

The first print statement allows us to confirm that `isIn` is looking for the right letter. The second statement confirms that we are looking in the right place.

Now the output looks like this:

```
isIn looking for n
in the String banana
```

Printing the parameters might seem silly, since we know what they are supposed to be. The point is to confirm that they are what we think they are.

3. To traverse the `String`, we can take advantage of the code from Section 7.3. In general, it is a great idea to reuse code fragments rather than writing them from scratch.

```
public static boolean isIn (char c, String s) {
    System.out.println ("isIn looking for " + c);
    System.out.println ("in the String " + s);

    int index = 0;
    while (index < s.length()) {
        char letter = s.charAt (index);
        System.out.println (letter);
        index = index + 1;
    }
    return false;
}
```

Now when we run the program it prints the characters in the `String` one at a time. If all goes well, we can confirm that the loop examines all the letters in the `String`.

4. So far we haven't given much thought to what this method is going to do. At this point we probably need to figure out an algorithm. The simplest algorithm is a linear search, which traverses the vector and compares each element to the target word.

Happily, we have already written the code that traverses the vector. As usual, we'll proceed by adding just a few lines at a time:

```
public static boolean isIn (char c, String s) {
    System.out.println ("isIn looking for " + c);
    System.out.println ("in the String " + s);

    int index = 0;
    while (index < s.length()) {
        char letter = s.charAt (index);
        System.out.println (letter);
        if (letter == c) {
```

```

        System.out.println ("found it");
    }
    index = index + 1;
}
return false;
}

```

As we traverse the String, we compare each letter to the target character. If we find it, we print something, so that when the new code executes it produces a visible effect.

5. At this point we are pretty close to working code. The next change is to return from the method if we find what we are looking for:

```

public static boolean isIn (char c, String s) {
    System.out.println ("isIn looking for " + c);
    System.out.println ("in the String " + s);

    int index = 0;
    while (index < s.length()) {
        char letter = s.charAt (index);
        System.out.println (letter);
        if (letter == c) {
            System.out.println ("found it");
            return true;
        }
        index = index + 1;
    }
    return false;
}

```

If we find the target character, we return **true**. If we get all the way through the loop without finding it, then the correct return value is **false**.

If we run the program at this point, we should get

```

isIn looking for n
in the String banana
b
a
n
found it
true

```

6. The next step is to make sure that the other test cases work correctly. First, we should confirm that the method returns **false** if the character is not in the String. Then we should check some of the typical troublemakers, like an empty String, "", or a String with a single character.

As always, this kind of testing can help find bugs if there are any, but it can't tell you if the method is correct.

7. The penultimate step is to remove or comment out the print statements.

```
public static boolean isIn (char c, String s) {
    int index = 0;
    while (index < s.length()) {
        char letter = s.charAt (index);
        if (letter == c) {
            return true;
        }
        index = index + 1;
    }
    return false;
}
```

Commenting out the print statements is a good idea if you think you might have to revisit this method later. But if this is the final version of the method, and you are convinced that it is correct, you should remove them.

Removing the comments allows you to see the code most clearly, which can help you spot any remaining problems.

If there is anything about the code that is not obvious, you should add comments to explain it. Resist the temptation to translate the code line by line. For example, no one needs this:

```
// if letter equals c, return true
if (letter == c) {
    return true;
}
```

You should use comments to explain non-obvious code, to warn about conditions that could cause errors, and to document any assumptions that are built into the code. Also, before each method, it is a good idea to write an abstract description of what the method does.

8. The final step is to examine the code and see if you can convince yourself that it is correct. At this point we know that the method is syntactically correct, because it compiles. To check for run time errors, you should find every statement that can cause an error and figure out what conditions cause the error.

In this method, the only statement that can cause a run-time error is `s.charAt (index)`. This statement will fail if `s` is `null` or if the index is out of bounds. Since we get `s` as a parameter, we can't be sure that it is not `null`; all we can do is check. In general, it is a good idea for methods to make sure their parameters are legal. The structure of the `while` loop ensures that `index` is always between 0 and `s.length-1`. If we check all the problem conditions, or prove that they cannot happen, then we can prove that this method will not cause a run time error.

We haven't proven yet that the method is semantically correct, but by proceeding incrementally, we have avoided many possible errors. For example, we already know that the method is getting parameters correctly and that the loop traverses the entire String. We also know that it is comparing characters successfully, and returning `true` if it finds the target. Finally, we know that if the loop exits, the target is not in the String.

Short of a formal proof, that is probably the best we can do.

## Appendix B

# Debugging

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

- Compile-time errors are produced by the compiler and usually indicate that there is something wrong with the syntax of the program. Example: omitting the semi-colon at the end of a statement.
- Run-time errors are produced by the run-time system if something goes wrong while the program is running. Most run-time errors are Exceptions. Example: an infinite recursion eventually causes a `StackOverflowException`.
- Semantic errors are problems with a program that compiles and runs, but doesn't do the right thing. Example: an expression may not be evaluated in the order you expect, yielding an unexpected result.

The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, there are some techniques that are applicable in more than one situation.

### B.1 Compile-time errors

The best kind of debugging is the kind you don't have to do, because you avoid making errors in the first place. In the previous section, I suggested a program development plan that minimizes the number of errors you will make and makes it easy to find them when you do. The key is to start with a working program and add small amounts of code at a time. That way, when there is an error, you will have a pretty good idea where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, I make some suggestions about how to proceed.

### **The compiler is spewing error messages.**

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it fails, and it reports spurious errors.

In general, only the first error message is reliable. I suggest that you only fix one error at a time, and then recompile the program. You may find that one semi-colon "fixes" 100 errors. Of course, if you see several legitimate error messages, you might as well fix more than one bug per compilation attempt.

### **I'm getting a weird compiler message and it won't go away.**

First of all, read the error message carefully. It is written in terse jargon, but often there is a kernel of information there that is carefully hidden.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally the error will be prior to the location of the error message, but there are cases where it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition.

If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Now is a good time to go through the whole program and make sure it is indented properly. I won't say that good indentation makes it easy to find syntax errors, but bad indentation sure makes it harder.

Now, start examining the code for the common syntax errors.

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.
2. Remember that upper case letters are not the same as lower case letters.
3. Check for semi-colons at the end of statements (and no semi-colons after squiggly-braces).

4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double-quotes for Strings and single quotes for characters.
5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).
6. For each method invocation, make sure that the arguments you provide are in the right order, and have right type, and that the object you are invoking the method on is the right type.
7. If you are invoking a fruitful method, make sure you are doing something with the result. If you are invoking a void method, make sure you are not *trying* to do something with the result.
8. If you are invoking an object method, make sure you are invoking it on an object with the right type. If you are invoking a class method from outside the class where it is defined, make sure you specify the class name.
9. Inside an object method you can refer to the instance variables without specifying an object. If you try that in a class method, you will get a confusing message like, “Static reference to non-static variable.”

If nothing works, move on to the next section...

### I can't get my program to compile no matter what I do.

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your development environment to make sure the program you are editing is the program the compiler is compiling. If you are not sure, try putting an obvious and deliberate syntax error right at the beginning of the program. Now compile again. If the compiler doesn't find the new error, there is probably something wrong with the way you set up the project.

Otherwise, if you have examined the code thoroughly, it is time for desperate measures. You should start over with a program that you can compile and then gradually add your code back.

- Make a copy of the file you are working on. If you are working on `Fred.java`, make a copy called `Fred.java.old`.
- Delete about half the code from `Fred.java`. Try compiling again.
  - If the program compiles now, then you know the error is in the other half. Bring back about half of the code you deleted and repeat.
  - If the program still doesn't compile, the error must be in this half. Delete about half of the code and repeat.

- Once you have found and fixed the error, start bringing back the code you deleted, a little bit at a time.

This process is called “debugging by bisection.” As an alternative, you can comment out chunks of code instead of deleting them. For really sticky syntax problems, though, I think deleting is more reliable—you don’t have to worry about the syntax of the comments, and by making the program smaller you make it more readable.

### **I did what the compiler told me to do, but it still doesn’t work.**

Some compiler messages come with tidbits of advice, like “class `Golfer` must be declared abstract. It does not define `compareTo(java.lang.Object)` from interface `java.lang.Comparable`.” It sounds like the compiler is telling you to declare `Golfer` as an abstract class, and if you are reading this book, you probably don’t know what that is or how to do it.

Fortunately, the compiler is wrong. The solution in this case is to make sure `Golfer` has a method called `compareTo` that takes an `Object` as a parameter.

In general, don’t let the compiler lead you by the nose. Error messages give you evidence that something is wrong, but they can be misleading, and their “advice” is often wrong.

## **B.2 Run-time errors**

### **My program hangs.**

If a program stops and seems to be doing nothing, we say it is “hanging.” Often that means that it is caught in an infinite loop or an infinite recursion.

- If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says “entering the loop” and another immediately after that says “exiting the loop.”

Run the program. If you get the first message and not the second, you’ve got an infinite loop. Go to the section titled “Infinite loop.”

- Most of the time an infinite recursion will cause the program to run for a while and then produce a `StackOverflowException`. If that happens, go to the section titled “Infinite recursion.”

If you are not getting a `StackOverflowException`, but you suspect there is a problem with a recursive method, you can still use the techniques in the infinite recursion section.

- If neither of those things works, start testing other loops and other recursive methods.



- If none of these suggestions helps, then it is possible that you don't understand the flow of execution in your program. Go to the section titled "Flow of execution."

### Infinite loop

If you think you have an infinite loop and think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition, and the value of the condition.

For example,

```
while (x > 0 && y < 0) {  
    // do something to x  
    // do something to y  
  
    System.out.println ("x: " + x);  
    System.out.println ("y: " + y);  
    System.out.println ("condition: " + (x > 0 && y < 0));  
}
```

Now when you run the program you will see three lines of output for each time through the loop. The last time through the loop, the condition should be **false**. If the loop keeps going, you will be able to see the values of **x** and **y** and you might figure out why they are not being updated correctly.

### Infinite recursion

Most of the time an infinite recursion will cause the program to run for a while and then produce a `StackOverflowException`.

If you know that a method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.

If there is a base case, but the program doesn't seem to be reaching it, add a print statement at the beginning of the method that prints the parameters. Now when you run the program you will see a few lines of output every time the method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

### Flow of execution

If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each method with a message like "entering method foo," where **foo** is the name of the method.

Now when you run the program it will print a trace of each method as it is invoked.

It is often useful to print the parameters each method receives when it is invoked. When you run the program, check whether the parameters are reasonable, and check for one of the classic errors—providing parameters in the wrong order.

### When I run the program I get an Exception.

If something goes wrong during run time, the Java run-time system prints a message that includes the name of the exception, the line of the program where the problem occurred, and a stack trace.

The stack trace includes the method that is currently running, and then the method that invoked it, and then the method that invoked *that*, and so on. In other words, it traces the stack of method invocations that got you to where you are.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened.

**NullPointerException:** You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out what variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an object type, it is initially `null`, until you assign a value to it. For example, this code causes a `NullPointerException`:

```
Point blank;  
System.out.println (blank.x);
```

**ArrayIndexOutOfBoundsException:** The index you are using to access an array is either negative or greater than `array.length-1`. If you can find the site where the problem is, add a print statement immediately before it to print the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backwards through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing.

If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

**StackOverflowException:** See “Infinite recursion.”

### I added so many print statements I get inundated with output.

One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: either simplify the output or simplify the program.

To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand. As you develop a program, you will conceive ways to visualize the execution of the program, and develop code that generates concise, informative visualizations.

To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a *small* array. If the program takes input from the user, give it the simplest input that causes the error.

Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the error is in a deeply-nested part of the program, try rewriting that part with simpler structure. If you suspect a large method, try splitting it into smaller methods and testing them separately.

Often the process of finding the minimal test case leads you to the bug. For example, if you find that a program works when the array has an even number of elements, but not when it has an odd number, that gives you a clue about what is going on.

Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

## B.3 Semantic errors

### My program doesn't work.

In some ways semantic errors are the hardest, because the compiler and the run-time system provide no information about what is wrong. Only you know what the program was supposed to do, and only you know that it isn't doing it.

The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that makes this hard is that computers run so fast. You will often wish that you could slow the program down to human speed, but there is no straightforward way to do that, and even if there were, it is not really a good way to debug.

Here are some questions to ask yourself:

- Is there something the program was supposed to do, but doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should. Add a print statement to the beginning of the suspect methods.
- Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.

- Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to built-in Java methods. Read the documentation for the methods you invoke. Try out the methods by invoking the methods directly with simple test cases, and check the results.

In order to program, you need to have a mental model of how programs work. If your program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.

The best way to correct your mental model is to break the program into its components (usually the classes and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.

Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

Here are some common semantic errors that you might want to check for:

- If you use the assignment operator, `=`, instead of the equality operator, `==`, in the condition of an `if`, `while` or `for` statement, you might get an expression that is syntactically legal, but it doesn't do what you expect.
- When you apply the equality operator, `==`, to an object, it checks shallow equality. If you meant to check deep equality, you should use the `equals` method (or define one, for user-defined objects).
- Some Java libraries expect user-defined objects to define methods like `equals`. If you don't define them yourself, you will inherit the default behavior from the parent class, which may not be what you want.
- Inheritance can lead to subtle semantic errors, because you may be executing inherited code without realizing it. To make sure you understand the flow of execution in your program, see the section titled "Flow of Execution."

### **I've got a big hairy expression and it doesn't do what I expect.**

Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
rect.setLocation (rect.getLocation().translate  
                  (-rect.getWidth(), -rect.getHeight()));
```

Can be rewritten as

```
int dx = -rect.getWidth();
int dy = -rect.getHeight();
Point location = rect.getLocation();
Point newLocation = location.translate (dx, dy);
rect.setLocation (newLocation);
```

The explicit version is easier to read, because the variable names provide additional documentation, and easier to debug, because we can check the types of the intermediate variables and display their values.

Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression  $\frac{x}{2\pi}$  into Java, you might write

```
double y = x / 2 * Math.PI;
```

That is not correct, because multiplication and division have the same precedence, and are evaluated from left to right. So this expression computes  $x\pi/2$ .

A good way to debug expressions is to add parentheses to make the order of evaluation explicit.

```
double y = x / (2 * Math.PI);
```

Any time you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intend); it will also be more readable for other people who haven't memorized the rules of precedence.

### **I've got a method that doesn't return what I expect.**

If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of

```
public Rectangle intersection (Rectangle a, Rectangle b) {
    return new Rectangle (
        Math.min (a.x, b.x),
        Math.min (a.y, b.y),
        Math.max (a.x+a.width, b.x+b.width)-Math.min (a.x, b.x)
        Math.max (a.y+a.height, b.y+b.height)-Math.min (a.y, b.y) );
}
```

You could write

```
public Rectangle intersection (Rectangle a, Rectangle b) {
    int x1 = Math.min (a.x, b.x);
    int y2 = Math.min (a.y, b.y);
    int x2 = Math.max (a.x+a.width, b.x+b.width);
    int y2 = Math.max (a.y+a.height, b.y+b.height);
    Rectangle rect = new Rectangle (x1, y1, x2-x1, y2-y1);
    return rect;
}
```

Now you have the opportunity to display any of the intermediate variables before returning.

## My print statement isn't doing anything

If you use the `println` method, the output gets displayed immediately, but if you use `print` (at least in some environments) the output gets stored without being displayed until the next newline character gets output. If the program terminates without producing a newline, you may never see the stored output.

If you suspect that this is happening to you, try changing some or all of the `print` statements to `println`.

## I'm really, really stuck and I need help

First of all, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing the following symptoms:

- Frustration and/or rage.
- Superstitious beliefs (“the computer hates me”) and magical thinking (“the program only works when I wear my hat backwards”).
- Random walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

Sometimes it just takes time to find a bug. I often find bugs when I am away from the computer and I let my mind wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

## No, I really need help.

It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.

Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.

When you bring someone in to help, be sure to give them the information they need.

- What kind of bug is it? Compile-time, run-time, or semantic?

- If the bug occurs at compile-time or run-time, what is the error message, and what part of the program does it indicate?
- What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
- What have you tried so far, and what have you learned?

Often you will find that by the time you have explained the problem to someone else, you will see the answer. This phenomenon is so pervasive that some people recommend a debugging technique called “rubber ducking.” Here’s how it works:

1. Buy a standard-issue rubber duck.
2. When you are really stuck on a problem, put the rubber duck on the desk in front of you and say, “Rubber duck, I am stuck on a problem. Here’s what’s happening...”
3. Explain the problem to the rubber duck.
4. See the solution.
5. Thank the rubber duck.

### **I found the bug!**

Most often, when you find a bug, it is obvious how to fix it. But not always. Sometimes what seems to be a bug is really an indication that you don’t really understand the program, or there is an error in your algorithm. In these cases, you might have to rethink the algorithm, or adjust your mental model of the program. Take some time away from the computer to think about the program, work through some test cases by hand, or draw diagrams to represent the computation.

When you fix a bug, don’t just dive in and start making new errors. Take a second to think about what kind of bug it was, why you made the error in the first place, how the error manifested itself, and what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.





## Appendix C

# Input and Output in Java

### System objects

`System` is the name of the built-in class that contains methods and objects used to get input from the keyboard, print text on the screen, and do file input/output (I/O).

`System.out` is the name of the object we use to display text on the screen. When you invoke `print` and `println`, you invoke them on the object named `System.out`.

Interestingly, you can print `System.out`:

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

The output is:

```
java.io.PrintStream@80cc0e5
```

As usual, when Java prints an object, it prints the type of the object, which is `PrintStream`, the package in which that type is defined, `java.io`, and a unique identifier for the object. On my machine the identifier is `80cc0e5`, but if you run the same code, you will probably get something different.

There is also an object named `System.in` that has type `BufferedInputStream`. `System.in` makes it possible to get input from the keyboard. Unfortunately, it does not make it easy to get input from the keyboard.

### Keyboard input

First, you have to use `System.in` to create a new `InputStreamReader`.

```
InputStreamReader in = new InputStreamReader (System.in);
```

Then you use `in` to create a new `BufferedReader`:

```
BufferedReader keyboard = new BufferedReader (in);
```

The point of all this manipulation is that there is a method you can invoke on a `BufferedReader`, called `readLine`, that gets input from the keyboard and converts it into a `String`. For example:

```
String s = keyboard.readLine ();
System.out.println (s);
```

reads a line from the keyboard and prints the result.

There is only one problem. There are things that can go wrong when you invoke `readLine`, and they might cause an `IOException`. There is a rule in Java that if a method might cause an exception, it should say so. The syntax looks like this:

```
public static void main (String[] args) throws IOException {
    // body of main
}
```

This indicates that `main` might “throw” an `IOException`. You can think of throwing an exception as similar to throwing a tantrum.

## File input

Reading input from a file is equally stupid. Here is an example:

```
public static void main (String[] args)
    throws FileNotFoundException, IOException {

    processFile ("/usr/dict/words");
}

public static void processFile (String filename)
    throws FileNotFoundException, IOException {

    FileReader fileReader = new FileReader (filename);
    BufferedReader in = new BufferedReader (fileReader);

    while (true) {
        String s = in.readLine();
        if (s == null) break;
        System.out.println (s);
    }
}
```

This program reads each line of the named file (`/usr/dict/words`) into a `String` and then prints the line. Again, the declaration `throws FileNotFoundException, IOException` is required by the compiler. The object types `FileReader` and `BufferedReader` are part of the insanely complicated class hierarchy Java uses to do incredibly common, simple things. Other than that, there is not much value in the details of how this code fragment works.

## Appendix D

# Graphics

### D.1 Slates and Graphics objects

There are a number of ways to create graphics in Java, some more complicated than others. To keep things simple, I have created a object type called **Slate** that represents a surface you can draw on. When you create a **Slate**, a new blank window appears. The **Slate** contains a **Graphics** object, which you use to draw on the slate.

The methods that pertain to **Graphics** objects are defined in the built-in **Graphics** class. The methods that pertain to **Slates** are defined in the **Slate** class, which is shown in Section D.6.

Use the **new** operator to create a new **Slate** object:

```
Slate slate = new Slate (500, 500);
```

The parameters are the width and height of the window. The return value gets assigned to a variable named **slate**. There is no conflict between the name of the class (with an upper-case “S”) and the name of the variable (with a lower-case “s”).

The next method we need is **getSlateGraphics**, which returns a **Graphics** object. You can think of a **Graphics** object as a piece of chalk.

```
Graphics g = slate.getSlateGraphics ();
```

Using the name **g** is conventional, but we could have called it anything.

### D.2 Invoking methods on a Graphics object

In order to draw things on the screen, you invoke methods on the graphics object.

```
g.setColor (Color.black);
```

`setColor` changes the current color, in this case to black. Everything that gets drawn will be black, until we use `setColor` again.

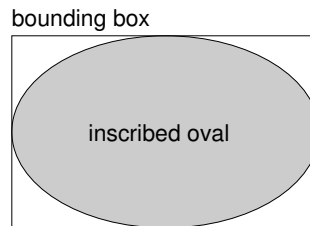
`Color.black` is a special value provided by the `Color` class, just as `Math.PI` is a special value provided by the `Math` class. `Color`, you will be happy to hear, provides a palette of other colors, including:

<code>black</code>	<code>blue</code>	<code>cyan</code>	<code>darkGray</code>	<code>gray</code>	<code>lightGray</code>
<code>magenta</code>	<code>orange</code>	<code>pink</code>	<code>red</code>	<code>white</code>	<code>yellow</code>

To draw on the `Slate`, we can invoke `draw` methods on the `Graphics` object. For example:

```
g.drawOval (x, y, width, height);
```

`drawOval` takes four integers as arguments. These arguments specify a **bounding box**, which is the rectangle in which the oval will be drawn (as shown in the figure). The bounding box itself is not drawn; only the oval is. The bounding box is like a guideline. Bounding boxes are always oriented horizontally or vertically; they are never at a funny angle.

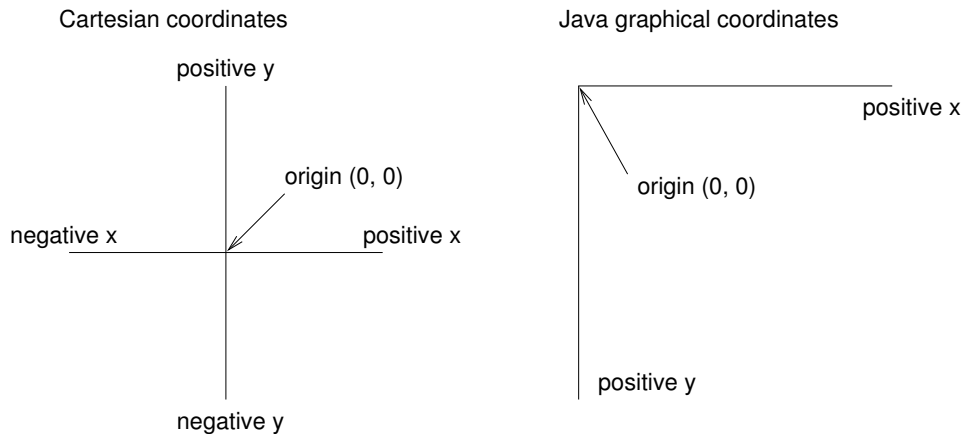


If you think about it, there are lots of ways to specify the location and size of a rectangle. You could give the location of the center or any of the corners, along with the height and width. Or, you could give the location of opposing corners. The choice is arbitrary, but in any case it will require the same number of parameters: four.

By convention, the usual way to specify a bounding box is to give the location of the *upper-left* corner and the width and height. The usual way to specify a location is to use a **coordinate system**.

## D.3 Coordinates

You are probably familiar with Cartesian coordinates in two dimensions, in which each location is identified by an x-coordinate (distance along the x-axis) and a y-coordinate. By convention, Cartesian coordinates increase to the right and up, as shown in the figure.



Annoyingly, it is conventional for computer graphics systems to use a variation on Cartesian coordinates in which the origin is in the upper-left corner of the screen or window, and the direction of the positive y-axis is *down*. Java follows this convention.

The unit of measure is called a **pixel**; a typical screen is about 1000 pixels wide. Coordinates are always integers. If you want to use a floating-point value as a coordinate, you have to round it off to an integer (See Section 3.2).

## D.4 A lame Mickey Mouse

Let's say we want to draw a picture of Mickey Mouse. We can use the oval we just drew as the face, and then add ears. Before we do that it is a good idea to break the program up into two methods. `main` will create the `Slate` and `Graphics` objects and then invoke `draw`, which does the actual drawing.

When we are done invoking `draw` methods, we have to invoke `slate.repaint` to make the changes appear on the screen.

```
public static void main (String[] args) {
    int width = 500;
    int height = 500;

    Slate slate = Slate.makeSlate (width, height);
    Graphics g = Slate.getGraphics (slate);

    g.setColor (Color.black);
    draw (g, 0, 0, width, height);
    slate.repaint ();
}

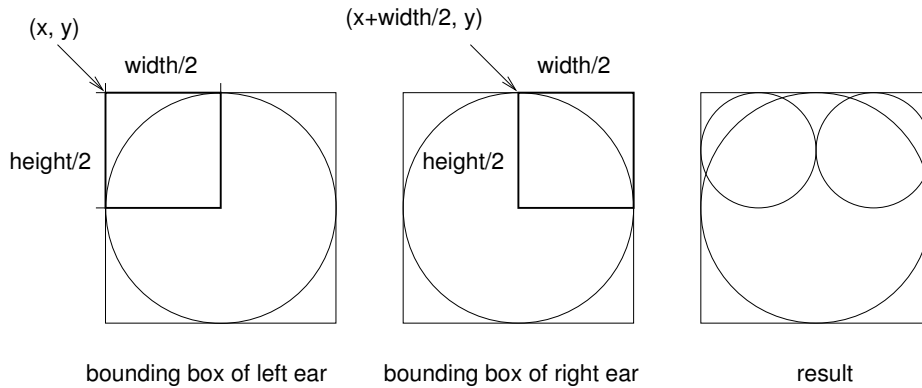
public static void draw
    (Graphics g, int x, int y, int width, int height) {
    g.drawOval (x, y, width, height);
}
```

```

    g.drawOval (x, y, width/2, height/2);
    g.drawOval (x+width/2, y, width/2, height/2);
}

```

The parameters for `draw` are the `Graphics` object and a bounding box. `draw` invokes `drawOval` three times, to draw Mickey's face and two ears. The following figure shows the bounding boxes for the ears.



As shown in the figure, the coordinates of the upper-left corner of the bounding box for the left ear are  $(x, y)$ . The coordinates for the right ear are  $(x+width/2, y)$ . In both cases, the width and height of the ears are half the width and height of the original bounding box.

Notice that the coordinates of the ear boxes are all relative to the location  $(x$  and  $y)$  and size (`width` and `height`) of the original bounding box. As a result, we can use `draw` to draw a Mickey Mouse (albeit a lame one) anywhere on the screen in any size.

**Exercise D.1** Modify the arguments passed to `draw` so that Mickey is one half the height and width of the screen, and centered.

## D.5 Other drawing commands

Another drawing command with the same parameters as `drawOval` is

```
drawRect (int x, int y, int width, int height)
```

Here I am using a standard format for documenting the name and parameters of methods. This information is sometimes called the method's **interface** or **prototype**. Looking at this prototype, you can tell what types the parameters are and (based on their names) infer what they do. Here's another example:

```
drawLine (int x1, int y1, int x2, int y2)
```

The use of parameter names `x1`, `x2`, `y1` and `y2` suggests that `drawLine` draws a line from the point  $(x1, y1)$  to the point  $(x2, y2)$ .

One other command you might want to try is

```
drawRoundRect (int x, int y, int width, int height,  
               int arcWidth, int arcHeight)
```

The first four parameters specify the bounding box of the rectangle; the remaining two parameters indicate how rounded the corners should be, specifying the horizontal and vertical diameter of the arcs at the corners.

There are also “fill” versions of these commands, that not only draw the outline of a shape, but also fill it in. The interfaces are identical; only the names have been changed:

```
fillOval (int x, int y, int width, int height)  
fillRect (int x, int y, int width, int height)  
fillRoundRect (int x, int y, int width, int height,  
               int arcWidth, int arcHeight)
```

There is no such thing as `fillLine`—it just doesn’t make sense.

## D.6 The Slate Class

```
import java.awt.*;  
  
class Example {  
  
    // demonstrate simple use of the Slate class  
  
    public static void main (String[] args) {  
        int width = 500;  
        int height = 500;  
  
        Slate slate = new Slate (width, height);  
        Graphics g = slate.getSlateGraphics ();  
  
        g.setColor (Color.blue);  
  
        draw (g, 0, 0, width, height);  
        slate.repaint ();  
  
        anim (slate);  
    }  
  
    // draw demonstrates a recursive pattern  
  
    public static void draw (Graphics g, int x, int y, int width, int height) {  
        if (height < 3) return;  
  
        g.drawOval (x, y, width, height);  
  
        draw (g, x, y+height/2, width/2, height/2);  
    }  
}
```

```

        draw (g, x+width/2, y+height/2, width/2, height/2);
    }

    // anim demonstrates a simple animation

    public static void anim (Slate slate) {
        Graphics g = slate.image.getGraphics ();
        g.setColor (Color.red);

        for (int i=-100; i<500; i+=10) {
            g.drawOval (i, 100, 100, 100);
            slate.repaint ();
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {}
        }
    }
}

class Slate extends Frame {

    // image is a buffer: when Slate users draw things, they
    // draw on the buffer.  When the Slate gets painted, we
    // copy the image onto the screen.
    Image image;

    public Slate (int width, int height) {
        setBounds (100, 100, width, height);
        setBackground (Color.white);
        setVisible (true);
        image = createImage (width, height);
    }

    // when a Slate user asks for a Graphics object, we give
    // them one from the off-screen buffer.

    public Graphics getSlateGraphics () {
        return image.getGraphics ();
    }

    // normally update erases the screen and invokes paint, but
    // since we are overwriting the whole screen anyway, it is
    // slightly faster to override update and avoid clearing the
    // screen

    public void update (Graphics g) {
        paint (g);
    }
}

```



```

    }

    // paint copies the off-screen buffer onto the screen

    public void paint (Graphics g) {
        if (image == null) return;
        g.drawImage (image, 0, 0, null);
    }
}

```

### Exercise D.2

The purpose of this assignment is to practice using methods as a way of organizing and encapsulating complex tasks.

WARNING: It is very important that you take this assignment one step at a time, and get each step working correctly before you proceed. More importantly, make sure you understand each step before you proceed.

- a. Create a new program named **Snowperson.java**, Type in the code or get it from . Run it. A new window should appear with a bunch of blue circles and a bunch of red circles. This window is the Slate.  
Unlike the other programs we ran, there is no console window. However, if you add a **print** or **println** statement, the console window will appear. For graphical applications, the console is useful for debugging.  
I found that I could not close the Slate window by clicking on it, which is probably good, because it will remind you to quit from the interpreter every time you run the program.
- b. Look over the source code as it currently exists and make sure you understand all the code in **draw**. The method named **anim** is there for your entertainment, but we will not be using it for this assignment. You should remove it.
- c. The **Slate** class appears immediately after the **cs151** class. You might want to check it out, although a lot of it will not make sense at this point.
- d. Fiddle with the statements in **draw** and see what effect your changes have. Try out the various drawing commands. For more information about them, see <http://java.sun.com/products/jdk/1.1/docs/api/java.awt.Graphics.html>
- e. Change the width or height of the **Slate** and run the program again. You should see that the image adjusts its size and proportions to fit the size of the window. You are going to write drawing programs that do the same thing. The idea is to use variables and parameters to make programs more general; in this case the generality is that we can draw images that are any size or location.

### Use a bounding box

The arguments that **draw** receives (not including the graphic object) make up a **bounding box**. The bounding box specifies the invisible rectangle in which **draw** should draw.

- a. Inside **draw**, create a new bounding box that is the same height as the **Slate** but only one-third of the width, and centered. When I say “create a bounding box” I mean define four local variables that will contain the location and size. You are going to pass this bounding box as an argument to **drawSnowperson**.

- b. Create a new method named **drawSnowperson** that takes the same parameters as **draw**. To start, it should draw a single oval that fills the entire bounding box (in other words, it should have the same position and location as the bounding box).
- c. Change the size of the **Slate** and run your program again. The size and proportion of the oval should adjust so that, no matter what the size the **Slate** is, the oval is the same height, one-third of the width, and centered.

### Make a snowperson

- a. Modify **drawSnowperson** so that it draws three ovals stacked on top of each other like a snowperson. The height and width of the snowperson should fill the bounding box, as in the figure on the quiz.
- b. Change the size of the **Slate** and run your program again. Again, the snowperson should adjust so that the snowperson always touches the top and bottom of the screen, and the three ovals touch, and the proportion of the three ovals stays the same.
- c. At this point, show your program to me so that I can make sure you understand the basic ideas behind this assignment. You should definitely not continue work on this assignment until I have seen your program.

### Draw the snowperson *American Gothic*

- a. Modify **draw** so that it draws two snowpeople side by side. One of them should be the same height as the window; the other should be 90% of the window height. (Snowpeople exhibit a sexual dimorphism in which females are roughly 10% smaller than males.)  
Their widths should be proportional to their heights, and their bounding boxes should be adjacent (which means that the drawn ovals probably will not quite touch). The pair should be centered, meaning that there is the same amount of space on each side.

### With a corn-cob pipe...

- a. Write a method called **drawFace** that takes the same number and type of parameters as **drawSnowperson**, and that draws a simple face within the given bounding box. Two eyes will suffice, but you can get as elaborate as you like. Again, as you resize the window, the proportions of the face should not change, and all the features should fit within the oval (with the possible exception of ears).
- b. Modify **drawSnowperson** so that it invokes **drawFace** in order to fill in the top oval (the face). The bounding box you pass to **drawFace** should be the same as the bounding box that created the face oval.

### Give the Snowpeople T-shirts

- a. Add a new method called **drawShirt** that takes the usual parameters and that draws some sort of T-shirt logo within the bounding box. You can use any of

the drawing commands, although you might want to avoid `drawString` because it is not easy to guarantee that the string will fit within the bounding box.

- b. Modify `drawSnowperson` so that it invokes `drawShirt` in order to emblazon something on the chest (middle oval) of each snowperson.

### Make Mrs. Snowperson pregnant

- a. I realize that this is a little risqué, but there is a point. Modify `draw` so that after drawing Mrs. Snowperson, it draws a baby snowperson inside her abdomen oval.

Notice that adding something to `draw` affects just one snowperson. Adding something to `drawSnowperson` affects all snowpeople.

Note: The next part of the assignment will not make sense until after class on Friday, so I recommend stopping here (if you get this far in lab).

### Make all snowpeople pregnant

Ok, now let's imagine that instead of making a particular snowperson pregnant, we want to make all snowpeople pregnant. Instead of adding a line to `draw`, we would add a line to `drawSnowperson`. What would that line do? It would invoke `drawSnowperson`! Can you do that? Can you invoke a method from within itself? Well, yes, you can, but you have to be careful.

SO DON'T DO THIS YET!!!

Think for a minute. What if we draw a big snowperson, and then put a small snowperson in the abdomen oval. Then we have to put an even smaller snowperson in the small abdomen oval, and so on and so on. The problem is, it would never stop. We would be drawing smaller and smaller snowpeople until doomsday!

One solution is to pick a minimum snowperson size, and say that below that size, we refuse to draw any more snowpeople.

- a. Add a line at the beginning of `drawSnowperson` that checks the height of the bounding box and returns immediately if it is less than 10.

```
if (height < 10) return;
```

- b. Now that you have that line, it is safe to add code at the end of the method so that after drawing the ovals, and the face, and the t-shirt, it invokes `drawSnowperson` to put a small snowperson in the abdomen.

### Exercise D.3

- a. Create a new program named `Moire.java`.
- b. Add the following code to your project, and replace the contents of `draw` with a single line that invokes `moire`.

```

public static void moire
    (Graphics g, int x, int y, int width, int height) {
    int i = 1;
    while (i < width) {
        g.drawOval (0, 0, i, i);
        i = i + 2;
    }
}

```

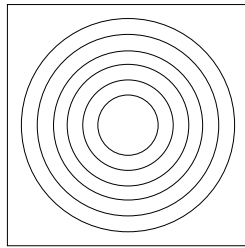
- c. Look at the code before you run it and draw a sketch of what you expect it to do. Now run it. Did you get what you expected? For a partial explanation of what is going on, see the following:

<http://math.hws.edu/xJava/other/Moire1.html>

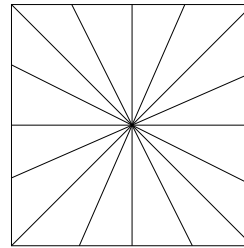
<http://tqd.advanced.org/3543/moirelesson.html>

- d. Modify the program so that the space between the circles is larger or smaller. See what happens to the image.
- e. Modify the program so that the circles are drawn in the center of the screen and concentric, as in the following figure. Unlike in the figure, the distance between the circles should be small enough that the Moiré interference is apparent.

concentric circles



radial Moire pattern



- f. Write a method named **radial** that draws a radial set of line segments as shown in the figure, but they should be close enough together to create a Moiré pattern.
- g. Just about any kind of graphical pattern can generate Moiré-like interference patterns. Play around and see what you can create.