отношении оператора `break` необходимо также иметь в виду следующее. Во-первых, в теле цикле может присутствовать несколько операторов `break`, но применять их следует очень аккуратно, поскольку чрезмерное количество операторов `break` обычно приводит к нарушению нормальной структуры кода. И во-вторых, оператор `break`, выполняющий выход из оператора `switch`, оказывает воздействие только на этот оператор, но не на объемлющие его циклы.

Оператор break в качестве оператора goto

Помимо оператора switch и циклов, оператор break может быть использован как "цивилизованный" вариант оператора goto. В языке Java оператор goto отсутствует, поскольку он дает возможность произвольного перехода в любую точку программы, что способствует созданию плохо структурированного кода. Программы, в которых часто используется оператор goto, обычно сложны для восприятия и сопровождения. Но в некоторых случаях оператор goto может оказаться полезным. Его удобно, например, использовать для экстренного выхода из многократно вложенных циклов.

Для решения подобных задач в Java определена расширенная форма оператора break. Используя этот вариант оператора break, можно, например, выйти за пределы одного или нескольких кодовых блоков. Эти блоки должны быть частью циклов или оператора switch. Кроме того, нужно явно указать точку, с которой должно быть продолжено выполнение программы. Для этой цели в расширенной форме оператора break предусмотрена метка. Как будет показано далее, оператор break позволяет воспользоваться всеми преимуществами оператора goto и в то же время избежать сложностей, связанных с его применением.

Ниже приведена общая форма оператора break с меткой.

```
break метка;
```

Как правило, *метка* — это имя, обозначающее кодовый блок. При выполнении расширенного оператора break управление передается за пределы именованного кодового блока. Оператор break с меткой может содержаться непосредственно в именованном кодовом блоке или в одном из блоков, входящих в состав именованного блока. Следовательно, рассматриваемый здесь вариант оператора break можно использовать для выхода из ряда вложенных блоков. Но это языковое средство не позволяет передать управление в кодовый блок, не содержащий оператор break.

Для того чтобы присвоить имя кодовому блоку, нужно поставить перед ним метку. Именовать можно как независимый блок, так и оператор, адресатом которого является кодовый блок. Роль *метки* может выполнять любой допустимый в Java идентификатор с двоеточием. Пометив кодовый блок, метку можно использовать в качестве адресата оператора break. Благодаря этому выполнение программы возобновляется с *конца* именованного блока. Например, в приведенном ниже фрагменте кода используются три вложенных блока.

```
// Применение оператора break с меткой.
class Break4 {
  public static void main(String args[]) {
    int i;

    for(i=1; i<4; i++) {
      one: {
      two: {
      three: {
                System.out.println("\ni is " + i);
                // Переходы по меткам.
                if(i==1) break one;
                if(i==2) break two;
                if(i==3) break three;

                  // Эта строка кода никогда не будет достигнута.
```

```
            System.out.println("won't print");
          }
          System.out.println("After block three.");
        }
        System.out.println("After block two.");
      }
      System.out.println("After block one.");
    }
    System.out.println("After for.");
  }
}
```

Выполнение этого фрагмента кода дает следующий результат:

```
i is 1
After block one.

i is 2
After block two.
After block one.

i is 3
After block three.
After block two.

After block one.
After for.
```

Рассмотрим подробнее приведенный выше фрагмент кода, чтобы лучше понять, каким образом получается именно такой результат его выполнения. Когда значение переменной `i` равно **1**, условие первого оператора `if` становится истинным и управление передается в конец блока с меткой `one`. В результате выводится сообщение `"After block one."` (После первого блока). Если значение переменной `i` равно **2**, то успешно завершается проверка условия во втором операторе `if` и выполнение программы продолжается с конца блока с меткой `two`. В результате выводятся по порядку сообщения `"After block two."` (После второго блока) и `"After block one."`. Когда же переменная `i` принимает значение **3**, истинным становится условие в третьем операторе **`if`** и управление передается в конец блока с меткой `three`. В этом случае выводятся все три сообщения: два упомянутых выше, а также сообщение `"After block three."` (После третьего блока), а затем еще и сообщение `"After for."` (После цикла `for`).

Обратимся к еще одному примеру. На этот раз оператор `break` будет использован для выхода за пределы нескольких вложенных циклов. Когда во внутреннем цикле выполняется оператор `break`, управление передается в конец блока внешнего цикла. Этот блок помечен меткой `done`. В результате происходит выход из всех трех циклов.

```
// Еще один пример применения оператора break с меткой.
class Break5 {
  public static void main(String args[]) {

    done:
      for(int i=0; i<10; i++) {
        for(int j=0; j<10; j++) {
          for(int k=0; k<10; k++) {
            System.out.println(k + " ");
```

```
            if(k == 5) break done; // переход по метке done
          }
          System.out.println("After k loop"); // не выполнится
        }
        System.out.println("After j loop"); // не выполнится
      }
      System.out.println("After i loop");
  }
}
```

Ниже приведен результат выполнения данного фрагмента кода.

```
0
1
2
3
4
5
After i loop
```

Расположение метки имеет большое значение, особенно когда речь идет о циклах. Рассмотрим в качестве примера следующий фрагмент кода:

```
// Расположение метки имеет большое значение.
class Break6 {
  public static void main(String args[]) {
    int x=0, y=0;

// Здесь метка располагается перед оператором.
stop1: for(x=0; x < 5; x++) {
         for(y = 0; y < 5; y++) {
           if(y == 2) break stop1;
             System.out.println("x and y: " + x + " " + y);
         }
       }

       System.out.println();

// А здесь метка располагается непосредственно перед
// открывающей фигурной скобкой.
      for(x=0; x < 5; x++)
stop2: {
         for(y = 0; y < 5; y++) {
           if(y == 2) break stop2;
             System.out.println("x and y: " + x + " " + y);
         }
       }
  }
}
```

Ниже приведен результат выполнения данного фрагмента кода.

```
x and y: 0 0
x and y: 0 1

x and y: 0 0
x and y: 0 1
```

```
x and y: 1 0
x and y: 1 1
x and y: 2 0
x and y: 2 1
x and y: 3 0
x and y: 3 1
x and y: 4 0
x and y: 4 1
```

Ряды вложенных циклов в данном фрагменте кода идентичны за исключением того, что в первом ряду метка находится перед внешним циклом `for`. В таком случае при выполнении оператора `break` управление передается в конец всего блока цикла `for`, а оставшиеся итерации внешнего цикла пропускаются. Во втором ряду метка находится перед открывающей фигурной скобкой кодового блока, определяющего тело внешнего цикла. Поэтому при выполнении оператора `break stop2` управление передается в конец тела внешнего цикла `for`, и далее выполняется очередной его шаг.

Не следует, однако, забывать, что в операторе `break` нельзя использовать метку, не определенную в объемлющем его цикле. Например, приведенный ниже фрагмент кода некорректен и не подлежит компиляции.

```
// Этот фрагмент кода содержит ошибку.
class BreakErr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // НЕВЕРНО!
      System.out.print(j + " ");
    }
  }
}
```

Кодовый блок, обозначенный меткой `one`, не содержит оператор `break`, и поэтому управление не может быть передано этому блоку.

Обращение к знатоку

ВОПРОС. Как пояснялось выше, оператор `goto` нарушает структуру программы, и поэтому более подходящим вариантом является оператор `break` с меткой. Но не нарушает ли структуру программы и переход в конец внешнего цикла, т.е. в точку, далеко отстоящую от оператора `break`?

ОТВЕТ. Действительно, нарушает. Но если явный переход все-таки необходим, то передача управления в конец кодового блока сохраняет хоть какое-то подобие структуры, чего нельзя сказать об операторе `goto`.

Применение оператора continue

С помощью оператора `continue` можно организовать преждевременное завершение шага итерации цикла в обход обычной структуры управления циклом. Оператор

continue осуществляет принудительный переход к следующему шагу цикла, пропуская любой код, оставшийся невыполненным. Таким образом, оператор continue служит своего рода дополнением оператора break. В приведенном ниже примере программы оператор continue используется в качестве вспомогательного средства для вывода четных чисел в пределах от 0 до 100.

```
// Применение оператора continue.
class ContDemo {
  public static void main(String args[]) {
    int i;

    // вывести четные числа в пределах от 0 до 100
    for(i = 0; i<=100; i++) {
      if((i%2) != 0) continue; // завершить шаг итерации цикла
        System.out.println(i);
    }
  }
}
```

В данном примере выводятся только четные числа, поскольку при обнаружении нечетного числа шаг итерации цикла завершается преждевременно в обход вызова метода println().

В циклах while и do-while оператор continue вызывает передачу управления непосредственно условному выражению, после чего продолжается процесс выполнения цикла. А в цикле for сначала вычисляется итерационное выражение, затем условное выражение, после чего цикл продолжается.

Как и в операторе break, в операторе continue может присутствовать метка, обозначающая тот объемлющий цикл, выполнение которого должно быть продолжено. Ниже приведен пример программы, демонстрирующий применение оператора continue с меткой.

```
// Применить оператор continue с меткой.
class ContToLabel {
  public static void main(String args[]) {

outerloop:
    for(int i=1; i < 10; i++) {
      System.out.print("\nOuter loop pass " + i +
                       ", Inner loop: ");
      for(int j = 1; j < 10; j++) {
       if(j == 5) continue outerloop; // продолжить внешний цикл
       System.out.print(j);
      }
    }
  }
}
```

Выполнение этой программы дает следующий результат:

```
Outer loop pass 1, Inner loop: 1234
Outer loop pass 2, Inner loop: 1234
Outer loop pass 3, Inner loop: 1234
Outer loop pass 4, Inner loop: 1234
Outer loop pass 5, Inner loop: 1234
```

```
Outer loop pass 6, Inner loop: 1234
Outer loop pass 7, Inner loop: 1234
Outer loop pass 8, Inner loop: 1234
Outer loop pass 9, Inner loop: 1234
```

Как следует из приведенного выше примера, при выполнении оператора `continue` управление передается внешнему циклу, и оставшиеся итерации внутреннего цикла пропускаются.

В реальных программах оператор `continue` применяется очень редко. И объясняется это, в частности, богатым набором в Java операторов цикла, удовлетворяющих бо́льшую часть потребностей в написании прикладных программ. Но в особых случаях, когда требуется преждевременное прекращение цикла, оператор `continue` позволяет сделать это, не нарушая структуру кода.

Пример для опробования 3.3.

Завершение построения справочной системы Java

Исходный файл: `Help3.java`

В этом проекте предстоит завершить построение справочной системы Java, начатое в предыдущих проектах. Данная версия будет дополнена сведениями о синтаксисе операторов `break` и `continue`, а также даст пользователю возможность запрашивать сведения о синтаксисе нескольких операторов. Эта цель достигается путем добавления внешнего цикла, который выполняется до тех пор, пока пользователь не введет с клавиатуры букву **q** вместо номера пункта меню.

Последовательность действий

1. Скопируйте файл `Help2.java` в новый файл `Help3.java`.
2. Поместите весь исходный код программы в бесконечный цикл `for`. Выход из этого цикла будет осуществляться с помощью оператора `break`, который получит управление тогда, когда пользователь введет с клавиатуры букву **q**. А поскольку этот цикл включает в себя весь код, то выход из него означает завершение программы.
3. Измените цикл отображения меню так, как показано ниже.

```
do {
  System.out.println("Help on:");
  System.out.println(" 1. if");
  System.out.println(" 2. switch");
  System.out.println(" 3. for");
  System.out.println(" 4. while");
  System.out.println(" 5. do-while");
  System.out.println(" 6. break");
  System.out.println(" 7. continue\n");
  System.out.print("Choose one (q to quit): ");

  choice = (char) System.in.read();
   do {
      ignore = (char) System.in.read();
   } while(ignore != '\n');
} while( choice < '1' | choice > '7' & choice != 'q');
```

Как видите, цикл теперь включает в себя операторы `break` и `continue`. Кроме того, буква **q** воспринимается в нем как допустимый вариант выбора.

4. Дополните оператор `switch` операторами `break` и `continue`, как показано ниже.

```
case '6':
  System.out.println("The break:\n");
  System.out.println("break; or break label;");
  break;
case '7':
  System.out.println("The continue:\n");
  System.out.println("continue; or continue label;");
  break;
```

5. Ниже приведен весь исходный код программы из файла `Help3.java`.

```
/*
  Пример для опробования 3.3.

  Завершенная справочная система по управляющим
  операторам Java, обрабатывающая многократные запросы.
*/
class Help3 {
  public static void main(String args[])
    throws java.io.IOException {
    char choice, ignore;

    for(;;) {
      do {
        System.out.println("Help on:");
        System.out.println(" 1. if");
        System.out.println(" 2. switch");
        System.out.println(" 3. for");
        System.out.println(" 4. while");
        System.out.println(" 5. do-while");
        System.out.println(" 6. break");
        System.out.println(" 7. continue\n");
        System.out.print("Choose one (q to quit): ");

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

        do {
          ignore = (char) System.in.read();
        } while(ignore != '\n');
      } while( choice < '1' | choice > '7' & choice != 'q');

      if(choice == 'q') break;

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

      switch(choice) {
        case '1':
          System.out.println("The if:\n");
          System.out.println("if(condition) statement;");
          System.out.println("else statement;");
          break;
        case '2':
```

```
            System.out.println("The switch:\n");
            System.out.println("switch(expression) {");
            System.out.println(" case constant:");
            System.out.println(" statement sequence");
            System.out.println(" break;");
            System.out.println(" // ...");
            System.out.println("}");
            break;
          case '3':
            System.out.println("The for:\n");
            System.out.print("for(init; condition; iteration)");
            System.out.println(" statement;");
            break;
          case '4':
            System.out.println("The while:\n");
            System.out.println("while(condition) statement;");
            break;
          case '5':
            System.out.println("The do-while:\n");
            System.out.println("do {");
            System.out.println(" statement;");
            System.out.println("} while (condition);");
            break;
          case '6':
            System.out.println("The break:\n");
            System.out.println("break; or break label;");
            break;
          case '7':
            System.out.println("The continue:\n");
            System.out.println("continue; or continue label;");
            break;
        }
        System.out.println();
      }
    }
}
```

6. Ниже приведен один из возможных вариантов выполнения данной программы в диалоговом режиме.

```
Help on:
  1. if
  2. switch
  3. for
  4. while
  5. do-while
  6. break
  7. Continue

Choose one (q to quit): 1

The if:

if(condition) statement;
```

```
  else statement;

  Help on:
    1. if
    2. switch
    3. for
    4. while
    5. do-while
    6. break
    7. Continue

  Choose one (q to quit): 6

  The break:

  break; or break label;

  Help on:
    1. if
    2. switch
    3. for
    4. while
    5. do-while
    6. break
    7. Continue

  Choose one (q to quit): q
```

Вложенные циклы

Как следует из предыдущих примеров программ, один цикл может быть вложен в другой. С помощью вложенных циклов решаются самые разные задачи. Поэтому, прежде чем завершить рассмотрение циклов в Java, уделим еще немного внимания вложенным циклам. Ниже приведен пример программы, содержащей вложенные циклы `for`. С помощью этих циклов для каждого числа от **2** до **100** определяется ряд множителей, на которые данное число делится без остатка.

```
/*
  Использовать вложенные циклы для выявления
  множителей чисел от 2 до 100.
*/
class FindFac {
  public static void main(String args[]) {

    for(int i=2; i <= 100; i++) {
      System.out.print("Factors of " + i + ": ");
      for(int j = 2; j < i; j++)
        if((i%j) == 0) System.out.print(j + " ");
          System.out.println();
    }
  }
}
```

Ниже приведена часть результата выполнения данной программы.

```
Factors of 2:
Factors of 3:
Factors of 4: 2
Factors of 5:
Factors of 6: 2 3
Factors of 7:
Factors of 8: 2 4
Factors of 9: 3
Factors of 10: 2 5
Factors of 11:
Factors of 12: 2 3 4 6
Factors of 13:
Factors of 14: 2 7
Factors of 15: 3 5
Factors of 16: 2 4 8
Factors of 17:
Factors of 18: 2 3 6 9
Factors of 19:
Factors of 20: 2 4 5 10
```

В данной программе переменная `i` из внешнего цикла последовательно принимает значения до **2** до **100**. А во внутреннем цикле для каждого числа от **2** до текущего значения переменной `i` выполняется проверка, делится ли это значение без остатка. В качестве упражнения попробуйте сделать данную программу более эффективной. (*Подсказка*: число итераций во внутреннем цикле можно уменьшить.)

Упражнение для самопроверки по материалу главы 3

1. Напишите программу, которая вводила бы символы с клавиатуры до тех пор, пока не встретится точка. Предусмотрите в программе счетчик числа пробелов. Сведения о количестве пробелов должны выводиться в конце программы.
2. Какова общая форма многоступенчатой конструкции `if-else-if`?
3. Допустим, имеется следующий фрагмент кода:

```
if(x < 10)
  if(y > 100) {
    if(!done) x = z;
    else y = z;
  }
else System.out.println("error"); // что если?
```

4. С каким из операторов `if` связан последний оператор `else`?
5. Напишите цикл `for`, в котором перебирались бы значения от **1000** до **0** с шагом **–2**.
6. Корректен ли следующий фрагмент кода?

```
for(int i = 0; i < num; i++)
  sum += i;

count = i;
```

7. Какие действия выполняет оператор **break**? Опишите оба варианта этого оператора.
8. Какое сообщение будет выведено после выполнения оператора `break` в приведенном ниже фрагменте кода?

```
for(i = 0; i < 10; i++) {
  while(running) {
    if(x<y) break;
      // ...
  }
  System.out.println("after while");
}
System.out.println("After for");
```

9. Что будет выведено на экран в результате выполнения следующего фрагмента кода?

```
for(int i = 0; i<10; i++) {
  System.out.print(i + " ");
  if((i%2) == 0) continue;
  System.out.println();
}
```

10. Итерационное выражение для цикла `for` не обязательно должно изменять переменную цикла на фиксированную величину. Эта переменная может принимать произвольные значения. Напишите программу, использующую цикл `for` для вывода чисел в геометрической прогрессии 1, 2, 4, 8, 16, 32 и т.д.
11. Код ASCII символов нижнего регистра отличается от кода соответствующих символов верхнего регистра на величину **32**. Следовательно, для преобразования строчной буквы в прописную нужно уменьшить ее код на **32**. Используйте это обстоятельство для написания программы, осуществляющей ввод символов с клавиатуры. При выводе результатов данная программа должна преобразовывать строчные буквы в прописные, а прописные — в строчные. Остальные символы не должны изменяться. Работа программы должна завершаться после того, как пользователь введет с клавиатуры точку. И наконец, сделайте так, чтобы программа отображала число символов, для которых был изменен регистр.
12. Что такое бесконечный цикл?
13. Должна ли метка, используемая вместе с оператором `break`, быть определена в кодовом блоке, содержащем этот оператор?