1.3 Conditionals and Loops

In the programs that we have examined to this point, each of the statements in the program is executed once, in the order given. Most programs are more complicated because the sequence of statements and the number of times each is executed can vary. We use the term control flow to refer to statement sequencing in a program. In this section, we introduce statements that allow us to change the control flow, using logic about the values of program variables. This feature is an essential component of programming.

Specifically, we consider Java statements that implement conditionals, where some other statements may or may not be executed depending on certain conditions, and loops, where some other statements may be executed multiple times, again depending on certain conditions. As you will see in numerous examples in this section, conditionals and loops truly harness the power of the computer and will equip you to write programs to accomplish a broad variety of tasks that you could not contemplate attempting without a computer.

If statements Most computations require different actions for different inputs. One way to express these differences in Java is the if statement:

```java
if (<boolean expression>) <statements>
```

This description introduces a formal notation known as a template that we will use to specify the format of Java constructs. We put within angle brackets (<>) a construct that we have already defined, to indicate that we can use any instance of that construct where specified. In this case, <boolean expression> represents an expression that has a boolean value, such as one involving a comparison operation, and <statements> represents a statement block (a sequence of Java statements, each terminated by a semicolon). This latter construct is familiar to you: the body of main() is such a sequence. If the sequence is a single statement, the curly braces are optional. It is possible to make formal definitions of <boolean expression> and <statements>, but we refrain from going into that level of detail. The meaning of an if statement is self-explanatory: the statement(s) in the sequence are to be executed if and only if the expression is true.

As a simple example, suppose that you want to compute the absolute value of an int value x. This statement does the job:

```java
if (x < 0) x = -x;
```

As a second simple example, consider the following statement:

```java
if (x > y)
{
    int t = x;
    x = y;
    y = t;
}
```

This code puts x and y in ascending order by exchanging them if necessary.

You can also add an else clause to an if statement, to express the concept of executing either one statement (or sequence of statements) or another, depending on whether the boolean expression is true or false, as in the following template:

```java
if (<boolean expression>) <statements>
else <statements>
```

As a simple example of the need for an else clause, consider the following code, which assigns the maximum of two int values to the variable max:

```java
if (x > y) max = x;
else max = y;
```

One way to understand control flow is to visualize it with a diagram called a flowchart. Paths through the flowchart correspond to flow-of-control paths in the program:

```
if (x < 0) x = -x;
```

```
if (x > y) max = x;
else max = y;
```

Flows through a flowchart correspond to flow-of-control paths in the program.
Elements of Programming

<table>
<thead>
<tr>
<th>absolute value</th>
<th>if ( x &lt; 0 ) ( x = -x );</th>
</tr>
</thead>
<tbody>
<tr>
<td>put ( x ) and ( y ) into sorted order</td>
<td>if ( x &gt; y )</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>( t = x; )</td>
</tr>
<tr>
<td></td>
<td>( y = x; )</td>
</tr>
<tr>
<td></td>
<td>( x = t; )</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>maximum of ( x ) and ( y )</td>
<td>if ( x &gt; y ) ( \text{max} = x; )</td>
</tr>
<tr>
<td></td>
<td>else ( \text{max} = y; )</td>
</tr>
<tr>
<td>error check for division operation</td>
<td>if ( \text{den} == 0 ) System.out.println(&quot;Division by zero&quot;);</td>
</tr>
<tr>
<td></td>
<td>else System.out.println(&quot;Quotient = ( \text{num}/\text{den} );</td>
</tr>
<tr>
<td>error check for quadratic formula</td>
<td>double ( \text{discriminant} = b^2 - 4ac; )</td>
</tr>
<tr>
<td></td>
<td>if ( \text{discriminant} &lt; 0.0 )</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>System.out.println(&quot;No real roots&quot;);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>System.out.println((-b + Math.sqrt(\text{discriminant}))/2.0);</td>
</tr>
<tr>
<td></td>
<td>System.out.println((-b - Math.sqrt(\text{discriminant}))/2.0);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Typical examples of using if and if-else statements

Program. In the early days of computing, when programmers used low-level languages and difficult-to-understand flows of control, flowcharts were an essential part of programming. With modern languages, we use flowcharts just to understand basic building blocks like the if statement.

The accompanying table contains some examples of the use of if and if-else statements. These examples are typical of simple calculations you might need in programs that you write. Conditional statements are an essential part of programming. Since the semantics (meaning) of statements like these is similar to their meanings as natural-language phrases, you will quickly grow used to them.

Program 1.3.1 is another example of the use of the if-else statement, in this case for the task of simulating a coin flip. The body of the program is a single statement, like the ones in the table above, but it is worth special attention because it introduces an interesting philosophical issue that is worth contemplating: can a computer program produce random values? Certainly not, but a program can produce numbers that have many of the properties of random numbers.

1.3 Conditionals and Loops

Program 1.3.1 Flipping a fair coin

```java
public class Flip {
    public static void main(String[] args) {
        // Simulate a coin flip.
        if (Math.random() < 0.5) System.out.println("Heads");
        else System.out.println("Tails");
    }
}
```

This program uses Math.random() to simulate a coin flip. Each time you run it, it prints either heads or tails. A sequence of flips will have many of the same properties as a sequence that you would get by flipping a fair coin, but it is not a truly random sequence.

**While loops** Many computations are inherently repetitive. The basic Java construct for handling such computations has the following format:

```java
while (<boolean expression>) {
    <statements>
}
```

The while statement has the same form as the if statement (the only difference being the use of the keyword while instead of if), but the meaning is quite different. It is an instruction to the computer to behave as follows: if the expression is false, do nothing; if the expression is true, execute the sequence of statements (just as with if) but then check the expression again, execute the sequence of statements again if the expression is true, and continue as long as the expression is true. We often refer to the statement block in a loop as the body of the loop. As with the if statement, the braces are optional if a while loop body has just one statement. The while statement is equivalent to a sequence of identical if statements:
if (<boolean expression>) { <statements> }
if (<boolean expression>) { <statements> }
if (<boolean expression>) { <statements> }

... At some point, the code in one of the statements must change something (such as the value of some variable in the boolean expression) to make the boolean expression false, and then the sequence is broken.

A common programming paradigm involves maintaining an integer value that keeps track of the number of times a loop iterates. We start at some initial value, and then increment the value by 1 each time through the loop, testing whether it exceeds a predetermined maximum before deciding to continue. TenHello's (PROGRAM 1.3.2) is a simple example of this paradigm that uses a while statement. The key to the computation is the statement

\[ i = i + 1; \]

As a mathematical equation, this statement is nonsense, but as a Java assignment statement it makes perfect sense: it says to compute the value \( i + 1 \) and then assign the result to the variable \( i \). If the value of \( i \) was 4 before the statement, it becomes 5 afterwards; if it was 5 it becomes 6; and so forth. With the initial condition in TenHello's that the value of \( i \) starts at 4, the statement block is executed five times until the sequence is broken, when the value of \( i \) becomes 11.

Using the while loop is barely worthwhile for this simple task, but you will soon be addressing tasks where you will need to specify that statements be repeated far too many times to contemplate doing it without loops. There is a profound difference between programs with while statements and programs without them, because while statements allow us to specify a potentially unlimited number of statements to be executed in a program. In particular, the while statement allows us to specify lengthy computations in short programs. This ability opens the door to writing programs for tasks that we could not contemplate addressing without a

Program 1.3.2  Your first while loop

```java
public class TenHello's {
    public static void main(String[] args) {
        // Print 10 Hello's.
        System.out.println("1st Hello");
        System.out.println("2nd Hello");
        System.out.println("3rd Hello");
        int i = 4;
        while (i <= 10) {
            // Print the i-th Hello.
            System.out.println("i + " + "th Hello");
            i++;
        }
    }
}
```

This program uses a while loop for the simple, repetitive task of printing the output shown below. After the third line, the lines to be printed differ only in the value of the index counting the line printed, so we define a variable \( i \) to contain that index. After initializing the value of \( i \) to 4, we enter into a while loop where we use the value of \( i \) in the System.out.println() statement and increment it each time through the loop. After printing 10th Hello, the value of \( i \) becomes 11 and the loop terminates.

Flowchart example (while statement)
computer. But there is also a price to pay: as your programs become more sophisticated, they become more difficult to understand.

PowersOfTwo (Program 1.3.3) uses a while loop to print out a table of the powers of 2. Beyond the loop control counter $i$, it maintains a variable $v$ that holds the powers of two as it computes them. The loop body contains three statements: one to print the current power of 2, one to compute the next (multiply the current one by 2), and one to increment the loop control counter.

There are many situations in computer science where it is useful to be familiar with powers of 2. You should know at least the first 10 values in this table and you should note that $2^{10}$ is about 1 thousand, $2^{20}$ is about 1 million, and $2^{30}$ is about 1 billion.

PowersOfTwo is the prototype for many useful computations. By varying the computations that change the accumulated value and the way that the loop control variable is incremented, we can print out tables of a variety of functions (see Exercise 1.3.11).

It is worthwhile to carefully examine the behavior of programs that use loops by studying a trace of the program. For example, a trace of the operation of PowersOfTwo should show the value of each variable before each iteration of the loop and the value of the conditional expression that controls the loop. Tracing the operation of a loop can be very tedious, but it is nearly always worthwhile to run a trace because it clearly exposes what a program is doing.

PowersOfTwo is nearly a self-tracing program, because it prints the values of its variables each time through the loop. Clearly, you can make any program produce a trace of itself by adding appropriate System.out.println() statements. Modern programming environments provide sophisticated tools for tracing, but

<table>
<thead>
<tr>
<th>$i$</th>
<th>$v$</th>
<th>$i \leq N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>true</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>true</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>true</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>true</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>true</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
<td>true</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
<td>true</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
<td>true</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
<td>true</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
<td>true</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
<td>true</td>
</tr>
<tr>
<td>17</td>
<td>131072</td>
<td>true</td>
</tr>
<tr>
<td>18</td>
<td>262144</td>
<td>true</td>
</tr>
<tr>
<td>19</td>
<td>524288</td>
<td>true</td>
</tr>
<tr>
<td>20</td>
<td>1048576</td>
<td>true</td>
</tr>
<tr>
<td>21</td>
<td>2097152</td>
<td>true</td>
</tr>
<tr>
<td>22</td>
<td>4194304</td>
<td>true</td>
</tr>
<tr>
<td>23</td>
<td>8388608</td>
<td>true</td>
</tr>
<tr>
<td>24</td>
<td>16777216</td>
<td>true</td>
</tr>
<tr>
<td>25</td>
<td>33554432</td>
<td>true</td>
</tr>
<tr>
<td>26</td>
<td>67108864</td>
<td>true</td>
</tr>
<tr>
<td>27</td>
<td>134217728</td>
<td>true</td>
</tr>
<tr>
<td>28</td>
<td>268435456</td>
<td>true</td>
</tr>
<tr>
<td>29</td>
<td>536870912</td>
<td>true</td>
</tr>
<tr>
<td>30</td>
<td>1073741824</td>
<td>false</td>
</tr>
</tbody>
</table>

Program 1.3.3  Computing powers of two

```java
public class PowersOfTwo {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int v = 1;
        int i = 0;
        while (i <= N) {
            // Print $i$th power of 2.
            System.out.println(i + " \times " + v);
            v = 2 * v;
            i = i + 1;
        }
    }
}
```

This program takes a command-line argument $N$ and prints a table of the powers of 2 that are less than or equal to $2^N$. Each time through the loop, we increment the value of $i$ and double the value of $v$. We show only the first three and the last three lines of the table; the program prints $N+1$ lines.

```
% java PowersOfTwo 5
 1
 1 2
 2 4
 3 8
 4 16
 5 32
```

```
% java PowersOfTwo 29
 1
 1
 2
 2 4
 3 8
 4 16
 5 32
... 134217728
26 268435456
28 536870912
```

This tried-and-true method is simple and effective. You certainly should add print statements to the first few loops that you write, to be sure that they are doing precisely what you expect.

There is a hidden trap in PowersOfTwo, because the largest integer in Java's int data type is $2^{31} - 1$ and the program does not test for that possibility. If you
invoke it with `java PowersOfTwo 31`, you may be surprised by the last line of output:

```
... 1073741824 234783648
```

The variable `v` becomes too large and takes on a negative value because of the way Java represents integers. The maximum value of an `int` is available for use as an `Integer.MAX_VALUE`. A better version of `PROGRAM 1.3.3` would use this value to test for overflow and print an error message if the user types too large a value, though getting such a program to work properly for all inputs is trickier than you might think. (For a similar challenge, see `EXERCISE 1.3.14`.)

As a more complicated example, suppose that we want to compute the largest power of two that is less than or equal to a given positive integer `N`. If `N` is 13 we want the result 8; if `N` is 1000, we want the result 512; if `N` is 64, we want the result 64; and so forth. This computation is simple to perform with a `while` loop:

```java
int v = 1;
while (v <= N/2)
  v = 2*v;
```

It takes some thought to convince yourself that this simple piece of code produces the desired result. You can do so by making these observations:

- `v` is always a power of 2.
- `v` is never greater than `N`.
- `v` increases each time through the loop, so the loop must terminate.
- After the loop terminates, `2^v` is greater than `N`.

Reasoning of this sort is often important in understanding how `while` loops work. Even though many of the loops you will write are much simpler than this one, you should be sure to convince yourself that each loop you write is going to behave as you expect.

The logic behind such arguments is the same whether the loop iterates just a few times, as in `TenHello`, dozens of times, as in `PowersOfTwo`, or millions of times, as in several examples that we will soon consider. That leap from a few tiny cases to a huge computation is profound. When writing loops, understanding how

the values of the variables change each time through the loop (and checking that understanding by adding statements to trace their values and running for a small number of iterations) is essential. Having done so, you can confidently remove those training wheels and truly unleash the power of the computer.

**For loops** As you will see, the `while` loop allows us to write programs for all manner of applications. Before considering more examples, we will look at an alternate Java construct that allows us even more flexibility when writing programs with loops. This alternate notation is not fundamentally different from the basic `while` loop, but it is widely used because it often allows us to write more compact and more readable programs than if we used only `while` statements.

**For notation.** Many loops follow this scheme: initialize an index variable to some value and then use a `while` loop to test a loop continuation condition involving the index variable, where the last statement in the `while` loop increments the index variable. You can express such loops directly with Java's `for` notation:

```java
for (<initialize>; <boolean expression>; <increment>)
{
  <statements>
}
```

This code is, with only a few exceptions, equivalent to:

```java
<initialize>
while (<boolean expression>)
{
  <statements>
  <increment>
}
```

Your Java compiler might even produce identical results for the two loops. In truth, `<initialize>` and `<increment>` can be any statements at all, but we nearly always use for loops to support this typical initialize-and-increment programming idiom. For example, the following two lines of code are equivalent to the corresponding lines of code in `TenHello` (`PROGRAM 1.3.2`):

```java
for (int i = 4; i <= 10; i = i + 1)
  System.out.println("th Hello");
```
1.3 Conditionals and Loops

The accompanying table includes several code fragments with typical examples of loops used in Java code. Some of these relate to code that you have already seen; others are new code for straightforward computations. To cement your understanding of loops in Java, put these code snippets into a class’s code that takes an integer $n$ from the command line (like PowersOfTwo) and compile and run them. Then, write some loops of your own for similar computations of your own invention, or do some of the early exercises at the end of this section. There is no substitute for experience running code that you create yourself, and it is imperative that you develop an understanding of how to write Java code that uses loops.

<table>
<thead>
<tr>
<th>print powers of two</th>
<th>int $v = 1$;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>for ($i = 0$; $i &lt; N$; $i++$)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>System.out.println($i + &quot;+ v&quot;);</td>
</tr>
<tr>
<td></td>
<td>$v = 2 * v$;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>compute a finite sum</th>
<th>int $v = 1$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2 + ... + $N$)</td>
<td>for ($i = 1$; $i &lt; N$; $i++$)</td>
</tr>
<tr>
<td></td>
<td>sum += $i$;</td>
</tr>
<tr>
<td></td>
<td>System.out.println(sum);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>compute a finite product</th>
<th>int $v = 1$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>($N! = 1 \times 2 \times \ldots \times N$)</td>
<td>for ($i = 1$; $i &lt; N$; $i++$)</td>
</tr>
<tr>
<td></td>
<td>product *= $i$;</td>
</tr>
<tr>
<td></td>
<td>System.out.println(product);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>print a table of function values</th>
<th>for ($i = 1$; $i &lt; N$; $i++$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System.out.println($i + &quot;+ Math.PI\times i/N$);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>print the ruler function</th>
<th>String ruler = &quot; &quot;;</th>
</tr>
</thead>
<tbody>
<tr>
<td>(see Program 1.2.1)</td>
<td>for ($i = 1$; $i &lt; N$; $i++$)</td>
</tr>
<tr>
<td></td>
<td>ruler = ruler + &quot;+ ruler;</td>
</tr>
<tr>
<td></td>
<td>System.out.println(ruler);</td>
</tr>
</tbody>
</table>

Typical examples of using for and while statements
Nesting: The if, while, and for statements have the same status as assignment statements or any other statements in Java. That is, we can use them whenever a statement is called for. In particular, we can use one or more of them in the <body> of another to make compound statements. As a first example, DivisorPattern (Program 1.3.4) has a for loop whose statements are a for loop (whose statement is an if statement) and a print statement. It prints a pattern of asterisks where the \( \text{i} \)th row has an asterisk in each position corresponding to divisors of \( \text{i} \) (the same holds true for the columns).

To emphasize the nesting, we use indentation in the program code. We refer to the \( \text{i} \) loop as the outer loop and the \( \text{j} \) loop as the inner loop. The inner loop iterates all the way through for each iteration of the outer loop. As usual, the best way to understand a new programming construct like this is to study a trace.

DivisorPattern has a complicated control structure, as you can see from its flowchart. A diagram like this illustrates the importance of using a limited number of simple control structures in programming. With nesting, you can compose loops and conditionals to build programs that are easy to understand even though they may have a complicated control structure. A great many useful computations can be accomplished with just one or two levels of nesting. For example, many programs in this book have the same general structure as DivisorPattern.

```
public class DivisorPattern {
    public static void main(String[] args) {
        // Print a square that visualizes divisors.
        int N = Integer.parseInt(args[0]);
        for (int i = 1; i < N; i++) {
            System.out.println(i);
            for (int j = 1; j < N; j++) {
                // Print the \( \text{j} \)th entry in the \( \text{i} \)th line.
                if (i % j == 0 || j % i == 0) System.out.print("*");
                else System.out.print("");
            }
            System.out.println();
        }
    }
}
```

This program takes an integer \( \text{N} \) as the command-line argument and uses nested for loops to print an \( \text{N} \) by \( \text{N} \) table with an asterisk in row \( \text{i} \) and column \( \text{j} \) if either \( \text{i} \) divides \( \text{j} \) or \( \text{j} \) divides \( \text{i} \). The loop control variables \( \text{i} \) and \( \text{j} \) control the computation.

```
% java DivisorPattern 3
* *
* * * *
*  * * * * *
* * * * * * *

% java DivisorPattern 16
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * *
```

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i % j</th>
<th>j % i</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
As a second example of nesting, consider the following program fragment, which a tax preparation program might use to compute income tax rates:

```java
if (income < 0) rate = 0.0;
else if (income < 47450) rate = .22;
else if (income < 114650) rate = .25;
else if (income < 174700) rate = .28;
else if (income < 311950) rate = .33;
else
    rate = .35;
```

In this case, a number of if statements are nested to test from among a number of mutually exclusive possibilities. This construct is a special one that we use often. Otherwise, it is best to use braces to resolve ambiguities when nesting if statements. This issue and more examples are addressed in the Q&A and exercises.

Applications The ability to program with loops immediately opens up the full world of computation. To emphasize this fact, we next consider a variety of examples. These examples all involve working with the types of data that we considered in Section 1.2, but rest assured that the same mechanisms serve us well for any computational application. The sample programs are carefully crafted, and by studying and appreciating them, you will be prepared to write your own programs containing loops, as requested in many of the exercises at the end of this section.

The examples that we consider involve computing with numbers, which has the advantage that natural, if not familiar, problems arise. Several of our examples are tied to problems faced by mathematicians and scientists throughout the past several centuries. While computers have existed for only 50 years or so, many of the computational methods that we use are based on a rich mathematical tradition tracing back to antiquity.

Finite sum. The computational paradigm used by PowersOfTwo is one that you will use frequently. It uses two variables—one as an index that controls a loop and the other to accumulate a computational result. Harmonic (Program 1.3.5) uses the same paradigm to evaluate the finite sum \( H_n = 1/1 + 1/2 + 1/3 + \ldots + 1/n \).

These numbers, which are known as the Harmonic numbers, arise frequently in discrete mathematics. Harmonic numbers are the discrete analog of the logarithm. They also approximate the area under the curve \( y = 1/x \). You can use Program 1.3.5 as a model for computing the values of other sums (see Exercise 1.3.16).

**Program 1.3.5 Harmonic numbers**

```java
public class Harmonic {
    public static void main(String[] args) {
        // Compute the Nth Harmonic number.
        double N = Integer.parseInt(args[0]);
        double sum = 0.0;
        for (int i = 1; i <= N; i++) {
            // Add the ith term to the sum
            sum += 1.0/i;
        }
        System.out.println(sum);
    }
}
```

This program computes the value of the Nth Harmonic number. The value is known from mathematical analysis to be about \( \ln(N) + 0.57721 \) for large N. Note that \( \ln(10000) \approx 9.21034 \).

\[
x = f(x)
\]

Computing the square root. How are functions in Java's Math library, such as `Math.sqrt(x)`, implemented? The `sqrt` function (Program 1.3.6) illustrates one technique. To compute the square root function, it uses an iterative computation that was known to the Babylonians over 4,000 years ago. It is also a special case of a general computational technique that was developed in the 17th century by Isaac Newton and Joseph Raphson and is widely known as Newton's method. Under generous conditions on a given function \( f(x) \), Newton's method is an effective way to find roots (values of \( x \) for which the function is 0). Start with an initial estimate, \( x_0 \). Given the
Program 1.3.6  Newton's method

```java
public class Sqrt {
    public static void main(String[] args) {
        double c = Double.parseDouble(args[0]);
        double epsilon = 1e-15;
        double t = c;
        while (Math.abs(t - c/t) > epsilon * t) {
            t = (c/t + t) / 2.0;
        }
        System.out.println(t);
    }
}
```

This program computes the square root of its command-line argument to 15 decimal places of accuracy, using Newton's method (see text).

<table>
<thead>
<tr>
<th>Iteration</th>
<th>t</th>
<th>c/t</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.000000000000000</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>1.500000000000000</td>
<td>1.333333333333333</td>
</tr>
<tr>
<td>2</td>
<td>1.416666666666666</td>
<td>1.411764705828355</td>
</tr>
<tr>
<td>3</td>
<td>1.4142156862745097</td>
<td>1.4142134837447870</td>
</tr>
<tr>
<td>4</td>
<td>1.4142135623740951</td>
<td>1.4142135623740951</td>
</tr>
</tbody>
</table>

Trance of java Sqrt 2.0

---

1.3 Conditionals and Loops

and c/t. With Newton's method, we get the value of the square root of 2 accurate to 15 places in just 5 iterations of the loop.

Newton's method is important in scientific computing because the same iterative approach is effective for finding the roots of a broad class of functions, including many for which analytic solutions are not known (so the Java Math library would be no help). Nowadays, we take for granted that we can find whatever values we need of mathematical functions before computers, scientists and engineers had to use tables or computed values by hand. Computational techniques that were developed to enable calculations by hand needed to be very efficient, so it is not surprising that many of those same techniques are effective when we use computers. Newton's method is a classic example of this phenomenon. Another useful approach for evaluating mathematical functions is due to Taylor series expansions (see Exercises 1.3.35–36).

**Number conversion.** Binary (Program 1.3.7) prints the binary (base 2) representation of the decimal number typed as the command-line argument. It is based on decomposing a number into a sum of powers of two. For example, the binary representation of 19 is 10011, which is the same as saying that $19 = 16 + 2 + 1$. To compute the binary representation of $N$, we consider the powers of 2 less than or equal to $N$ in decreasing order to determine which belong in the binary decomposition (and therefore correspond to a 1 bit in the binary representation). The process corresponds precisely to using a balance scale to weigh an object, using weights whose values are powers of two. First, we find largest weight not heavier than the object. Then, considering the weights in decreasing order, we add each weight to test whether the object is lighter. If so, we remove the

```
16
2
1
```

Scale analog to binary conversion
1.3 Conditionals and Loops

<table>
<thead>
<tr>
<th>n</th>
<th>binary representation</th>
<th>v</th>
<th>v &gt; 0</th>
<th>binary representation</th>
<th>n &lt; v</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>10011</td>
<td>16</td>
<td>true</td>
<td>10000</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>8</td>
<td>true</td>
<td>1000</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>4</td>
<td>true</td>
<td>100</td>
<td>true</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>2</td>
<td>true</td>
<td>10</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>true</td>
<td>1</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Trace of casting-out powers of two loop for Java Binary 19

simulate leaving a weight on the balance, we just subtract that weight from n. The value of v decreases through the powers of two. When it is larger than n, Binary prints 0; otherwise, it prints 1 and subtracts v from n. As usual, a trace (of the values of n, v, n < v, and the output bit for each loop iteration) can be very useful in helping you to understand the program. Read from top to bottom in the rightmost column of the trace, the output is 10011, the binary representation of 19.

Converting data from one representation to another is a frequent theme in writing computer programs. Thinking about conversion emphasizes the distinction between an abstraction (an integer like the number of hours in a day) and a representation of that abstraction (24 or 11000). The irony here is that the computer’s representation of an integer is actually based on its binary representation.

Simulation. Our next example is different in character from the ones we have been considering, but it is representative of a common situation where we use computers to simulate what might happen in the real world so that we can make informed decisions. The specific example that we consider now is from a thoroughly studied class of problems known as gambler’s ruin. Suppose that a gambler makes a series of fair $1 bets, starting with some given initial stake. The gambler always goes broke eventually, but when we set other limits on the game, various questions arise. For example, suppose that the gam-
public class Gambler {
    public static void main(String[] args) {
        // Run T experiments that start with $stake
        // and terminate on $0 or $goal.
        int stake = Integer.parseInt(args[0]);
        int goal = Integer.parseInt(args[1]);
        int T = Integer.parseInt(args[2]);
        int bets = 0;
        int wins = 0;
        for (int t = 0; t < T; t++) {
            // Run one experiment.
            int cash = stake;
            while (cash > 0 && cash < goal) {
                // Simulate one bet.
                bets++;
                if (Math.random() < 0.5) cash++;
                else cash--;
            }
            // Cash is either 0 (ruin) or $goal (win).
            if (cash == goal) wins++;
        }
        System.out.println(100*wins/T + "% wins");
        System.out.println("Avg # bets: "+bets/T);
    }
}

The inner while loop in this program simulates a gambler who makes a series of $1 bets, continuing until going broke or reaching $goal. The running time of this program is proportional to T times the average number of bets. For example, the third command below causes nearly 100 million random numbers to be generated.

% java Gambler 10 20 1000
50% wins
Avg # bets: 100
% java Gambler 50 250 100
19% wins
Avg # bets: 11050
% java Gambler 500 2500 100
21% wins
Avg # bets: 998071

1.3 Conditionals and Loops

Gambler (Program 1.3.8) is a simulation that can help answer these questions. It does a sequence of trials, using Math.random() to simulate the sequence of bets, continuing until the gambler is broke or the goal is reached, and keeping track of the number of wins and the number of bets. After running the experiment for the specified number of trials, it averages and prints out the results. You might wish to run this program for various values of the command-line arguments, not necessarily just to plan your next trip to the casino, but to help you think about the following questions: Is the simulation an accurate reflection of what would happen in real life? How many trials are needed to get an accurate answer? What are the computational limits on performing such a simulation? Simulations are widely used in applications in economics, science, and engineering, and questions of this sort are important in any simulation.

In the case of Gambler, we are verifying classical results from probability theory, which say the probability of success is the ratio of the stake to the goal and that the expected number of bets is the product of the stake and the desired gain (the difference between the goal and the stake). For example, if you want to go to Monte Carlo to try to turn $50 into $2,500, you have a reasonable (20%) chance of success, but you should expect to make a million $1 bets! If you try to turn $1 into $1,000, you have a 1% chance and can expect to be done (ruin, most likely) in about 999 bets.

Simulation and analysis go hand-in-hand, each validating the other. In practice, the value of simulation is that it can suggest answers to questions that might be too difficult to resolve with analysis. For example, suppose that our gambler, recognizing that there will never be enough time to make a million bets, decides ahead of time to set an upper limit on the number of bets. How much money can the gambler expect to take home in that case? You can address this question with an easy change to Program 1.3.8 (see Exercise 1.3.24), but addressing it with mathematical analysis is not so easy.
1.3 Conditionals and Loops

Program 1.3.9  Factoring integers

```java
public class Factors
{
    public static void main(String[] args)
    {
        // Print the prime factors of N.
        long N = Long.parseLong(args[0]);
        long n = N;
        for (long i = 2; i <= n/i; i++)
        {
            // Test whether i is a factor.
            while (n % i == 0)
            {
                // Cast out and print i factors.
                n /= i;
                System.out.print(i + " ");
            } // Any factors of n are greater than i.
        }
    if (n > 1) System.out.print(n);
    System.out.println();
}
```

This program prints the prime factorization of any positive integer in Java’s `long` data type. The code is simple, but it takes some thought to convince oneself that it is correct (see text).

% java Factors 3757208
2 2 2 7 13 13 397
% java Factors 287994832722311
17 1739347 9739789

learn the effectiveness of this simple change. On a computer that can do billions of operations per second, we could factor numbers on the order of 10^9 in a few seconds; with the (i <= n/i) test we can factor numbers on the order of 10^18 in a comparable amount of time. Loops give us the ability to solve difficult problems, but they also give us the ability to construct simple programs that run slowly, so we must always be cognizant of performance.

In modern applications in cryptography, there are important situations where we wish to factor truly huge numbers (with, say, hundreds or thousands of digits). Such a computation is prohibitively difficult even with the use of a computer.

Factoring. A prime is an integer greater than one whose only positive divisors are one and itself. The prime factorization of an integer is the multiset of primes whose product is the integer. For example, 3757208 = 2^2 * 2^2 * 7 * 13 * 13 * 397. Factors (Program 1.3.9) computes the prime factorization of any given positive integer. In contrast to many of the other programs that we have seen (which we could do in a few minutes with a calculator or even a pencil and paper), this computation would not be feasible without a computer. How would you go about trying to find the factors of a number like 287994832722311? You might find the factor 17 quickly, but even with a calculator it would take you quite a while to find 1739347.

Although Factors is compact and straightforward, it certainly will take some thought to for you to convince yourself that it produces the desired result for any given integer. As usual, following a trace that shows the values of the variables at the beginning of each iteration of the outer loop is a good way to understand the computation. For the case where the initial value of n is 3757208, the inner while loop iterates three times when i is 2, to remove the three factors of 2; then zero times when i > 3, 4, and 5, since none of those numbers divide 469651; and so forth. Tracing the program for a few example inputs clearly reveals its basic operation. To convince ourselves that the program will behave as expected for all inputs, we reason about what we expect each of the loops to do. The while loop clearly prints and removes from n all factors of i, but the key to understanding the program is to see that the following fact holds at the beginning of each iteration of the for loop: n has no factors between 2 and i−1. Thus, if i is not prime, it will not divide n; if i is prime, the while loop will do its job. Once we know that n has no factors less than or equal to i, we also know that it has no factors greater than n/i, so we need look no further when i is greater than n/i.

In a more naïve implementation, we might simply have used the condition (i < n) to terminate the for loop. Even given the blazing speed of modern computers, such a decision would have a dramatic effect on the size of the numbers that we could factor. Exercise 1.3.26 encourages you to experiment with the program to

<table>
<thead>
<tr>
<th>i</th>
<th>N</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3757208</td>
<td>2 2 2</td>
</tr>
<tr>
<td>3</td>
<td>469651</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>469651</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>469651</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>469651</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>469651</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>67093</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>67093</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>67093</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>67093</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>67093</td>
<td>13 13</td>
</tr>
<tr>
<td>13</td>
<td>67093</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>397</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>397</td>
<td>397</td>
</tr>
</tbody>
</table>

Trace of Java Factors 3757208
Other conditional and loop constructs. To more fully cover the Java language, we consider here four more control-flow constructs. You need not think about using these constructs for every program that you write, because you are likely to encounter them much less frequently than the if, while, and for statements. You certainly do not need to worry about using these constructs until you are comfortable using if, while, and for. You might encounter one of them in a program in a book or on the web, but many programmers do not use them at all and we do not use any of them outside this section.

Break statement. In some situations, we want to immediately exit a loop without letting it run to completion. Java provides the break statement for this purpose. For example, the following code is an effective way to test whether a given integer N>1 is prime:

```java
int i;
for (i = 2; i <= N/i; i++)
if (N % i == 0) break;
if (i > N/i) System.out.println(N + " is prime");
```

There are two different ways to leave this loop: either the break statement is executed (because i divides N, so N is not prime) or the for loop condition is not satisfied (because no i with i <= N/2 was found that divides N, which implies that N is prime). Note that we have to declare i outside the for loop instead of in the initialization statement so that its scope extends beyond the loop.

Continue statement. Java also provides a way to skip to the next iteration of a loop: the continue statement. When a continue is executed within a loop body, the flow of control transfers directly to the increment statement for the next iteration of the loop.

Switch statement. The if and if-else statements allow one or two alternatives in directing the flow of control. Sometimes, a computation naturally suggests more than two mutually exclusive alternatives. We could use a sequence or a chain of if-else statements, but the Java switch statement provides a direct solution. Let us move right to a typical example. Rather than printing an int variable day in a program that works with days of the weeks (such as a solution to Exercise 1.2.29), it is easier to use a switch statement, as follows:

```java
switch (day) {
    case 0: System.out.println("Sun"); break;
    case 1: System.out.println("Mon"); break;
    case 2: System.out.println("Tue"); break;
    case 3: System.out.println("Wed"); break;
    case 4: System.out.println("Thu"); break;
    case 5: System.out.println("Fri"); break;
    case 6: System.out.println("Sat"); break;
}
```

When you have a program that seems to have a long and regular sequence of if statements, you might consider consulting the booksite and using a switch statement, or using an alternate approach described in Section 1.4.

Do-while statement. Another way to write a loop is to use the template

```java
    do { <statements> } while (<boolean expression>);
```

The meaning of this statement is the same as

```java
    while (<boolean expression>) { <statements> }
```

except that the first test of the condition is omitted. If the condition initially holds, there is no difference. For an example in which do-while is useful, consider the problem of generating points that are randomly distributed in the unit disk. We can use Math.random() to generate x and y coordinates independently to get points that are randomly distributed in the 2-by-2 square centered on the origin. Most points fall within the unit disk, so we just reject those that do not. We always want to generate at least one point, so a do-while loop is ideal for this computation. The following code sets x and y such that the point (x; y) is randomly distributed in the unit disk:

```java
    do // Scale x and y to be random in (-1, 1).
    x = 2.0*Math.random() - 1.0;
    y = 2.0*Math.random() - 1.0;
    while (Math.sqrt(x*x + y*y) > 1.0);
```

Since the area of the disk is \(\pi\) and the area of the square is 4, the expected number of times the loop is iterated is \(4/\pi\) (about 1.27).
### Infinite loops

Before you write programs that use loops, you need to think about the following issue: what if the loop-continuation condition in a while loop is always satisfied? With the statements that you have learned so far, one of two bad things could happen, both of which you need to learn to cope with.

First, suppose that such a loop calls `System.out.println()`. For example, if the condition in `TenHello`s was `(1 > 3)` instead of `(1 <= 10)`, it would always be true. What happens? Nowadays, we use `print` as an abstraction to mean `display in a terminal window` and the result of attempting to display an unlimited number of lines in a terminal window is dependent on operating-system conventions. If your system is set up to have `print mean print characters on a piece of paper`, you might run out of paper or have to unplug the printer. In a terminal window, you need a `stop printing operation`. Before running programs with loops on your own, you make sure that you know what to do to "pull the plug" on an infinite loop of `System.out.println()` calls and then test out the strategy by making the change to `TenHello`s indicated above and trying to stop it. On most systems, `<ctlr-c>` means `stop the current program`, and should do the.

Second, nothing might happen. If your program has an infinite loop that does not produce any output, it will spin through the loop and you will see no results at all. When you find yourself in such a situation, you can inspect the loops to make sure that the loop exit condition always happens, but the program may not be easy to identify. One way to locate such a bug is to insert calls to `System.out.println()` to produce a trace. If these calls fall within an infinite loop, this strategy reduces the problem to the case discussed in the previous paragraph, but the output might give you a clue about what to do.

You might not know (or it might not matter) whether a loop is infinite or just very long. Even `Babbage`s eventually would terminate after printing over a billion lines because of overflow. If you invoke `Program 1.3.8` with arguments such as `Java Gambler 1000000 2000000 100`, you may not want to wait for the answer. You will learn to be aware of and to estimate the running time of your programs.

Why not have `Java` detect infinite loops and warn us about them? You might be surprised to know that it is not possible to do so, in general. This counterintuitive fact is one of the fundamental results of theoretical computer science.

### Summary

For reference, the accompanying table lists the programs that we have considered in this section. They are representative of the kinds of tasks we can address with short programs comprised of `if`, `while`, and for statements processing built-in types of data. These types of computations are an appropriate way to become familiar with the basic Java flow-of-control constructs. The time that you spend now working with as many such programs as you can will certainly pay off for you in the future.

To learn how to use conditionals and loops, you must practice writing and debugging programs with `if`, `while`, and for statements. The exercises at the end of this section provide many opportunities for you to begin this process. For each exercise, you will write a Java program, then run and test it. All programmers know that it is unusual to have a program work as planned the first time it is run, so you will want to have an understanding of your program and an expectation of what it should do, step by step. At first, use explicit traces to check your understanding and expectation. As you gain experience, you will find yourself thinking in terms of what a trace might produce as you compose your loops. Ask yourself the following kinds of questions: What will be the values of the variables after the loop iterates the first time? The second time? The final time? Is there any way this program could get stuck in an infinite loop?

Loops and conditionals are a giant step in our ability to compute: `if`, `while`, and for statements take us from simple straight-line programs to arbitrarily complicated flow of control. In the next several chapters, we will take more giant steps that will allow us to process large amounts of input data and allow us to define and process types of data other than simple numeric types. The `if`, `while`, and for statements of this section will play an essential role in the programs that we consider as we take these steps.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Flip</code></td>
<td>simulate a coin flip your first loop</td>
</tr>
<tr>
<td><code>EightHello</code>s</td>
<td>compute and print a table of values your first nested loop</td>
</tr>
<tr>
<td><code>DivisorsPattern</code></td>
<td>compute finite sum</td>
</tr>
<tr>
<td><code>Harmonic</code></td>
<td>classic iterative algorithm</td>
</tr>
<tr>
<td><code>Newton</code></td>
<td>basic iterative algorithm</td>
</tr>
<tr>
<td><code>Binary</code></td>
<td>simulation with nested loops</td>
</tr>
<tr>
<td><code>Gambler</code></td>
<td>while loop with a for loop</td>
</tr>
<tr>
<td><code>Factor</code></td>
<td></td>
</tr>
</tbody>
</table>
Q. What is the difference between == and =?

A. We repeat this question here to remind you to be sure not to use = when you mean == in a conditional expression. The expression (x = y) assigns the value of y to x, whereas the expression (x == y) tests whether the two variables currently have the same values. In some programming languages, this difference can wreak havoc in a program and be difficult to detect, but Java’s type safety usually will come to the rescue. For example, if we make the mistake of typing (t = goal) instead of (t == goal) in Program 1.3.8, the compiler finds the bug for us:

```
javac Gambler.java
Gambler.java:18: incompatible types
found: int
required: boolean
if (t = goal) wins++;
1 error
```

Be careful about writing if (x = y) when x and y are boolean variables, since this will be treated as an assignment statement, which assigns the value of y to x and evaluates to the true value of y. For example, instead of writing if (isPrime = false), you should write if (isPrime).

Q. So I need to pay attention to using == instead of = when writing loops and conditionals. Is there something else in particular that I should watch out for?

A. Another common mistake is to forget the braces in a loop or conditional with a multi-statement body. For example, consider this version of the code in Gambler:

```
for (int t = 0; t < T; t++)
    for (cash = stake; cash > 0 && cash < goal; bets++)
        if (Math.random() < 0.5) cash++;
        else cash--;
    if (cash == goal) wins++;  
```

The code appears correct, but it is dysfunctional because the second if is outside both for loops and gets executed just once. Our practice of using explicit braces for long statements is precisely to avoid such insidious bugs.

Q. Anything else?

A. The third classic pitfall is ambiguity in nested if statements:

```
if <expr1> if <expr2> <stmtA> else <stmtB>
```

In Java this is equivalent to

```
if <expr1> { if <expr2> <stmtA> else <stmtB> }
```

even if you might have been thinking

```
if <expr1> { if <expr2> <stmtA> } else <stmtB>
```

Again, using explicit braces is a good way to avoid this pitfall.

Q. Are there cases where I must use a for loop but not a while, or vice versa?

A. No. Generally, you should use a for loop when you have an initialization, an increment, and a loop continuation test (if you do not need the loop control variable outside the loop). But the equivalent while loop still might be fine.

Q. What are the rules on where we declare the loop-control variables?

A. Opinions differ. In older programming languages, it was required that all variables be declared at the beginning of a <body>, so many programmers are in this habit and there is a lot of code out there that follows this convention. But it makes a lot of sense to declare variables where they are first used, particularly in for loops, when it is normally the case that the variable is not needed outside the loop. However, it is not uncommon to need to test (and therefore declare) the loop-control variable outside the loop, as in the primality-testing code we considered as an example of the break statement.

Q. What is the difference between ++i and i++?

A. As statements, there is no difference. In expressions, both increment i, but ++i has the value after the increment and i++ the value before the increment. In this book, we avoid statements like x = ++i that have the side effect of changing variable values. So, it is safe to not worry much about this distinction and just use i++
in for loops and as a statement. When we do use ++i in this book, we will call attention to it and say why we are using it.

Q. So, initialize and increment can be any statements whatever in a for loop. How can I take advantage of that?

A. Some experts take advantage of this ability to create compact code fragments, but, as a beginner, it is best for you to use a while loop in such situations. In fact, the situation is even more complicated because initialize and increment can be sequences of statements, separated by commas. This notation allows for code that initializes and modifies other variables besides the loop index. In some cases, this ability leads to compact code. For example, the following two lines of code could replace the last eight lines in the body of the main() method in PowersOfTwo (Program 1.3.3):

```java
    for (int i = 0, v = 1; i <= n; i++, v *= 2)
        System.out.println(i + " " + v);
```

Such code is rarely necessary and better avoided, particularly by beginners.

Q. Can I use a double value as in index in a for loop?

A. It is legal, but generally bad practice to do so. Consider the following loop:

```java
    for (double x = 0.0; x < 1.0; x += 0.1)
        System.out.println(x + " + Math.sin(x));
```

How many times does it iterate? The number of iterations depends on an equality test between double values, which may not always give the result that you expect.

Q. Anything else tricky about loops?

A. Not all parts of a for loop need to be filled in with code. The initialization statement, the boolean expression, the increment statement, and the loop body can each be omitted. It is generally better style to use a while statement than null statements in a for loop. In the code in this book, we avoid null statements.

```
int i = 1;
while (v <= N/2)
    v *= 2;
```

```
for (int i = 1; v <= N/2; i++)
    v *= 2;
```

```
for (int i = 1; v <= N/2; i++)
    v *= 2;
```

Three equivalent loops

### Exercises

1.3.1 Write a program that takes three integer command-line arguments and prints equal if all three are equal, and not equal otherwise.

1.3.2 Write a more general and more robust version of Quadratic (Program 1.3.3) that prints the roots of the polynomial ax^2 + bx + c. prints an appropriate message if the discriminant is negative, and behaves appropriately (avoiding division by zero) if a is zero.

1.3.3 What (if anything) is wrong with each of the following statements?
   a. if (a > b) then c = 0;
   b. if a > b { c = 0; }
   c. if (a > b) c = 0;
   d. if (a > b) c = 0 else b = 0;

1.3.4 Write a code fragment that prints true if the double variables x and y are both strictly less than 0 and x and false otherwise.

1.3.5 Improve your solution to Exercise 1.2.25 by adding code to check that the values of the command-line arguments fall within the range of validity of the formula, and also adding code to print out an error message if that is not the case.

1.3.6 Suppose that i and j are both of type int. What is the value of j after each of the following statements is executed?
   a. for (i = 0; j = 0; i < 10; i++) j += i;
   b. for (i = 0; j = 1; i < 10; i++) j += i;
   c. for (j = 0; j < 10; j++) j += j;
   d. for (i = 0; j = 0; i < 10; i++) j += j;

1.3.7 Rewrite TenHells to make a program Hellos that takes the number of lines to print as a command-line argument. You may assume that the argument is less than 1000. Hint: Use %10 and %100 to determine when to use st, nd, rd, or th for printing the st, nd, rd, or th Hellos.

1.3.8 Write a program that, using one for loop and one if statement, prints the
1.3.9 Write a program that takes an integer \( N \) as a command-line argument, uses \( \text{Math.random()} \) to print \( N \) uniform random values between 0 and 1, and then prints their average value (see Exercise 1.2.30).

1.3.10 Describe what happens when you invoke Program 1.3.4 with an argument that is too large, such as Java Rule 400.

1.3.11 Write a program `FunctionGrowth` that prints a table of the values \( \log N, N, N \log N, N^2, N^3, \) and \( 2^N \) for \( N = 16, 32, 64, \ldots, 2048 \). Use tabs (\( \backslash t \) characters) to line up columns.

1.3.12 What are the values of \( m \) and \( n \) after executing the following code?

```java
int n = 123456789;
int m = 0;
while (n != 0)
{
    m = (10 * m) + (n % 10);
    n = n / 10;
}
```

1.3.15 Expand your solution to Exercise 1.2.24 to print a table giving the total amount paid and the remaining principal after each monthly payment.

1.3.16 Unlike the harmonic numbers, the sum \( 1/1^2 + 1/2^2 + \ldots + 1/N^2 \) does converge to a constant as \( N \) grows to infinity. (Indeed, the constant is \( \pi^2/6 \), so this formula can be used to estimate the value of \( \pi \).) Which of the following for loops computes this sum? Assume that \( N \) is an `int` initialized to 1000000 and `sum` is a double initialized to 0.0.

a. for (int `i` = 1; `i` <= `N`; `i`++) `sum` += 1 / (`i` * `i`);

b. for (int `i` = 1; `i` <= `N`; `i`++) `sum` += 1.0 / `i`;

c. for (int `i` = 1; `i` <= `N`; `i`++) `sum` += 1.0 / (`i` * `i`);

d. for (int `i` = 1; `i` <= `N`; `i`++) `sum` += 1 / (1.0 + `i`);

1.3.17 Show that Program 1.3.6 implements Newton's method for finding the square root of \( c \). Hint: Use the fact that the slope of the tangent to \( f(x) = x^2 \) at \( x = t \) is \( f'(t) \) to find the equation of the tangent line and then use that equation to find the point where the tangent line intersects the \( x \)-axis to show that you can use Newton's method to find a root of any function as follows: at each iteration, replace the estimate \( r \) by \( 1 - f(t) / f'(t) \).

1.3.18 Using Newton's method, develop a program that takes integers \( N \) and \( k \) as command-line arguments and prints the \( k \)th root of \( N \) (Hint: see Exercise 1.3.17).

1.3.19 Modify `BinarySearch` to get a program `Kary` that takes \( i \) and \( k \) as command-line arguments and converts \( i \) to base \( k \). Assume that \( i \) is an integer in Java's `long` data type and that \( k \) is an integer between 2 and 16. For bases greater than 10, use the letters A through F to represent the 11th through 16th digits, respectively.
1.3.20 Write a code fragment that puts the binary representation of a positive integer \( N \) into a String \( s \).

Solution. Java has a built-in method `Integer.toBinaryString(N)` for this job, but the point of the exercise is to see how such a method might be implemented. Working from Program 1.3.7, we get the solution

```java
String s = "";
int v = 1;
while (v <= N/2) v = 2*v;
while (v > 0)
{
    if (n >= v) { s = s + 0; }
    else { s = s + 1; n = n - v; }
    v = v/2;
}
```

A simpler option is to work from right to left:

```java
String s = "";
for (int n = N; n > 0; n /= 2)
    s = (n % 2) + s;
```

Both of these methods are worthy of careful study.

1.3.21 Write a version of `Gambler` that uses two nested while loops or two nested for loops instead of a while loop inside a for loop.

1.3.22 Write a program `GamblerPlot` that traces a gambler's ruin simulation by printing a line after each bet \( n \) which one asterisk corresponds to each dollar held by the gambler.

1.3.23 Modify `Gambler` to take an extra command-line argument that specifies the (fixed) probability that the gambler wins each bet. Use your program to try to learn how this probability affects the chance of winning and the expected number of bets. Try a value of \( p \) close to .5 (say, .48).

1.3.24 Modify `Gambler` to take an extra command-line argument that specifies the number of bets the gambler is willing to make, so that there are three possible ways for the game to end: the gambler wins, loses, or runs out of time. Add to the output to give the expected amount of money the gambler will have when the game ends. Extra credit: Use your program to plan your next trip to Monte Carlo.

1.3.25 Modify `Factors` to print just one copy each of the prime divisors.

1.3.26 Run quick experiments to determine the impact of using the termination condition \((i <= N/i)\) instead of \((i < N)\) in `Factors` in Program 1.3.9. For each method, find the largest \( n \) such that when you type in an \( n \) digit number, the program is sure to finish within 10 seconds.

1.3.27 Write a program `Checkerboard` that takes one command-line argument \( N \) and uses a loop within a loop to print out a two-dimensional \( N \times N \) checkerboard pattern with alternating spaces and asterisks.

1.3.28 Write a program `GCD` that finds the greatest common divisor (gcd) of two integers using Euclid's algorithm, which is an iterative computation based on the following observation: if \( x \) is greater than \( y \), then if \( y \) divides \( x \), the gcd of \( x \) and \( y \) is \( y \); otherwise, the gcd of \( x \) and \( y \) is the same as the gcd of \( x \) \( \% \) \( y \) and \( y \).

1.3.29 Write a program `RelativelyPrime` that takes one command-line argument \( N \) and prints out an \( N \times N \) table such that there is an * in row \( i \) and column \( j \) if the gcd of \( i \) and \( j \) is 1 (i and \( j \) are relatively prime) and a space in that position otherwise.

1.3.30 Write a program `Powers0FK` that takes an integer \( k \) as command-line argument and prints all the positive powers of \( k \) in the Java `long` data type. Note: The constant `Long.MAX_VALUE` is the value of the largest integer in `long`.

1.3.31 Generate a random point \((x, y, z)\) on the surface of a sphere using Marsaglia’s method: Pick a random point \((a, b)\) in the unit disk using the method described at the end of this section. Then, set \( x = 2a \sqrt{1-a^2-b^2} \), \( y = 2b \sqrt{1-a^2-b^2} \), and \( z = 1-2(a^2+b^2) \).
1.3.32 Ramanujan's taxi. Srinivasa Ramanujan was an Indian mathematician who became famous for his intuition for numbers. When the English mathematician G. H. Hardy came to visit him one day, Hardy remarked that the number of his taxi was 1729, a rather dull number. To which Ramanujan replied, "No, Hardy! No, Hardy! It is a very interesting number. It is the smallest number expressible as the sum of two cubes in two different ways." Verify this claim by writing a program that takes a command-line argument N and prints out all integers less than or equal to N that can be expressed as the sum of two cubes in two different ways. In other words, find distinct positive integers a, b, c, and d such that \( a^3 + b^3 = c^3 + d^3 \). Use four nested for loops.

1.3.33 Checksum. The International Standard Book Number (ISBN) is a 10-digit code that uniquely specifies a book. The rightmost digit is a checksum digit that can be uniquely determined from the other 9 digits, from the condition that \( d_1 + 2d_2 + 3d_3 + \ldots + 10d_{10} \) must be a multiple of 11 (here \( d_i \) denotes the ith digit from the right). The checksum digit \( d_{10} \) can be any value from 0 to 10. The ISBN convention is to use the character 'X' to denote 10. Example: the checksum digit corresponding to 020131452 is 5 since 5 is the only value of \( x \) between 0 and 10 for which

\[
100 + 92 + 80 + 71 + 63 + 51 + 44 + 35 + 22 + 12 + x
\]

is a multiple of 11. Write a program that takes a 9-digit integer as a command-line argument, computes the checksum, and prints out the ISBN number.

1.3.34 Counting primes. Write a program PrimeCounter that takes a command-line argument N and finds the number of primes less than or equal to N. Use it to print out the number of primes less than or equal to 10 million. Note: if you are not careful, your program may not finish in a reasonable amount of time!

1.3.35 2D random walk. A two-dimensional random walk simulates the behavior of a particle moving in a grid of points. At each step, the random walker moves north, south, east, or west with probability equal to 1/4, independent of previous moves. Write a program RandomWalker that takes a command-line argument N and estimates how long it will take a random walker to hit the boundary of a 2N-by-2N square centered at the starting point.

1.3.36 Exponential function. Assume that x is a positive variable of type double. Write a code fragment that uses the Taylor series expansion to set the value of sum to \( e^x = 1 + x + x^2/2! + x^3/3! + \ldots \).

Solution. The purpose of this exercise is to get you to think about how a library function like Math.exp() might be implemented in terms of elementary operators. Try solving it, then compare your solution with the one developed here.

We start by considering the problem of computing one term. Suppose that x and term are variables of type double and n is a variable of type int. The following code fragment sets term to \( x^n / n! \) using the direct method of having one loop for the numerator and another loop for the denominator, then dividing the results:

```java
double num = 1.0, den = 1.0;
for (int i = 1; i <= n; i++) num *= x;
for (int i = 1; i <= n; i++) den *= i;
double term = num/den;
```

A better approach is to use just a single for loop:

```java
double term = 1.0;
for (int i = 1; i <= n; i++) term *= x/i;
```

Besides being more compact and elegant, the latter solution is preferable because it avoids inaccuracies caused by computing with huge numbers. For example, the two-loop approach breaks down for values like \( x = 10 \) and \( N = 100 \) because 100! is too large to represent as a double.

To compute \( e^x \), we nest this for loop within another for loop:

```java
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum + term < n); n++)
{
    sum += term;
    term = 1.0;
    for (int i = 1; i <= n; i++) term *= x/i;
}
```

The number of times the loop iterates depends on the relative values of the next term and the accumulated sum. Once the value of the sum stops changing, we
leave the loop. (This strategy is more efficient than using the termination condition \((\text{term} > 0)\) because it avoids a significant number of iterations that do not change the value of the sum.) This code is effective, but it is inefficient because the inner for loop recomputes all the values it computed on the previous iteration of the outer for loop. Instead, we can make use of the term that was added in on the previous loop iteration and solve the problem with a single for loop:

```java
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; n++)
{
    sum += term;
    term *= x/n;
}
```

1.3.37 Trigonometric functions. Write two programs, \(\sin\) and \(\cos\), that compute the sine and cosine functions using their Taylor series expansions

\[
\sin x = -x + x^3/3! - x^5/5! + \ldots\quad \text{and}\quad \cos x = 1 + x^2/2! - x^4/4! + \ldots .
\]

1.3.38 Experimental analysis. Run experiments to determine the relative costs of \text{Math.exp()} and the methods from Exercise 1.3.36 for computing \(e^n\): the direct method with nested for loops, the improvement with a single for loop, and the latter with the termination condition \((\text{term} > 0)\). Use trial-and-error with a command-line argument to determine how many times your computer can perform each computation in 10 seconds.

1.3.39 Pepys problem. In 1693 Samuel Pepys asked Isaac Newton which is more likely: getting 1 at least once when rolling a fair die six times or getting 1 at least twice when rolling it 12 times. Write a program that could have provided Newton with a quick answer.

1.3.40 Game simulation. In the 1970s game \textit{Let's Make a Deal}, a contestant is presented with three doors. Behind one of them is a valuable prize. After the contestant chooses a door, the host opens one of the other two doors (never revealing the prize, of course). The contestant is then given the opportunity to switch to the other unopened door. Should the contestant do so? Intuitively, it might seem that

the contestant's initial choice door and the other unopened door are equally likely to contain the prize, so there would be no incentive to switch. Write a program \texttt{MonteHall} to test this intuition by simulation. Your program should take a command-line argument \(N\), play the game \(N\) times using each of the two strategies (switch or do not switch), and print the chance of success for each of the two strategies.

1.3.41 Median-of-5. Write a program that takes five distinct integers from the command line and prints the median value (the value such that two of the others are smaller and two are larger). \textit{Extra credit:} Solve the problem with a program that compares values fewer than seven times for any given input.

1.3.42 Sorting three numbers. Suppose that the variables \(a, b, c,\) and \(t\) are all of the same numeric primitive type. Prove that the following code puts \(a, b,\) and \(c\) in ascending order:

```java
if (a > b) { t = a; a = b; b = t; }
if (a > c) { t = a; a = c; c = t; }
if (b > c) { t = b; b = c; c = t; }
```

1.3.43 Chaos. Write a program to study the following simple model for population growth, which might be applied to study fish in a pond, bacteria in a test tube, or any of a host of similar situations. We suppose that the population ranges from 0 (extinct) to 1 (maximum population that can be sustained). If the population at time \(t\) is \(x\), then we suppose the population at time \(t+1\) to be \(rx(1-x)\), where the argument \(r\), known as the \textit{fecundity parameter}, controls the rate of growth. Start with a small population—say, \(x = 0.01\)—and study the result of iterating the model, for various values of \(r\). For which values of \(r\) does the population stabilize at \(x = 1 - 1/r\)? Can you say anything about the population when \(r = 3.5, 3.8, 5\)?

1.3.44 Euler's sum-of-powers conjecture. In 1769 Leonhard Euler formulated a generalized version of Fermat's Last Theorem, conjecturing that at least \(n\) \(n\)-th powers are needed to obtain a sum that is itself an \(n\)-th power, for \(n > 2\). Write a program to disprove Euler's conjecture (which stood until 1967), using a quintuply nested loop to find four positive integers whose 5th power sums to the 5th power of another positive integer. That is, find \(a, b, c, d,\) and \(e\) such that

\[
a^5 + b^5 + c^5 + d^5 = e^5.
\]

Use the \texttt{long} data type.