SESSION 3

Programming Languages for Objects

Programming in the Small II: Control

THE BASIC BUILDING BLOCKS of programs -- variables, expressions, assignment statements, and subroutine call statements -- were covered in the previous chapter. Starting with this chapter, we look at how these building blocks can be put together to build complex programs with more interesting behavior.

Since we are still working on the level of "programming in the small" in this chapter, we are interested in the kind of complexity that can occur within a single subroutine. On this level, complexity is provided by control structures. The two types of control structures, loops and branches, can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. Java includes several control structures of each type, and we will look at each of them in some detail.

Program complexity can be seen not just in control structures but also in data structures. A data structure is an organized collection of data, chunked together so that it can be treated as a unit. Section 3.8 in this chapter includes an introduction to one of the most common data structures: arrays.

The chapter will also begin the study of program design. Given a problem, how can you come up with a program to solve that problem? We'll look at a partial answer to this question in Section 3.2. Finally, Section 3.9 is a very brief first look at GUI programming

Programming in the Large I: Subroutines

ONE WAY TO BREAK UP A COMPLEX PROGRAM into manageable pieces is to use subroutines. A subroutine consists of the instructions for carrying out a certain task, grouped together and given a name. Elsewhere in the program, that name can be used as a stand-in for the whole set of instructions. As a computer executes a program, whenever it encounters a subroutine name, it executes all the instructions necessary to carry out the task associated with that subroutine.

Subroutines can be used over and over, at different places in the program. A subroutine can even be used inside another subroutine. This allows you to write simple subroutines and then use them to help write more complex subroutines, which can then be used in turn in other subroutines. In this way, very complex programs can be built up step-by-step, where each step in the construction is reasonably simple.

Subroutines in Java can be either static or non-static. This chapter covers static subroutines. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

Blocks, Loops, and Branches

THE ABILITY OF A COMPUTER TO PERFORM complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are just six such structures that are used to determine the normal flow of control in a program -- and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the block, the while loop, the do..while loop, the for loop, the if statement, and the switch statement. Each of these structures is considered to be a single "statement," but a **structured** statement that can contain one or more other statements inside itself.

3.1.1 Blocks

The block is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{
      statements
}
```

That is, it consists of a sequence of statements enclosed between a pair of braces, "{" and "}". In fact, it is possible for a block to contain no statements at all; such a block is called an empty block, and can actually be useful at times. An empty block consists of nothing but an empty pair of braces. Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit. However, a block can be legally used wherever a statement can occur. There is one place where a block is required: As you might have already noticed in the case of the `main` subroutine of a program, the definition of a subroutine is a block, since it is a sequence of statements enclosed inside a pair of braces.

I should probably note again at this point that Java is what is called a free-format language. There are no syntax rules about how the language has to be arranged on a page. So, for example, you could write an entire block on one line if you want. But as a matter of good programming style, you should lay out your program on the page in a way that will make its structure as clear as possible. In general, this means putting one statement per line and using indentation to indicate statements that are contained inside control structures. This is the format that I will generally use in my examples.

Here are two examples of blocks:

```
{
   System.out.print("The answer is ");
   System.out.println(ans);
}


{  // This block exchanges the values of x and y
   int temp;      // A temporary variable for use in this block.
   temp = x;      // Save a copy of the value of x in temp.
   x = y;         // Copy the value of y into x.
   y = temp;      // Copy the value of temp into y.
}
```

In the second example, a variable, `temp`, is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block. When the computer executes the variable declaration statement, it allocates memory to hold the value of the variable. When the block ends, that memory is discarded (that is, made available for reuse). The variable is said to be local to the block. There is a general concept called the "scope" of an identifier. The scope of an identifier is the part of the program in which that identifier is valid. The scope of a variable defined inside a block is limited to that block, and more specifically to the part of the block that comes after the declaration of the variable.

3.1.2 The Basic While Loop

The block statement by itself really doesn't affect the flow of control in a program. The five remaining control structures do. They can be divided into two classes: loop statements and branching statements. You really just need one control structure from each category in order to have a completely general-purpose programming language. More than that is just convenience. In this section, I'll introduce the `while` loop and the `if` statement. I'll give the full details of these statements and of the other three control structures in later sections.

A while loop is used to repeat a given statement over and over. Of course, it's not likely that you would want to keep repeating it forever. That would be an infinite loop, which is generally a bad thing. (There is an old story about computer pioneer Grace Murray Hopper, who read instructions on a bottle of shampoo telling her to "lather, rinse, repeat." As the story goes, she claims that she tried to follow the directions, but she ran out of shampoo. (In case you don't get it, this is a joke about the way that computers mindlessly follow instructions.))

To be more specific, a `while` loop will repeat a statement over and over, but only so long as a specified condition remains true. A `while` loop has the form:

```
while (boolean-expression)
     statement
```

Since the statement can be, and usually is, a block, most `while` loops have the form:

```
while (boolean-expression) {
    statements
}
```

Some programmers think that the braces should always be included as a matter of style, even when there is only one statement between them, but I don't always follow that advice myself.

The semantics of the `while` statement go like this: When the computer comes to a `while` statement, it evaluates the **boolean-expression**, which yields either `true` or `false` as its value. If the value is `false`, the computer skips over the rest of the `while` loop and proceeds to the next command in the program. If the value of the expression is `true`, the computer executes the **statement** or block of **statements** inside the loop. Then it returns to the beginning of the `while` loop and repeats the process. That is, it re-evaluates the **boolean-expression**, ends the loop if the value is `false`, and continues it if the value is `true`. This will continue over and over until the value of the expression is `false` when the computer evaluates it; if that never happens, then there will be an infinite loop.

Here is an example of a `while` loop that simply prints out the numbers 1, 2, 3, 4, 5:

```
int number;    // The number to be printed.
number = 1;    // Start with 1.
while ( number < 6 ) {  // Keep going as long as number is < 6.
    System.out.println(number);
    number = number + 1;  // Go on to the next number.
}
System.out.println("Done!");
```

The variable `number` is initialized with the value 1. So when the computer evaluates the expression "`number < 6`" for the first time, it is asking whether 1 is less than 6, which is `true`. The computer therefore proceeds to execute the two statements inside the loop. The first statement prints out "1". The second statement adds 1 to `number` and stores the result back into the variable `number`; the value of `number` has been changed to 2. The computer has reached the end of the loop, so it returns to the beginning and asks again whether `number` is less than 6. Once again this is true, so the computer executes the loop again, this time printing out 2 as the value of `number` and then changing the value of `number` to 3. It continues in this way until eventually `number` becomes equal to 6. At that point, the expression "`number < 6`" evaluates to `false`. So, the computer jumps past the end of the loop to the next statement and prints out the message "Done!". Note that when the loop ends, the value of `number` is 6, but the last value that was printed was 5.

By the way, you should remember that you'll never see a `while` loop standing by itself in a real program. It will always be inside a subroutine which is itself defined inside some class. As an example of a `while` loop used inside a complete program, here is a little program that computes the interest on an investment over several years. This is an improvement over examples from the previous chapter that just reported the results for one year:

```
/**
 *  This class implements a simple program that will compute the
amount of
 *  interest that is earned on an investment over a period of 5
years.  The
 *  initial amount of the investment and the interest rate are
input by the
 *  user.  The value of the investment at the end of each year is
output.
 */

public class Interest3 {


   public static void main(String[] args) {

      double principal;  // The value of the investment.
      double rate;       // The annual interest rate.

      /* Get the initial investment and interest rate from the
user. */

      System.out.print("Enter the initial investment: ");
      principal = TextIO.getlnDouble();

      System.out.println();
      System.out.println("Enter the annual interest rate.");
      System.out.print("Enter a decimal, not a percentage: ");
      rate = TextIO.getlnDouble();
      System.out.println();

      /* Simulate the investment for 5 years. */

      int years;  // Counts the number of years that have passed.

      years = 0;
      while (years < 5) {
         double interest;  // Interest for this year.
         interest = principal * rate;
         principal = principal + interest;     // Add it to
principal.
         years = years + 1;    // Count the current year.
         System.out.print("The value of the investment after ");
         System.out.print(years);
         System.out.print(" years is $");
         System.out.printf("%1.2f", principal);
         System.out.println();
      } // end of while loop

   } // end of main()

} // end of class Interest3
```

You should study this program, and make sure that you understand what the computer does step-by-step as it executes the `while` loop.

3.1.3 The Basic If Statement

An if statement tells the computer to take one of two alternative courses of action, depending on whether the value of a given boolean-valued expression is true or false. It is an example of a "branching" or "decision" statement. An `if` statement has the form:

```
if ( boolean-expression )
    statement1
else
    statement2
```

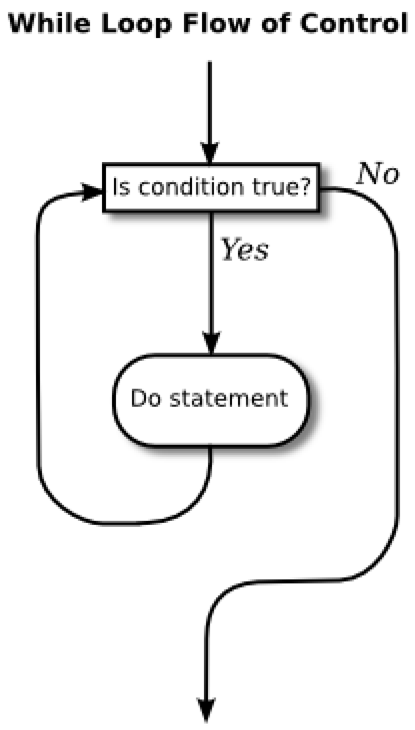
When the computer executes an `if` statement, it evaluates the boolean expression. If the value is `true`, the computer executes the first statement and skips the statement that follows the "`else`". If the value of the expression is `false`, then the computer skips the first statement and executes the second one. Note that in any case, one and only one of the two statements inside the `if` statement is executed. The two statements represent alternative courses of action; the computer decides between these courses of action based on the value of the boolean expression.

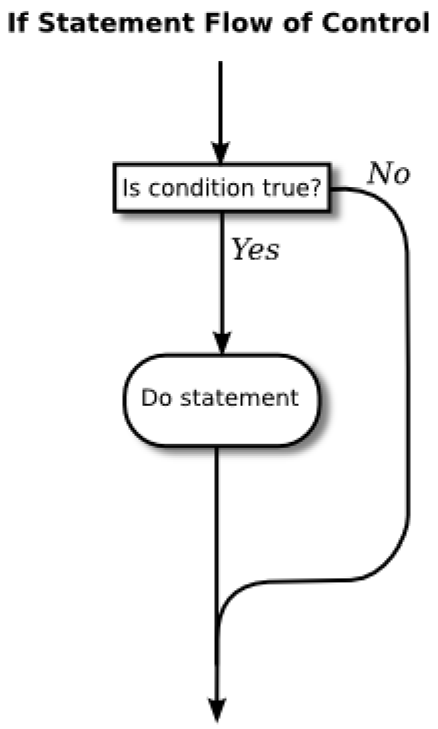
In many cases, you want the computer to choose between doing something and not doing it. You can do this with an `if` statement that omits the `else` part:

```
if ( boolean-expression )
    statement
```

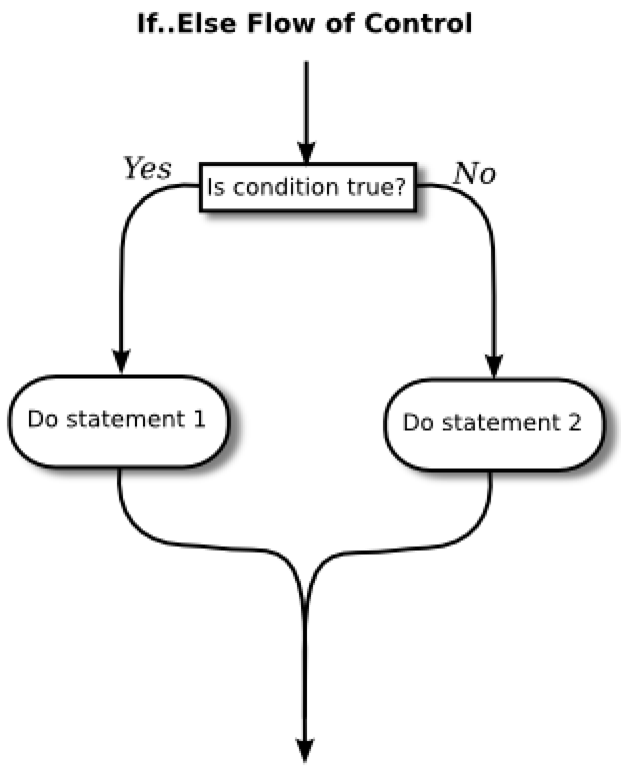
To execute this statement, the computer evaluates the expression. If the value is `true`, the computer executes the **statement** that is contained inside the `if` statement; if the value is `false`, the computer skips over that **statement**. In either case, the computer then continues with whatever follows the `if` statement in the program.

Sometimes, novice programmers confuse `while` statements with simple `if` statements (with no `else` part), although their meanings are quite different. The **statement** in an `if` is executed at most once, while the **statement** in a `while` can be executed any number of times. It can be helpful to look at diagrams of the the flow of control for `while` and simple `if` statements:





In these diagrams, the arrows represent the flow of time as the statement is executed. Control enters the diagram at the top and leaves at the bottom. Similarly, a flow control diagram for an `if..else` statement makes it clear that exactly one of the two nested statements is executed:



Of course, either or both of the **statements** in an `if` statement can be a block, and again many programmers prefer to add the braces even when they contain just a single statement. So an `if` statement often looks like:

```
if ( boolean-expression ) {
    statements
}
else {
    statements
}
```

or:

```
if ( boolean-expression ) {
    statements
}
```

As an example, here is an `if` statement that exchanges the value of two variables, `x` and `y`, but only if `x` is greater than `y` to begin with. After this `if` statement has been executed, we can be sure that the value of `x` is definitely less than or equal to the value of `y`:

```
if ( x > y ) {
    int temp;      // A temporary variable for use in this block.
    temp = x;      // Save a copy of the value of x in temp.
    x = y;         // Copy the value of y into x.
```

```
        y = temp;      // Copy the value of temp into y.
    }
```

Finally, here is an example of an `if` statement that includes an `else` part. See if you can figure out what it does, and why it would be used:

```
if ( years > 1 ) {  // handle case for 2 or more years
    System.out.print("The value of the investment after ");
    System.out.print(years);
    System.out.print(" years is $");
}
else {  // handle case for 1 year
    System.out.print("The value of the investment after 1 year is
$");
}  // end of if statement
System.out.printf("%1.2f", principal);  // this is done in any case
```

I'll have more to say about control structures later in this chapter. But you already know the essentials. If you never learned anything more about control structures, you would already know enough to perform any possible computing task. Simple looping and branching are all you really need!

3.1.4 Definite Assignment

I will finish this introduction to control structures with a somewhat technical issue that you might not fully understand the first time you encounter it. Consider the following two code segments, which seem to be entirely equivalent:

```
int y;                          int y;
if (x < 0) {                    if (x < 0) {
    y = 1;                           y = 1;
}                               }
else {                          if (x >= 0) {
    y = 2;                           y = 2;
}                               }
System.out.println(y);          System.out.println(y);
```

In the version on the left, `y` is assigned the value 1 if `x < 0` and is assigned the value 2 otherwise, that is, if `x >= 0`. Exactly the same is true of the version on the right. However, there is a subtle difference. In fact, the Java compiler will report an error for the `System.out.println` statement in the code on the right, while the code on the left is perfectly fine!

The problem is that in the code on the right, the computer can't tell that the variable `y` has definitely been assigned a value. When an `if` statement has no `else` part, the statement inside the `if` might or might not be executed, depending on the value of the condition. The compiler can't tell whether it will be executed or not, since the condition will only be evaluated when the program is running. For the code on the right above, as far as the compiler is concerned, it is possible that **neither** statement, `y = 1` or `y = 2`, will be evaluated, so it is possible that the output statement is trying to print an undefined value. The compiler considers this to be an error. The value of a variable can only be used if the compiler can **verify** that the variable will have been assigned a value at that point when the program is running. This is called definite assignment. (It doesn't matter that **you** can tell that `y` will always be assigned a value in this example. The question is whether the compiler can tell.)

Note that in the code on the left above, `y` is definitely assigned a value, since in an `if..else` statement, one of the two alternatives will be executed no matter what the value of the condition in the `if`. It is important that you understand that there is a difference between an `if..else` statement and a pair of plain `if` statements. Here is another pair of code segments that might seem to do the same thing, but don't. What's the value of `x` after each code segment is executed?

```
int x;
x = -1;
if (x < 0)
    x = 1;
else
    x = 2;
```

```
int x;
x = -1;
if (x < 0)
    x = 1;
if (x >= 0)
    x = 2;
```

After the code on the left is executed, `x` is 1; after the code on the right, `x` is 2.

PROGRAMMING IS DIFFICULT (like many activities that are useful and worthwhile -- and like most of those activities, it can also be rewarding and a lot of fun). When you write a program, you have to tell the computer every small detail of what to do. And you have to get everything exactly right, since the computer will blindly follow your program exactly as written. How, then, do people write any but the most simple programs? It's not a big mystery, actually. It's a matter of learning to think in the right way.

A program is an expression of an idea. A programmer starts with a general idea of a task for the computer to perform. Presumably, the programmer has some idea of how to perform the task by hand, at least in general outline. The problem is to flesh out that outline into a complete, unambiguous, step-by-step procedure for carrying out the task. Such a procedure is called an "algorithm." (Technically, an algorithm is an unambiguous, step-by-step procedure that always terminates after a finite number of steps. We don't want to count procedures that might go on forever.) An algorithm is not the same as a program. A program is written in some particular programming language. An algorithm is more like the **idea** behind the program, but it's the idea of the **steps** the program will take to perform its task, not just the idea of the **task** itself. When describing an algorithm, the steps don't necessarily have to be specified in complete detail, as long as the steps are unambiguous and it's clear that carrying out the steps will accomplish the assigned task. An algorithm can be expressed in any language, including English. Of course, an algorithm can only be expressed as an actual program if all the details have been filled in.

So, where do algorithms come from? Usually, they have to be developed, often with a lot of thought and hard work. Skill at algorithm development is something that comes with practice, but there are techniques and guidelines that can help. I'll talk here about some techniques and guidelines that are relevant to "programming in the small," and I will return to the subject several times in later chapters.

3.2.1 Pseudocode and Stepwise Refinement

When programming in the small, you have a few basics to work with: variables, assignment statements, and input/output routines. You might also have some subroutines, objects, or other building blocks that have already been written by you or someone else. (Input/output routines fall into this class.) You can build sequences of these basic instructions, and you can also combine them into more complex control structures such as `while` loops and `if` statements.

Suppose you have a task in mind that you want the computer to perform. One way to proceed is to write a description of the task, and take that description as an outline of the algorithm you want to develop. Then you can refine and elaborate that description, gradually adding steps and detail, until you have a complete algorithm that can be translated directly into programming language. This method is called stepwise refinement, and it is a type of top-down design. As you proceed through the stages of stepwise refinement, you can write out descriptions of your algorithm in pseudocode -- informal instructions that imitate the structure of programming languages without the complete detail and perfect syntax of actual program code.

As an example, let's see how one might develop the program from the previous section, which computes the value of an investment over five years. The task that you want the program to perform is: "Compute and display the value of an investment for each of the next five years, where the initial investment and interest rate are to be specified by the user." You might then write -- or more likely just think -- that this can be expanded as:

```
Get the user's input
Compute the value of the investment after 1 year
Display the value
Compute the value after 2 years
Display the value
Compute the value after 3 years
Display the value
Compute the value after 4 years
Display the value
Compute the value after 5 years
Display the value
```

This is correct, but rather repetitive. And seeing that repetition, you might notice an opportunity to use a loop. A loop would take less typing. More important, it would be more **general**: Essentially the same loop will work no matter how many years you want to process. So, you might rewrite the above sequence of steps as:

```
Get the user's input
while there are more years to process:
    Compute the value after the next year
    Display the value
```

Following this algorithm would certainly solve the problem, but for a computer we'll have to be more explicit about how to "Get the user's input," how to "Compute the value after the next year," and what it means to say "there are more years to process." We can expand the step, "Get the user's input" into

```
Ask the user for the initial investment
Read the user's response
Ask the user for the interest rate
Read the user's response
```

To fill in the details of the step "Compute the value after the next year," you have to know how to do the computation yourself. (Maybe you need to ask your boss or professor for clarification?) Let's say you know that the value is computed by adding some interest to the previous value. Then we can refine the `while` loop to:

```
while there are more years to process:
    Compute the interest
    Add the interest to the value
    Display the value
```

As for testing whether there are more years to process, the only way that we can do that is by counting the years ourselves. This displays a very common pattern, and you should expect to use something similar in a lot of programs: We have to start with zero years, add one each time we process a year, and stop when we reach the desired number of years. This is sometimes called a counting loop. So the `while` loop becomes:

```
years = 0
while years < 5:
    years = years + 1
    Compute the interest
    Add the interest to the value
    Display the value
```

We still have to know how to compute the interest. Let's say that the interest is to be computed by multiplying the interest rate by the current value of the investment. Putting this together with the part of the algorithm that gets the user's inputs, we have the complete algorithm:

```
Ask the user for the initial investment
Read the user's response
Ask the user for the interest rate
Read the user's response
years = 0
while years < 5:
    years = years + 1
    Compute interest = value * interest rate
    Add the interest to the value
    Display the value
```

Finally, we are at the point where we can translate pretty directly into proper programming-language syntax. We still have to choose names for the variables, decide exactly what we want to say to the user, and so forth. Having done this, we could express our algorithm in Java as:

```
double principal, rate, interest;  // declare the variables
int years;
System.out.print("Type initial investment: ");
principal = TextIO.getlnDouble();
System.out.print("Type interest rate: ");
rate = TextIO.getlnDouble();
years = 0;
while (years < 5) {
   years = years + 1;
   interest = principal * rate;
   principal = principal + interest;
   System.out.println(principal);
}
```

This still needs to be wrapped inside a complete program, it still needs to be commented, and it really needs to print out more information in a nicer format for the user. But it's essentially the same program as the one in the previous section. (Note that the pseudocode algorithm used indentation to show which statements are inside the loop. In Java, indentation is completely ignored by the computer, so you need a pair of braces to tell the computer which statements are in the loop. If you leave out the braces, the only statement inside the loop would be "`years = years + 1;`". The other statements would only be executed once, after the loop ends. The nasty thing is that the computer won't notice this error for you, like it would if you left out the parentheses around "`(years < 5)`". The parentheses are required by the syntax of the `while` statement. The braces are only required semantically. The computer can recognize syntax errors but not semantic errors.)

One thing you should have noticed here is that my original specification of the problem -- "Compute and display the value of an investment for each of the next five years" -- was far from being complete. Before you start writing a program, you should make sure you have a complete specification of exactly what the program is supposed to do. In particular, you need to know what information the program is going to input and output and what computation it is going to perform. Here is what a reasonably complete specification of the problem might look like in this example:

"Write a program that will compute and display the value of an investment for each of the next five years. Each year, interest is added to the value. The interest is computed by multiplying the current value by a fixed interest rate. Assume that the initial value and the rate of interest are to be input by the user when the program is run."

3.2.2 The 3N+1 Problem

Let's do another example, working this time with a program that you haven't already seen. The assignment here is an abstract mathematical problem that is one of my favorite programming exercises. This time, we'll start with a more complete specification of the task to be performed:

"Given a positive integer, N, define the '3N+1' sequence starting from N as follows: If N is an even number, then divide N by two; but if N is odd, then multiply N by 3 and add 1. Continue to generate numbers in this way until N becomes equal to 1. For example, starting from N = 3, which is odd, we multiply by 3 and add 1, giving N = 3*3+1 = 10. Then, since N is even, we divide by 2, giving N = 10/2 = 5. We continue in this way, stopping when we reach 1. The complete sequence is: 3, 10, 5, 16, 8, 4, 2, 1.

"Write a program that will read a positive integer from the user and will print out the 3N+1 sequence starting from that integer. The program should also count and print out the number of terms in the sequence."

A general outline of the algorithm for the program we want is:

```
Get a positive integer N from the user.
Compute, print, and count each number in the sequence.
Output the number of terms.
```

The bulk of the program is in the second step. We'll need a loop, since we want to keep computing numbers until we get 1. To put this in terms appropriate for a `while` loop, we need to know when to **continue** the loop rather than when to stop it: We want to continue as long as the number is **not** 1. So, we can expand our pseudocode algorithm to:

```
Get a positive integer N from the user;
while N is not 1:
    Compute N = next term;
    Output N;
    Count this term;
Output the number of terms;
```

In order to compute the next term, the computer must take different actions depending on whether N is even or odd. We need an `if` statement to decide between the two cases:

```
Get a positive integer N from the user;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
    Output N;
    Count this term;
Output the number of terms;
```

We are almost there. The one problem that remains is counting. Counting means that you start with zero, and every time you have something to count, you add one. We need a variable to do the counting. The variable must be set to zero once, **before** the loop starts, and it must be incremented within the loop. (Again, this is a common pattern that you should expect to see over and over.) With the counter added, we get:

```
Get a positive integer N from the user;
Let counter = 0;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
    Output N;
    Add 1 to counter;
Output the counter;
```

We still have to worry about the very first step. How can we get a **positive** integer from the user? If we just read in a number, it's possible that the user might type in a negative number or zero. If you follow what happens when the value of N is negative or zero, you'll see that the program will go on forever, since the value of N will never become equal to 1. This is bad. In this case, the problem is probably no big deal, but in general you should try to write programs that are foolproof. One way to fix this is to keep reading in numbers until the user types in a positive number:

```
Ask user to input a positive number;
Let N be the user's response;
while N is not positive:
   Print an error message;
   Read another value for N;
Let counter = 0;
while N is not 1:
    if N is even:
       Compute N = N/2;
    else
       Compute N = 3 * N + 1;
    Output N;
    Add 1 to counter;
Output the counter;
```

The first `while` loop will end only when N is a positive number, as required. (A common beginning programmer's error is to use an `if` statement instead of a `while` statement here: "If N is not positive, ask the user to input another value." The problem arises if the second number input by the user is also non-positive. The `if` statement is only executed once, so the second input number is never tested, and the program proceeds into an infinite loop. With the `while` loop, after the second number is input, the computer jumps back to the beginning of the loop and tests whether the second number is positive. If not, it asks the user for a third number, and it will continue asking for numbers until the user enters an acceptable input. After the while loop ends, we can be absolutely sure that `N` is a positive number.)

Here is a Java program implementing this algorithm. It uses the operators <= to mean "is less than or equal to" and != to mean "is not equal to." To test whether N is even, it uses "`N % 2 == 0`". All the operators used here were discussed in Section 2.5.

```
/**
 * This program prints out a 3N+1 sequence starting from a positive
 * integer specified by the user.  It also counts the number of
 * terms in the sequence, and prints out that number.
 */
 public class ThreeN1 {

      public static void main(String[] args) {

         int N;       // for computing terms in the sequence
         int counter; // for counting the terms

         System.out.print("Starting point for sequence: ");
         N = TextIO.getlnInt();
         while (N <= 0) {
            System.out.print(
                   "The starting point must be positive. Please try
again: " );
            N = TextIO.getlnInt();
         }
         // At this point, we know that N > 0

         counter = 0;
         while (N != 1) {
             if (N % 2 == 0)
                N = N / 2;
             else
                N = 3 * N + 1;
             System.out.println(N);
             counter = counter + 1;
         }

         System.out.println();
         System.out.print("There were ");
         System.out.print(counter);
         System.out.println(" terms in the sequence.");

     }  // end of main()

 }  // end of class ThreeN1
```

Two final notes on this program: First, you might have noticed that the first term of the sequence -- the value of N input by the user -- is not printed or counted by this program. Is this an error? It's hard to say. Was the specification of the program careful enough to decide? This is the type of thing that might send you back to the boss/professor for clarification. The problem (if it is one!) can be fixed easily enough. Just replace the line "counter = 0" before the while loop with the two lines:

```
System.out.println(N);   // print out initial term
counter = 1;       // and count it
```

Second, there is the question of why this problem might be interesting. Well, it's interesting to mathematicians and computer scientists because of a simple question about the problem that they haven't been able to answer: Will the process of computing the 3N+1 sequence finish after a finite number of steps for all possible starting values of N? Although individual sequences are easy to compute, no one has been able to answer the general question. To put this another way, no one knows whether the process of computing 3N+1 sequences can properly be called an algorithm, since an algorithm is required to terminate after a finite number of steps! (Note: This discussion really applies to integers, not to values of type int! That is, it assumes that the value of `N` can take on arbitrarily large integer values, which is not true for a variable of type int in a Java program. When the value of `N` in the program becomes too large to be represented as a 32-bit int, the values output by the program are no longer mathematically correct. So the Java program does not compute the correct 3N+1 sequence if `N` becomes too large. See Exercise 8.2.)

3.2.3 Coding, Testing, Debugging

It would be nice if, having developed an algorithm for your program, you could relax, press a button, and get a perfectly working program. Unfortunately, the process of turning an algorithm into Java source code doesn't always go smoothly. And when you do get to the stage of a working program, it's often only working in the sense that it does **something**. Unfortunately not what you want it to do.

After program design comes coding: translating the design into a program written in Java or some other language. Usually, no matter how careful you are, a few syntax errors will creep in from somewhere, and the Java compiler will reject your program with some kind of error message. Unfortunately, while a compiler will always detect syntax errors, it's not very good about telling you exactly what's wrong. Sometimes, it's not even good about telling you where the real error is. A spelling error or missing "{" on line 45 might cause the compiler to choke on line 105. You can avoid lots of errors by making sure that you really understand the syntax rules of the language and by following some basic programming guidelines. For example, I never type a "{" without typing the matching "}". Then I go back and fill in the statements between the braces. A missing or extra brace can be one of the hardest errors to find in a large program. Always, always indent your program nicely. If you change the program, change the indentation to match. It's worth the trouble. Use a consistent naming scheme, so you don't have to struggle to remember whether you called that variable `interestrate` or `interestRate`. In general, when the compiler gives multiple error messages, don't try to fix the second error message from the compiler until you've fixed the first one. Once the compiler hits an error in your program, it can get confused, and the rest of the error messages might just be guesses. Maybe the best advice is: Take the time to understand the error before you try to fix it. Programming is not an experimental science.

When your program compiles without error, you are still not done. You have to test the program to make sure it works correctly. Remember that the goal is not to get the right output for the two sample inputs that the professor gave in class. The goal is a program that will work correctly for all reasonable inputs. Ideally, when faced with an unreasonable input, it should respond by gently chiding the user rather than by crashing. Test your program on a wide variety of inputs. Try to find a set of inputs that will test the full range of functionality that you've coded into your program. As you begin writing larger programs, write them in stages and test each stage along the way. You might even have to write some extra code to do the testing -- for example to call a subroutine that you've just written. You don't want to be faced, if you can avoid it, with 500 newly written lines of code that have an error in there *somewhere.*

The point of testing is to find bugs -- semantic errors that show up as incorrect behavior rather than as compilation errors. And the sad fact is that you will probably find them. Again, you can minimize bugs by careful design and careful coding, but no one has found a way to avoid them altogether. Once you've detected a bug, it's time for debugging. You have to track down the cause of the bug in the program's source code and eliminate it. Debugging is a skill that, like other aspects of programming, requires practice to master. So don't be afraid of bugs. Learn from them. One essential debugging skill is the ability to read source code -- the ability to put aside preconceptions about what you *think* it does and to follow it the way the computer does -- mechanically, step-by-step -- to see what it really does. This is hard. I can still remember the time I spent hours looking for a bug only to find that a line of code that I had looked at ten times had a "1" where it should have had an "i", or the time when I wrote a subroutine named `WindowClosing` which would have done exactly what I wanted except that the computer was looking for `windowClosing` (with a lower case "w"). Sometimes it can help to have someone who doesn't share your preconceptions look at your code.

Often, it's a problem just to find the part of the program that contains the error. Most programming environments come with a debugger, which is a program that can help you find bugs. Typically, your program can be run under the control of the debugger. The debugger allows you to set "breakpoints" in your program. A breakpoint is a point in the program where the debugger will pause the program so you can look at the values of the program's variables. The idea is to track down exactly when things start to go wrong during the program's execution. The debugger will also let you execute your program one line at a time, so that you can watch what happens in detail once you know the general area in the program where the bug is lurking.

I will confess that I only occasionally use debuggers myself. A more traditional approach to debugging is to insert debugging statements into your program. These are output statements that print out information about the state of the program. Typically, a debugging statement would say something like

```
System.out.println("At start of while loop, N = " + N);
```

You need to be able to tell from the output where in your program the output is coming from, and you want to know the value of important variables. Sometimes, you will find that the computer isn't even getting to a part of the program that you think it should be executing. Remember that the goal is to find the first point in the program where the state is not what you expect it to be. That's where the bug is.

And finally, remember the golden rule of debugging: If you are absolutely sure that everything in your program is right, and if it still doesn't work, then one of the things that you are absolutely sure of is wrong.

The while and do..while Statements

STATEMENTS IN JAVA CAN be either simple statements or compound statements. Simple statements, such as assignment statements and subroutine call statements, are the basic building blocks of a program. Compound statements, such as `while` loops and `if` statements, are used to organize simple statements into complex structures, which are called control structures because they control the order in which the statements are executed. The next five sections explore the details of control structures that are available in Java, starting with the `while` statement and the `do..while` statement in this section. At the same time, we'll look at examples of programming with each control structure and apply the techniques for designing algorithms that were introduced in the previous section.

3.3.1 The while Statement

The `while` statement was already introduced in Section 3.1. A `while` loop has the form

```
while ( boolean-expression )
   statement
```

The **statement** can, of course, be a block statement consisting of several statements grouped together between a pair of braces. This statement is called the body of the loop. The body of the loop is repeated as long as the **boolean-expression** is true. This boolean expression is called the continuation condition, or more simply the test, of the loop. There are a few points that might need some clarification. What happens if the condition is false in the first place, before the body of the loop is executed even once? In that case, the body of the loop is never executed at all. The body of a while loop can be executed any number of times, including zero. What happens if the condition is true, but it becomes false somewhere in the **middle** of the loop body? Does the loop end as soon as this happens? It doesn't, because the computer continues executing the body of the loop until it gets to the end. Only then does it jump back to the beginning of the loop and test the condition, and only then can the loop end.

Let's look at a typical problem that can be solved using a `while` loop: finding the average of a set of positive integers entered by the user. The average is the sum of the integers, divided by the number of integers. The program will ask the user to enter one integer at a time. It will keep count of the number of integers entered, and it will keep a running total of all the numbers it has read so far. Here is a pseudocode algorithm for the program:

```
Let sum = 0     // The sum of the integers entered by the user.
Let count = 0   // The number of integers entered by the user.
while there are more integers to process:
    Read an integer
    Add it to the sum
    Count it
Divide sum by count to get the average
Print out the average
```

But how can we test whether there are more integers to process? A typical solution is to tell the user to type in zero after all the data have been entered. This will work because we are assuming that all the data are positive numbers, so zero is not a legal data value. The zero is not itself part of the data to be averaged. It's just there to mark the end of the real data. A data value used in this way is sometimes called a sentinel value. So now the test in the while loop becomes "while the input integer is not zero". But there is another problem! The first time the test is evaluated, before the body of the loop has ever been executed, no integer has yet been read. There is no "input integer" yet, so testing whether the input integer is zero doesn't make sense. So, we have to do something **before** the while loop to make sure that the test makes sense. Setting things up so that the test in a `while` loop makes sense the first time it is executed is called priming the loop. In this case, we can simply read the first integer before the beginning of the loop. Here is a revised algorithm:

```
Let sum = 0
Let count = 0
Read an integer
while the integer is not zero:
    Add the integer to the sum
    Count it
    Read an integer
Divide sum by count to get the average
Print out the average
```

Notice that I've rearranged the body of the loop. Since an integer is read before the loop, the loop has to begin by processing that integer. At the end of the loop, the computer reads a new integer. The computer then jumps back to the beginning of the loop and tests the integer that it has just read. Note that when the computer finally reads the sentinel value, the loop ends before the sentinel value is processed. It is not added to the sum, and it is not counted. This is the way it's supposed to work. The sentinel is not part of the data. The original algorithm, even if it could have been made to work without priming, was incorrect since it would have summed and counted all the integers, including the sentinel. (Since the sentinel is zero, the sum would still be correct, but the count would be off by one. Such so-called off-by-one errors are very common. Counting turns out to be harder than it looks!)

We can easily turn the algorithm into a complete program. Note that the program cannot use the statement "`average = sum/count;`" to compute the average. Since `sum` and `count` are both variables of type int, the value of `sum/count` is an integer. The average should be a real number. We've seen this problem before: we have to convert one of the int values to a double to force the computer to compute the quotient as a real number. This can be done by type-casting one of the variables to type double. The type cast "(double)sum" converts the value of `sum` to a real number, so in the program the average is computed as "`average =`

((double)sum) / count;". Another solution in this case would have been to declare sum to be a variable of type double in the first place.

One other issue is addressed by the program: If the user enters zero as the first input value, there are no data to process. We can test for this case by checking whether count is still equal to zero after the while loop. This might seem like a minor point, but a careful programmer should cover all the bases.

Here is the full source code for the program:

```
/**
 * This program reads a sequence of positive integers input
 * by the user, and it will print out the average of those
 * integers.  The user is prompted to enter one integer at a
 * time.  The user must enter a 0 to mark the end of the
 * data.  (The zero is not counted as part of the data to
 * be averaged.)  The program does not check whether the
 * user's input is positive, so it will actually add up
 * both positive and negative input values.
 */

public class ComputeAverage {

   public static void main(String[] args) {

      int inputNumber;   // One of the integers input by the user.
      int sum;           // The sum of the positive integers.
      int count;         // The number of positive integers.
      double average;    // The average of the positive integers.

      /* Initialize the summation and counting variables. */

      sum = 0;
      count = 0;

      /* Read and process the user's input. */

      System.out.print("Enter your first positive integer: ");
      inputNumber = TextIO.getlnInt();

      while (inputNumber != 0) {
         sum += inputNumber;   // Add inputNumber to running sum.
         count++;              // Count the input by adding 1 to
count.
         System.out.print("Enter your next positive integer, or 0
to end: ");
         inputNumber = TextIO.getlnInt();
      }

      /* Display the result. */

      if (count == 0) {
         System.out.println("You didn't enter any data!");
      }
      else {
         average = ((double)sum) / count;
         System.out.println();
         System.out.println("You entered " + count + " positive
integers.");
         System.out.printf("Their average is %1.3f.\n", average);
      }

   } // end main()

} // end class ComputeAverage
```

3.3.2 The do..while Statement

Sometimes it is more convenient to test the continuation condition at the end of a loop, instead of at the beginning, as is done in the `while` loop. The `do..while` statement is very similar to the `while` statement, except that the word "while," along with the condition that it tests, has been moved to the end. The word "do" is added to mark the beginning of the loop. A `do..while` statement has the form

```
do
    statement
while ( boolean-expression );
```

or, since, as usual, the **statement** can be a block,

```
do {
    statements
} while ( boolean-expression );
```

Note the semicolon, ';', at the very end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it is a syntax error. (More generally, **every** statement in Java ends either with a semicolon or a right brace, '}'.)

To execute a `do` loop, the computer first executes the body of the loop -- that is, the statement or statements inside the loop -- and then it evaluates the boolean expression. If the value of the expression is `true`, the computer returns to the beginning of the `do` loop and repeats the process; if the value is `false`, it ends the loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a `do` loop is always executed at least once.

For example, consider the following pseudocode for a game-playing program. The `do` loop makes sense here instead of a `while` loop because with the `do` loop, you know there will be at least one game. Also, the test that is used at the end of the loop wouldn't even make sense at the beginning:

```
do {
      Play a Game
      Ask user if he wants to play another game
      Read the user's response
   } while ( the user's response is yes );
```

Let's convert this into proper Java code. Since I don't want to talk about game playing at the moment, let's say that we have a class named `Checkers`, and that the `Checkers` class contains a static member subroutine named `playGame()` that plays one game of checkers against the user. Then, the pseudocode "Play a game" can be expressed as the subroutine call statement "`Checkers.playGame();`". We need a variable to store the user's response. The *TextIO* class makes it convenient to use a boolean variable to store the answer to a yes/no question. The input function `TextIO.getlnBoolean()` allows the user to enter the value as "yes" or "no" (among other acceptable responses). "Yes" is considered to be `true`, and "no" is considered to be `false`. So, the algorithm can be coded as

```
   boolean wantsToContinue;  // True if user wants to play again.
   do {
      Checkers.playGame();
      System.out.print("Do you want to play again? ");
      wantsToContinue = TextIO.getlnBoolean();
   } while (wantsToContinue == true);
```

When the value of the boolean variable is set to `false`, it is a signal that the loop should end. When a boolean variable is used in this way -- as a signal that is set in one part of the program and tested in another part -- it is sometimes called a flag or flag variable (in the sense of a signal flag).

By the way, a more-than-usually-pedantic programmer would sneer at the test "`while (wantsToContinue == true)`". This test is exactly equivalent to "`while (wantsToContinue)`". Testing whether "`wantsToContinue == true`" is true amounts to the same thing as testing whether "`wantsToContinue`" is true. A little less offensive is an expression of the form "`flag == false`", where `flag` is a boolean variable. The value of "`flag == false`" is exactly the same as the value of "`!flag`", where `!` is the boolean negation operator. So you can write "`while (!flag)`" instead of "`while (flag == false)`", and you can write "`if (!flag)`" instead of "`if (flag == false)`".

Although a `do..while` statement is sometimes more convenient than a `while` statement, having two kinds of loops does not make the language more powerful. Any problem that can be solved using `do..while` loops can also be solved using only `while` statements, and vice versa. In fact, if **doSomething** represents any block of program code, then

```
   do {
       doSomething
   } while ( boolean-expression );
```

has exactly the same effect as

```
doSomething
while ( boolean-expression ) {
    doSomething
}
```

Similarly,

```
while ( boolean-expression ) {
    doSomething
}
```

can be replaced by

```
if ( boolean-expression ) {
   do {
       doSomething
   } while ( boolean-expression );
}
```

without changing the meaning of the program in any way.

3.3.3 break and continue

The syntax of the `while` and `do..while` loops allows you to test the continuation condition at either the beginning of a loop or at the end. Sometimes, it is more natural to have the test in the middle of the loop, or to have several tests at different places in the same loop. Java provides a general method for breaking out of the middle of any loop. It's called the `break` statement, which takes the form

```
break;
```

When the computer executes a `break` statement in a loop, it will immediately jump out of the loop. It then continues on to whatever follows the loop in the program. Consider for example:

```
while (true) {  // looks like it will run forever!
   System.out.print("Enter a positive number: ");
   N = TextIO.getlnInt();
   if (N > 0)   // the input value is OK, so jump out of loop
      break;
   System.out.println("Your answer must be > 0.");
}
// continue here after break
```

If the number entered by the user is greater than zero, the `break` statement will be executed and the computer will jump out of the loop. Otherwise, the computer will print out "Your answer must be > 0." and will jump back to the start of the loop to read another input value.

The first line of this loop, "`while (true)`" might look a bit strange, but it's perfectly legitimate. The condition in a `while` loop can be any boolean-valued expression. The computer evaluates this expression and checks whether the value is `true` or `false`. The boolean literal "`true`" is just a boolean expression that always evaluates to true. So "`while (true)`" can be used to write an infinite loop, or one that will be terminated by a `break` statement.

A `break` statement terminates the loop that immediately encloses the `break` statement. It is possible to have nested loops, where one loop statement is contained inside another. If you use a `break` statement inside a nested loop, it will only break out of that loop, not out of the loop that contains the nested loop. There is something called a labeled break statement that allows you to specify which loop you want to break. This is not very common, so I will go over it quickly. Labels work like this: You can put a label in front of any loop. A label consists of a simple identifier followed by a colon. For example, a `while` with a label might look like "`mainloop: while...`". Inside this loop you can use the labeled break statement "`break mainloop;`" to break out of the labeled loop. For example, here is a code segment that checks whether two strings, `s1` and `s2`, have a character in common. If a common character is found, the value of the flag variable `nothingInCommon` is set to `false`, and a labeled break is used to end the processing at that point:

```
boolean nothingInCommon;
nothingInCommon = true;  // Assume s1 and s2 have no chars in
common.
int i,j;  // Variables for iterating through the chars in s1 and
s2.

i = 0;
bigloop: while (i < s1.length()) {
   j = 0;
   while (j < s2.length()) {
      if (s1.charAt(i) == s2.charAt(j)) { // s1 and s2 have a
common char.
          nothingInCommon = false;
          break bigloop;  // break out of BOTH loops
      }
      j++;  // Go on to the next char in s2.
   }
   i++;  //Go on to the next char in s1.
}
```

The `continue` statement is related to `break`, but less commonly used. A `continue` statement tells the computer to skip the rest of the current iteration of the loop. However, instead of jumping out of the loop altogether, it jumps back to the beginning of the loop and continues with the next iteration (including evaluating the loop's continuation condition to see whether any further iterations are required). As with `break`, when a `continue` is in a nested loop, it will continue the loop that directly contains it; a "labeled continue" can be used to continue the containing loop instead.

break and continue can be used in while loops and do..while loops. They can also be used in for loops, which are covered in the next section. In Section 3.6, we'll see that break can also be used to break out of a switch statement. A break can occur inside an if statement, but only if the if statement is nested inside a loop or inside a switch statement. In that case, it does **not** mean to break out of the if. Instead, it breaks out of the loop or switch statement that contains the if statement. The same consideration applies to continue statements inside ifs.

The for Statement

WE TURN IN THIS SECTION to another type of loop, the for statement. Any for loop is equivalent to some while loop, so the language doesn't get any additional power by having for statements. But for a certain type of problem, a for loop can be easier to construct and easier to read than the corresponding while loop. It's quite possible that in real programs, for loops actually outnumber while loops.

3.4.1 For Loops

The for statement makes a common type of while loop easier to write. Many while loops have the general form:

```
initialization
while ( continuation-condition ) {
    statements
    update
}
```

For example, consider this example, copied from an example in Section 3.2:

```
years = 0;  // initialize the variable years
while ( years < 5 ) {   // condition for continuing loop

    interest = principal * rate;    //
    principal += interest;          // do three statements
    System.out.println(principal);  //

    years++;   // update the value of the variable, years
}
```

This loop can be written as the following equivalent for statement:

```
for ( years = 0;  years < 5;  years++ ) {
   interest = principal * rate;
   principal += interest;
```

```
    System.out.println(principal);
}
```

The initialization, continuation condition, and updating have all been combined in the first line of the `for` loop. This keeps everything involved in the "control" of the loop in one place, which helps make the loop easier to read and understand. The `for` loop is executed in exactly the same way as the original code: The initialization part is executed once, before the loop begins. The continuation condition is executed before each execution of the loop, and the loop ends when this condition is `false`. The update part is executed at the end of each execution of the loop, just before jumping back to check the condition.

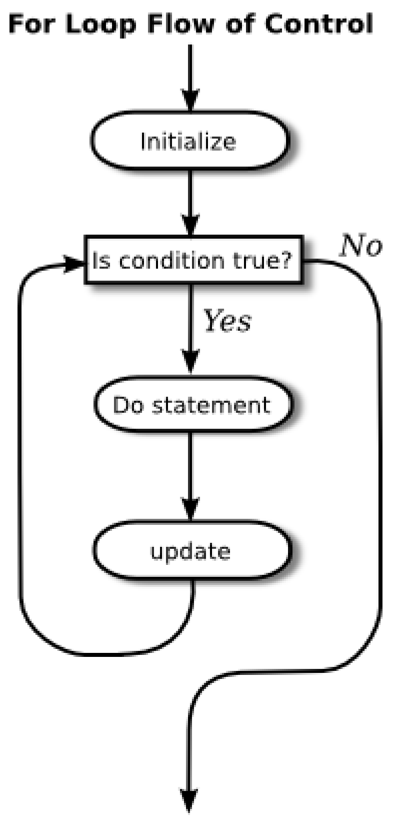
The formal syntax of the `for` statement is as follows:

```
for ( initialization; continuation-condition; update )
     statement
```

or, using a block statement:

```
for ( initialization; continuation-condition; update ) {
     statements
}
```

The **continuation-condition** must be a boolean-valued expression. The **initialization** is usually a declaration or an assignment statement, but it can be any expression that would be allowed as a statement in a program. The **update** can be any simple statement, but is usually an increment, a decrement, or an assignment statement. Any of the three parts can be empty. If the continuation condition is empty, it is treated as if it were "`true`," so the loop will be repeated forever or until it ends for some other reason, such as a `break` statement. (Some people like to begin an infinite loop with "`for (;;)`" instead of "`while (true)`".) Here's a flow control diagram for a `for` statement:



Usually, the initialization part of a `for` statement assigns a value to some variable, and the update changes the value of that variable with an assignment statement or with an increment or decrement operation. The value of the variable is tested in the continuation condition, and the loop ends when this condition evaluates to `false`. A variable used in this way is called a loop control variable. In the example given above, the loop control variable was `years`.

Certainly, the most common type of `for` loop is the counting loop, where a loop control variable takes on all integer values between some minimum and some maximum value. A counting loop has the form

```
for ( variable = min;  variable <= max; variable++ ) {
     statements
}
```

where **min** and **max** are integer-valued expressions (usually constants). The **variable** takes on the values **min**, **min**+1, **min**+2, ..., **max**. The value of the loop control variable is often used in the body of the loop. The `for` loop at the beginning of this section is a counting loop in which the loop control variable, `years`, takes on the values 1, 2, 3, 4, 5. Here is an even simpler example, in which the numbers 1, 2, ..., 10 are displayed on standard output:

```
for ( N = 1 ;  N <= 10 ;  N++ )
   System.out.println( N );
```

For various reasons, Java programmers like to start counting at 0 instead of 1, and they tend to use a "<" in the condition, rather than a "<=". The following variation of the above loop prints out the ten numbers 0, 1, 2, ..., 9:

```
for ( N = 0 ;  N < 10 ;  N++ )
   System.out.println( N );
```

Using < instead of <= in the test, or vice versa, is a common source of off-by-one errors in programs. You should always stop and think, Do I want the final value to be processed or not?

It's easy to count down from 10 to 1 instead of counting up. Just start with 10, decrement the loop control variable instead of incrementing it, and continue as long as the variable is greater than or equal to one.

```
for ( N = 10 ;  N >= 1 ;  N-- )
   System.out.println( N );
```

Now, in fact, the official syntax of a `for` statement actually allows both the initialization part and the update part to consist of several expressions, separated by commas. So we can even count up from 1 to 10 and count down from 10 to 1 at the same time!

```
for ( i=1, j=10;  i <= 10;  i++, j-- ) {
   System.out.printf("%5d", i); // Output i in a 5-character wide
column.
   System.out.printf("%5d", j); // Output j in a 5-character column
   System.out.println();       //     and end the line.
}
```

As a final introductory example, let's say that we want to use a `for` loop that prints out just the even numbers between 2 and 20, that is: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. There are several ways to do this. Just to show how even a very simple problem can be solved in many ways, here are four different solutions (three of which would get full credit):

```
 (1)   // There are 10 numbers to print.
       // Use a for loop to count 1, 2,
       // ..., 10.  The numbers we want
       // to print are 2*1, 2*2, ... 2*10.

       for (N = 1; N <= 10; N++) {
          System.out.println( 2*N );
       }


 (2)   // Use a for loop that counts
       // 2, 4, ..., 20 directly by
       // adding 2 to N each time through
       // the loop.

       for (N = 2; N <= 20; N = N + 2) {
          System.out.println( N );
       }
```

```
(3)    // Count off all the numbers
       // 2, 3, 4, ..., 19, 20, but
       // only print out the numbers
       // that are even.

       for (N = 2; N <= 20; N++) {
          if ( N % 2 == 0 ) // is N even?
             System.out.println( N );
       }


(4)    // Irritate the professor with
       // a solution that follows the
       // letter of this silly assignment
       // while making fun of it.

       for (N = 1; N <= 1; N++) {
          System.out.println("2 4 6 8 10 12 14 16 18 20");
       }
```

Perhaps it is worth stressing one more time that a `for` statement, like any statement except for a variable declaration, never occurs on its own in a real program. A statement must be inside the `main` routine of a program or inside some other subroutine. And that subroutine must be defined inside a class. I should also remind you that every variable must be declared before it can be used, and that includes the loop control variable in a `for` statement. In all the examples that you have seen so far in this section, the loop control variables should be declared to be of type int. It is not required that a loop control variable be an integer. Here, for example, is a `for` loop in which the variable, `ch`, is of type char, using the fact that the ++ operator can be applied to characters as well as to numbers:

```
// Print out the alphabet on one line of output.
char ch;  // The loop control variable;
          //       one of the letters to be printed.
for ( ch = 'A';  ch <= 'Z';  ch++ )
    System.out.print(ch);
System.out.println();
```

3.4.2 Example: Counting Divisors

Let's look at a less trivial problem that can be solved with a `for` loop. If `N` and `D` are positive integers, we say that `D` is a divisor of `N` if the remainder when `D` is divided into `N` is zero. (Equivalently, we could say that `N` is an even multiple of `D`.) In terms of Java programming, `D` is a divisor of `N` if `N % D` is zero.

Let's write a program that inputs a positive integer, `N`, from the user and computes how many different divisors `N` has. The numbers that could possibly be divisors of `N` are 1, 2, ..., `N`. To compute the number of divisors of N, we can just test each possible divisor of N and count the ones that actually do divide N evenly. In pseudocode, the algorithm takes the form

```
Get a positive integer, N, from the user
Let divisorCount = 0
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
Output the count
```

This algorithm displays a common programming pattern that is used when some, but not all, of a sequence of items are to be processed. The general pattern is

```
for each item in the sequence:
   if the item passes the test:
       process it
```

The for loop in our divisor-counting algorithm can be translated into Java code as

```
for (testDivisor = 1; testDivisor <= N; testDivisor++) {
   if ( N % testDivisor == 0 )
      divisorCount++;
}
```

On a modern computer, this loop can be executed very quickly. It is not impossible to run it even for the largest legal int value, 2147483647. (If you wanted to run it for even larger values, you could use variables of type long rather than int.) However, it does take a significant amount of time for very large numbers. So when I implemented this algorithm, I decided to output a dot every time the computer has tested one million possible divisors. In the improved version of the program, there are two types of counting going on. We have to count the number of divisors and we also have to count the number of possible divisors that have been tested. So the program needs two counters. When the second counter reaches 1000000, the program outputs a '.' and resets the counter to zero so that we can start counting the next group of one million. Reverting to pseudocode, the algorithm now looks like

```
Get a positive integer, N, from the user
Let divisorCount = 0  // Number of divisors found.
Let numberTested = 0  // Number of possible divisors tested
                      //       since the last period was output.
for each number, testDivisor, in the range from 1 to N:
    if testDivisor is a divisor of N:
        Count it by adding 1 to divisorCount
    Add 1 to numberTested
    if numberTested is 1000000:
        print out a '.'
        Reset numberTested to 0
Output the count
```

Finally, we can translate the algorithm into a complete Java program:

```
/**
```

```
 * This program reads a positive integer from the user.
 * It counts how many divisors that number has, and
 * then it prints the result.
 */

public class CountDivisors {

   public static void main(String[] args) {

      int N;  // A positive integer entered by the user.
              // Divisors of this number will be counted.

      int testDivisor;  // A number between 1 and N that is a
                        // possible divisor of N.

      int divisorCount;  // Number of divisors of N that have been
found.

      int numberTested;  // Used to count how many possible
divisors
                         // of N have been tested.  When the number
                         // reaches 1000000, a period is output and
                         // the value of numberTested is reset to
zero.

      /* Get a positive integer from the user. */

      while (true) {
         System.out.print("Enter a positive integer: ");
         N = TextIO.getlnInt();
         if (N > 0)
            break;
         System.out.println("That number is not positive.  Please
try again.");
      }

      /* Count the divisors, printing a "." after every 1000000
tests. */

      divisorCount = 0;
      numberTested = 0;

      for (testDivisor = 1; testDivisor <= N; testDivisor++) {
         if ( N % testDivisor == 0 )
            divisorCount++;
         numberTested++;
         if (numberTested == 1000000) {
            System.out.print('.');
            numberTested = 0;
         }
      }

      /* Display the result. */

      System.out.println();
      System.out.println("The number of divisors of " + N
                          + " is " + divisorCount);
   } // end main()

} // end class CountDivisors
```

3.4.3 Nested for Loops

Control structures in Java are statements that contain other, simpler statements. In particular, control structures can contain control structures. You've already seen several examples of `if` statements inside loops, and one example of a `while` loop inside another `while`, but any combination of one control structure inside another is possible. We say that one structure is nested inside another. You can even have multiple levels of nesting, such as a `while` loop inside an `if` statement inside another `while` loop. The syntax of Java does not set a limit on the number of levels of nesting. As a practical matter, though, it's difficult to understand a program that has more than a few levels of nesting.

Nested `for` loops arise naturally in many algorithms, and it is important to understand how they work. Let's look at a couple of examples. First, consider the problem of printing out a multiplication table like this one:

```
 1   2   3   4   5   6   7   8   9  10  11  12
 2   4   6   8  10  12  14  16  18  20  22  24
 3   6   9  12  15  18  21  24  27  30  33  36
 4   8  12  16  20  24  28  32  36  40  44  48
 5  10  15  20  25  30  35  40  45  50  55  60
 6  12  18  24  30  36  42  48  54  60  66  72
 7  14  21  28  35  42  49  56  63  70  77  84
 8  16  24  32  40  48  56  64  72  80  88  96
 9  18  27  36  45  54  63  72  81  90  99 108
10  20  30  40  50  60  70  80  90 100 110 120
11  22  33  44  55  66  77  88  99 110 121 132
12  24  36  48  60  72  84  96 108 120 132 144
```

The data in the table are arranged into 12 rows and 12 columns. The process of printing them out can be expressed in a pseudocode algorithm as

```
for each rowNumber = 1, 2, 3, ..., 12:
   Print the first twelve multiples of rowNumber on one line
   Output a carriage return
```

The first step in the `for` loop can itself be expressed as a `for` loop. We can expand "Print the first twelve multiples of `rowNumber` on one line" as:

```
for N = 1, 2, 3, ..., 12:
   Print N * rowNumber
```

so a refined algorithm for printing the table has one `for` loop nested inside another:

```
for each rowNumber = 1, 2, 3, ..., 12:
   for N = 1, 2, 3, ..., 12:
      Print N * rowNumber
   Output a carriage return
```

We want to print the output in neat columns, with each output number taking up four spaces. This can be done using formatted output with format specifier `%4d`. Assuming that `rowNumber` and `N` have been declared to be variables of type int, the algorithm can be expressed in Java as

```
for ( rowNumber = 1;  rowNumber <= 12;  rowNumber++ ) {
   for ( N = 1;  N <= 12;  N++ ) {
               // print in 4-character columns
      System.out.printf( "%4d", N * rowNumber );  // No carriage
return !
   }
   System.out.println();  // Add a carriage return at end of the
line.
}
```

This section has been weighed down with lots of examples of numerical processing. For our next example, let's do some text processing. Consider the problem of finding which of the 26 letters of the alphabet occur in a given string. For example, the letters that occur in "Hello World" are D, E, H, L, O, R, and W. More specifically, we will write a program that will list all the letters contained in a string and will also count the number of different letters. The string will be input by the user. Let's start with a pseudocode algorithm for the program.

```
Ask the user to input a string
Read the response into a variable, str
Let count = 0  (for counting the number of different letters)
for each letter of the alphabet:
   if the letter occurs in str:
      Print the letter
      Add 1 to count
Output the count
```

Since we want to process the entire line of text that is entered by the user, we'll use `TextIO.getln()` to read it. The line of the algorithm that reads "for each letter of the alphabet" can be expressed as "`for (letter='A'; letter<='Z'; letter++)`". But the `if` statement inside the `for` loop needs still more thought before we can write the program. How do we check whether the given letter, `letter`, occurs in `str`? One idea is to look at each character in the string in turn, and check whether that character is equal to `letter`. We can get the `i`-th character of `str` with the function call `str.charAt(i)`, where `i` ranges from 0 to `str.length() - 1`.

One more difficulty: A letter such as 'A' can occur in `str` in either upper or lower case, 'A' or 'a'. We have to check for both of these. But we can avoid this difficulty by converting `str` to upper case before processing it. Then, we only have to check for the upper case letter. We can now flesh out the algorithm fully:

```
Ask the user to input a string
```

```
Read the response into a variable, str
Convert str to upper case
Let count = 0
for letter = 'A', 'B', ..., 'Z':
    for i = 0, 1, ..., str.length()-1:
        if letter == str.charAt(i):
            Print letter
            Add 1 to count
            break  // jump out of the loop, to avoid counting
letter twice
Output the count
```

Note the use of `break` in the nested `for` loop. It is required to avoid printing or counting a given letter more than once (in the case where it occurs more than once in the string). The `break` statement breaks out of the inner `for` loop, but not the outer `for` loop. Upon executing the `break`, the computer continues the outer loop with the next value of `letter`. You should try to figure out exactly what `count` would be at the end of this program, if the `break` statement were omitted. Here is the complete program:

```
/**
 * This program reads a line of text entered by the user.
 * It prints a list of the letters that occur in the text,
 * and it reports how many different letters were found.
 */

public class ListLetters {

   public static void main(String[] args) {

      String str;  // Line of text entered by the user.
      int count;   // Number of different letters found in str.
      char letter; // A letter of the alphabet.

      System.out.println("Please type in a line of text.");
      str = TextIO.getln();

      str = str.toUpperCase();

      count = 0;
      System.out.println("Your input contains the following
letters:");
      System.out.println();
      System.out.print("   ");
      for ( letter = 'A'; letter <= 'Z'; letter++ ) {
          int i;  // Position of a character in str.
          for ( i = 0; i < str.length(); i++ ) {
              if ( letter == str.charAt(i) ) {
                  System.out.print(letter);
                  System.out.print(' ');
                  count++;
                  break;
              }
          }
      }
      System.out.println();
      System.out.println();
      System.out.println("There were " + count + " different
letters.");

   } // end main()

} // end class ListLetters
```

In fact, there is actually an easier way to determine whether a given letter occurs in a string, `str`. The built-in function `str.indexOf(letter)` will return `-1` if `letter` does **not** occur in the string. It returns a number greater than or equal to zero if it does occur. So, we could check whether `letter` occurs in `str` simply by checking "`if (str.indexOf(letter) >= 0)`". If we used this technique in the above program, we wouldn't need a nested `for` loop. This gives you a preview of how subroutines can be used to deal with complexity.

The if Statement

THE FIRST OF THE TWO BRANCHING STATEMENTS in Java is the `if` statement, which you have already seen in Section 3.1. It takes the form

```
if (boolean-expression)
     statement-1
else
     statement-2
```

As usual, the statements inside an `if` statement can be blocks. The `if` statement represents a two-way branch. The `else` part of an `if` statement -- consisting of the word "else" and the statement that follows it -- can be omitted.

3.5.1 The Dangling else Problem

Now, an `if` statement is, in particular, a statement. This means that either **statement-1** or **statement-2** in the above `if` statement can itself be an `if` statement. A problem arises, however, if **statement-1** is an `if` statement that has no `else` part. This special case is effectively forbidden by the syntax of Java. Suppose, for example, that you type

```
if ( x > 0 )
    if (y > 0)
       System.out.println("First case");
else
    System.out.println("Second case");
```

Now, remember that the way you've indented this doesn't mean anything at all to the computer. You might think that the `else` part is the second half of your "`if (x > 0)`" statement, but the rule that the computer follows attaches the `else` to "`if (y > 0)`", which is closer. That is, the computer reads your statement as if it were formatted:

```
if ( x > 0 )
    if (y > 0)
       System.out.println("First case");
    else
        System.out.println("Second case");
```

You can force the computer to use the other interpretation by enclosing the nested `if` in a block:

```
if ( x > 0 ) {
    if (y > 0)
       System.out.println("First case");
}
else
    System.out.println("Second case");
```

These two `if` statements have different meanings: In the case when `x <= 0`, the first statement doesn't print anything, but the second statement prints "Second case".

3.5.2 Multiway Branching

Much more interesting than this technicality is the case where **statement-2**, the `else` part of the `if` statement, is itself an `if` statement. The statement would look like this (perhaps without the final else part):

```
if (boolean-expression-1)
     statement-1
else
     if (boolean-expression-2)
         statement-2
     else
         statement-3
```

However, since the computer doesn't care how a program is laid out on the page, this is almost always written in the format:

```
if (boolean-expression-1)
     statement-1
else if (boolean-expression-2)
     statement-2
else
     statement-3
```

You should think of this as a single statement representing a three-way branch. When the computer executes this, one and only one of the three statements -- **statement-1**, **statement-2**, or **statement-3** -- will be executed. The computer starts by evaluating **boolean-expression-1**. If it is `true`, the computer executes **statement-1** and then jumps all the way to the end of the outer if statement, skipping the other two **statements**. If **boolean-expression-1** is `false`, the computer skips **statement-1** and executes the second, nested if statement. To do this, it tests the value of **boolean-expression-2** and uses it to decide between **statement-2** and **statement-3**.

Here is an example that will print out one of three different messages, depending on the value of a variable named `temperature`:

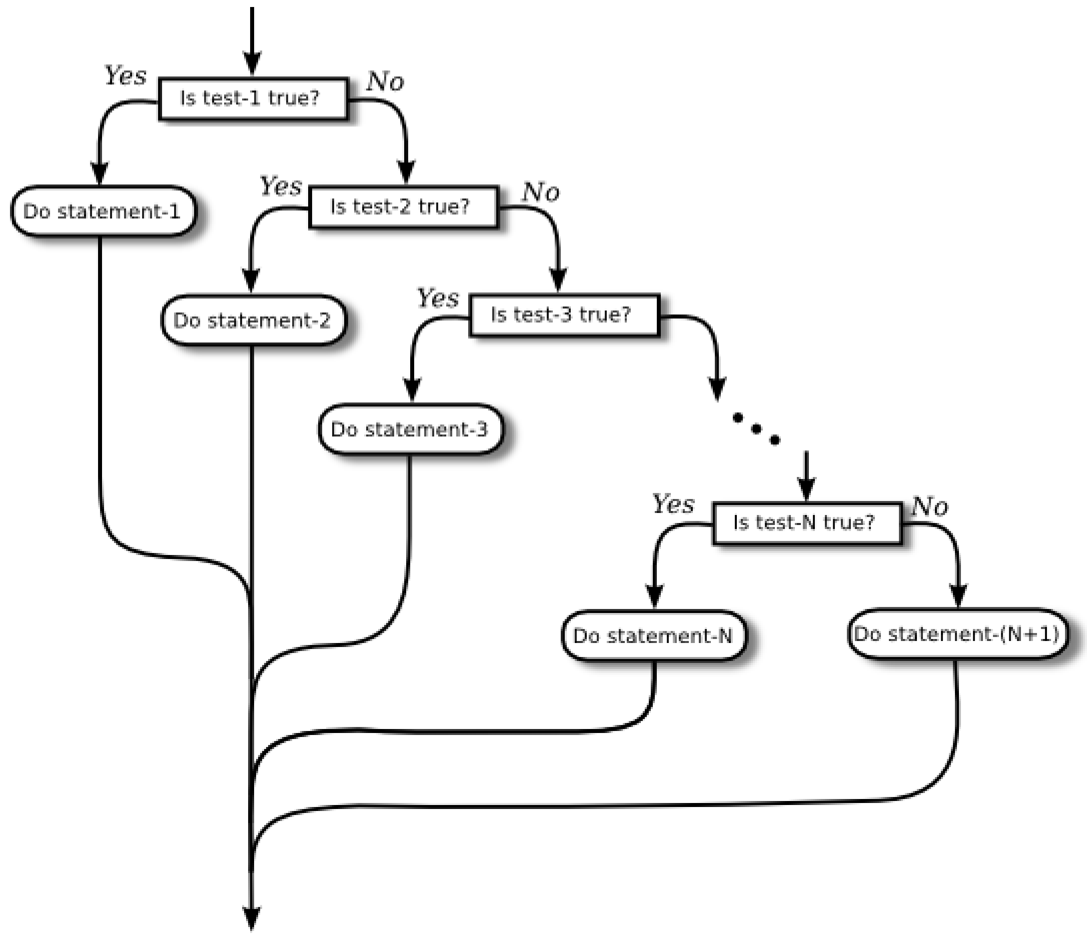
```
if (temperature < 50)
   System.out.println("It's cold.");
else if (temperature < 80)
   System.out.println("It's nice.");
else
   System.out.println("It's hot.");
```

If `temperature` is, say, 42, the first test is `true`. The computer prints out the message "It's cold", and skips the rest -- without even evaluating the second condition. For a temperature of 75, the first test is `false`, so the computer goes on to the second test. This test is `true`, so the computer prints "It's nice" and skips the rest. If the temperature is 173, both of the tests evaluate to `false`, so the computer says "It's hot" (unless its circuits have been fried by the heat, that is).

You can go on stringing together "else-if's" to make multi-way branches with any number of cases:

```
if (test-1)
     statement-1
else if (test-2)
     statement-2
else if (test-3)
     statement-3
  .
  . // (more cases)
  .
else if (test-N)
     statement-N
else
     statement-(N+1)
```

The computer evaluates the tests, which are boolean expressions, one after the other until it comes to one that is `true`. It executes the associated statement and skips the rest. If none of the boolean expressions evaluate to `true`, then the statement in the `else` part is executed. This statement is called a multi-way branch because one and only one of the statements will be executed. The final `else` part can be omitted. In that case, if all the boolean expressions are false, none of the statements are executed. Of course, each of the statements can be a block, consisting of a number of statements enclosed between { and }. Admittedly, there is lot of syntax here; as you study and practice, you'll become comfortable with it. It might be useful to look at a flow control diagram for the general "if..else if" statement shown above:



3.5.3 If Statement Examples

As an example of using `if` statements, let's suppose that `x`, `y`, and `z` are variables of type int, and that each variable has already been assigned a value. Consider the problem of printing out the values of the three variables in increasing order. For examples, if the values are 42, 17, and 20, then the output should be in the order 17, 20, 42.

One way to approach this is to ask, where does `x` belong in the list? It comes first if it's less than both `y` and `z`. It comes last if it's greater than both `y` and `z`. Otherwise, it comes in the middle. We can express this with a 3-way `if` statement, but we still have to worry about the order in which `y` and `z` should be printed. In pseudocode,

```
if (x < y && x < z) {
    output x, followed by y and z in their correct order
}
else if (x > y && x > z) {
    output y and z in their correct order, followed by x
}
else {
    output x in between y and z in their correct order
}
```

Determining the relative order of `y` and `z` requires another `if` statement, so this becomes

```
if (x < y && x < z) {        // x comes first
    if (y < z)
       System.out.println( x + " " + y + " " + z );
    else
       System.out.println( x + " " + z + " " + y );
}
else if (x > y && x > z) {   // x comes last
    if (y < z)
       System.out.println( y + " " + z + " " + x );
    else
       System.out.println( z + " " + y + " " + x );
}
else {                       // x in the middle
    if (y < z)
       System.out.println( y + " " + x + " " + z);
    else
       System.out.println( z + " " + x + " " + y);
}
```

You might check that this code will work correctly even if some of the values are the same. If the values of two variables are the same, it doesn't matter which order you print them in.

Note, by the way, that even though you can say in English "if x is less than y and z," you can't say in Java "`if (x < y && z)`". The `&&` operator can only be used between boolean values, so you have to make separate tests, `x<y` and `x<z`, and then combine the two tests with `&&`.

There is an alternative approach to this problem that begins by asking, "which order should `x` and `y` be printed in?" Once that's known, you only have to decide where to stick in `z`. This line of thought leads to different Java code:

```
if ( x < y ) {  // x comes before y
   if ( z < x )   // z comes first
      System.out.println( z + " " + x + " " + y);
   else if ( z > y )   // z comes last
      System.out.println( x + " " + y + " " + z);
   else   // z is in the middle
      System.out.println( x + " " + z + " " + y);
}
else {          // y comes before x
   if ( z < y )   // z comes first
      System.out.println( z + " " + y + " " + x);
   else if ( z > x )  // z comes last
      System.out.println( y + " " + x + " " + z);
   else  // z is in the middle
      System.out.println( y + " " + z + " " + x);
```

```
          }
```

Once again, we see how the same problem can be solved in many different ways. The two approaches to this problem have not exhausted all the possibilities. For example, you might start by testing whether `x` is greater than `y`. If so, you could swap their values. Once you've done that, you know that `x` should be printed before `y`.

Finally, let's write a complete program that uses an `if` statement in an interesting way. I want a program that will convert measurements of length from one unit of measurement to another, such as miles to yards or inches to feet. So far, the problem is extremely under-specified. Let's say that the program will only deal with measurements in inches, feet, yards, and miles. It would be easy to extend it later to deal with other units. The user will type in a measurement in one of these units, such as "17 feet" or "2.73 miles". The output will show the length in terms of **each** of the four units of measure. (This is easier than asking the user which units to use in the output.) An outline of the process is

```
Read the user's input measurement and units of measure
Express the measurement in inches, feet, yards, and miles
Display the four results
```

The program can read both parts of the user's input from the same line by using `TextIO.getDouble()` to read the numerical measurement and `TextIO.getlnWord()` to read the unit of measure. The conversion into different units of measure can be simplified by first converting the user's input into inches. From there, the number of inches can easily be converted into feet, yards, and miles. Before converting into inches, we have to test the input to determine which unit of measure the user has specified:

```
Let measurement = TextIO.getDouble()
Let units = TextIO.getlnWord()
if the units are inches
   Let inches = measurement
else if the units are feet
   Let inches = measurement * 12         // 12 inches per foot
else if the units are yards
   Let inches = measurement * 36         // 36 inches per yard
else if the units are miles
   Let inches = measurement * 12 * 5280  // 5280 feet per mile
else
   The units are illegal!
   Print an error message and stop processing
Let feet = inches / 12.0
Let yards = inches / 36.0
Let miles = inches / (12.0 * 5280.0)
Display the results
```

Since `units` is a *String*, we can use `units.equals("inches")` to check whether the specified unit of measure is "inches". However, it would be nice to allow the units to be specified as "inch" or abbreviated to "in". To allow these three possibilities, we can check `if` `(units.equals("inches") || units.equals("inch") || units.equals("in"))`. It would also be nice to allow upper case letters, as in "Inches" or "IN". We can do this by converting `units` to lower case before testing it or by substituting the function `units.equalsIgnoreCase` for `units.equals`.

In my final program, I decided to make things more interesting by allowing the user to repeat the process of entering a measurement and seeing the results of the conversion for each measurement. The program will end only when the user inputs 0. To program that, I just had to wrap the above algorithm inside a `while` loop, and make sure that the loop ends when the user inputs a 0. Here's the complete program:

```
/**
 * This program will convert measurements expressed in inches,
 * feet, yards, or miles into each of the possible units of
 * measure.  The measurement is input by the user, followed by
 * the unit of measure.  For example:  "17 feet", "1 inch", or
 * "2.73 mi".  Abbreviations in, ft, yd, and mi are accepted.
 * The program will continue to read and convert measurements
 * until the user enters an input of 0.
 */

 public class LengthConverter {

    public static void main(String[] args) {

       double measurement;  // Numerical measurement, input by
user.
       String units;        // The unit of measure for the input,
also
                            //    specified by the user.

       double inches, feet, yards, miles;  // Measurement expressed
in
                                           //   each possible unit
of
                                           //   measure.

       System.out.println("Enter measurements in inches, feet,
yards, or miles.");
       System.out.println("For example:  1 inch    17 feet    2.73
miles");
       System.out.println("You can use abbreviations:   in   ft  yd
mi");
       System.out.println("I will convert your input into the other
units");
       System.out.println("of measure.");
       System.out.println();

       while (true) {

          /* Get the user's input, and convert units to lower case.
*/
```

```
          System.out.print("Enter your measurement, or 0 to end:
");
          measurement = TextIO.getDouble();
          if (measurement == 0)
             break;  // Terminate the while loop.
          units = TextIO.getlnWord();
          units = units.toLowerCase();  // convert units to lower
case

          /* Convert the input measurement to inches. */

          if (units.equals("inch") || units.equals("inches")
                                        || units.equals("in")) {
              inches = measurement;
          }
          else if (units.equals("foot") || units.equals("feet")
                                        || units.equals("ft")) {
              inches = measurement * 12;
          }
          else if (units.equals("yard") || units.equals("yards")
                                         || units.equals("yd")) {
              inches = measurement * 36;
          }
          else if (units.equals("mile") || units.equals("miles")
                                           || units.equals("mi"))
{
              inches = measurement * 12 * 5280;
          }
          else {
              System.out.println("Sorry, but I don't understand \""
                                                 + units +
"\".");
              continue;  // back to start of while loop
          }

          /* Convert measurement in inches to feet, yards, and
miles. */

          feet = inches / 12;
          yards = inches / 36;
          miles = inches / (12*5280);

          /* Output measurement in terms of each unit of measure.
*/

          System.out.println();
          System.out.println("That's equivalent to:");
          System.out.printf("%12.5g", inches);
          System.out.println(" inches");
          System.out.printf("%12.5g", feet);
          System.out.println(" feet");
          System.out.printf("%12.5g", yards);
          System.out.println(" yards");
          System.out.printf("%12.5g", miles);
          System.out.println(" miles");
          System.out.println();
```

```
            } // end while

            System.out.println();
            System.out.println("OK!  Bye for now.");

         } // end main()

      } // end class LengthConverter
```

(Note that this program uses formatted output with the "g" format specifier. In this program, we have no control over how large or how small the numbers might be. It could easily make sense for the user to enter very large or very small measurements. The "g" format will print a real number in exponential form if it is very large or very small, and in the usual decimal form otherwise. Remember that in the format specification `%12.5g`, the 5 is the total number of significant digits that are to be printed, so we will always get the same number of significant digits in the output, no matter what the size of the number. If we had used an "f" format specifier such as `%12.5f`, the output would be in decimal form with 5 digits after the decimal point. This would print the number 0.000000000745482 as `0.00000`, with no **significant** digits at all! With the "g" format specifier, the output would be `7.4549e-10`.)

3.5.4 The Empty Statement

As a final note in this section, I will mention one more type of statement in Java: the empty statement. This is a statement that consists simply of a semicolon and which tells the computer to do nothing. The existence of the empty statement makes the following legal, even though you would not ordinarily see a semicolon after a } :

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

The semicolon is legal after the }, but the computer considers it to be an empty statement, not part of the `if` statement. Occasionally, you might find yourself using the empty statement when what you mean is, in fact, "do nothing." For example, the rather contrived `if` statement

```
if ( done )
   ;  // Empty statement
else
   System.out.println( "Not done yet.");
```

does nothing when the boolean variable `done` is true, and prints out "Not done yet" when it is false. You can't just leave out the semicolon in this example, since Java syntax requires an actual statement between the `if` and the `else`. I prefer, though, to use an empty block, consisting of { and } with nothing between, for such cases.

Occasionally, stray empty statements can cause annoying, hard-to-find errors in a program. For example, the following program segment prints out "Hello" just **once**, not ten times:

```
for (int i = 0; i < 10; i++);
    System.out.println("Hello");
```

Why? Because the ";" at the end of the first line is a statement, and it is this empty statement that is executed ten times. The `System.out.println` statement is not really inside the `for` statement at all, so it is executed just once, after the `for` loop has completed. The `for` loop just does nothing, ten times!

THE SECOND BRANCHING STATEMENT in Java is the `switch` statement, which is introduced in this section. The `switch` statement is used far less often than the `if` statement, but it is sometimes useful for expressing a certain type of multiway branch.

3.6.1 The Basic switch Statement

A switch statement allows you to test the value of an expression and, depending on that value, to jump directly to some location within the switch statement. Only expressions of certain types can be used. The value of the expression can be one of the primitive integer types int, short, or byte. It can be the primitive char type. It can be *String*. Or it can be an enum type (see Subsection 2.3.4 for an introduction to enums). In particular, note that the expression **cannot** be a double or float value.

The positions within a switch statement to which it can jump are marked with case labels that take the form: "case **constant**:". The **constant** here is a literal of the same type as the expression in the `switch`. A case label marks the position the computer jumps to when the expression evaluates to the given **constant** value. As the final case in a switch statement you can, optionally, use the label "default:", which provides a default jump point that is used when the value of the expression is not listed in any case label.

A `switch` statement, as it is most often used, has the form:

```
switch (expression) {
   case constant-1:
      statements-1
      break;
   case constant-2:
      statements-2
      break;
      .
      .   // (more cases)
      .
   case constant-N:
      statements-N
      break;
   default:  // optional default case
      statements-(N+1)
} // end of switch statement
```

This has exactly the same effect as the following multiway `if` statement, but the `switch` statement can be more efficient because the computer can evaluate one expression and jump directly to the correct case, whereas in the `if` statement, the computer must evaluate up to `N` expressions before it knows which set of statements to execute:

```
if (expression == constant-1) { // but use .equals for String!!
    statements-2
}
else if (expression == constant-2) {
    statements-3
}
else
    .
    .
    .
else if (expression == constant-N) {
    statements-N
}
else {
    statements-(N+1)
}
```

The `break` statements in the `switch` are technically optional. The effect of a `break` is to make the computer jump past the end of the switch statement, skipping over all the remaining cases. If you leave out the break statement, the computer will just forge ahead after completing one case and will execute the statements associated with the next case label. This is rarely what you want, but it is legal. (I will note here -- although you won't understand it until you get to the next chapter -- that inside a subroutine, the `break` statement is sometimes replaced by a `return` statement, which terminates the subroutine as well as the switch.)

Introduction to Exceptions and try..catch

IN ADDITION TO THE CONTROL structures that determine the normal flow of control in a program, Java has a way to deal with "exceptional" cases that throw the flow of control off its normal track. When an error occurs during the execution of a program, the default behavior is to terminate the program and to print an error message. However, Java makes it possible to "catch" such errors and program a response different from simply letting the program crash. This is done with the try..catch statement. In this section, we will take a preliminary and incomplete look the `try..catch` statement, leaving out a lot of the rather complex syntax of this statement. Error handling is a complex topic, which we will return to in Chapter 8, and we will cover the full syntax of `try..catch` at that time.

3.7.1 Exceptions

The term exception is used to refer to the type of error that one might want to handle with a `try..catch`. An exception is an exception to the normal flow of control in the program. The term is used in preference to "error" because in some cases, an exception might not be considered to be an error at all. You can sometimes think of an exception as just another way to organize a program.

Exceptions in Java are represented as objects of type *Exception*. Actual exceptions are usually defined by subclasses of *Exception*. Different subclasses represent different types of exceptions. We will look at only two types of exception in this section: *NumberFormatException* and *IllegalArgumentException*.

A *NumberFormatException* can occur when an attempt is made to convert a string into a number. Such conversions are done by the functions `Integer.parseInt` and `Double.parseDouble`. (See Subsection 2.5.7.) Consider the function call `Integer.parseInt(str)` where `str` is a variable of type *String*. If the value of `str` is the string `"42"`, then the function call will correctly convert the string into the int 42. However, if the value of `str` is, say, `"fred"`, the function call will fail because `"fred"` is not a legal string representation of an int value. In this case, an exception of type *NumberFormatException* occurs. If nothing is done to handle the exception, the program will crash.

An *IllegalArgumentException* can occur when an illegal value is passed as a parameter to a subroutine. For example, if a subroutine requires that a parameter be greater than or equal to zero, an *IllegalArgumentException* might occur when a negative value is passed to the subroutine. How to respond to the illegal value is up to the person who wrote the subroutine, so we can't simply say that every illegal parameter value will result in an *IllegalArgumentException*. However, it is a common response.

3.7.2 try..catch

When an exception occurs, we say that the exception is "thrown". For example, we say that `Integer.parseInt(str)` throws an exception of type *NumberFormatException* when the value of `str` is illegal. When an exception is thrown, it is possible to "catch" the exception and prevent it from crashing the program. This is done with a try..catch statement. In simplified form, the syntax for a `try..catch` can be:

```
try {
   statements-1
}
catch ( exception-class-name  variable-name ) {
   statements-2
}
```

The **exception-class-name** could be *NumberFormatException*, *IllegalArgumentException*, or some other exception class. When the computer executes this `try..catch` statement, it executes the statements in the `try` part. If no exception occurs during the execution of **statements-1**, then the computer just skips over the `catch` part and proceeds with the rest of the program. However, if an exception of type **exception-class-name** occurs during the execution of **statements-1**, the computer immediately jumps from the point where the exception occurs to the `catch` part and executes **statements-2**, skipping any remaining statements in **statements-1**. Note that only one type of exception is caught; if some other type of exception occurs during the execution of **statements-1**, it will crash the program as usual.

During the execution of **statements-2**, the **variable-name** represents the exception object, so that you can, for example, print it out. The exception object contains information about the cause of the exception. This includes an error message, which will be displayed if you print out the exception object.

After the end of the `catch` part, the computer proceeds with the rest of the program; the exception has been caught and handled and does not crash the program.

By the way, note that the braces, { and }, are part of the syntax of the `try..catch` statement. They are required even if there is only one statement between the braces. This is different from the other statements we have seen, where the braces around a single statement are optional.

As an example, suppose that `str` is a variable of type *String* whose value might or might not represent a legal real number. Then we could say:

```
double x;
try {
   x = Double.parseDouble(str);
   System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
   System.out.println( "Not a legal number." );
   x = Double.NaN;
}
```

If an error is thrown by the call to `Double.parseDouble(str)`, then the output statement in the `try` part is skipped, and the statement in the `catch` part is executed. (In this example, I set `x` to be the value `Double.NaN` when an exception occurs. `Double.NaN` is the special "not-a-number" value for type double.)

It's **not** always a good idea to catch exceptions and continue with the program. Often that can just lead to an even bigger mess later on, and it might be better just to let the exception crash the program at the point where it occurs. However, sometimes it's possible to recover from an error.

Suppose, for example, we want a program that will find the average of a sequence of real numbers entered by the user, and we want the user to signal the end of the sequence by entering a blank line. (This is similar to the sample program *ComputeAverage.java* from Section 3.3, but in that program the user entered a zero to signal end-of-input.) If we use `TextIO.getlnInt()` to read the user's input, we will have no way of detecting the blank line, since that function simply skips over blank lines. A solution is to use `TextIO.getln()` to read the user's input. This allows us to detect a blank input line, and we can convert non-blank inputs to numbers using `Double.parseDouble`. And we can use `try..catch` to avoid crashing the program when the user's input is not a legal number. Here's the program:

```
public class ComputeAverage2 {

   public static void main(String[] args) {
       String str;     // The user's input.
       double number;  // The input converted into a number.
       double total;   // The total of all numbers entered.
       double avg;     // The average of the numbers.
       int count;      // The number of numbers entered.
       total = 0;
       count = 0;
       System.out.println("Enter your numbers, press return to
end.");
       while (true) {
          System.out.print("? ");
          str = TextIO.getln();
          if (str.equals("")) {
             break; // Exit the loop, since the input line was
blank.
          }
          try {
              number = Double.parseDouble(str);
              // If an error occurs, the next 2 lines are skipped!
              total = total + number;
              count = count + 1;
          }
          catch (NumberFormatException e) {
              System.out.println("Not a legal number!  Try
again.");
          }
       }
       avg = total/count;
       System.out.printf("The average of %d numbers is %1.6g%n",
count, avg);
   }

}
```

3.7.3 Exceptions in TextIO

When `TextIO` reads a numeric value from the user, it makes sure that the user's response is legal, using a technique similar to the `while` loop and `try..catch` in the previous example. However, `TextIO` can read data from other sources besides the user. (See Subsection 2.4.4.) When it is reading from a file, there is no reasonable way for `TextIO` to recover from an illegal value in the input, so it responds by throwing an exception. To keep things simple, `TextIO` only throws exceptions of type *IllegalArgumentException*, no matter what type of error it encounters. For example, an exception will occur if an attempt is made to read from a file after all the data in the file has already been read. In `TextIO`, the exception is of type *IllegalArgumentException*. If you have a better response to file errors than to let the program crash, you can use a `try..catch` to catch exceptions of type *IllegalArgumentException*.

As an example, we will look at yet another number-averaging program. In this case, we will read the numbers from a file. Assume that the file contains nothing but real numbers, and we want a program that will read the numbers and find their sum and their average. Since it is unknown how many numbers are in the file, there is the question of when to stop reading. One approach is simply to try to keep reading indefinitely. When the end of the file is reached, an exception occurs. This exception is not really an error -- it's just a way of detecting the end of the data, so we can catch the exception and finish up the program. We can read the data in a `while (true)` loop and break out of the loop when an exception occurs. This is an example of the somewhat unusual technique of using an exception as part of the expected flow of control in a program.

To read from the file, we need to know the file's name. To make the program more general, we can let the user enter the file name, instead of hard-coding a fixed file name in the program. However, it is possible that the user will enter the name of a file that does not exist. When we use `TextIO.readfile` to open a file that does not exist, an exception of type *IllegalArgumentException* occurs. We can catch this exception and ask the user to enter a different file name. Here is a complete program that uses all these ideas:

```
/**
 * This program reads numbers from a file.  It computes the sum and
 * the average of the numbers that it reads.  The file should
contain
 * nothing but numbers of type double; if this is not the case, the
 * output will be the sum and average of however many numbers were
 * successfully read from the file.  The name of the file will be
 * input by the user.
 */
public class AverageNumbersFromFile {

   public static void main(String[] args) {

      while (true) {
         String fileName;  // The name of the file, to be input by
the user.
         System.out.print("Enter the name of the file: ");
         fileName = TextIO.getln();
         try {
            TextIO.readFile( fileName );  // Try to open the file
for input.
            break;  // If that succeeds, break out of the loop.
         }
         catch ( IllegalArgumentException e ) {
            System.out.println("Can't read from the file \"" +
fileName + "\".");
            System.out.println("Please try again.\n");
         }
      }

      /* At this point, TextIO is reading from the file. */

      double number;  // A number read from the data file.
      double sum;     // The sum of all the numbers read so far.
      int count;      // The number of numbers that were read.

      sum = 0;
      count = 0;

      try {
         while (true) { // Loop ends when an exception occurs.
             number = TextIO.getDouble();
             count++;  // This is skipped when the exception occurs
             sum += number;
         }
      }
      catch ( IllegalArgumentException e ) {
         // We expect this to occur when the end-of-file is
encountered.
         // We don't consider this to be an error, so there is
nothing to do
         // in this catch clause.  Just proceed with the rest of
the program.
      }

      // At this point, we've read the entire file.

      System.out.println();
      System.out.println("Number of data values read: " + count);
      System.out.println("The sum of the data values: " + sum);
      if ( count == 0 )
         System.out.println("Can't compute an average of 0
values.");
      else
         System.out.println("The average of the values:  " +
(sum/count));
```

3.8.1 Creating and Using Arrays

A data structure consists of a number of data items chunked together so that they can be treated as a unit. An array is a data structure in which the items are arranged as a numbered sequence, so that each individual item can be referred to by its position number. In Java -- but not in other programming languages -- all the items must be of the same type, and the numbering always starts at zero. You will need to learn several new terms to talk about arrays: The number of items in an array is called the length of the array. The type of the individual items in an array is called the base type of the array. And the position number of an item in an array is called the index of that item.

Suppose that you want to write a program that will process the names of, say, one thousand people. You will need a way to deal with all that data. Before you knew about arrays, you might have thought that the program would need a thousand variables to hold the thousand names, and if you wanted to print out all the names, you would need a thousand print statements. Clearly, that would be ridiculous! In reality, you can put all the names into an array. The array is a represented by a single variable, but it holds the entire list of names. The length of the array would be 1000, since there are 1000 individual names. The base type of the array would be *String* since the items in the array are strings. The first name would be at index 0 in the array, the second name at index 1, and so on, up to the thousandth name at index 999.

The base type of an array can be any Java type, but for now, we will stick to arrays whose base type is *String* or one of the eight primitive types. If the base type of an array is int, it is referred to as an "array of ints." An array with base type *String* is referred to as an "array of *Strings*." However, an array is not, properly speaking, a list of integers or strings or other **values**. It is better thought of as a list of **variables** of type int, or a list of variables of type *String*, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array), just as a variable can hold a value. The value can be changed at any time, just as the value of a variable can be changed. The items in an array -- really, the individual variables that make up the array -- are more often referred to as the elements of the array.

As I mentioned above, when you use an array in a program, you can use a variable to refer to array as a whole. But you often need to refer to the individual elements of the array. The name for an element of an array is based on the name for the array and the index number of the element. The syntax for referring to an element looks, for example, like this: `namelist[7]`. Here, `namelist` is the variable that names the array as a whole, and `namelist[7]` refers to the element at index 7 in that array. That is, to refer to an element of an array, you use the array name, followed by element index enclosed in square brackets. An element name of this form can be used like any other variable: You can assign a value to it, print it out, use it in an expression.

An array also contains a kind of variable representing its length. For example, you can refer to the length of the array `namelist` as `namelist.length`. However, you cannot assign a value to `namelist.length`, since the length of an array cannot be changed.

Before you can use a variable to refer to an array, that variable must be declared, and it must have a type. For an array of *Strings*, for example, the type for the array variable would be `String[]`, and for an array of ints, it would be `int[]`. In general, an array type consists of the base type of the array followed by a pair of empty square brackets. Array types can be used to declare variables; for example,

```
String[] namelist;
int[] A;
double[] prices;
```

and variables declared in this way can refer to arrays. However, declaring a variable does not make the actual array. Like all variables, an array variable has to be assigned a value before it can be used. In this case, the value is an array. Arrays have to be created using a special syntax.

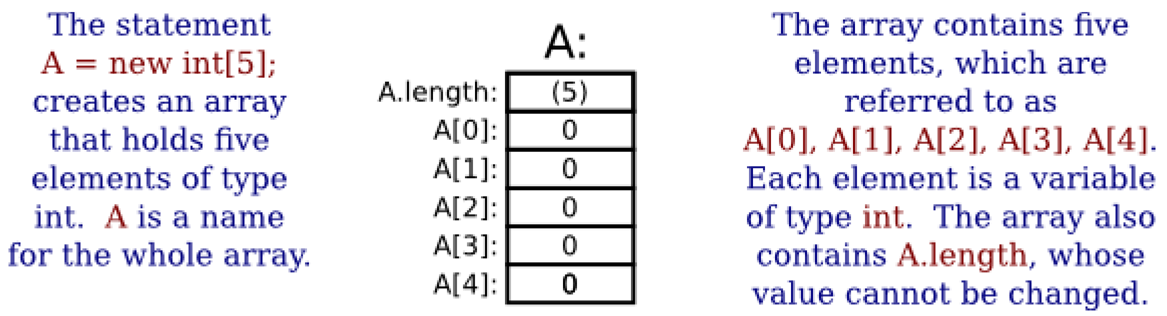
(The syntax is related to the fact that arrays in Java are actually objects, but that doesn't need to concern us here.) Arrays are created with an operator named new. Here are some examples:

```
namelist = new String[1000];
A = new int[5];
prices = new double[100];
```

The general syntax is

```
array-variable = new base-type[array-length];
```

The length of the array can be given as either an integer or an integer-valued expression. For example, after the assignment statement "`A = new int[5];`", `A` is an array containing the five integer elements `A[0]`, `A[1]`, `A[2]`, `A[3]`, and `A[4]`. Also, `A.length` would have the value 5. It's useful to have a picture in mind:


The statement
A = new int[5];
creates an array that holds five elements of type int. A is a name for the whole array.
A:
A.length: (5)
A[0]: 0
A[1]: 0
A[2]: 0
A[3]: 0
A[4]: 0
The array contains five elements, which are referred to as
A[0], A[1], A[2], A[3], A[4].
Each element is a variable of type int. The array also contains A.length, whose value cannot be changed.

When you create an array of int, each element of the array is automatically initialized to zero. Any array of numbers is filled with zeros when it is created. An array of boolean is filled with the value `false`. And an array of char is filled with the character that has Unicode code number zero. (For an array of *String*, the initial value is `null`, a special value used for objects that we won't encounter officially until Section 5.1.)

3.8.2 Arrays and For Loops

A lot of the real power of arrays comes from the fact that the index of an element can be given by an integer variable or even an integer-valued expression. For example, if `list` is an array and `i` is a variable of type int, then you can use `list[i]` and even `list[2*i+1]` as variable names. The meaning of `list[i]` depends on the value of `i`. This becomes especially useful when we want to process all the elements of an array, since that can be done with a `for` loop. For example, to print out all the items in an array, `list`, we can just write

```
int i;  // the array index
for (i = 0; i < list.length; i++) {
    System.out.println( list[i] );
}
```

The first time through the loop, `i` is 0, and `list[i]` refers to `list[0]`. So, it is the value stored in the variable `list[0]` that is printed. The second time through the loop, `i` is 1, and the value stored in `list[1]` is printed. The loop ends after printing the value of `list[4]`, when `i` becomes equal to 5 and the continuation condition "`i < list.length`" is no longer true. This is a typical example of using a loop to process an array.

Let's look at a few more examples. Suppose that `A` is an array of double, and we want to find the average of all the elements of the array. We can use a `for` loop to add up the numbers, and then divide by the length of the array to get the average:

```
double total;     // The sum of the numbers in the array.
double average;   // The average of the numbers.
int i;  // The array index.
total = 0;
for ( i = 0; i < A.length; i++ ) {
    total = total + A[i];  // Add element number i to the total.
}
average = total / A.length;  // A.length is the number of items
```

Another typical problem is to find the largest number in the array `A`. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called `max`. As we look through the array, whenever we find a number larger than the current value of `max`, we change the value of `max` to that larger value. After the whole array has been processed, `max` is the largest item in the array overall. The only question is, what should the original value of `max` be? One possibility is to start with `max` equal to `A[0]`, and then to look through the rest of the array, starting from `A[1]`, for larger items:

```
double max;  // The largest number seen so far.
max = A[0];   // At first, the largest number seen is A[0].
int i;
for ( i = 1; i < A.length; i++ ) {
    if (A[i] > max) {
       max = A[i];
    }
}
// at this point, max is the largest item in A
```

Sometimes, you only want to process some elements of the array. In that case, you can use an `if` statement inside the `for` loop to decide whether or not to process a given element. Let's look at the problem of averaging the elements of an array, but this time, suppose that we only want to average the non-zero elements. In this case, the number of items that we add up can be less than the length of the array, so we will need to keep a count of the number of items added to the sum:

```
double total;     // The sum of the non-zero numbers in the array.
int count;        // The number of non-zero numbers.
double average;   // The average of the non-zero numbers.
int i;
total = 0;
count = 0;
for ( i = 0; i < A.length; i++ ) {
    if ( A[i] != 0 ) {
        total = total + A[i];   // Add element to the total
        count = count + 1;      //      and count it.
    }
}
if (count == 0) {
    System.out.println("There were no non-zero elements.");
}
else {
    average = total / count;  // Divide by number of items
    System.out.printf("Average of %d elements is %1.5g%n",
                            count, average);
}
```

3.8.3 Random Access

So far, my examples of array processing have used sequential access. That is, the elements of the array were processed one after the other in the sequence in which they occur in the array. But one of the big advantages of arrays is that they allow random access. That is, every element of the array is equally accessible at any given time.

As an example, let's look at a well-known problem called the birthday problem: Suppose that there are `N` people in a room. What's the chance that there are two people in the room who have the same birthday? (That is, they were born on the same day in the same month, but not necessarily in the same year.) Most people severely underestimate the probability. We will actually look at a different version of the question: Suppose you choose people at random and check their birthdays. How many people will you check before you find one who has the same birthday as someone you've already checked? Of course, the answer in a particular case depends on random factors, but we can simulate the experiment with a computer program and run the program several times to get an idea of how many people need to be checked on average.

To simulate the experiment, we need to keep track of each birthday that we find. There are 365 different possible birthdays. (We'll ignore leap years.) For each possible birthday, we need to keep track of whether or not we have already found a person who has that birthday. The answer to this question is a boolean value, true or false. To hold the data for all 365 possible birthdays, we can use an array of 365 boolean values:

```
boolean[] used;
used = new boolean[365];
```

For this problem, the days of the year are numbered from 0 to 364. The value of `used[i]` is true if someone has been selected whose birthday is day number `i`. Initially, all the values in the array are false. (Remember that this is done automatically when the array is created.) When we select someone whose birthday is day number `i`, we first check whether `used[i]` is `true`. If it is `true`, then this is the second person with that birthday. We are done. On the other hand, if `used[i]` is `false`, we set `used[i]` to be `true` to record the fact that we've encountered someone with that birthday, and we go on to the next person. Here is a program that carries out the simulated experiment (of course, in the program, there are no simulated people, only simulated birthdays):

```
/**
 * Simulate choosing people at random and checking the day of the
year they
 * were born on.  If the birthday is the same as one that was seen
previously,
 * stop, and output the number of people who were checked.
 */
public class BirthdayProblem {

   public static void main(String[] args) {

       boolean[] used;  // For recording the possible birthdays
                        //   that have been seen so far.  A value
                        //   of true in used[i] means that a person
                        //   whose birthday is the i-th day of the
                        //   year has been found.

       int count;       // The number of people who have been
checked.

       used = new boolean[365];  // Initially, all entries are
false.

       count = 0;

       while (true) {
             // Select a birthday at random, from 0 to 364.
             // If the birthday has already been used, quit.
             // Otherwise, record the birthday as used.

          int birthday;  // The selected birthday.
          birthday = (int)(Math.random()*365);
          count++;

          System.out.printf("Person %d has birthday number %d",
count, birthday);
          System.out.println();

          if ( used[birthday] ) {
                // This day was found before; it's a duplicate.  We
are done.
             break;
          }

          used[birthday] = true;

       } // end while

       System.out.println();
       System.out.println("A duplicate birthday was found after "
                                               + count + " tries.");
   }
} // end class BirthdayProblem
```

You should study the program to understand how it works and how it uses the array. Also, try it out! You will probably find that a duplicate birthday tends to occur sooner than you expect.

3.8.4 Partially Full Arrays

Consider an application where the number of items that we want to store in an array changes as the program runs. Since the size of the array can't be changed, a separate counter variable must be used to keep track of how many spaces in the array are in use. (Of course, every space in the array has to contain something; the question is, how many spaces contain useful or valid items?)

Consider, for example, a program that reads positive integers entered by the user and stores them for later processing. The program stops reading when the user inputs a number that is less than or equal to zero. The input numbers can be kept in an array, `numbers`, of type `int[]`. Let's say that no more than 100 numbers will be input. Then the size of the array can be fixed at 100. But the program must keep track of how many numbers have actually been read and stored in the array. For this, it can use an integer variable. Each time a number is stored in the array, we have to count it; that is, value of the counter variable must be incremented by one. One question is, when we add a new item to the array, where do we put it? Well, if the number of items is `count`, then they would be stored in the array in positions number 0, 1, ..., (count-1). The next open spot would be position number `count`, so that's where we should put the new item.

As a rather silly example, let's write a program that will read the numbers input by the user and then print them in the reverse of the order in which they were entered. Assume that an input value equal to zero marks the end of the data. (This is, at least, a processing task that requires that the numbers be saved in an array. Note that many types of processing, such as finding the sum or average or maximum of the numbers, can be done without saving the individual numbers.)

```
public class ReverseInputNumbers {

   public static void main(String[] args) {

     int[] numbers;  // An array for storing the input values.
     int count;      // The number of numbers saved in the array.
     int num;        // One of the numbers input by the user.

     numbers = new int[100];   // Space for 100 ints.
     count = 0;                // No numbers have been saved yet.

     System.out.println("Enter up to 100 positive integers; enter 0 
to end.");

     while (true) {   // Get the numbers and put them in the array.
        System.out.print("? ");
        num = TextIO.getlnInt();
        if (num <= 0) {
              // Zero marks the end of input; we have all the
numbers.
           break;
        }
        numbers[count] = num;  // Put num in position count.
        count++;  // Count the number
     }

     System.out.println("\nYour numbers in reverse order are:\n");

     for (int i = count - 1; i >= 0; i--) {
         System.out.println( numbers[i] );
     }

   } // end main();

}  // end class ReverseInputNumbers
```

It is especially important to note how the variable `count` plays a dual role. It is the number of items that have been entered into the array. But it is also the index of the next available spot in the array.

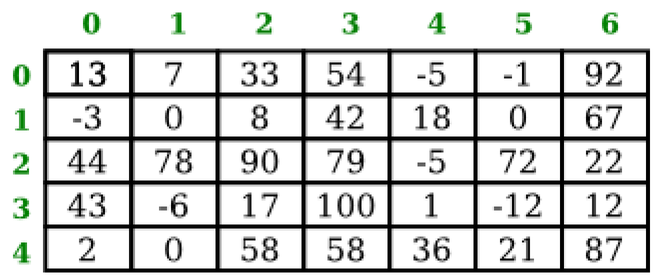
When the time comes to print out the numbers in the array, the last occupied spot in the array is location `count - 1`, so the `for` loop prints out values starting from location `count - 1` and going down to 0. This is also a nice example of processing the elements of an array in reverse order.

You might wonder what would happen in this program if the user tries to input more than 100 numbers. The result would be an error that would crash the program. When the user enters the 101-st number, the program tries to store that number in an array element `number[100]`. However, there is no such array element. There are only 100 items in the array, and the index of the last item is 99. The attempt to use `number[100]` generates an exception of type *ArrayIndexOutOfBoundsException*. Exceptions of this type are a common source of run-time errors in programs that use arrays.

3.8.5 Two-dimensional Arrays

The arrays that we have considered so far are "one-dimensional." This means that the array consists of a sequence of elements that can be thought of as being laid out along a line. It is also possible to have two-dimensional arrays, where the elements can be laid out in a rectangular grid. We consider them only briefly here, but will return to the topic in Section 7.5.

In a two-dimensional, or "2D," array, the elements can be arranged in rows and columns. Here, for example, is a 2D array of int that has five rows and seven columns:



	0	1	2	3	4	5	6
0	13	7	33	54	-5	-1	92
1	-3	0	8	42	18	0	67
2	44	78	90	79	-5	72	22
3	43	-6	17	100	1	-12	12
4	2	0	58	58	36	21	87

This 5-by-7 grid contains a total of 35 elements. The rows in a 2D array are numbered 0, 1, 2, ..., up to the number of rows minus one. Similarly, the columns are numbered from zero up to the number of columns minus one. Each individual element in the array can be picked out by specifying its row number and its column number. (The illustration shown here is not what the array actually looks like in the computer's memory, but it does show the logical structure of the array.)

In Java, the syntax for two-dimensional arrays is similar to the syntax for one-dimensional arrays, except that an extra index is involved, since picking out an element requires both a row number and a column number. For example, if `A` is a 2D array of int, then `A[3][2]` would be the element in row 3, column 2. That would pick out the number 17 in the array shown above. The type for `A` would be given as `int[][]`, with two pairs of empty brackets. To declare the array variable and create the array, you could say,

```
int[][]  A;
A  =  new int[5][7];
```

The second line creates a 2D array with 5 rows and 7 columns. Two-dimensional arrays are often processed using nested `for` loops. For example, the following code segment will print out the elements of `A` in neat columns:

```
int row, col;  // loop-control-variables for accessing rows and
columns in A
for ( row = 0; row < 5; row++ ) {
    for ( col = 0; col < 7; col++ ) {
        System.out.printf( "%7d",  A[row][col] );
    }
    System.out.println();
}
```

The base type of a 2D array can be anything, so you can have arrays of type `double[][]`, `String[][]`, and so on.

There are some natural uses for 2D arrays. For example, a 2D array can be used to store the contents of the board in a game such as chess or checkers. And an example in Subsection 4.6.3 uses a 2D array to hold the colors of a grid of colored squares. But sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2014. If the stores are numbered from 0 to 24, and if the twelve months from

January 2014 through December 2014 are numbered from 0 to 11, then the profit data could be stored in an array, `profit`, created as follows:

```
double[][]  profit;
profit  =  new double[25][12];
```

`profit[3][2]` would be the amount of profit earned at store number 3 in March, and more generally, `profit[storeNum][monthNum]` would be the amount of profit earned in store number `storeNum` in month number `monthNum` (where the numbering, remember, starts from zero).

Let's assume that the `profit` array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company -- for the whole year from all its stores -- can be calculated by adding up all the entries in the array:

```
double totalProfit;  // Company's total profit in 2014.
int store, month;  // variables for looping through the stores and
the months
totalProfit = 0;
for ( store = 0; store < 25; store++ ) {
   for ( month = 0; month < 12; month++ )
      totalProfit += profit[store][month];
}
```

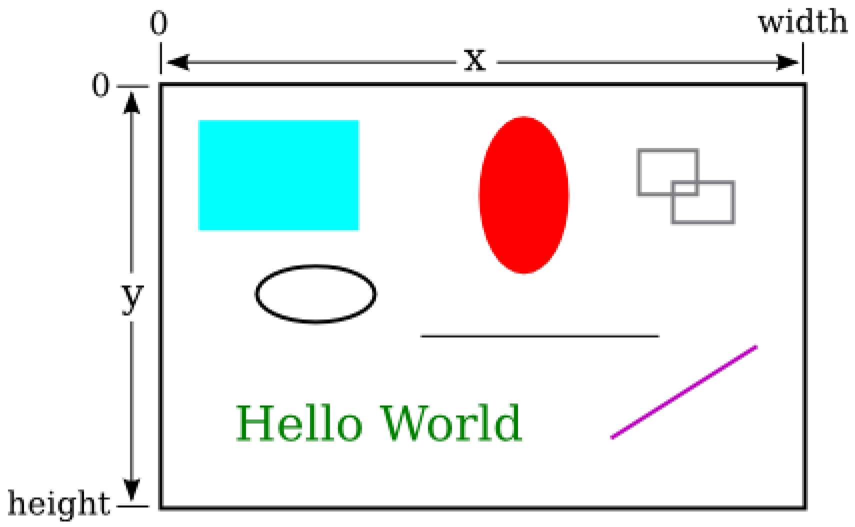
Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit;
int storeNum;
doubleProfit = 0.0;
for ( storeNum = 0; storeNum < 25; storeNum++ ) {
   decemberProfit += profit[storeNum][11];
}
```

Two-dimensional arrays are sometimes useful, but they are much less common than one-dimensional arrays. Java actually allows arrays of even higher dimension, but they are only rarely encountered in practice.

3.9.1 Drawing Shapes

To understand computer graphics, you need to know a little about pixels and coordinate systems. The computer screen is made up of small squares called pixels, arranged in rows and columns, usually about 100 pixels per inch. The computer controls the color of the pixels, and drawing is done by changing the colors of individual pixels. Each pixel has a pair of integer coordinates, often called *x* and *y*, that specify the pixel's horizontal and vertical position. For a graphics context drawing to a rectangular area on the screen, the coordinates of the pixel in the upper left corner of the rectangle are (0,0). The *x* coordinate increases from the left to right, and the *y* coordinate increases from top to bottom. Shapes are specified using pixels. For example, a rectangle is specified by the *x* and *y* coordinates of its upper left corner and by its width and height measured in pixels. Here's a picture of a rectangular drawing area, showing the ranges of *x* and *y* coordinates. The "width" and "height" in this picture are give the size of the drawing area, in pixels:



Assuming that the drawing area is 800-by-500 pixels, the rectangle in the upper left of the picture would have, approximately, width 200, height 150, and upper left corner at coordinates (50,50).

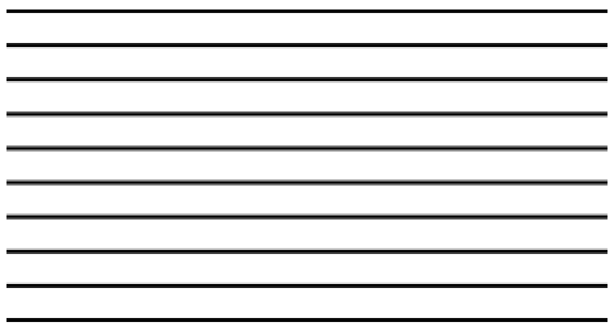
Drawing in Java is done using a graphics context. A graphics context is an object. As an object, it can include subroutines and data. Among the subroutines in a graphics context are routines for drawing basic shapes such as lines, rectangles, ovals, and text. (When text appears on the screen, the characters have to be drawn there by the computer, just like the computer draws any other shapes.) Among the data in a graphics context are the color and font that are currently selected for drawing. (A font determines the style and size of characters.) One other piece of data in a graphics context is the "drawing surface" on which the drawing is done. Generally, the drawing surface is a rectangle on the computer screen, although it can be other surfaces such as a page to be printed. Different graphics context objects can draw to different drawing surfaces. For us, the drawing surface will be the content area of a window, not including its border or title bar.

A graphics context is represented by a variable. The type for the variable is *Graphics* (just like the type for a string variable is *String*). The variable is often named *g*, but the name of the variable is of course up to the programmer. Here are a few of the subroutines that are available in a graphics context *g*:

- `g.setColor(c)`, is called to set the color to be used for drawing. The parameter, `c` is an object belonging to a class named *Color*. There are about a dozen constants representing standard colors that can be used as the parameter in this subroutine. The standard colors include `Color.BLACK`, `Color.WHITE`, `Color.LIGHT_GRAY`, `Color.RED`, Color.GREEN, and Color.BLUE. (Later, we will see that it is also possible to create new colors.) For example, if you want to draw in red, you would say "g.setColor(Color.RED);". The specified color is used for all subsequent drawing operations up until the next time g.setColor() is called.

- g.drawLine(x1,y1,x2,y2) draws a line from the point with coordinates (x1,y1) to the point with coordinates (x2,y2).
- g.drawRect(x,y,w,h) draws the outline of a rectangle with vertical and horizontal sides. The parameters x, y, w, and h must be integers or integer-valued expressions. This subroutine draws the outline of the rectangle whose top-left corner is x pixels from the left edge of the drawing area and y pixels down from the top. The width of the rectangle is w pixels, and the height is h pixels. The color that is used is black, unless a different color has been set by calling g.setColor().
- g.fillRect(x,y,w,h) is similar to g.drawRect() except that it fills in the inside of the rectangle instead of drawing an outline.
- g.drawOval(x,y,w,h) draws the outline of an oval. The oval just fits inside the rectangle that would be drawn by g.drawRect(x,y,w,h). To get a circle, use the same values for w and for h.
- g.fillOval(x,y,w,h) is similar to g.drawOval() except that it fills in the inside of the oval instead of drawing an outline.

This is enough information to draw some pictures using Java graphics. To start with something simple, let's say that we want to draw a set of ten parallel lines, something like this:



Let's say that the lines are 200 pixels long and that the distance from each line to the next is 10 pixels, and let's put the start of the first line at the pixel with coordinates (100,50). To draw one line, we just have to call g.drawLine(x1,y1,x2,y2) with appropriate values for the parameters. Now, all the lines start at *x*-coordinate 100, so we can use the constant 100 as the value for x1. Since the lines are 200 pixels long, we can use the constant 300 as the value for x2. The *y*-coordinates of the lines are different, but we can see that both endpoints of a line have the **same** *y*-coordinates, so we can use a single variable as the value for y1 and for y2. Using y as the name of that variable, the command for drawing one of the lines becomes g.drawLine(100,y,300,y). The value of y is 50 for the top line and increases by 10 each time we move down from one line to the next. We just need to make sure that y takes on the correct sequence of values. We can use a for loop that counts from 1 to 10:

```
int y;    // y-coordinate for the line
int i;    // loop control variable
```

```
y = 50;  // y starts at 50 for the first line
for ( i = 1; i <= 10; i++ ) {
    g.drawLine( 100, y, 300, y );
    y = y + 10;  // increase y by 10 before drawing the next line.
}
```

Alternatively, we could use `y` itself as the loop control variable, noting that the value of `y` for the last line is 140:

```
int y;
for ( y = 50; y <= 140; y = y + 50 )
    g.drawLine( 100, y, 300, y );
```

If we wanted to set the color of the lines, we could do that by calling `g.setColor()` **before** drawing them. If we just draw the lines without setting the color, they will be black.

For something a little more complicated, let's draw a large number of randomly colored, randomly positioned, filled circles. Since we only know a few colors, I will randomly select the color to be red, green, or blue. That can be done with a simple switch statement, similar to the ones in Subsection 3.6.4:

```
switch ( (int)(3*Math.random()) ) {
    case 0:
        g.setColor( Color.RED );
        break;
    case 1:
        g.setColor( Color.GREEN );
        break;
    case 2:
        g.setColor( Color.BLUE );
        break;
}
```

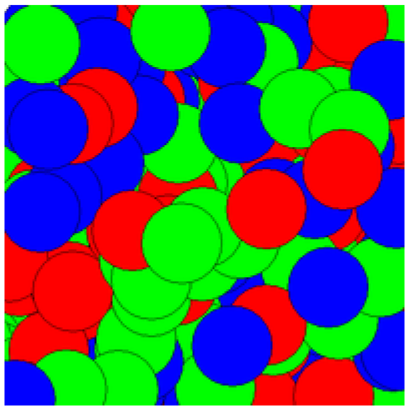
I will choose the center points of the circles at random. Let's say that the width of the drawing area is given by a variable, `width`. Then we want a random value in the range `0` to `width-1` for the horizontal position of the center. Similarly, the vertical position of the center will a random value in the range `0` to `height-1`. That leaves the size of the circle to be determined; I will make the radius of each circle equal to 50 pixels. We can draw the circle with a statement of the form `g.fillOval(x,y,w,h)`. However, in this command, `x` and `y` are not the coordinates of the center of the circle; they are the upper left corner of a rectangle drawn around the circle. To get values for `x` and `y`, we have to move back from the center of the circle by 50 pixels, an amount equal to the radius of the circle. The parameters `w` and `h` give the width and height of the rectangle, which has to be twice the radius, or 100 pixels in this case. Taking all this into account, here is a code segment for drawing a random circle:

```
centerX = (int)(width*Math.random());
centerY = (int)(height*Math.random());
g.fillOval( centerX - 50, centerY - 50, 100, 100 );
```

This code comes after the color-setting code given above. In the end, I found that the picture looks better if I also draw a black outline around each filled circle, so I added this code at the end:

```
          g.setColor( Color.BLACK );
          g.drawOval( centerX - 50, centerY - 50, 100, 100 );
```

Finally, to get a large number of circles, I put all of the above code into a `for` loop that runs for 500 executions. Here's a typical drawing from the program, shown at reduced size:



3.9.2 Drawing in a Program

Now, as you know, you can't just have a bunch of Java code standing by itself. The code has to be inside a subroutine definition that is itself inside a class definition. In fact, for my circle-drawing program, the complete subroutine for drawing the picture looks like this:

```
          public void drawFrame(Graphics g, int frameNumber, int width, int
          height) {

              int centerX;     // The x-coord of the center of a disk.
              int centerY;     // The y-coord of the center of a disk.
              int colorChoice; // Used to select a random color.
              int count;       // Loop control variable for counting disks.

              for (count = 0; count < 500; count++) {

                  colorChoice = (int)(3*Math.random());
                  switch (colorChoice) {
                  case 0:
                      g.setColor(Color.RED);
                      break;
                  case 1:
                      g.setColor(Color.GREEN);
                      break;
                  case 2:
                      g.setColor(Color.BLUE);
                      break;
```

```
            }

            centerX = (int)(width*Math.random());
            centerY = (int)(height*Math.random());

            g.fillOval( centerX - 50, centerY - 50, 100, 100 );
            g.setColor(Color.BLACK);
            g.drawOval( centerX - 50, centerY - 50, 100, 100 );

        }
    }
```

This is the first subroutine definition that you have seen, other than `main()`, but you will learn all about defining subroutines in the next chapter. The first line of the definition makes available certain values that are used in the subroutine: the graphics context `g` and the `width` and `height` of the drawing area. (Ignore `frameNumber` for now.) These values come from outside the subroutine, but the subroutine can use them. The point here is that to draw something, you just have to fill in the inside of the subroutine, just as you write a program by filling in the inside of `main()`.

The subroutine definition still has to go inside a class that defines the program. In this case, the class is named *RandomCircles*, and the complete program is available in the sample source code file *RandomCircles.java*. You can run that program to see the drawing.

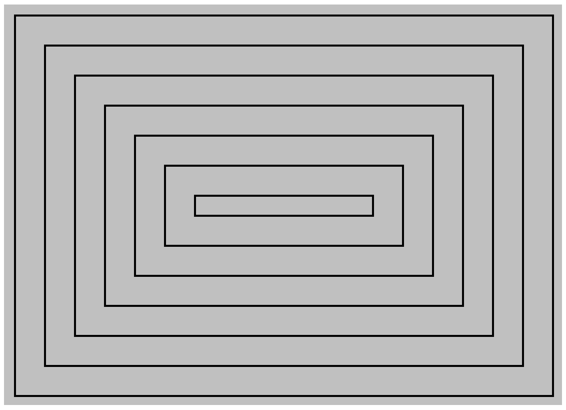
There's a lot in the program that you won't understand. To make your own drawing, all you have to do is erase the inside of the `drawFrame()` routine in the source code and substitute your own drawing code. You don't need to understand the rest. The source code file *SimpleAnimationStarter.java* can be used as a basis for your own first graphics explorations. It's essentially the same as `RandomCircles.java` but with the drawing code omitted. You'll need it to do some of the exercises for this chapter.

(By the way, you might notice that the `main()` subroutine uses the word `static` in its definition, but `drawFrame()` does not. This has to do with the fact that the drawing area in this program is an object, and `drawFrame` is a subroutine in that object. The difference between static and non-static subroutines is important but not something that we need to worry about for the time being. It will become important for us in Chapter 5.)

3.9.3 Animation

The name "SimpleAnimationStarter" should give you a clue that we are looking at the possibility of more than just individual drawings here. A computer animation is simply a sequence of individual pictures, displayed quickly one after the other. If the change from each picture to the next is small, the user will perceive the sequence of images as a continuous animation. Each picture in the animation is called a frame. `SimpleAnimationStarter.java` is configured to draw fifty frames every second, although that can be changed. (In `RandomCircles.java`, it has been changed to one frame every three seconds, so that the program actually draws a new set of random circles every three seconds.) The frames in the animation are numbered 0, 1, 2, 3, ..., and the value of `frameNumber` in the `drawFrame()` subroutine tells you which frame you are drawing. The key to programming the animation is to base what you draw on the `frameNumber`.

As an example of animation, we look at drawing a set of nested rectangles. The rectangles will shrink towards the center of the drawing, giving an illusion of infinite motion. Here's one frame from the animation:



Consider how to draw a picture like this one. The rectangles can be drawn with a `while` loop, which draws the rectangles starting from the one on the outside and moving in. Think about what variables will be needed and how they change from one iteration of the while loop to the next. Each time through the loop, the rectangle that is drawn is smaller than the previous one and is moved down and over a bit. The difference between two rectangles is in their size and in the coordinates of the upper left corner. We need a variable to represent the `size`. The x and y-coordinates are the same, and they can be represented by the same variable. I call that variable `inset`, since it is the amount by which the edges of the rectangle are inset from the edges of the drawing area. The `size` decreases from one rectangle to the next, while the `inset` increases. The while loop ends when `size` becomes less than or equal to zero. In general outline, the algorithm for drawing one frame is

```
Set the drawing color to light gray  (using the g.setColor
subroutine)
Fill in the entire picture (using the g.fillRect subroutine)
Set the drawing color to black
Set the amount of inset for the first rectangle
Set the rectangle width and height for the first rectangle
while the width and height are both greater than zero:
    draw a rectangle (using the g.drawRect subroutine)
    increase the inset (to move the next rectangle over and down)
    decrease width the height (the make the next rectangle smaller)
```

In my program, each rectangle is 15 pixels away from the rectangle that surrounds it, so the `inset` is increased by 15 each time through the `while` loop. The rectangle shrinks by 15 pixels on the left **and** by 15 pixels on the right, so the width of the rectangle shrinks by **30** before drawing the next rectangle. The height also shrinks by 30 pixels each time through the loop.

The pseudocode is then easy to translate into Java, except that we need to know what initial values to use for the inset, width, and height of the first rectangle. To figure that out, we have to think about the fact that the picture is animated, so that what we draw will depend in some way on the frame number. From one frame to the next frame of the animation, the top-left corner of the outer rectangle moves over and down; that is, the `inset` for the outer rectangle increase from one frame to the next. We can make this happen by setting the inset for frame number 0 to 0, the inset for frame number 1 to 1, and so on. But that can't go on forever, or eventually all the rectangles would disappear. In fact, when the animation gets to frame 15, a new rectangle should appear at the outside of the drawing area -- but it's not really a "new rectangle," it's just that the `inset` for the outer rectangle goes back to zero. So, as the animation proceeds, the inset should go through the sequence of values 0, 1, 2, ..., 14 over and over. We can accomplish that very easily by setting

```
inset = frameNumber % 15;
```

Finally, note that the first rectangle fills the drawing area except for a border of size `inset` around the outside of the rectangle. This means that the the width of the rectangle is the width of the drawing area minus two times the inset, and similarly for the height. Here, then is the `drawFrame()` subroutine for the moving rectangle example:

```
public void drawFrame(Graphics g, int frameNumber, int width, int
height) {

    int inset; // Gap between edges of drawing area and the outer
rectangle.

    int rectWidth, rectHeight;   // The size of one of the
rectangles.

    g.setColor(Color.LIGHT_GRAY);
    g.fillRect(0,0,width,height);  // Fill drawing area with light
gray.

    g.setColor(Color.BLACK);  // Draw the rectangles in black.

    inset = frameNumber % 15;  // inset for the outer rectangle

    rectWidth = width - 2*inset;   // drawing area width minus two
insets
    rectHeight = height - 2*inset; // drawing area height minus two
insets

    while (rectWidth >= 0 && rectHeight >= 0) {
        g.drawRect(inset, inset, rectWidth, rectHeight);
        inset += 15;       // rectangles are 15 pixels apart
        rectWidth -= 30;
        rectHeight -= 30;
    }
}
```