Program 1.3.1 Flipping a fair coin

```
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.*

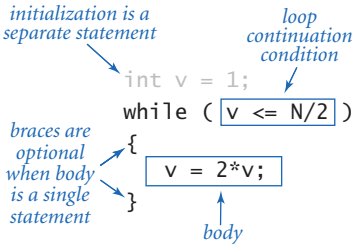
```
% java Flip
Heads
% java Flip
Tails
% java Flip
Tails
```

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

```
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> }
...
```



Anatomy of a `while` *loop*

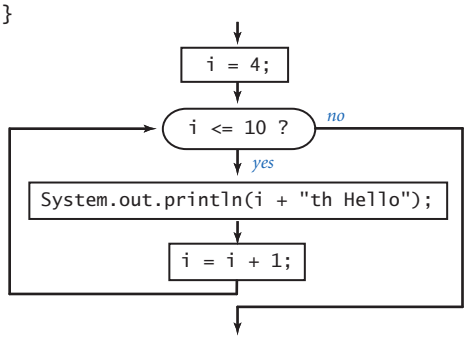
At some point, the code in one of the statements must change something (such as the value of some variable in the boolean expression) to make the boolean expression `false`, and then the sequence is broken.

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. `TenHellos` (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 `TenHellos` 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`.

```
int i = 4;
while (i <= 10)
{
   System.out.println(i + "th Hello");
   i = i + 1;
}
```



Flowchart example (while statement)

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

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

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.*

```
% java TenHellos
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

i	i <= 10	*output*
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Trace of `java TenHellos`

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

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

Trace of `java PowersOfTwo 29`

Program 1.3.3 Computing powers of two

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

N	*loop termination value*
i	*loop control counter*
v	*current power of 2*

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
0 1
1 2
2 4
3 8
4 16
5 32
```

```
% java PowersOfTwo 29
0 1
1 2
2 4
...
27 134217728
28 268435456
29 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

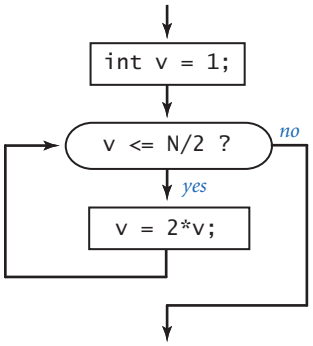
invoke it with `java PowersOfTwo 31`, you may be surprised by the last line of output:

```
...
1073741824
-2147483648
```

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 us to use as `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:

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



Flowchart for the statements

```
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 `TenHellos`, 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:

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

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

```
<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 `TenHellos` (PROGRAM 1.3.2):

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

Typically, we work with a slightly more compact version of this code, using the shorthand notation discussed next.

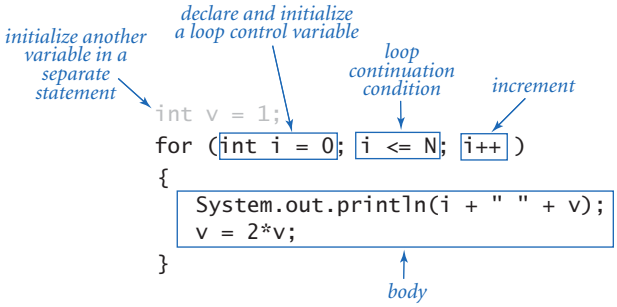
Compound assignment idioms. Modifying the value of a variable is something that we do so often in programming that Java provides a variety of different shorthand notations for the purpose. For example, the following four statements all increment the value of `i` by `1` in Java:

```
i = i + 1;   i++;   ++i;   i += 1;
```

You can also say `i--` or `--i` or `i -= 1` or `i = i-1` to decrement that value of `i` by 1. Most programmers use `i++` or `i--` in `for` loops, though any of the others would do. The `++` and `--` constructs are normally used for integers, but the *compound assignment* constructs are useful operations for any arithmetic operator in any primitive numeric type. For example, you can say `v *= 2` or `v += v` instead of `v = 2*v`. All of these idioms are for notational convenience, nothing more. This combination of shortcuts came into widespread use with the C programming language in the 1970s and have become standard. They have survived the test of time because they lead to compact, elegant, and easily understood programs. When you learn to write (and to read) programs that use them, you will be able to transfer that skill to programming in numerous modern languages, not just Java.

Scope. The scope of a variable is the part of the program where it is defined. Generally the scope of a variable is comprised of the statements that follow the declaration in the same block as the declaration. For this purpose, the code in the `for` loop header is considered to be in the same block as the `for` loop body. Therefore, the `while` and `for` formulations of loops are not quite equivalent: in a typical `for` loop, the incrementing variable is *not* available for use in later statements; in the corresponding `while` loop, it is. This distinction is often a reason to use a `while` instead of a `for` loop.

Choosing among different formulations of the same computation is a matter of each programmer's taste, as when a writer picks from among synonyms or chooses between using active and passive voice when composing a sentence. You will not find good hard-and-fast rules on how to compose a program any more than you will find such rules on how to compose a paragraph. Your goal should be to find a style that suits you, gets the computation done, and can be appreciated by others.



Anatomy of a `for` *loop (that prints powers of 2)*

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 the experience gained by running code that you create yourself, and it is imperative that you develop an understanding of how to write Java code that uses loops.

print largest power of two less than or equal to N	`int v = 1;`
	`while (v <= N/2)`
	`  v = 2*v;`
	`System.out.println(v);`

compute a finite sum (1 + 2 + . . . + N)	`int sum = 0;`
	`for (int i = 1; i <= N; i++)`
	`  sum += i;`
	`System.out.println(sum);`

compute a finite product (N! = 1 × 2 × . . . × N)	`int product = 1;`
	`for (int i = 1; i <= N; i++)`
	`  product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0; i <= N; i++)` `  System.out.println(i + " " + 2*Math.PI*i/N);`

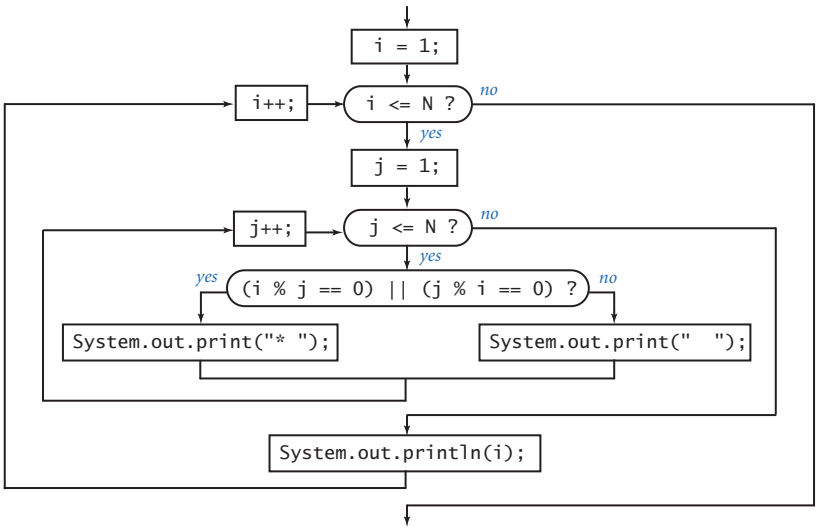
print the ruler function (see Program 1.2.1)	`String ruler = " ";`
	`for (int i = 1; i <= N; i++)`
	`  ruler = ruler + i + ruler;`
	`System.out.println(ruler);`

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 *i*th row has an asterisk in each position corresponding to divisors of *i* (the same holds true for the columns).

To emphasize the nesting, we use indentation in the program code. We refer to the `i` loop as the *outer* loop and the `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`.



Flowchart for `DivisorPattern`

Program 1.3.4 Your first nested loops

```
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++)
      {  // Print the ith line
         for (int j = 1; j <= N; j++)
         {  // Print the jth entry in the ith line.
            if ((i % j == 0) || (j % i == 0))
               System.out.print("* ");
            else
               System.out.print("  ");
         }
         System.out.println(i);
      }
   }
}
```

N	*number of rows and columns*
i	*row index*
j	*column index*

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

```
% java DivisorPattern 3
* * *  1
* *    2
*   *  3

% java DivisorPattern 16
* * * * * * * * * * * * * * * * 1
* *   *   *   *   *   *   *   * 2
*   *     *     *     *     *   3
* *   *       *       *       * 4
*       *         *         *   5
* * *     *           *         6
*           *               *   7
* *   *       *               * 8
*   *           *               9
* *     *         *             10
*                   *           11
* * * *   *           *         12
*                       *       13
* *         *               *   14
*   *   *                   *   15
* *   *       *               * 16
```

i	j	i % j	j % i	*output*
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
				1
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
				2
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

Trace of java DivisorPattern 3

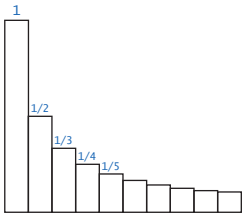
As a second example of nesting, consider the following program fragment, which a tax preparation program might use to compute income tax rates:

```
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 here involve computing with numbers. 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 $H_N = 1 + 1/2 + 1/3 + ... + 1/N$. 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

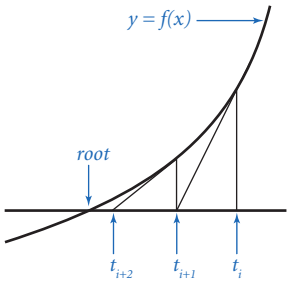
```
public class Harmonic
{
   public static void main(String[] args)
   {  // Compute the Nth Harmonic number.
      int N = Integer.parseInt(args[0]);
      double sum = 0.0;
      for (int i = 1; i <= N; i++)
      {  // Add the ith term to the sum
         sum += 1.0/i;
      }
      System.out.println(sum);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

N	*number of terms in sum*
i	*loop index*
sum	*cumulated 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) ≈ 9.21034.

```
% java Harmonic 2
1.5
% java Harmonic 10
2.9289682539682538
% java Harmonic 10000
9.787606036044348
```

Computing the square root. How are functions in Java's `Math` library, such as `Math.sqrt()`, implemented? `Sqrt` (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, t_0. Given the



Newton's method

Program 1.3.6 Newton's method

```
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)
      {  // Replace t by the average of t and c/t.
         t = (c/t + t) / 2.0;
      }
      System.out.println(t);
   }
}
```

c	*argument*
epsilon	*error tolerance*
t	*estimate of c*

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

```
% java Sqrt 2.0
1.414213562373095
% java Sqrt 2544545
1595.1630010754388
```

iteration	t	c/t
	2.0000000000000000	1.0
1	1.5000000000000000	1.3333333333333333
2	1.4166666666666665	1.4117647058823530
3	1.4142156862745097	1.4142114384748700
4	1.4142135623746899	1.4142135623715002
5	1.4142135623730950	1.4142135623730951

Trace of `java Sqrt 2.0`

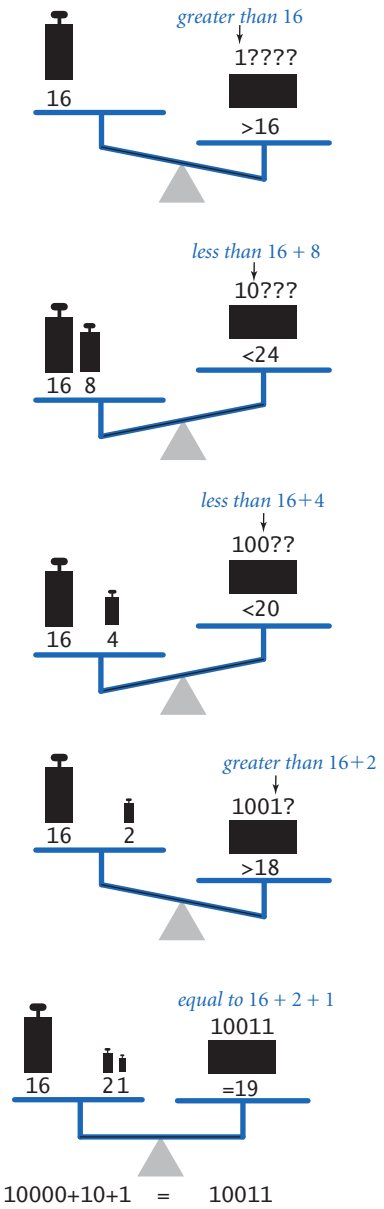
estimate t_i, compute a new estimate by drawing a line tangent to the curve $y = f(x)$ at the point $(t_i, f(t_i))$ and set t_{i+1} to the x-coordinate of the point where that line hits the x-axis. Iterating this process, we get closer to the root.

Computing the square root of a positive number c is equivalent to finding the positive root of the function $f(x) = x^2 - c$. For this special case, Newton's method amounts to the process implemented in `Sqrt` (see EXERCISE 1.3.17). Start with the estimate $t = c$. If t is equal to c/t, then t is equal to the square root of c, so the computation is complete. If not, refine the estimate by replacing t with the average of t

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 to use 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



Scale analog to binary conversion

Program 1.3.7 Converting to binary

```
public class Binary
{
   public static void main(String[] args)
   {  // Print binary representation of N.
      int N = Integer.parseInt(args[0]);
      int v = 1;
      while (v <= N/2)
         v = 2*v;
      // Now v is the largest power of 2 <= N.

      int n = N;
      while (v > 0)
      {  // Cast out powers of 2 in decreasing order.
         if (n < v) { System.out.print(0);         }
         else       { System.out.print(1); n -= v; }
         v = v/2;
      }
      System.out.println();
   }
}
```

N	*integer to convert*
v	*current power of 2*
n	*current excess*

This program prints the binary representation of a positive integer given as the command-line argument, by casting out powers of 2 in decreasing order (see text).

```
% java Binary 19
10011
% java Binary 100000000
101111101011110000100000000
```

weight; if not, we leave the weight and try the next one. Each weight corresponds to a bit in the binary representation of the weight of the object: leaving a weight corresponds to a 1 bit in the binary representation of the object's weight, and removing a weight corresponds to a 0 bit in the binary representation of the object's weight.

In `Binary`, the variable `v` corresponds to the current weight being tested, and the variable `n` accounts for the excess (unknown) part of the object's weight (to

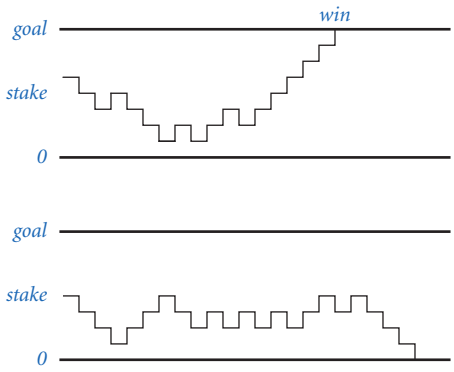
n	*binary representation*	v	v > 0	*binary representation*	n < v	*output*
19	10011	16	true	10000	false	1
3	0011	8	true	1000	true	0
3	011	4	true	100	true	0
3	01	2	true	10	false	1
1	1	1	true	1	false	1
0		0	false			

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-



Gambler simulation sequences

Program 1.3.8 Gambler's ruin simulation

```
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);
   }
}
```

stake	*initial stake*
goal	*walkaway goal*
T	*number of trials*
bets	*bet count*
wins	*win count*
cash	*cash on hand*

The inner `while` *loop in this program simulates a gambler with* `$stake` *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
```

bler decides ahead of time to walk away after reaching a certain goal. What are the chances that the gambler will win? How many bets might be needed to win or lose the game? What is the maximum amount of money that the gambler will have during the course of the game?

`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 $500 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.

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*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 `287994837222311`? You might find the factor `17` quickly, but even with a calculator it would take you quite a while to find `1739347`.

i	N	*output*
2	3757208	2 2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	
16	397	
17	397	
18	397	
19	397	
20	397	
		397

Trace of `java Factors 3757208`

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 `for` 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` is `3`, `4`, `5`, and `6`, 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 blinding 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

Program 1.3.9 Factoring integers

```
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();
   }
}
```

N	*integer to factor*
n	*unfactored part*
i	*potential factor*

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 287994837222311
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.

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:

```
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/i` 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:

```
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 loop. Another way to write a loop is to use the template

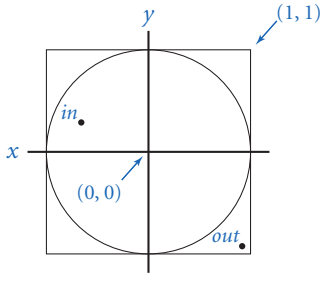
```
do { <statements> } while (<boolean expression>);
```

The meaning of this statement is the same as

```
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:

```
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 π 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 `TenHellos` were `(i > 3)` instead of `(i <= 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 `TenHellos` indicated above and trying to stop it. On most systems, `<ctrl-c>` means *stop the current program*, and should do the job.

```
public class BadHellos
...
int i = 4;
while (i > 3)
{
   System.out.println
      (i + "th Hello");
   i = i + 1;
}
...
```

```
% java BadHellos
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
...
```

An infinite loop

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 problem 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 `BadHellos` 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 100000 200000 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.

program	*description*
`Flip`	simulate a coin flip
`TenHellos`	your first loop
`PowersOfTwo`	compute and print a table of values
`DivisorPattern`	your first nested loop
`Harmonic`	compute finite sum
`Sqrt`	classic iterative algorithm
`Binary`	basic number conversion
`Gambler`	simulation with nested loops
`Factors`	`while` loop within a `for` loop

Summary of programs in this section

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.

Q&A

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 truth 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> <stmntA> else <stmntB>
```

In Java this is equivalent to

```
if <expr1> { if <expr2> <stmntA> else <stmntB> }
```

even if you might have been thinking

```
if <expr1> { if <expr2> <stmntA> } else <stmntB>
```

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 whatsoever 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):

```
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 an index in a `for` loop?

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

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

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

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

null increment statement

null loop body

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 1.2.3) 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 between `0` and `1` 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 ranges 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 += j;
```

c.
```
for (j = 0; j < 10; j++) j += j;
```

d.
```
for (i = 0, j = 0; i < 10; i++) j += j++;
```

1.3.7 Rewrite `TenHellos` 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 `i % 10` and `i % 100` to determine when to use `st`, `nd`, `rd`, or `th` for printing the `i`th `Hello`.

1.3.8 Write a program that, using one `for` loop and one `if` statement, prints the

integers from 1,000 to 2,000 with five integers per line. Hint: Use the % operation.

1.3.9 Write a program that takes an integer N as a command-line argument, uses 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 try to print a ruler function (see the table on page 57) with a value of N that is too large, such as 100.

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 (\t characters) to line up columns.

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

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

1.3.13 What does the following program print ?

```
int f = 0, g = 1;
for (int i = 0; i <= 15; i++)
{
   System.out.println(f);
   f = f + g;
   g = f - g;
}
```

Solution. Even an expert programmer will tell you that the only way to understand a program like this is to trace it. When you do, you will find that it prints the values 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, 233, 377, and 610. These numbers are the first sixteen of the famous *Fibonacci sequence*, which are defined by the following formulas: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$. The Fibonacci sequence arises in a surprising variety of contexts, they have been studied for centuries, and

many of their properties are well-known. For example, the ratio of successive numbers approaches the *golden ratio* ϕ (about 1.618) as n approaches infinity.

1.3.14 Write a program that takes a command-line argument N and prints all the positive powers of two less than or equal to N. Make sure that your program works properly for all values of N. (`Integer.parseInt()` will generate an error if N is too large, and your program should print nothing if N is negative.)

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 $1/1^2 + 1/2^2 + ... + 1/N^2$ *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 π.) 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*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*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 a (differentiable) function $f(x)$ 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 t by $t - f(t) / f'(t)$.

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 `Binary` 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

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

A simpler option is to work from right to left:

```
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 in 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`-by-`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`-by-`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 `PowersOfK` 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 $x = 2\,a\,\sqrt{1 - a^2 - b^2}$. $y = 2\,b\,\sqrt{1 - a^2 - b^2}$. and $z = 1 - 2\,(a^2 + b^2)$.

Creative Exercises

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 $d_1 + 2d_2 + 3d_3 + ... + 10d_{10}$ must be a multiple of 11 (here d_i denotes the ith digit from the right). The checksum digit d_i 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

$$10 \cdot 0 + 9 \cdot 2 + 8 \cdot 0 + 7 \cdot 1 + 6 \cdot 3 + 5 \cdot 1 + 4 \cdot 4 + 3 \cdot 5 + 2 \cdot 2 + 1 \cdot x$$

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 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:

```
double num = 1.0, dem = 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:

```
double term = 1.0;
for (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:

```
double term = 1.0;
double sum = 0.0;
for (int n = 1; sum != sum + term; 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 `(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:

```
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 $\sin x = x - x^3/3! + x^5/5! - \ldots$ and $\cos x = 1 - x^2/2! + x^4/4! - \ldots$.

1.3.38 *Experimental analysis.* Run experiments to determine the relative costs of `Math.exp()` and the methods from EXERCISE 1.3.36 for computing e^x: the direct method with nested `for` loops, the improvement with a single `for` loop, and the latter with the termination condition `(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 show *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 `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). *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:

```
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 *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 is 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 nth powers are needed to obtain a sum that is itself an nth 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 `long` data type.