The science of computing: Java supplement

first edition

by Carl Burch

Copyright ©2004, by Carl Burch. This publication may be redistributed, in part or in whole, provided that this page is included. A complete version, and additional resources, are available on the Web at

`http://www.cburch.com/socs/`

Contents

Chapter J1

Programming overview

In this book, we examine the basics of using high-level programming languages. We will study this using one particular language — Java — but the point is not to master Java (which would takes several courses). Instead, we will use the language primarily as a tool for learning the fundamentals of programming. By the end of this course, you should be familiar with writing simple procedural programs using Java.

J1.1 About Java

In the early 1990's, a computer company called Sun Microsystems began a project to develop a platform for *embedded systems* — that is, they wanted to build software for devices that have a special-purpose computer inside them, like a microwave, a telephone, or a car. The people assigned to the project began by asking what sort of language they would want to use for such a system. They decided that none of the alternatives were suitable, and so they developed Java. In the wake of the effort to develop a new language, their embedded systems mission fell by the wayside.

When Sun released Java in 1996, the company had enough marketing sense to throw in features to support the hottest technology of the day, the World Wide Web, via the concept of an *applet*. That idea was only a moderate success — except that the association with the Web gave the language all the hype that it needed to become widely known.

In developing Java, its designers drew on a long history of programming languages. Historically, Java derives primarily from two languages: C and Smalltalk. Designed in the 1970's for developing an operating system called UNIX, C is a small, efficient language. But it provides few *abstractions* — a C programmer thinks in terms of how the computer actually computes, which makes it difficult to build large programs to do something useful. It became very popular in the industrial world, however, due largely to the increasing popularity of UNIX. In fact, the primary reason Java builds on C is so that the vast body of C programmers would accept Java.

Smalltalk, also designed in the 1970's for an operating system, is an elegant, inefficient language. In contrast to C, Smalltalk provides many abstractions; a Smalltalk programmer rarely considers or even understands how the program works within the computer. Although Smalltalk never achieved the popularity of C, the basic abstraction it introduced — the *object* — makes the language historically very important. In the 1980's, programmers began to appreciate the value of using this abstraction.

Java's designers sought to combine the elegance of Smalltalk with the efficiency of C. Their design has proven a sound response to a long-established need, which has enabled it to gain a foothold in industrial programming. It's difficult to quantify how widely used a language is, as much software is developed by individual companies for their own internal use. But Java is likely among the top five most-used languages in large-scale systems today.

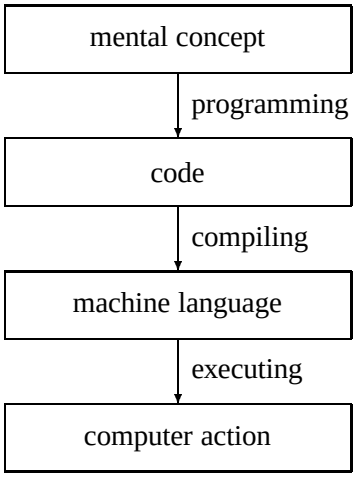


Figure J1.1: The programming process.

It is easier to quantify Java's success in education. Most university computer science curricula now use Java as their primary language. This future workforce trained in Java virtually guarantees its endurance in the workplace.

J1.2 The programming process

The pipeline from idea to action consists of three phases: *programming*, *compiling*, and *executing*. Figure J1.1 illustrates the process.

In **programming**, you as the programmer translate your mental concept of how the computer should behave to corresponding **code** written in a **programming language** such as C, Java, or Ada. A programming language is a compromise between what humans find natural for expressing procedure and what computers can interpret.

Computers, as built, cannot actually understand programming languages; they are built to understand a much more primitive language called **machine language**. (Different types of machines have different machine languages — the machine language for a PC is totally different from the machine language for a Macintosh.) In **compiling**, the computer runs a program to translate the programmer-written code to machine language. This program is called a **compiler**, and it is the primary programming tool for the compile phase.

In the final phase, the computer **executes** the machine language that the compiler produced. At this point the machine finally does the job that the programmer originally conceived... at least if all the phases proceed flawlessly.

During these phases, errors crop up due to programmer errors. For all these errors, the solution is for the programmer to discover where the code is wrong, to fix the code, and to repeat the compile and execution phases.

Programming: A **logic error** arises when the programmer has written code that compiles and executes normally, but the machine's behavior does not correspond with the original concept. For example, if the user chooses Save from a program's menu, and the computer does nothing, this is a logic error in the program.

Compiling: A **compile-time error** is an error that prevents the compiler from interpreting the program. If

```
 1 import socs.*;
 2
 3 public class ShowWindow {
 4     public static void main(String[] args) {
 5         // shows a window
 6         RobotWindow win;
 7         win = new RobotWindow();
 8         win.show();
 9     }
10 }
```

Figure J1.2: The `ShowWindow` program.

Figure J1.3: Running `ShowWindow`.

the code contains a mistyped name, for example, then this causes a compile-time error. The result of a compile-time error is that the compiler refuses to compile the program, instead issuing a description of what is wrong with the program. This is the easiest type of error to fix, since the compiler usually points directly to the problem.

Executing: A **run-time error** occurs during execution; one example of a common run-time error is when the machine code instructs the computer to divide a number by zero. Often such errors *crash* the program; that is, execution stops abruptly.

J1.3 A simple program

To begin our study of Java, we'll look at the program of Figure J1.2.* When executed on a computer, this program brings up an empty window on the screen; see Figure J1.3. It's not too impressive, but we have to start somewhere.

For the next few chapters, much of this small program you should mindlessly copy into every program you write without thinking about it. That's scant consolation to the curious, however — and we don't want to squelch curiosity. So let's do a quick overview to see what this program says.

Line 1: This tells the Java compiler that this program is going to use pre-written code defined somewhere else — specifically, in the `socs` library.

*The line numbers are for reference only; they aren't part of the program.

Line 2: Java ignores blank lines. They make the program a little easier to read for humans. The program works just as well without line 2, but it would be more confusing for a programmer to understand.

Line 3: This tells the Java compiler that we are defining a program named `ShowWindow`. The program goes between the left brace at the end of line 3 and the corresponding right brace in line 10.

Line 4: This tells the Java compiler that we are defining what the program is to do when asked to run. This is something else that you should just copy — we'll get to it later on. Everything between the left brace on line 4 and the corresponding right brace on line 9 contains what the program should do when it runs.

Line 5: This is a **comment** — that is, it is part of the program that the computer simply ignores. This is useful for describing what the program is trying to do to a human trying to understand it. You can indicate a comment in Java using a double slash (`/`); the comment continues to the end of the line.

Lines 6–8: This specifes what the program actually does. What we can put in place of lines 6 to 8 is what we're going to study for the next several chapters. We'll look at these particular three lines in more detail in the next chapter.

The body of a program is separated into **statements** — a single piece of program telling the computer to do one particular thing. When the computer runs the program, it executes each of these statements in sequence. In this program, there are three statements, separated by semicolons.

Of this program, then, only lines 5 to 8 are important for now. The other pieces we'll cover more deeply later. Until then, it's not necessary to understand lines 1 to 4 well.

J1.4 Programming style

When you write a program, you're really writing for two audiences. The obvious one is the computer that will be executing the program. But just as important is the human audience — somebody who needs to evaluate your program's correctness or perhaps even to modify it to incorporate new features. You should think of this person as coming in with an understanding of Java and how your program runs on the computer, but who has no real idea of how your program accomplishes. That person may be your teacher, your boss, your coworker, or it may even be you after you've been doing other things for a few months and then decide to go back to that program you were working on before.

For these people, readability is important. They need help to understand easily how the program is structured to do what it does. In recognition of this need, Java includes three major aspects to help: *white space*, *naming*, and *comments*.

White space The term ***white space*** refers to characters of a file that you can't see — spaces, tabs, and line breaks. In Java, white space does not matter. In fact, Java regards the following program as being identical to that of Figure J1.2.

```
import socs.*;class ShowWindow{public static void main(String
[]args){RobotWindow win;win=new RobotWindow();win.show();}}
```

White space is useful to humans, however, in indicating the structure of a program. It's helpful for us in reading a program if it systematically uses white space to reflect its structure.

We'll follow one particular approach for using white space faithfully in this book.

- We put one "concept" per line. Typically, the concept will be a single statement.

- We insert a blank line between largely independent pieces, akin to how authors insert paragraph breaks in English text.
- We'll indent everything within a set of braces four spaces. The `ShowWindow` program demonstrates this: Lines 4 to 9 are all within the set of braces begun at the end of line 3, so they're all indented four spaces. Moreover, lines 5 to 8 are within the set of braces begun at the end of line 4, so they're indented an additional four spaces.

These are conventions followed by most Java programmers, based on years of experience writing millions of lines of code. You should get into the habit of using these conventions too. (If you have programmed a lot before and are already in the habit of another widely used convention, that's fine too. But don't go off and develop your own — it will confuse others and cause more headaches than it's worth in the long run.)

Naming The second element of good programming style is good names. When you program, you have lots of opportunities to assign names to things. Choosing representative names is worth some consideration. For example, in the `ShowWindow` program, I arbitrarily chose the name `win` that appears in lines 6, 7, and 8. I could have chosen virtually anything.

```
6         RobotWindow i_like_java;
7         i_like_java = new RobotWindow();
8         i_like_java.show();
```

Arbitrary names quickly become unwieldy, however: With lots of arbitrary names, it's hard to remember them all. It's better to choose a good name that indicates the purpose of the variable and that you can remember. I chose `win`, short for *window*. The name `window` would be better, but I didn't want to keep retyping it. The name `w`, however, is too brief.

Comments Finally, a good program will have comments to indicate what the program is doing. Finding a good balance between comments and silence is important. Beginners, given the edict to insert comments into their programs, often go through each line and explain what that line does. This is a silly exercise — it's reasonable to assume that anybody reading your program knows Java well enough to read any individual line. But they won't necessarily be able to discern the *purpose* of that action; indicating the purpose is where comments are useful.

A good approach to commenting is to break long programs into blocks of up to 10 lines. Use a blank line to separate the blocks, and include a comment at the top of each block explaining what that block does. Figure J7.1 (page J45) contains a program that illustrates this technique.

If being able to run a program on a computer were the only important thing, Java wouldn't include white space, naming, or comments. But human readers are important too, and so Java includes these features. A certain sign of a novice programmer is somebody who doesn't use these features to advantage. It's best to get into the habit of using them as soon as possible.

Exercise J1.1: Using a compiler, compile and run the `ShowWindow` program of Figure J1.2. Try removing the semicolon on line 1 and compiling again; the compiler should show you a compile-time error.

Exercise J1.2: (Solution, J57) Take the following program and insert white space according to the system outlined in Section J1.4.

```
import csbsju.cs150.*;public class DrawTriangle{public static
void main(String[]args){RobotWindow win;win=new RobotWindow()
;win.show();Robot rob;rob=new Robot(win,50,150);rob.move(100)
;rob.turn(120);rob.move(100);rob.turn(120);rob.move(100);rob.
switchOff();}}
```

Chapter J2

Variables and objects

J2.1 The `ShowWindow` program

Now we'll look at the body of our `ShowWindow` program (lines 6 to 8) in detail.

```
6         RobotWindow win;
7         win = new RobotWindow();
8         win.show();
```

J2.1.1 Variable declaration

The first line that the computer executes when the program runs is line 6, which tells the computer that our program uses a variable.

```
RobotWindow win;
```

A **variable** is a name for a value. In Java, each variable has an associated **type**, which indicates what sort of value the variable will be naming. Line 6 declares a variable named `win`, of type `RobotWindow`; we call this a **variable declaration**.

The Java syntax for declaring a variable is to give the type first, followed by the variable's name, followed by a semicolon. The semicolon is there to indicate to the compiler that the statement is over.

⟨***typeName***⟩ ⟨***variableName***⟩;

`RobotWindow` is a **class** — it's a type whose values will be what Java calls *objects*, abstract entities representing a complex structure, like a window. This particular class is defined in the `socs` library we imported on line 1. As programmers using the class, we don't need to understand how it works. We just need to understand how to use it. And we'll get to that in a moment.

A detail worth remembering	Name your variables with care. It is much easier to write and understand a program when its variables' names indicate their purpose. In Java, you can choose virtually anything you want as a name for a variable. Java requires that the name contain only letters, digits, and underscore characters ('_'). And it can't begin with a digit. Java reserves a few dozen words for special uses (like `import` and `class`); but other than that, any name is good. Letter case is significant, so for example `win` and `Win` are different variables.

So line 6 creates the name `win` for a `RobotWindow` object. This does *not* mean that `win` is a name for a `RobotWindow` yet. It's just a name right now that can potentially be a name for a particular `RobotWindow` object, just as you understand *Ariel* is a woman's name, but you won't understand who I mean by it until I introduce her to you. Internally, the computer allocates some memory for remembering

to what object `win` is referring. But it hasn't been told what object `win` should name yet, so the computer will remember the variable `win` as referring to an undefined object for the moment.

J2.1.2 Variable assignment

To assign a particular value to the name, we need to use an **assignment statement**, as in line 7 of the program.

```
win = new RobotWindow();
```

In general, an assignment statement looks like the following.

⟨***variableName***⟩ `=` ⟨***valueToGiveIt***⟩`;`

An assignment statement consists of two parts, separated by an equal sign ('`=`'). On the left side of the equal sign is the variable to which you want a value assigned. And on the right side is the value you want to assign to it. Again, a semicolon must terminate the statement to enable the compiler to find the statement's end easily. In line 7, we create a new `RobotWindow` object on the right side, and on the left side we say that we want `win` to be a name for this object.

A detail worth remembering	Don't let the equal sign '`=`' confuse you: We are not discussing algebraic equality here. The assignment statement actually *changes* the value of the variable mentioned. That is, we are reassigning the name to a different value, not comparing the variable's value to anything.

A detail worth remembering	Note the order of the assignment statement: The thing to be changed goes on the *left* side. Beginners often want to swap the two sides, but this leads to trouble.
	`new RobotWindow() = win; // WRONG!!!`
	The impulse is natural: The computer evaluates the expression before it assigns its value to the variable, and it would make sense to write this in left-to-right order. Besides, in algebra, equality is commutative. But assignment is not equality, so order is significant. And Java chose to have the variable name first. (It chooses this because the variable being assigned is the most important part of the statement, and it's easier to find the first part of the statement.)

The way you create an object in Java is a little peculiar, but it turns out to be convenient. The `RobotWindow` class defines a **constructor method**, which allows you as the user to create new objects of that type. To use a constructor method, you use the `new` keyword, followed by the type of object to be created, followed by a set of parentheses and a semicolon.

Incidentally, Java allows you to combine a variable declaration with a variable assignment in the same statement. We could combine lines 6 and 7 thus.

```
RobotWindow win = new RobotWindow();
```

We'll keep the two separate for the next few chapters, however, to emphasize that these are two separate concepts. But you're welcome to use this handy space-saver in your own programs.

J2.1.3 Instance methods

Line 8 sends a message to the object to perform a specific task. The `RobotWindow` class defines several behaviors that a `RobotWindow` object can perform; each of these is called an **instance method**. ***Instance*** is

a synonym for *object* in Java; it's called an *instance method*, since it's something you would tell a particular object to perform.

In line 8, we tell the object named by `win` to perform its `show` method. In the `RobotWindow` class, this method is defined to bring a window up on the screen. We shouldn't worry about exactly how it accomplishes this feat.

```
win.show();
```

To tell an object to do something, you give the name of the object first, followed by a period, followed by the name of the instance method you want to use. After the instance method name is a set of parentheses and a semicolon.

Each class defines a set of instance methods, specifying exactly what behaviors are defined for objects. There's nothing particularly sacred about the word `show` — it's just what the `RobotWindow` class happened to name one of its instance methods. Other classes will have other instance methods, with different names.

J2.2 Classes and methods

J2.2.1 Class documentation

There are *many* classes in Java, more than anyone could possibly remember. To use them, you need documentation that lists the methods of each class. To use Java effectively, therefore, you need to understand how to read the documentation.

As an example, let's look at how the documentation for the `RobotWindow` methods that we've seen so far would look. Recall that the two methods from the `RobotWindow` class that we've seen are its constructor method and its `show` instance method. Here's documentation for them.

`RobotWindow()`
: (Constructor method) Constructs an object to represent a window on the screen, 200 pixels wide and 200 pixels tall.

`void show()`
: Displays this window on the screen.

For now, ignore the word `void` written before `show` above; we'll get to it in the next chapter.

Recall that a constructor method is used for creating a new object of that type, using a `new` keyword. A constructor method is always named the same as the class; thus, in the `RobotWindow` class documentation, the constructor method is listed as `RobotWindow()`.

Now let's look at another class defined in the `socs` library, the `Robot` class. If you were to look at the documentation for this class, you'd see something like the following, listing all the methods of the `Robot` class.

`Robot(RobotWindow world, double x, double y)`
: (Constructor method) Constructs a robot object, located in the `world` window at coordinates $(\texttt{x}, \texttt{y})$, facing east.

`void move(double dist)`
: Moves this robot forward `dist` pixels in its current direction, tracing a line along the path.

`void turn(double angle)`
: Turns this robot `angle` degrees counterclockwise.

`void switchOff()`
: Removes this robot from its window.

As you should be able to determine from this documentation, this class has a constructor method and three different instance methods, named `move`, `turn`, and `switchOff`.

Appendix JA, by the way, contains documentation for all the classes and methods described in this book. We'll introduce each item in the text as it becomes needed, but the appendix is a handy reference.

J2.2.2 Parameters

One thing that's new with the methods of the `Robot` class is that many of them take a **parameter**. A parameter is some additional information given to a method to further specify what to do. For example, when we tell a robot to go forward (using its `move` method), we'll also tell it how far it should go forward. That the `move` method requires such a parameter is cryptically specified in the documentation when it says the following.

```
void move(double dist)
```

Inside the parentheses you see "`double dist`." This syntax is akin to the variable declaration syntax as in line 6 of `ShowWindow` ("`RobotWindow win;`"): You have a type, followed by a name. In this case, "`double dist`" indicates that, when you use this method, you need to include a parameter. Specifically, it says that the parameter, named `dist`, needs to be of the `double` type.

The name `dist` is really irrelevant — it's only there so that the English description of the method can indicate what the parameter means. The type `double` is important, though. The `double` type is Java's type for numbers.* Thus, this documentation says that the `move` method requires a number as a parameter.

When we tell a `Robot` to move, therefore, we will place a number in the parentheses after the method name. For example, if `robin` is a name for a `Robot` object, we could tell `robin` to move forward 100 pixels as follows.

```
robin.move(100);
```

The `Robot` constructor method is more complicated: It takes *three* parameters. The first parameter, of the `RobotWindow` type, describes which window the robot should live in. The second and third parameters, of `double` type, indicate the robot's initial x- and y-coordinates. As the documentation for the constructor method states, when a robot is newly created, it is facing east.

To pass parameters into a constructor method, you include them in the parentheses following the class name (which itself follows the `new` keyword).

```
robin = new Robot(win, 50, 150);
```

This would create a `Robot` object in whatever window `win` names, at coordinates $(50, 150)$. (As the documentation for the constructor method states, when a robot is newly created, it is facing east.) And then it would assign `robin` to be a name for this `Robot` object.

Note that the following is illegal.

```
robin = new Robot();   // Illegal!
```

This is because the `Robot` class doesn't include a constructor method that takes no parameters. The *only* way we have for creating a `Robot` object is to use the constructor method with three parameters.

```
 1 import socs.*;
 2
 3 public class DrawTriangle {
 4     public static void main(String[] args) {
 5         RobotWindow win;
 6         win = new RobotWindow();
 7         win.show();
 8
 9         Robot rob;
10         rob = new Robot(win, 50, 150);
11         rob.move(100);
12         rob.turn(120);
13         rob.move(100);
14         rob.turn(120);
15         rob.move(100);
16         rob.switchOff();
17     }
18 }
```

Figure J2.1: The `DrawTriangle` program.

J2.2.3 Drawing a triangle

Now let's look at an example program using the `Robot` class. Figure J2.1 contains a program to draw a window containing a triangle. It shares quite a bit from our earlier `ShowWindow` program: The only change in lines 1 to 7 is that the word `ShowWindow` is now `DrawTriangle` in line 3, representing the fact that this is a different program. But, after line 7, the program begins working with a robot, and beyond that it is all new.

Let's trace what happens when the computer runs this program.

Line 5: Declares a variable `win` that can potentially be a name for a `RobotWindow` object.

Line 6: Creates a new `RobotWindow` object using the constructor method of `RobotWindow` and assigns `win` to be a name for this object.

Line 7: Tells the `win` object to execute its `show` instance method.

Line 9: Declares a variable `rob` that can potentially be a name for a `Robot` object.

Line 10: Creates a new `Robot` object using the constructor method of `Robot` and assigns `rob` to be a name for this object.

Graphical coordinates on computer systems start with $(0, 0)$ at the upper left corner and increase as you move rightward and *downward*. This contrasts with the typical mathematical systems, where the y-coordinate increases as you move up. Thus, when we place the robot at $(50, 150)$ in a 200×200 window, we are placing it in the lower left quadrant of the window. See Figure J2.2(a).

Line 11: Tells `rob` to execute its `move` instance method, with 100 as a parameter. Since the robot is facing east, this will move the robot from $(50, 150)$ to $(150, 150)$. The robot traces a line, which will form the base of the triangle. See Figure J2.2(b).

*Java's designers chose `double` for historical reasons. Its predecessor, C, used `double` to indicate a number that is stored in twice as much memory as regular numbers. The added memory allows the computer to remember the number more precisely (with 15 digits of precision, instead of merely 6) and to remember larger numbers. Over the years, as memory became cheaper and hence more voluminous, programmers began using `double` almost exclusively, since there was little reason to risk erroneous results in order to skimp on memory.

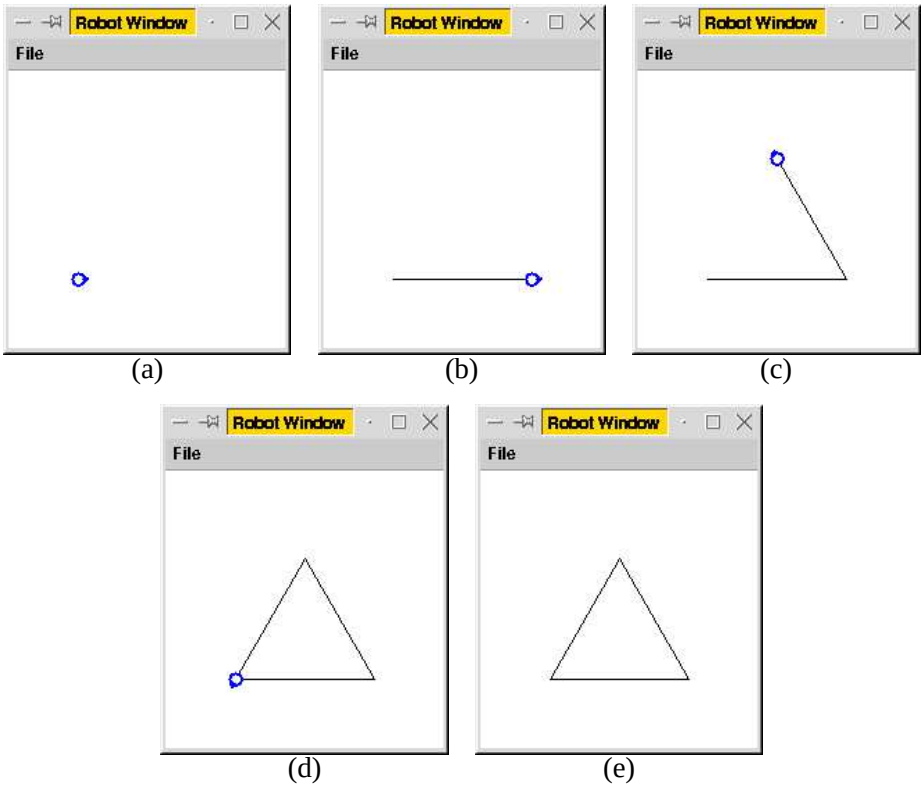


Figure J2.2: Running `DrawTriangle`.

```
 1 import socs.*;
 2
 3 public class Race {
 4     public static void main(String[] args) {
 5         RobotWindow win;
 6         win = new RobotWindow();
 7         win.show();
 8
 9         Robot upper;
10         Robot cur;
11         cur = new Robot(win, 10, 60);
12         upper = cur;
13         cur.move(90);
14         cur = new Robot(win, 10, 140);
15         cur.move(90);
16         cur = upper;
17         cur.move(90);
18     }
19 }
```

Figure J2.3: The `Race` program.

Line 12: Tells `rob` to execute its `turn` instance method, with 120 as a parameter. As the documentation for the `Robot` class indicates, its `turn` method takes a number as a parameter, specifying how far the robot is to turn counterclockwise. So, while the robot was facing at 3 o'clock before, now it is facing 120° counterclockwise of that, at 11 o'clock.

Line 13: Tells `rob` to `move` 100 pixels forward. This draws the right side of the triangle. See Figure J2.2(c).

Line 14: Tells `rob` to `turn` 120 pixels counterclockwise. The robot is now facing at 7 o'clock.

Line 15: Tells `rob` to `move` 100 pixels forward. This draws the left side of the triangle. See Figure J2.2(d).

Line 16: Tells `rob` to `switchOff`. As the `Robot` documentation indicates, this removes `rob` from the window. The path the robot traced, however, remains. Thus we have a triangle in the window. See Figure J2.2(e).

J2.3 Multiple names for the same object

Figure J2.3 contains a program that's a little confusing. Though I'd never write this program in practice, understanding how it works illustrates a principle that's important to understand in Java: Namely, it illustrates how Java works when a program assigns the same object to multiple variables.

Lines 5–7: Creates a `RobotWindow` object assigned to `win` and tells it to show itself.

Lines 9–10: Creates two `Robot` variables, named `upper` and `cur`.

Line 11: Creates a `Robot` object located at $(10, 60)$ in `win`, named `cur`. We'll also call this Robot A for the purposes of our walk-through.

Line 12: Assigns `cur` to `upper`; since `upper` currently refers to Robot A, so will `upper`.

```
import socs.*;

class RobotMystery {
    public static void main(String[] args) {
        RobotWindow win;
        win = new RobotWindow();
        win.show();

        Robot robin;
        robin = new Robot(win, 50, 50);
        robin.turn(-90);
        robin.move(30);
        robin.turn(90);
        robin.move(40);
        robin.turn(90);
        robin.move(30);
        robin.switchOff();

        robin = new Robot(win, 70, 30);
        robin.turn(-45);
        robin.move(30);
        robin.switchOff();

        robin = new Robot(win, 70, 30);
        robin.turn(-135);
        robin.move(30);
        robin.switchOff();
    }
}
```

Figure J2.4: The `RobotMystery` program.

Line 13: Tells `cur` (which is Robot A) to move forward 90 pixels. Now Robot A is at $(100, 60)$.

Line 14: Reassigns `cur` to be a new `Robot` object located at $(10, 140)$ in `win`, which we'll name Robot B for the purposes of this walk-through. Since line 14 does not change `upper`, and `upper` was referring to Robot A from before, `upper` still refers to Robot A.

Line 15: Tells `cur` (which is now Robot B) to move forward 90 pixels. Now Robot B is at $(100, 140)$.

Line 16: Assigns `upper` to `cur`; since `cur` currently refers to Robot A, so will `cur`.

Line 17: Tells `cur` (which is now Robot A) to move forward 90 pixels. Now Robot A is at $(190, 60)$.

Thus, at the end of the program, there are two robots in the window. One (Robot A) has moved from $(10, 60)$ to $(190, 60)$. And the other (Robot B) has moved from $(10, 60)$ to $(100, 60)$.

Exercise J2.1: (Solution, J57) Without running the program on a computer, draw a picture of what the `RobotMystery` program of Figure J2.4 would draw on the screen.

Exercise J2.2: Insert code into the `Race` program of Figure J2.3, so that both robots turn off after Robot A makes its last move (line 17). You may only add to it; do not delete or modify any of the code already in the program. To accomplish this, you will need to add a new variable to remember Robot B, since both `upper` and `cur` refer to Robot A after line 16.

Exercise J2.3: Write a program to draw a letter *E*, as in Figure J2.5. In the screen shot, the letter is 120 pixels wide and 140 pixels tall.

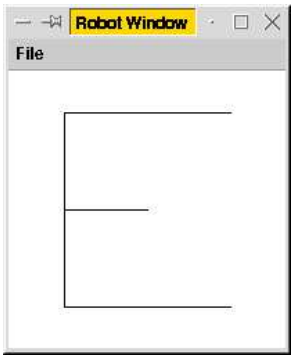


Figure J2.5: Running program of Exercise J2.3.

Chapter J3

Using numbers

J3.1 Numeric types

We've glanced at the `double` type in Java, which we can use to represent a number. Using this type, we can, if we want, make a variable to refer to a number.

```
double scale;
```

This line creates a variable named `scale`, of type `double`. We can assign a number to the variable using an assignment statement.

```
scale = 2.4;
```

Notice that we don't have to use `new` here; we use `new` only in conjunction with constructor methods, and the numeric types don't really have constructor methods.

Java also has an `int` type for representing integer values. (An integer is a number with no fractional part, like 0 or -3, but not 1.4.)

```
int i;
i = 5;
```

You cannot make an `int` variable refer to a `double` value — even if the value has no fractional part. The following is illegal.

```
i = 5.0;       // Illegal!!
```

This is illegal, since 5.0 is a `double` value (due to the decimal point), and `i` can name only `int` values.

Java will happily allow a `double` variable to be assigned an `int` value, however, on the grounds that `int` is more restricted than `double`; any `int` value can be converted to a legitimate `double` value.

```
scale = 2;     // Ok
```

You may be wondering: Why would you ever use `int` instead of `double`? Why limit yourself? There are three reasons for this. First, computers use scientific notation with limited precision to remember `double` values. While this allows the computer to remember a wider range of numbers, it can only approximate most of them. An `int` variable always represents its value exactly. Second, computers work with `double`s somewhat more slowly. And, finally, it rarely causes much trouble: To experienced programmers, the distinction is second nature. In practice, most programs use very few `double` values.

J3.2 Return values

We've already seen that we can send a value to a method via a parameter. But, often, a method needs to provide some sort of response; for this, it can use a **return value**.

For example, the `RobotWindow` class includes the following two instance methods in addition to the `show` method we saw earlier.

`double requestDouble()`
: Shows the user a dialog box prompting for a number, and returns this number.

`int requestInt()`
: Shows the user a dialog box prompting for an integer, and returns this integer.

The type written before the method name in the method descriptor (`double` for `requestDouble` and `int` for `requestInt`) specifies what type of value the method will return. Although a method can take multiple values as parameters, a method can return only one value.

We can use an assignment statement to assign a name to the value that a method returns.

```
scale = win.requestDouble();
```

Here, we've assigned the name `scale` to the value returned when we tell `win` to request a `double`. Since `requestDouble` returns what the user types into a dialog box, this statement assigns `scale` to be a name for whatever it is the user types.

If you look at the documentation for the `show` method of `RobotWindow`, you'll see that the method descriptor for its `show` method indicates that the method returns a value of type `void`.

```
void show()
```

The word `void` as a return type is special: It indicates that the method never returns anything.

J3.3 Arithmetic

Though they are valid types, neither `double` nor `int` are classes defined in some library: They are **primitive types**. Java has exactly eight primitive types, of which `double` and `int` are two.* (We won't see any of the other six in this book.)

As primitive values, we cannot apply instance methods to them as we can to object values. Instead, to work with primitive values, we use **operators**, special symbols that can be used to compute a value. Java includes many operators, including several arithmetic operators for numeric values.

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus (remainder)

We can combine these as we like. Java will follow the algebraic order of operations: Multiplication and division (and modulus) have priority over addition and subtraction. And, within the same level of precedence, the computation proceeds left to right. Suppose I write something like the following.

```
scale = 100 - length / 2.0;
```

*Java rigorously follows a naming convention to distinguish primitive types and classes: Primitive type names are all lower case (like `double`), while classes conventionally begin with a capital letter (`RobotWindow`).

The first thing that happens is that `length` is divided by 2. And then this result is subtracted from 100. (And `scale` is assigned to this value. The value of `length` remains unchanged by this statement.) If we really wanted to do the subtraction first, then we would need to use parentheses.

```
scale = (100 - length) / 2.0;
```

When you operate on two integers, Java will produce an integer result. And if either side is a `double`, Java will produce a `double` result. This works quite smoothly in practice, except for one thing: When you divide two integers, of course the true result is not necessarily an integer. In order to get an integer result, Java will simply ignore any remainder. Thus, "`8 / 3`" has the value 2. (Its true value is $2.666\ldots$, but the remainder of 2 is ignored since both `8` and `3` are `int` values.)

A detail worth remembering	This dropping of the remainder is guaranteed to bite you some day! It will happen even if the result is converted into a `double`. `scale = 1 / 2;  // Sets scale to 0.0` It looks like this line assigns 0.5 to `scale`, but in fact it assigns 0.0 to it, since both sides of the division are integers. The solution is simple: Convert either side of the division to a `double` value. One easy way to convert an `int` value into a `double` is to multiply it by `1.0`.

The modulus operator (`%`) works only with two integers, and its result is the remainder. For example, "`11 % 3`" would yield the integer 2, since the remainder when you divide 11 by 3 is 2.

J3.4 An extended example

Figure J3.1 contains a program that asks the user for a number (Figure J3.2(a)) and then draws a square exactly that big in the center of the window (Figures J3.2(b) and J3.2(c)).

When run, the computer begins at line 5 and works through the lines one by one.

Lines 5–7: Creates a `RobotWindow` object named `win` and tells it to show itself to the user.

Lines 9–10: Creates a `double` variable called `length`. This variable will represent the length of each side of the square to be drawn. Then it asks the user for an integer and assigns the value the user types to this `length` variable. In Figure J3.2(a), the user types 60, so in the example of Figure J3.2, `length` will represent the number 60.

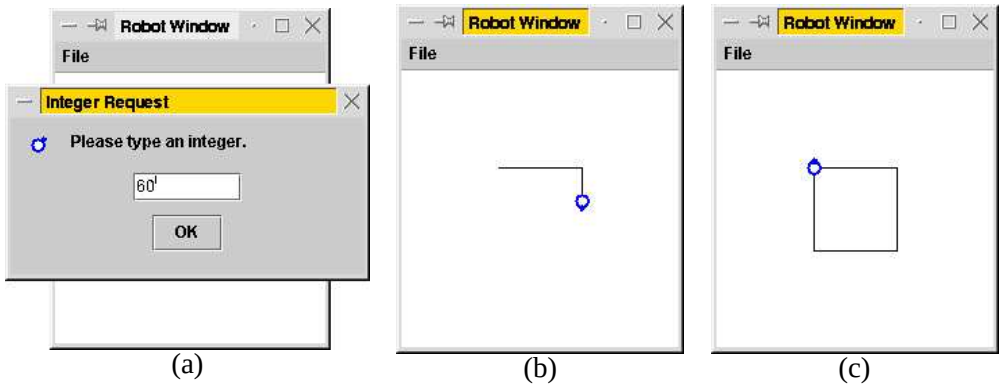
Lines 11–12: Creates another variable called `start` that will represent the x-coordinate of the top left corner of the square. Its value is computed in line 12 using an arithmetic expression. Since the window is 200 pixels wide, 100 is halfway between the left and right sides; we want half of the side length to extend to the left of this, so we want to subtract `length`/2 from 100. Thus, this is the expression you see in line 12, whose value is assigned to `start`. In our example, `start` would be assigned the value 70.

Lines 14–15: Creates another variable `rob` for referring to a `Robot` object. In line 15, a `Robot` object is created and assigned to `rob`. The robot begins in the window named `win`, with its x- and y-coordinates being `start`. Thus, this robot would be placed at $(70, 70)$ facing east.

Line 16: Moves `rob` forward `length` pixels, to draw the top side of the square. Notice that this line uses a variable for parameter values. We're saying that we want the value of the variable `length` to be

```
 1 import socs.*;
 2
 3 public class DrawSquare {
 4     public static void main(String[] args) {
 5         RobotWindow win;
 6         win = new RobotWindow();
 7         win.show();
 8
 9         double length;
10         length = win.requestInt();
11         double start;
12         start = 100 - length / 2;
13
14         Robot rob;
15         rob = new Robot(win, start, start);
16         rob.move(length);
17         rob.turn(-90);
18         rob.move(length);
19         rob.turn(-90);
20         rob.move(length);
21         rob.turn(-90);
22         rob.move(length);
23         rob.switchOff();
24     }
25 }
```

Figure J3.1: The `DrawSquare` program.



(a) (b) (c)

Figure J3.2: Running `DrawSquare`

passed as the parameter value to `move`. In our example, the robot would go forward (east) 60 pixels, ending at $(130, 70)$.

Lines 17–18: Turns `rob` 90 degrees clockwise, now facing south, and then moves `rob` forward `length` pixels again to draw the right side of the square. In Figure J3.2(b), the robot is halfway toward its destination. The robot reaches $(130, 130)$ facing south.

Lines 19–22: Draws the bottom and left sides of the square. See Figure J3.2(c).

Line 23: Removes `rob` from the window, leaving only the square remaining.

In line 16, we passed a variable's value as a parameter, instead of simply a number. More generally, we could write any expression for a parameter, and the computer will compute its value before calling the method. When we create the robot in line 15, for example, we could write the following instead.

```
rob = new Robot(win, 100 - length / 2, 100 - length / 2);
```

This would remove the need for the `start` variable, so that we could remove lines 11 and 12.

Exercise J3.1: Modify the `DrawTriangle` program of Figure J2.1 (page J11) so that it reads two numbers from the user, x and y, and places the triangle with its lower left corner at (x, y).

Chapter J4

Repetition

J4.1 The `while` loop

One of the great virtues of computers is that they will repeat mindless tasks without complaint. To facilitate this, Java includes a concept called a **loop**, which allows you to specify some sequence of instructions that should be repeated several times.

The most fundamental of loops in Java is the `while` loop. The `while` loop enables you to specify that some code should be done as long as (*while*) some condition is true. The template for a `while` loop looks like the following.

```
while(⟨thisIsTrue⟩) {
    ⟨statementsToRepeat⟩
}
```

You include the word `while`, with a true/false expression in parentheses (the parentheses are required), followed by a set of braces including the lines you want to repeat. The parenthesized expression is called the **condition** of the loop. The part between the braces is the loop's **body**.

When the computer reaches a `while` loop, it tests the condition inside the parentheses to determine whether it is true. If so, it executes the body (between the braces) and then rechecks the condition again. It repeatedly checks the condition and executes the body, until finally it finishes the body and the condition turns out to be false. Each time it executes the body of the loop is called an **iteration**. Once the computer gets to this point, it continues to the first statement following the body (after the closing brace). (If the condition never held in the first place, the computer skips past the body immediately.) Figure J4.1 illustrates this process.

To see an example of a `while` loop, let's look back to our `DrawSquare` program (Figure J3.1, page J20). That program involved some repetition — we had to draw four identical sides. We could replace lines 11 to 23 with a `while` loop, with each iteration of the loop drawing a single side of the square. Figure J4.2 contains this program.

The computer would execute the modified `DrawSquare` program as follows.

Lines 11–12: The computer creates the name `rob` and assigns to it a newly created `Robot` object.

Lines 13–14: The computer creates the name `drawn` and assigns the value 0 to it. This variable will count how many sides of the square `rob` has drawn. So far, `rob` hasn't drawn any; hence the program gives `drawn` the value 0.

Line 15: The computer checks whether the condition `drawn < 4` holds. It is, so the computer proceeds to execute each of the statements in the braces.

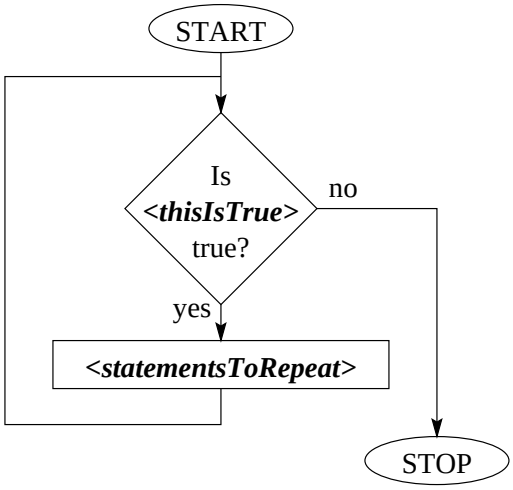


Figure J4.1: A flowchart for the `while` statement.

```
 1 import socs.*;
 2
 3 public class DrawSquare {
 4     public static void main(String[] args) {
 5         RobotWindow win;
 6         win = new RobotWindow();
 7         win.show();
 8
 9         double length;
10         length = win.requestInt();
11         Robot rob;
12         rob = new Robot(win, 100 - length / 2, 100 - length / 2);
13         int drawn;
14         drawn = 0;
15         while(drawn < 4) {
16             rob.move(length);
17             drawn = drawn + 1;
18             rob.turn(-90);
19         }
20         rob.switchOff();
21     }
22 }
```

Figure J4.2: The modified `DrawSquare` program.

Line 16: The computer tells `rob` to move forward `length` pixels.

Line 17: The computer now reassigns `drawn` to a new value: In particular, it is assigned to the value of `drawn + 1`. Since `drawn` is currently assigned to the value 0, the value of `drawn + 1` is 1, and so this is the value now assigned to the `drawn` name.

Line 18: The computer tells `rob` to turn 90° clockwise, so that `rob` now faces south.

Line 19: Having reached the end of the `while` loop, the computer proceeds to the top of the loop again to check the condition again, in line 15.

Lines 15–19: The computer checks whether the condition `drawn < 4` still holds. Since `drawn` is 1, it is does. Hence we proceed: The computer tells `rob` to move forward, reassigns 2 to `drawn`, and tells `rob` to turn facing west.

Lines 15–19: The condition `drawn < 4` still holds. The computer tells `rob` to move forward, reassigns 3 to `drawn`, and tells `rob` to turn facing north.

Lines 15–19: The condition `drawn < 4` still holds. The computer tells `rob` to move forward, reassigns 4 to `drawn`, and tells `rob` to turn facing east.

Line 15: The condition `drawn < 4` is finally false. So the computer is done with the `while` loop and now proceeds to the first statement following the loop's body, line 20.

Line 20: The computer tells `rob` to switch off.

One subtlety of `while` loops is that the computer checks the condition only when it reaches the end of the loop's body. In the last iteration of the `while` loop of this example, after `drawn` is assigned the value 4 in line 17, the robot nonetheless turns to face east in line 18, because the computer must complete the body of the `while` loop before checking the condition again.

A detail worth remembering	Notice that a `while` loop involves no semicolons! Beginners are often tempted to add one anyway.
	`while(i < 27); {  // Wrong!`
	`    i = i * 2;`
	`}`
	The irritating thing about this is that Java will accept it, understanding the semicolon as being the body of the loop, a single non-statement. The program will end up stuck in an infinite loop.

J4.2 Conditions

Alongside `int` and `double` among the primitive types is the `boolean` type, which is used to represent whether some fact holds. There are only two valid `boolean` values: `true` and `false`. (And that's how you write these values in Java programs: `true` or `false`.)

The most common usage of the `boolean` type is implicit: When you compare two numerical values (as done using the '`<`' operator on line 15 of Figure J4.2), the comparison operator takes two numeric values and computes a `boolean` values. Java has several comparison operators, all of which operate on two numerical values and result in a `boolean` value.

`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal

When the computer reaches a `while` loop as on line 15 of Figure J4.2, it evaluates the expression in the parentheses. Because the expression must compute a `boolean` value, the result will be either `true` or `false`, and the computer will use this result to determine whether to go through the body of the loop.

A detail worth remembering	Notice the doubled equal sign (==) for testing equality! The single equal sign (=) is already being used in Java for assignment. Beginners often find this somewhat confusing, but their meaning is quite different: Use == when you want to see whether two values are equal, and = when you want to change the value of a variable.

A detail worth remembering	One occassional pitfall when comparing `double`s is that round-off errors can cause unexpected results. For example, `1.2 - 1.1` turns out to be `0.09999999999999985`, which is different from `0.1`. So if we evaluated `1.2 - 1.1 == 0.1`, the value would would unexpectedly turn out to be `false`. To avoid errors, when you are testing to see if two `double`s are equal, you should instead test to see if the absolute value of their difference is small.

Sometimes you will want to combine conditions together to create a more complex condition. Java incorporates other operators to permit this.

`&&`	*and* (`true` if both sides hold)
`||`	*or* (`true` if either side holds, or if both hold)
`!`	*not* (`true` if expression doesn't hold)

For example, to see if `n` represents a value between 1 and 10, we would want to see if `n` is at least 1 and at most 10. We can do this using the && operator.

```
n >= 1 && n <= 10
```

(Java would *not* allow you to write "`1 <= n <= 10`.")

Here are some other examples.

`!(n >= 1 && n <= 10)`	`true` if `n` is not between 1 and 10
`n < 1 || n > 10`	`true` if `n` is less than 1 or greater than 10

J4.3 Incrementing assignments

You'll find that, in the context of loops, it's quite frequent that you write a statement like line 17 of our modified `DrawSquare`.

```
drawn = drawn + 1;
```

This reassigns `drawn` to one more than it was before. This operation is so common that Java has a way of abbreviating it, to save you the bother of having to type the variable name twice.

```
drawn++;
```

```
import socs.*;

public class WhileMystery {
    public static void main(String[] args) {
        RobotWindow win = new RobotWindow();
        win.show();

        Robot rob = new Robot(win, 100, 100);
        int len = 10;
        while(len < 200) {
            rob.move(len);
            rob.turn(90);
            rob.move(len);
            rob.turn(90);
            len += 10;
        }
        rob.switchOff();
    }
}
```

Figure J4.3: The `WhileMystery` class.

Similarly, Java also has an abbreviation for decrementing: "`i--;`" is equivalent to "`i = i - 1;`".

For those cases when you want to add something other than 1 to a variable, Java has another shortcut, the `+=` operator.

```
drawn += 10;
```

This line would be equivalent to "`drawn = drawn + 10;`". You can analogously use `-=`, `*=`, `/=`, and `%=`. .

Exercise J4.1: (Solution, J57) For each of the following conditions, describe the variable values for which it is true.

a. `x * x == x && x > -1`
b. `score > 90 || (bonus && score == 89)`
c. `!(k == 1)`

Exercise J4.2: (Solution, J57) Write a condition to test whether the `int` variable `year` represents a leap year. (Remember that a year is a leap year if it is a multiple of 4, except for years that are multiples of 100 but not 400. For example, 1992 and 2000 are leap years; 2100 is not.)

Exercise J4.3: (Solution, J57) Without running the program on a computer, draw a picture of what the `WhileMystery` program of Figure J4.3 would draw on the screen.

Exercise J4.4: (Solution, J58) Modify the `DrawSquare` program of Figure J4.2 so that it insists the user types a number between 5 and 195: As long as the user types a number outside this range, the compiler should ask again and again.

Exercise J4.5: Modify the `DrawSquare` program of Figure J4.2 so that it draws an octagon in the center of the window, with each side of the length the user specifies. The left vertex of the centered octagon's top horizontal edge would be at $(100 - 0.5\ell, 100 - 1.2071\ell)$, where ℓ is the edge length.

Chapter J5

Strings

J5.1 The `String` class

One particularly useful class included with Java is the `String` class, to represent a sequence of characters like a word or a sentence.

```
String sentence;
sentence = "Hello, world.";
```

In this code fragment, we create a variable named `sentence`, and we assign it to refer to the string "`Hello, world.`" To create a `String` object in Java, we simply enclose the characters it should contain in quotation marks, as in this short example. (The `String` class is unusual, in that it does not require the word `new` to create a new object. Instead, Java provides this quotation mark syntax to allow for easy creation of `String` objects.)

Java also allows you create a larger string through addition.*

```
String value;
value = "Drawn is " + drawn + ".";
```

If the `drawn` variable names the integer value 3, then this would make `value` refer to the string "`Drawn is 3.`".

Several instance methods in the `RobotWindow` class accept a `String` object as a parameter.

`double requestDouble(String message)`
: Shows the user a dialog box prompting for a number using `message`, and returns this number.

`int requestInt(String message)`
: Shows the user a dialog box prompting for an integer using `message`, and returns this integer.

`void notify(String message)`
: Shows the user a dialog box containing `message` and returns after the user clicks a button labeled OK.

These methods allow us to refine our `DrawSquare` class of Figure J3.1 (page J20) further. In particular, we can replace line 10 with the following.

```
10          length = win.requestInt("How long should each side be?");
```

*Notice that we're applying an operator (`+`) to an object here. Java normally prohibits this —most operators apply only to primitive values like numbers —, but it makes an exception for addition with the `String` class because of its usefulness.

Figure J5.1: Running the modified `DrawSquare` program.

Here we have passed a `String` object into the `requestInt` method that takes a `String` object as a parameter.[†] This particular method uses its `String` parameter to tell the user what it wants, instead of giving the unhelpful message, "Please type an integer." Figure J5.1 illustrates what the user would see (instead of Figure J3.2(a), page J20).

J5.2 The `IOWindow` class

The `IOWindow` class in the `socs` library allows you to interface with the user in a text format. It includes the following methods.

`IOWindow()`
: (Constructor method) Constructs an empty window for textual communication with the user. It automatically appears on the screen (there is no `show` method as with `RobotWindow`).

`void print(String message)`
: Prints `message` at the end of the transcript contained in the window.

`void println(String message)`
: Prints `message` at the end of the transcript contained in the window, followed by a line break.

`double readDouble()`
: Waits for the user to type a number into the window, followed by the Enter key, and returns this number.

`int readInt()`
: Waits for the user to type an integer into the window, followed by the Enter key, and returns this integer.

`String readLine()`
: Waits for the user to type a string into the window, followed by the Enter key, and returns this string.

For example, we might write the program of Figure J5.2 that reads in a sequence of five numbers and then displays their average.[‡] An execution of the program might place the following into the window. (The boldface characters indicate what the user typed.)

[†] There are two separate `requestInt` methods in the `RobotWindow` class: one that takes no parameters and one that takes a `String` parameter. Java determines which to use by looking at what's inside the parentheses. In this case, the parentheses contain a `String` value, so Java opts for the `requestInt` method taking a `String` parameter.

[‡] Lines 5 to 7 of this program employ the space-saving technique of combining a variable declaration with its first assignment. We've avoided this shortcut until now to emphasize that these are two separate concepts, but from now on we'll combine them freely.

Also, lines 10 and 11 illustrate the incrementing shortcuts of Section J4.3.

```
 1 import socs.*;
 2
 3 public class Average {
 4     public static void main(String[] args) {
 5         IOWindow io = new IOWindow();
 6         double total = 0.0; // accumulates the sum of user's numbers
 7         int done = 0;       // counts how many numbers have been read
 8         while(done < 5) {
 9             io.print("Number? ");
10             total += io.readDouble();
11             done++;
12         }
13         io.println("Average is " + (total / done));
14     }
15 }
```

Figure J5.2: The `Average` program.

```
Number? 95
Number? 87
Number? 74
Number? 93
Number? 82
Average is 86.2
```

J5.3 Testing and debugging

While it's not too difficult to write a program that the compiler will accept, and it's only a mild challenge to write a program that seems to work, it's impossible to avoid writing erroneous programs. Testing your program is an important skill in programming; one could argue that the ability to test well is what separates the good programmers from the mediocre ones.

To check whether your program is correct, it's important that you try it on a variety of test cases. And don't just choose several similar test cases: Try a wide variety. For example, in our `DrawSquare` program, we should try several possible inputs. We should choose a good "typical case" run, like 60. But some extreme cases are good, too: 0 and 1 for the small end, and 1000 on the large end. If we were feeling particularly adventurous (and, as a tester, that's a good feeling to have), we'd try a negative number, too, like -60, to see what happens.

Until you've run your program on a wide variety of test cases, you can't be confident that your program works. Even *after* you try the wide variety, you won't be sure — there's frequently some unforeseen problem. But in larger programs, testing is sure to reveal some problems.

Frequently, during testing, you find that the program needs some work. Now you need to **debug** your program. (Programmers call each error in the program a **bug**, which irks some who think this term trivializes errors.) If you understand your program well, you most often know exactly where to fix it. But occassionally, you're stumped: How does my program do *that*? In my opinion, this is the most fun aspect of programming — the detective work to track down how things went awry.

One of the most useful techniques for figuring out what's going wrong is to slow things down and get more information about what the computer is doing. You can do this by displaying more information to the user. For example, in our new `DrawSquare` program (Figure J4.2), we might add the following line after line 18.

```
win.notify("Drawn is " + drawn);
```

If we added this line, then the computer would pause each time through the loop with a message notifying the user of the value of `drawn`. This gives us as programmers the chance to check that the computer is on the right track.

If you have lots of debugging information to dump out for inspection, lots of dialog boxes popping up quickly gets irritating. The `IOWindow` class is handy for avoiding this: You can create a single `IOWindow` object and dump all the information into it.

J5.4 `String` methods

As a class defined in the Java libraries, the `String` class has a host of methods that you can use on a particular string.

`int length()`
: Returns the number of characters in this string.

`String substring(int begin)`
: Returns a string containing the characters of this string beginning at index `begin` and going to this string's end.

`String substring(int begin, int end)`
: Returns a string containing the characters of this string beginning at index `begin` and going up to — but not including — index `end`.[§]

It's important to understand that the index of the first character of the string is 0, the second character's index is 1, and so on.

Suppose we want to print a particular string in reverse; for example, if the user types "straw," we want the program to print "warts." The following code segment accomplishes this.

```
String str = io.readLine();
int index = str.length() - 1;
while(index >= 0) {
    io.print(str.substring(index, index + 1));
    index--;
}
```

This fragment begins at the index of the final character of the string, which is 1 less than the length of the string, since the indices begin at 0. And then, for each iteration of the loop, it prints out that character and then decrements the index in order to move to the previous character. Notice that, in calling the `substring` method, it passes `index` as the beginning point, since that is the character it wants to retrieve, and it passes `index + 1` as the finishing point, since this is the first character it does *not* want.

J5.5 Equality testing

Suppose we wanted to write a program to repeatedly ask the user for a password until the user types the correct password, which we'll say is "friend." You might be tempted to accomplish this with the following program fragment.

[§] Omitting the last character named is not what you would expect, but it often turns out to be convenient. For example, the number of characters fetched by the `substring` method is simply `end - begin`.

```
1 String password;
2 io.print("Password? ");
3 password = io.readLine();
4 while(password != "friend") {   // Wrong!
5     io.print("Wrong. Password? ");
6     password = io.readLine();
7 }
8 io.print("You're in.");
```

This says to read in a word from the user (line 3). As long as the word the user types isn't "friend" (line 4), the program continues asking for another word (lines 5 and 6).

The idea's good, and the program will compile and run fine, but when we test it, it won't work. It will keep saying that we were wrong, even when we type "friend" at the prompt.

The problem is that a test to see if two objects are equal is a test to see if they're the same. As it turns out, `password` and the string created by enclosing "friend" in double-quotations marks, are two different strings, even if they contain the same letters. They're identical, but they're not the same. It's the same principle that leads me to assert that two identical blue M&M's laid side by side are nonetheless different: They're not the same piece of candy, even if they are indistinguishible.

Thus, the condition of line 4 will always turn out to be true — `password` will never equal `"friend"`, even if `password` contains the same letters.

So how can we compare two strings to see if they contain the same letters? Luckily, Java's `String` class includes some methods to help with this dilemma.

`boolean equals(String other)`
: Return `true` if this string contains the same sequence of characters as `other` does.

`boolean equalsIgnoreCase(String other)`
: Return `true` if this string contains the same sequence of characters as `other` does, treating lowercase letters and capital letters the same.

We can use the `equals` method to compare the two strings in line 4 of our program fragment.

```
4 while(!password.equals("friend")) {
```

The `equals` method will look through the characters of `password` and see if they all match up with the corresponding characters of `"friend"`. If they do, it returns `true`, and this `while` condition (with its exclamation point for representing *not*) will have a value of `false`, so that the computer will proceed to the first statement following the `while` loop, line 8. If they don't match, the `equals` method returns `false`, so that the value of the condition is `true`, and so the computer will go through another iteration of the loop, asking the user to try again.

Exercise J5.1: (Solution, J58) Suppose a user executes the program of Figure J5.3 and types `5` when prompted. What does the program display?

Exercise J5.2: Write a program using `IOWindow` that reads in a number and prints its square right-justified over 10 columns.

```
Number? 456
    207936
```

To convert the number into a string, you can add it to an empty string: "`"" + 45`" gives you the string containing the characters "`45`". Then you can use the `length` instance method in the `String` class to determine how many spaces you need to print.

Exercise J5.3: Write a program using `IOWindow` that prints a triangle of asterisks of a width given by the user.

```
public class Bomb {
    public static void main(String[] args) {
        IOWindow io = new IOWindow();
        io.print("Integer? ");
        int num = io.readInt();
        String str = "B";
        while(num > 0) {
            io.println("" + num);
            str += "O";
            num--;
        }
        io.println(str + "M!");
    }
}
```

Figure J5.3: The `Bomb` program.

```
How long a side? 5
*****
****
***
**
*
```

Chapter J6

Conditional execution

J6.1 The `if` statement

An **`if` statement** tells the computer to execute a sequence of statements only *if* a particular condition holds. This is a type of **conditional statement**, since it allows us to indicate for the computer to execute some statements only in some circumstances.

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
}
```

This corresponds to the flowchart in Figure J6.1. When the computer reaches the `if` statement, it evaluates the condition. If it turns out to be `true`, then the computer executes the `if` statement's body and then continues to any statements following. If the condition turns out to be `false`, it skips over the body and goes directly to whatever follows the closing brace.

An `if` statement looks a lot like a `while` loop. The only differences are that it uses the `if` keyword in place of `while` and that the computer does not recheck the condition after executing the statement's body.

As an example of an `if` statement in action, consider the following code fragment.

```
double abs = num;
if(num < 0.0) {
    abs = -num;
}
io.println("Absolute value = " + abs);
```

In this fragment, we create a variable `abs`, which initially refers to the value of `num`. If `num` is less than 0, then we change the reference to the value of "`-num`," and then we continue to the statement printing this as the absolute value. But if `num` is not negative, we skip the "`abs = -num`" statement and go directly to print `abs` as the absolute value (in this case, the printed value is the same as that of `num`).

J6.2 The `else` clause

Sometimes we want to do one thing if the condition is true and another thing if the condition is false. In this case the `else` keyword comes in handy.

```
if(⟨thisIsTrue⟩) {
    ⟨statementsToDoIfTrue⟩
} else {
    ⟨statementsToDoIfFalse⟩
}
```

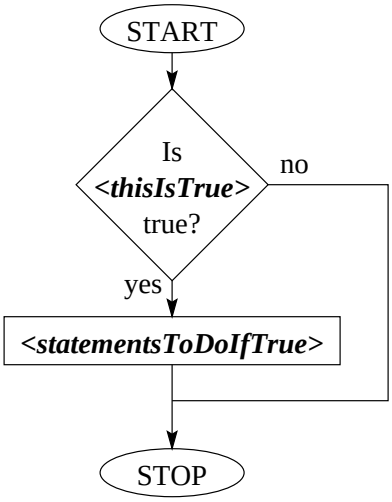


Figure J6.1: A flowchart for the `if` statement.

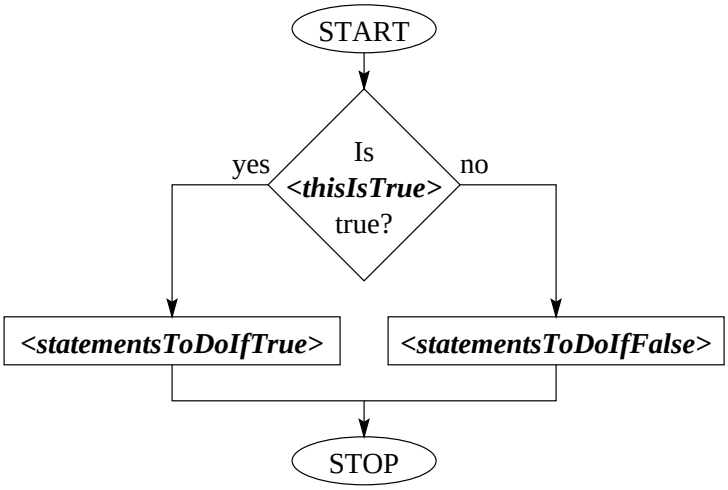


Figure J6.2: A flowchart for the `else` clause.

Figure J6.2 contains a flowchart diagramming this type of statement.

For example, if we wanted to compute the larger of two values `x` and `y`, then we might use the following code fragment.

```
int max;
if(x > y) {
    max = x;
} else {
    max = y;
}
```

This function says assign the value of `x` to `max` if `x` holds a larger value than `y`; otherwise — it says — assign it the value of `y`.

Sometimes it's useful to string several possibilities together. This is possible by inserting "`else if`" clauses into the code.

```
io.println("Guess a number.");
int guess = io.readDouble();
if(guess < 21) {
    io.print("That's way too small.");
} else if(guess < 23) {
    io.print("That's too small.");
} else if(guess > 23) {
    io.print("That's too large.");
} else {
    io.println("That's just right!");
}
```

You can string together as many `else if` clauses as you want. Having an `else` clause at the end is not required.

Note that, if the user types `5` as the computer runs the above fragment, the computer would print only, "That's way too small." It would *not* also print, "That's too small," even though it's true that `guess < 23`. This is because the computer checks an `else if` clause only if the preceding conditions all turn out to be `false`.

J6.3 Braces

When the body of a `while` or `if` statement holds only a single statement, the braces are optional. Thus, we could write our `max` code fragment as follows, since both the `if` and the `else` clauses contain a single statement. (We could also include braces on just one of the two bodies.)

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

I recommend that you include the braces anyway. This saves you trouble later, should you decide to add additional statements into the body of the statement. And it makes it easier to keep track of braces, since each indentation level requires a closing right brace.

In fact, Java technically doesn't have an `else if` clause. We could have formatted our "Guess a number" fragment as follows.

```
io.println("Guess a number.");
int guess = io.readDouble();
if(guess < 21) {
    io.print("That's way too small.");
} else
    if(guess < 23) {
        io.print("That's too small.");
    } else
        if(guess > 23) {
            io.print("That's too large.");
        } else {
            io.println("That's just right!");
        }
```

In this fragment, each `else` clause includes exactly one statement — which happens to be an `if` statement with an accompanying `else` clause in each case. Thus, the only thing we were really seeing when we were talking about `else if` clauses is a more convenient way of inserting white space for the special case where an `else` clause contains a single `if` statement.

J6.4 Variables and compile errors

J6.4.1 Variable scope

Java allows you to declare variables within the body of a `while` or `if` statement, but it's important to remember the following: A variable is available only from its declaration down to the end of the braces in which it is declared. This region of the program text where the variable is valid is called its **scope**.

Check out this illegal code fragment.

```
1 if(x > y) {
2     int max = x;
3 } else {
4     int max = y;
5 }
6 io.print("The maximum is " + max);   // Illegal!
```

If you tried to compile this, the compiler would point to line 6 and print a message saying something like, "Cannot resolve symbol," referring to the symbol `max`. What's going on here is that the declaration of `max` in line 2 only persists until the closing brace on line 3; and the declaration in line 4 lasts only until the closing brace on line 5. Thus, `max` is an invalid variable on line 6.

A novice programmer might try to patch over this error by declaring `max` before the `if` statement.

```
1 int max = 0;
2 if(x > y) {
3     int max = x;
4 } else {
5     int max = y;
6 }
7 io.print("The maximum is " + max);   // Illegal!
```

A good compiler will respond in line 3 with a message like "`max` is already defined in `main`," pointing out that you're trying to declare two variables of the same name. In Java, each time you label a variable with a type, it counts as a declaration. So line 3 is a variable declaration, which is not what we want; we want lines 3 and 5 to refer to the `max` variable already declared in line 1.

You should declare each variable once and only once. The following is a valid way to write this fragment.

```
1 int max;
2 if(x > y) {
3     max = x;
4 } else {
5     max = y;
6 }
7 io.print("The maximum is " + max);
```

J6.4.2 Variable initialization

Another common mistake of beginners is to use an `else if` clause where an `else` clause is completely appropriate.

```
1 int max;
2 if(x > y) {
3     max = x;
4 } else if(x <= y) {
5     max = y;
6 }
7 io.print("The maximum is " + max);   // Illegal!
```

Surprisingly, the compiler will complain about line 7 here, with a message like "variable `max` might not have been initialized." This message is a bit perplexing to the novice who thinks that line 1 initializes the variable; it does not — it only *declares* the variable. In programming, **initialization** refers to the first assignment of a value to a variable.

In this particular fragment, the computer is reasoning that it may be possible for the computer to reach line 7 without any assignment of a value to `max`: The computer recognizes that this assignment would occur on line 3 if the condition of line 2 turns out `true`, or on line 5 if the condition of line 4 turns out `true`. But what if both are `false`? You and I know this can't happen, but the compiler's not sophisticated enough to see this.

The solution in this case is to recognize that, in fact, we don't need the extra `if` test in line 4: If the condition of line 2 turns out to be `false`, then of course the condition of line 4 will be `true`. Thus, doing that extra test in line 4 both adds extra verbiage and slows down the program slightly. If we delete that extra test, the compiler will reason successfully that `max` will be initialized (on either line 3 or line 5, depending on whether the condition of line 2 turns out `true`), and it won't complain.

(Novice programmers are sometimes tempted to simply initialize `max` on line 1 to get around this message.

```
1 int max = 0;
```

This removes the compiler message, and the program works. But patching over compiler errors like this is a bad habit. You're better off addressing the problem at its root.)

By the way, Java compilers are some of strictest I've seen about these sorts of things. It's actually somewhat irritating, at times, when it refuses to compile something like a program with an alleged uninitialized variable, even when it's actually the case that the program would run flawlessly. The reason it does this is that it's aggressively trying to help you debug your programs. Suppose, for the sake of argument, that it was a real problem in the program. If the compiler didn't complain, you'd be relying on the test cases to catch the problem. If the test cases didn't catch it, you'd be misled into releasing a erroneous program. Even if they did catch it, you're stuck trying to figure out the cause, which can take quite a while. A problem like an uninitialized variable can be very difficult to track down by trial and error; but with the compiler pointing to it for you, it becomes much easier.

```
 1 import socs.*;
 2
 3 public class Primality {
 4     public static void main(String[] args) {
 5         IOWindow io = new IOWindow();
 6         io.print("What do you want to test? ");
 7         int to_test = io.readInt();
 8         int i = 2;
 9         while(i * i <= to_test) {
10             if(to_test % i == 0) {
11                 io.println("It is not prime");
12                 break;
13             }
14             i++;
15         }
16         if(i * i > to_test) {
17             io.println("It is prime");
18         }
19     }
20 }
```

Figure J6.3: The `Primality` program.

J6.5 The `break` statement

Sometimes, in going through an iteration of a loop, it becomes appropriate to immediately stop work on the loop. In this case, the `break` statement is handy. It's a simple statement, just the word `break` followed by a semicolon. When the computer reaches a `break` statement, it immediately skips to the first statement outside of the loop in which the statement occurs.

Figure J6.3 contains a program to determine whether a number is prime. Here's a sample run of this program.

```
What do you want to test? 25
It is not prime.
```

Let us do a step-by-step trace to see how this came about. First the computer prompts the user (line 6) for a number and reads the number from the user (line 7). The user types **25**, so the variable `to_test` now represents the number 25. Now we create a new variable `i` before entering the `while` loop (line 8). We assign `i` to be 2 as line 8 instructs.

First iteration: Since $\texttt{i}^2 \leq 25$, we go through our first iteration. In line 10, we test to see if "`to_test % i`" is 0, which it is not (the remainder is 1); thus we skip past the statements in these braces (lines 11 and 12) to what follows them (line 14). This reassigns `i` to the value of `i + 1`, which is 3. We reach line 15, which closes off the body of the loop, and the iteration is done.

Second iteration: Still $\texttt{i}^2 \leq 25$ (that is, $3^2 < 25$), so we go through another iteration. This time "`to_test % i`" has the value 1, so we skip the statements within the braces to line 14. We execute "`i++`"; now `i` represents the value 4.

Third iteration: Still $\texttt{i}^2 \leq 25$, so we go through another iteration. This time "`to_test % i`" has the value 1, so we skip the statements within the braces to line 14. We execute "`i++`"; now `i` represents the value 5.

Fourth iteration: Still $\texttt{i}^2 \leq 25$, so we go through another iteration. This time "`to_test % i`" has the value 0, so we execute the statements within the braces. In this case, the statements tell us to print that `to_test` is not prime (line 11) and then to break out of the loop on which we're working. The computer immediately skips down to line 16, the first statement following the loop's body. It does *not* execute line 14. (And it does not collect $200.)

In line 16, the computer notices that $\texttt{i}^2 > 25$ is `false`, so it skips the body of that `if` statement.

For nested loops, where one loop lies within the body of another, the `break` statement applies to the innermost loop only.

Exercise J6.1: (Solution, J58) Write a program to tell whether an integer divides another exactly. It should behave something like the following. (You may find the modulus operator % useful.)

```
What is the numerator? 13
What is the denominator? 2
2 does not divide 13.
```

Exercise J6.2: Write a program that determines whether a string read from the user is a palindrome — that is, whether it reads the same forwards and backwards.

```
String? RACINGCAR
It is not a palindrome.
```

Examples of palindromes include *redder* and *civic*.

Exercise J6.3: Write a program to help balance a checkbook. A run of the program should look exactly like the following.

```
To add? 30.25
+ 30.25 = 30.25
To add? -20.30
- 20.3 = 9.95
To add? 998.23
+ 998.23 = 1008.18
To add? -447.87
- 447.87 = 560.31
To add? 0.0
```

The user should be able to type as many entries as desired; but when the user types zero, the program exits. (Using zero this way allows you to read the user-typed value into a `double` instead of doing something more complicated.)

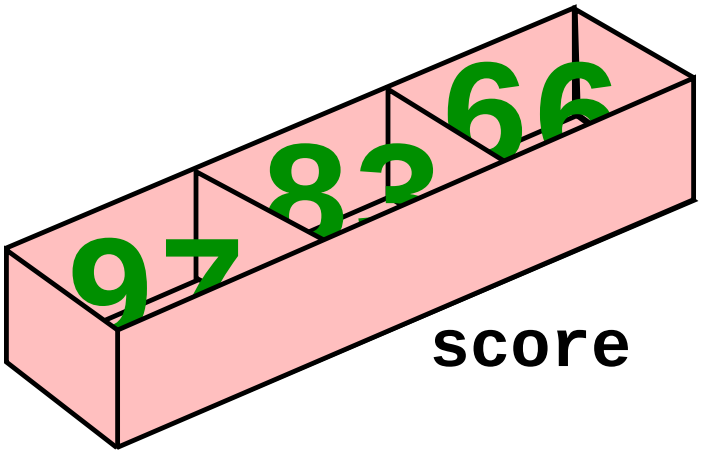
Chapter J7

Arrays

J7.1 Arrays

Besides allowing for values representing a single object, Java also allows for values representing an entire sequence of objects. Such an object is an array.

An **array** holds a sequence of values of the same type. It's useful when you want to store a large group of related data, like data points in a graphing program or students' scores in a gradebook program. Each individual value in the array is called an **array element**.



You can declare a variable to be of an array type using the following format.

⟨***typeOfElement***⟩`[]` ⟨***variableToDefine***⟩`;`

For example, the following creates a variable `score` that can name an array of numbers.

```
double[] score;
```

This creates a variable that can refer to an array. Like object variables, it does not refer to an array yet.

To assign a newly created array to the variable, we can use the `new` keyword.

```
score = new double[3];
```

This is the syntax for creating an array: the `new` keyword, followed by the type name of each array element, followed by the length of the array enclosed in brackets. The integer in brackets can be any expression ("`2 * num_students`" instead of "`3`", for example).

To work with an array, we must refer to individual elements using their **array indices**. The array elements are automatically numbered from 0 up to one less than the array length.

A detail worth remembering	Yes, that's *one less than the array length*. So if you declare an array `score` of length 3, the array indices are 0, 1, and 2. Java always begins at 0. (This turns out to be more convenient than the intuitive choice of 1.) If you try to access an undefined array index, the program will crash, giving some message including the word "`ArrayIndexOutOfBoundsException`." So be careful with array indices.

To refer to an array element in an expression, type the array variable name, followed by the element's array index enclosed in brackets. You can also do this on the left-hand side of an assignment statement to alter an array element's contents.

```
double[] score = new double[3];
score[0] = 97.0;
score[1] = 83.0;
score[2] = 66.0;
io.println("Average = " + ((score[0] + score[1] + score[2]) / 3.0));
```

In these statements we create an array of three numbers, called `score`. We assigned its three boxes to refer to three test scores, 97, 83, and 66. And finally we printed the average of these. The computer will display `82.0`.

The significance of arrays comes when you use an expression to access a particular element of the array. Figure J7.1 contains a short program that reads a sequence of numbers into an array and then prints the numbers in reverse order. For example, the user might experience the following in running the program. (What the user types is in boldface.)

```
How many scores? 3
Type the scores now.
2
3
5
Here they are in reverse order.
  5
  3
  2
```

There's no way we could write a program to accomplish this using what we had seen in previous chapters. Using arrays, however, allows us to store an arbitrarily large sequence of data with no troubles.

J7.2 The `length` attribute

Java includes a special technique for accessing the length of an array, using the word `length`. In any expression, you can write the array name, followed by a period and the word `length`, and the value will be the number of items that the array was created to hold. As an example, we could rewrite line 14 of `PrintReverse` as follows.

```
14          while(i < scores.length) {
```

It's preferable to use the `length` keyword when appropriate, in favor of using some other variable that happens to represent the length of the array.

Exercise J7.1: (Solution, J60) Write a program that computes the mode of a set of integer test scores between 0 and 100. (The **mode** is the element that occurs most frequently in the list.) Here is a sample transcript.

```
 1 import socs.*;
 2
 3 public class PrintReverse {
 4     public static void main(String[] args) {
 5         // create the array
 6         IOWindow io = new IOWindow();
 7         io.print("How many scores? ");
 8         int num_scores = io.readInt();
 9         double[] scores = new double[num_scores];
10
11         // fill the array
12         io.println("Type the scores now.");
13         int i = 0;
14         while(i < num_scores) {
15             scores[i] = io.readDouble();
16             i++;
17         }
18
19         // print it in reverse
20         io.println("Here they are in reverse order.");
21         i = num_scores - 1;
22         while(i >= 0) {
23             io.println("  " + scores[i]);
24             i--;
25         }
26     }
27 }
```

Figure J7.1: The `PrintReverse` program.

.

```
How many numbers? 5
#1: 83
#2: 32
#3: 83
#4: 71
#5: 65
Mode = 83
```

Hint: The easiest way of doing this is not obvious. Use an array indexed from 0 to 100 to tally how many times each of the numbers occurs. Then you can go through the array to see which index contains the largest element.

Chapter J8

Class methods

J8.1 Using class methods

Java includes three types of methods: constructor methods, instance methods, and class methods. We've already seen constructor methods and their use for creating objects. And we've seen how instance methods allow a program to send a message to an object. A **class method** is a method associated directly with the class.

For example, if you look in the documentation for the `Math` class, you'll see the following method. (The `Math` class defines several methods like this for computing mathematical functions.)

`static double pow(double base, double exp)`
: Returns the result of raising `base` to the `exp`th power.

This looks like a regular instance method, except for one thing: It includes the word `static` in the descriptor. That word `static` marks this as a class method.

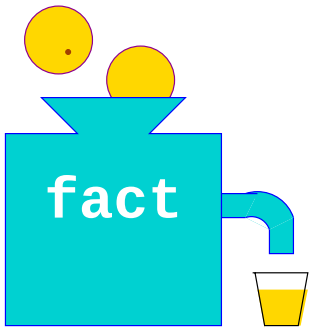
To execute a class method, you send the message to the class in general (in this case, the `Math` class); you don't send it to a particular instance. The following fragment illustrates a use of this method.

```
io.println("The square root of 5 is " + Math.pow(5.0, 0.5));
```

Since `println` is an *instance* method in the `IOWindow` class, we need a particular `IOWindow` object to which to send it; we send it to the `IOWindow` object named by `io`. Since `pow` is a *class* method in the `Math` class, we perform `pow` on the entire class by writing `Math.pow`. No `Math` object is required.

J8.2 Defining a class method

The libraries define several useful class methods (especially in the `Math` class); but it's also often useful to define our own. You can think of a class method as a packaged sequence of statements to accomplish a certain task. A useful analogy is to think of a it as a sort of juicer.



This juicer takes some parameters and produces a return value. In our juicer analogy, the parameters would be the fruit, and the juice produced would be the return value. If you give it oranges, it makes orange juice. Give it lemons, and it makes lemonade.

Defining new class methods can serve two purposes.

- They allow you to describe a procedure once, even if the program executes the procedure many times. For example, maybe your program determines whether a number is prime in many places. By making a class method to test primality, you can write your algorithm only once (as the machinery for that method) and then easily use this method in many places through simple calls to the class method. This simplifies the task of writing the program, and it also makes modifying the primality-testing method very easy later, should you need to change it.

- They permit decomposing a larger program into subtasks. Even if there is only one location in your program where you decide whether a number is prime, it may be useful to make it a separate method anyway, just because in the larger program the details of exactly how you determine this are irrelevant. By using methods in this way, a program becomes easier to read, and it becomes possible to write much larger programs.

So it's useful to be able to define class methods. To accomplish this in Java, we place the method definition inside the class. The method definition looks like the following.

```
public static ⟨returnValueType⟩ ⟨methodName⟩(⟨parameterList⟩) {
    ⟨methodBody⟩
}
```

Figure J8.1 illustrates a program containing a class method definition. Actually, this program defines two methods: a `fact` method (taking an `int` as a parameter and returning a `double`) and a `main` method (taking an array of `String`s as a parameter and returning nothing).*

To better understand class method definitions, let's break apart the `fact` method in lines 4 to 10.

`public static` Think of these words as simply being required when you define any class method.

`double` Then we say that this method will be returning a `double` value. (Some methods may actually return nothing; they'd be useful due to their side effects, like displaying something to the user. For these, we would write `void` for the return type.)

`fact` Then we find the name of class the method — `fact` in this case. The rules for method names are the same as for variable names. Always think carefully about how to name your methods so as to best communicate the method's purpose.

*Don't worry about the array of `String`s being a parameter to the `main` method. Few programs use this array, but Java requires that the `main` method must take it as a parameter anyway.

```
 1 import socs.*;
 2
 3 public class Choose {
 4     public static double fact(int k) {
 5         double ret = 1; // this will be the factorial of k
 6         int i = 1;
 7         while(i <= k) {
 8             ret *= i; // multiply the value of i into ret
 9             i++;
10         }
11         return ret;
12     }
13
14     public static void main(String[] args) {
15         IOWindow io = new IOWindow();
16         io.print("Choose how many of how many? ");
17         int r = io.readInt();
18         int n = io.readInt();
19         io.println((Choose.fact(n) / Choose.fact(r) / Choose.fact(n - r))
20             + " choices");
21         return;
22     }
23 }
```

Figure J8.1: The `Choose` program.

`(int n)` In the parentheses are the method's *parameters* — the lemons for our juicer. In this case, the method takes one parameter, which is of type `int`. This parameter defines the variable `n` to represent the value given to the method.

`{...}` The brace characters surround the statements that say what the method does. This is called the **method body**. This corresponds to the machinery within the juicer.

When a method is called, the computer will assign the parameter values to the parameter variables and then begin executing the statements contained within the method body. It will continue executing these until it finally reaches a `return` statement, at which time it immediately stops its work and exits the method.

For methods with a return type (non-`void`), the `return` statement will contain some expression specifying what value to return (the lemonade, according to our analogy). In the above example, the method returns the value of the `ret` variable.

Here is a sample run of this program.

```
Choose how many of how many? 2 6
15.0 choices
```

To illustrate how class methods work, let's trace through a complete run to see how this came about.

Lines 14–18: We begin at the beginning of the `main` method. We create two variables `n` and `r` and wait for the user to give their values. Now `r` represents 2 and `n` represents 6.

Line 19–20: In order to compute the first expression to be printed, we compute the value of "`fact(n)`". Since `n` represents 6, we call `fact` with its parameter `k` representing 6.

Lines 5–10: We run through the code of `fact` with `k` representing 6. This code multiplies all the integers between 1 and 6 together; we finally reach line 10 with `ret` representing 720.

Line 11: We return the value of `ret` (that is, 720) and continue with line 18, where we called the method.

Line 19–20: Now that we know the dividend is 720, we compute the divisor. Again, we call `fact`, this time with `k` representing what `r` represents, the value 2.

Lines 5–11: We run through `fact` with `k` representing 2. When we reach line 10, `ret` represents 2 and so 2 is the return value.

Line 19–20: Now that we have computed the first two calls, we divide to get $720/2 = 360$. But we still have to perform another division. We know the dividend is 360; to get the divisor, we call `fact` again, this time with `k` representing the value of "`n - r`", which is 4.

Lines 5–11: We run through `fact` one more time, this time with `k` representing 4. The return value is 24.

Line 19–20: Now that we have 24 from `fact`, we perform the second division to get $360/24 = 15$. We send the string "`15.0 choices`" to the `println` instance method for printing in the window.

Line 21: We reach the `return` statement in `main`, so we are finished with running the program.

The `return` statement in line 19 was optional: For methods that return `void`, Java will assume that we mean to return if the computer reaches the end of the method.

Java permits a shortcut for calling class methods within the same class where they are defined: You can omit the class name. Thus, we can write lines 17 and 18 of `Choose` as follows instead.

```
17          io.println((fact(n) / fact(r) / fact(n - r))
18              + " choices");
```

Often we want a class method with several parameters. In this case we list the parameters, separated by commas. For example, we might want to add a class method named `choose`. To do this, we would insert the following into the `Choose` class.

```
public static double choose(int n, int r) {
    return fact(n) / fact(r) / fact(n - r);
}
```

To call such a method, you list the expressions for the arguments in the same order they are defined, separated by commas.

```
io.println(Choose.choose(6, 2) + " choices");
```

This will call our `choose` method with the first parameter `n` representing the value 6 and the second parameter `r` representing the value 2.

J8.3 Parameters and variables

The variables available within a class method are exactly those declared within the method. The code within a method cannot see variables defined in methods. (These variables declared in other methods, by the way, are technically called **local variables**, since they are available only within the method where they are declared.) For this reason, for any method you define, you should include among its parameters any information that the method needs.

Note that when the computer calls a method, it copies the values sent into the parameter variables. So if we happen to change a parameter variable's value, this does not alter anything outside the method. As an example, consider the following program.

```
 1 import socs.*;
 2
 3 public class ZeroExample {
 4     public static void setToZero(int n) {
 5         n = 0;
 6     }
 7
 8     public static void main(String[] args) {
 9         IOWindow io = new IOWindow();
10         int i = 1;
11         setToZero(i);
12         io.println(i);
13     }
14 }
```

This program will print the value `1`. Setting the value of `n` to 0 on line 5 in `setToZero` has no effect on the value of `i` declared on line 10 of `main`. This system of parameter passing is called **call by value**.

Changing the value assigned to a parameter variable has no effect on what is passed into the method, but there is an effect if the method changes the values contained within whatever the parameter variable refers to. This effect is noticeable with arrays. Suppose we were to use the following program instead.

```
 1 import socs.*;
 2
 3 public class ZeroExample {
 4     public static void setToZero(int[] array) {
 5         array[0] = 0;
 6     }
 7
 8     public static void main(String[] args) {
 9         IOWindow io = new IOWindow();
10         int[] a = new int[1];
11         a[0] = 1;
12         setToZero(a);
13         io.println(a[0]);
14     }
15 }
```

Here, we create an array on line 10, set its first value to be 1 on line 11, and pass the array into the `setToZero` method. The parameter variable `array` refers to the same array that `a` refers to in the `main` method, so when the computer reassigns the first element in the array in line 5, it's changing what's within that array named by both of these variables. Thus, in line 13 of this program, the computer would print 0.

Exercise J8.1: Define a class method to find the greatest common divisor of two numbers, and use this to modify the checkbook exercise of Exercise J6.3 to work with fractions rather than `double`s. The program should keep the total in lowest terms.

```
To add? 5 6
+ 5/6 = 5/6
To add? 3 4
+ 3/4 = 19/12
To add? 5 12
+ 5/12 = 2/1
To add? 0 0
```

Exercise J8.2: (Solution, J60) Write a class including the following class method.

`public static int removeDuplicates(int[] arr)`

Removes all adjacent duplicates within `arr` and returns return the elements remaining. For example, given the array $\langle 3, 7, 7, 7, 8, 3, 3\rangle$, the method should replace the elements with $\langle 3, 7, 8, 3, 0, 0, 0, 0\rangle$ and return 4.

Appendix JA

Class documentation

JA.1 `IOWindow` class

Represents a window in which a program can interact with the user using a simple text interface.

You must import the `socs` library to use this class.

`IOWindow()`
: (Constructor method) Constructs an empty window for textual communication with the user. It automatically appears on the screen (there is no `show` method as with `RobotWindow`).

`void print(double value)`
: Prints the decimal representation of `value` into this window.

`void print(int value)`
: Prints the decimal representation of `value` into this window.

`void print(String message)`
: Prints `message` at the end of the transcript contained in the window.

`void println()`
: Prints a newline into this window.

`void println(double value)`
: Prints the decimal representation of `value` into this window, followed by a newline.

`void println(int value)`
: Prints the decimal representation of `value` into this window, followed by a newline.

`void println(String message)`
: Prints `message` at the end of the transcript contained in the window, followed by a line break.

`double readDouble()`
: Waits for the user to type a number into the window, followed by the Enter key, and returns this number.

`int readInt()`
: Waits for the user to type an integer into the window, followed by the Enter key, and returns this integer.

`String readLine()`
: Waits for the user to type a string into the window, followed by the Enter key, and returns this string.

JA.2 `Math` class

Holds class methods for computing mathematical functions.

`static double abs(double value)`
: Returns the absolute value of the given double-precision number.

`static int abs(int value)`
: Returns the absolute value of the given integer.

`static double acos(double value)`
: Returns the arccosine of the given number. (That is, the angle (in radians) whose cossine is `value`.) The result is between 0 and π, or NaN if no such angle exists.

`static double asin(double value)`
: Returns the arcsine of the given number. (That is, the angle (in radians) whose sine is `value`.) The result is between $-\pi/2$ and $\pi/2$, or NaN if no such angle exists.

`static double atan(double value)`
: Returns the arctangent of the given number. (That is, the angle (in radians) whose tangent is `value`.) The result is between $-\pi/2$ and $\pi/2$.

`static double atan2(double deltay, double deltax)`
: Returns the arctangent of `deltay`/`deltax`. This may be understood as the slope of the line that rises `deltay` units for each `deltax` units to the right. The result is between $-\pi$ and π.

`static double ceil(double value)`
: Returns the double-precision value representing the least integer that is at least `value`.

`static double cos(double radians)`
: Returns the cosine of the angle given in radians.

`static double exp(double exp)`
: Returns the value of Euler's constant (2.71828...) to the power `exponent`.

`static double floor(double value)`
: Returns the double-precision value representing the greatest integer that is at most `value`.

`static double log(double value)`
: Returns the natural logarithm of the given value.

`static double max(double a, double b)`
: Returns the greater of the two given double-precision values.

`static int max(int a, int b)`
: Returns the greater of the two given integer values.

`static double min(double a, double b)`
: Returns the lesser of the two given double-precision values.

`static int min(int a, int b)`
: Returns the lesser of the two given integer values.

`static double pow(double base, double exp)`
: Returns the result of raising `base` to the `exp`th power.

`static double random()`
: Returns a pseudorandom number that is at least 0.0 and less than 1.0.

`static double rint(double value)`
: Returns the double-precision value representing the integer that is closest to `value`.

`static double sin(double radians)`
: Returns the sine of the angle given in radians.

`static double sqrt(double value)`
: Returns the square root of the given value.

`static double tan(double radians)`
: Returns the tangent of the angle given in radians.

JA.3 `Robot` class

Represents a robot who can move around a window.

You must import the `socs` library to use this class.

`Robot(RobotWindow world, double x, double y)`
: (Constructor method) Constructs a robot object, located in the `world` window at coordinates $(\texttt{x}, \texttt{y})$, facing east.

`void move(double dist)`
: Moves this robot forward `dist` pixels in its current direction, tracing a line along the path.

`void turn(double angle)`
: Turns this robot `angle` degrees counterclockwise.

`void switchOff()`
: Removes this robot from its window.

JA.4 `RobotWindow` class

Represents a window in which robots can move.

You must import the `socs` library to use this class.

`RobotWindow()`
: (Constructor method) Constructs an object to represent a window on the screen, 200 pixels wide and 200 pixels tall.

`void notify(String message)`
: Shows the user a dialog box containing `message` and returns after the user clicks a button labeled OK.

`double requestDouble()`
: Shows the user a dialog box prompting for a number, and returns this number.

`double requestDouble(String message)`
: Shows the user a dialog box prompting for a number using `message`, and returns this number.

`int requestInt()`
: Shows the user a dialog box prompting for an integer, and returns this integer.

`int requestInt(String message)`
: Shows the user a dialog box prompting for an integer using `message`, and returns this integer.

`void show()`
: Displays this window on the screen.

JA.5 `String` class

Represents a sequence of characters, like a word or a sentence.

`boolean equals(String other)`
: Return `true` if this string contains the same sequence of characters as `other` does.

`boolean equalsIgnoreCase(String other)`
: Return `true` if this string contains the same sequence of characters as `other` does, treating lowercase letters and capital letters the same.

`int length()`
: Returns the number of characters in this string.

`String substring(int begin)`

Returns a string containing the characters of this string beginning at index `begin` and going to this string's end.

`String substring(int begin, int end)`

Returns a string containing the characters of this string beginning at index `begin` and going up to — but not including — index `end`.*

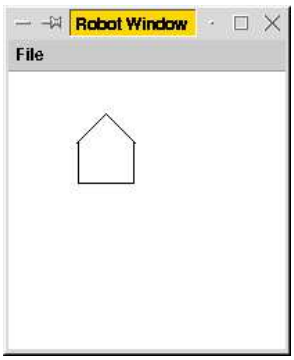
*Omitting the last character named is not what you would expect, but it often turns out to be convenient. For example, the number of characters fetched by the `substring` method is simply `end - begin`.

Appendix JB

Exercise solutions

Exercise J1.2: (page J5) Figure J2.1 (page J11) contains the properly indented program.

Exercise J2.1: (page J14)



Exercise J4.1: (page J27) Expressions (logical)

a. `true` if `x` represents 0 or 1.
b. `true` if `score` exceeds 90, or if it is 89 and `bonus` is `true`.
c. `true` if `k` represents anything other than 1.

Exercise J4.2: (page J27) If `year` is a multiple of 4 but not 100, it is a leap year. It is also a leap year if `year` is a multiple of 400. The following encodes both these cases.

```
(year % 4 == 0 && year % 100 != 0) || year % 400 == 0
```

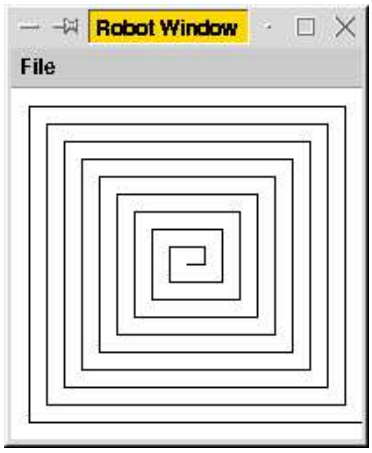
Exercise J4.3: (page J27) It draws the following.

```
import socs.*;

public class TestDivisor {
    public static void main(String[] args) {
        IOWindow io = new IOWindow();
        io.print("What is the numerator? ");
        int num = io.readInt();
        io.print("What is the denominator? ");
        int den = io.readInt();

        if(den == 0) {
            io.println("Cannot divide by 0.");
        } else if(num % den == 0) {
            io.println(den + " divides " + num + ".");
        } else {
            io.println(den + " does not divide " + num + ".");
        }
    }
}
```

Figure JB.1: The `TestDivisor` program.



Exercise J4.4: (page J27) Add the following code after line 10 of Figure J4.2 (page J24).

```
while(length < 5 || length > 195) {
    length = win.requestInt();
}
```

Exercise J5.1: (page J33)

```
5
4
3
2
1
BOOOOOM!
```

Exercise J6.1: (page J41) See Figure JB.1. Notice how it tests whether the denominator is 0 before performing the division. This is a feature of good programs: They are very careful with user input when the user might type something causing a run-time error. It is much better to print something meaningful in these cases than to let the program crash.

```
import socs.*;

public class Median {
    public static void main(String[] args) {
        IOWindow io = new IOWindow();

        // create empty tally boxes
        int[] tally = new int[101];
        int pos = 0;
        while(pos < tally.length) {
            tally[pos] = 0;
            pos++;
        }

        // read and tally the scores
        io.print("How many numbers? ");
        int num_scores = io.readInt();
        pos = 0;
        while(pos < num_scores) {
            io.print("#" + (pos + 1) + ": ");
            int x = io.readInt();
            tally[x]++;
            pos++;
        }

        // now find and print the mode
        int mode = 0;
        pos = 0;
        while(pos < tally.length) {
            if(tally[pos] > tally[mode])
                mode = pos;

            pos++;
        }
        io.println("Mode = " + mode);
    }
}
```

Figure JB.2: The `Median` program.

Exercise J7.1: (page J44) See Figure JB.2.

Exercise J8.2: (page J51)

```
public static int removeDuplicates(int[] arr) {
    int j = 1; // position in array with duplicates removed
    int i = 0;
    while(i < arr.length) {
        if(arr[i] != arr[i - 1]) { // this is not a duplicate
            arr[j] = arr[i]; // put it in array with duplicates removed
            j++;             // increase our position
        }
        i++;
    }
    return j;
}
```

Index