How Java Works

by Marshall Brain

Have you ever wondered how computer programs work? Have you ever wanted to learn how to write your own computer programs? Whether you are 14 years old and hoping to learn how to write your first game, or you are 70 years old and have been curious about computer programming for 20 years, this article is for you. In this edition of **HowStuffWorks**, I'm going to teach you how computer programs work by teaching you how to program in the **Java programming language**.

In order to teach you about computer programming, I am going to make several assumptions from the start:

- I am going to assume that you know nothing about computer programming now. If you already know something then the first part of this article will seem elementary to you. Please feel free to skip forward until you get to something you don't know.
- I am going to assume you *do* know something about the computer you are using. That is, I am going to assume you already know how to edit a file, copy and delete files, rename files, find information on your system, etc.
- For simplicity, I am going to assume that you are using a machine running Windows 95, 98, 2000, NT or XP. It should be relatively straightforward for people running other operating systems to map the concepts over to those.
- I am going to assume that you have a desire to learn.

All of the tools you need to start programming in Java are widely available on the Web for free. There is also a huge amount of educational material for Java available on the Web, so once you finish this article you can easily go learn more to advance your skills. You can learn Java programming here without spending any money on compilers, development environments, reading materials, etc. Once you learn Java it is easy to learn other languages, so this is a good place to start.

Having said these things, we are ready to go. Let's get started!

A Little Terminology

Keep in mind that I am assuming that you know nothing about programming. Here are several vocabulary terms that will make things understandable:

- **Computer program** - A computer program is a set of instructions that tell a computer exactly what to do. The instructions might tell the computer to add up a set of numbers, or compare two numbers and make a decision based on the result, or whatever. But a computer program is simply a set of instructions for the computer, like a recipe is a set of instructions for a cook or musical notes are a set of instructions for a musician. The computer follows your instructions exactly and in the process does something useful -- like balancing a checkbook or displaying a game on the screen or implementing a word processor.
- **Programming language** - In order for a computer to recognize the instructions you give it, those instructions need to be written in a language the computer understands -- a programming language. There are many computer programming languages -- Fortran, Cobol, Basic, Pascal, C, C++, Java, Perl -- just like there are many spoken languages. They all express approximately the same concepts in different ways.
- **Compiler** - A compiler translates a computer program written in a human-readable computer language (like Java) into a form that a computer can **execute**. You have probably seen EXE files on your computer. These EXE files are the output of compilers. They contain **executables** -- machine-readable programs translated from human-readable programs.

In order for you to start writing computer programs in a programming language called Java, you need a compiler for the Java language. The next section guides you through the process of downloading and installing a compiler. Once you have a compiler, we can get started. This process is going to take several hours, much of that time being download time for several large files. You are also going to need about 40 megabytes of free disk space (make sure you have the space available before you get started).

Downloading the Java Compiler

In order to get a Java development environment set up on your machine -- you "develop" (write) computer programs using a "development environment" -- you will have to complete the following steps:

1. Download a large file containing the Java development environment (the compiler and other tools).
2. Download a large file containing the Java documentation.
3. If you do not already have WinZip (or an equivalent) on your machine, you will need to download a large file containing WinZip and install it.
4. Install the Java development environment.
5. Install the documentation.
6. Adjust several environment variables.
7. Test everything out.

Before getting started, it would make things easier if you create a new directory in your temp directory to hold the files we are about to download. We will call this the **download directory**.

Step 1 - Download the Java development environment

Go to the page http://java.sun.com/j2se/1.4.1/download.html. Download the SDK software by selecting your operating system and clicking the link on the next page. You will be shown a licensing agreement. Click Accept. Download the file to your download directory. This is a huge file -- almost 35 megabytes -- and it will take several hours to download over a normal phone-line modem. The next two files are also large.

Step 2 - Download the Java documentation

Download the documentation by selecting your operating system and clicking the SDK 1.4.1 documentation link.

Step 3 - Download and install WinZip

If you do not have a version of WinZip or an equivalent on your machine, go to the page http://www.winzip.com/ and download an evaluation copy of WinZip. Run the EXE you get to install it. We will use it in a moment to install the documentation.

Step 4 - Install the development kit

Run the j2sdk-1_4_1-*.exe file that you downloaded in step 1. It will unpack and install the development kit automatically.

Step 5 - Install the documentation

Read the installation instructions for the documentation. They will instruct you to move the documentation file to same directory as that containing the development kit you just installed. Unzip the documentation and it will drop into the proper place.

Step 6 - Adjust your environment

As instructed on this page, you need to change your path variable. This is most easily done by opening an MS-DOS prompt and typing PATH to see what the path is set to currently. Then open autoexec.bat in Notepad and make the changes to PATH specified in the instructions.

Step 7 - Test

Now you should be able to open another MS-DOS window and type **javac**. If everything is set up properly, then you should see a two-line blob of text come out that tells you how to use **javac**. That means you are ready to go. If you see the message "Bad Command or File Name" it means you are not ready to go. Figure out what you did wrong by rereading the installation instructions. Make sure the PATH is set properly and working. Go back and reread the Programmer's Creed above and be persistent until the problem is resolved.

You are now the proud owner of a machine that can compile Java programs. You are ready to start writing software!

By the way, one of the things you just unpacked is a **demo** directory full of neat examples. All of the examples are ready to run, so you might want to find the directory and play with some of the samples. Many of them make sounds, so be sure to turn on your speakers. To run the examples, find pages with names like **example1.html** and load them into your usual Web browser.

Your First Program

Your first program will be short and sweet. It is going to create a drawing area and draw a diagonal line across it. To create this program you will need to:

1. Open Notepad and type in (or cut and paste) the program
2. Save the program
3. Compile the program with the Java compiler to create a **Java applet**
4. Fix any problems
5. Run the Java applet

Here is the program we will use for this demonstration:

```
import java.awt.Graphics;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        g.drawLine(0, 0, 200, 200);
    }
}
```

Step 1 - Type in the program

Create a new directory to hold your program. Open up Notepad (or any other text editor that can create TXT files). Type or cut and paste the program into the Notepad window. This is important: When you type the program in, **case matters**. That means that you must type the uppercase and lowercase characters exactly as they appear in the program. Review the programmer's creed above. If you do not type it EXACTLY as shown, it is not going to work.

Step 2 - Save the file

Save the file to the filename **FirstApplet.java** in the directory that you created in step 1. **Case matters** in the filename. Make sure the 'F' and 'A' are uppercase and all other characters are lowercase, as shown.

Step 3 - Compile the program

Open an MS-DOS window. Change directory ("cd") to the directory containing **FirstApplet.java**. Type:

```
javac FirstApplet.java
```

Case matters! Either it will work, in which case nothing will be printed to the window, or there will be errors. If there are no errors, a file named FirstApplet.class will be created in the directory right next to FirstApplet.java.

(Make sure that the file is saved to the name **FirstApplet.java** and not **FirstApplet.java.txt**. This is most easily done by typing **dir** in the MS-DOS window and looking at the file name. If it has a **.txt** extension, remove it by renaming the file. Or run the Windows Explorer and select Options in the View menu. Make sure that the "Hide MD-DOS File Extensions for file types that are registered" box is NOT checked, and then look at the filename with the explorer. Change it if necessary.)

Step 4 - Fix any problems

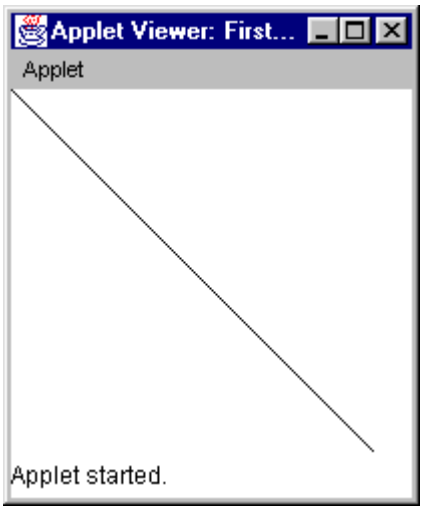
If there are errors, fix them. Compare your program to the program above and get them to match exactly. Keep recompiling until you see no errors. If **javac** seems to not be working, look back at the previous section and fix your installation.

Step 5 - Run the Applet

In your MS-DOS window, type:

```
appletviewer applet.htm
```

You should see a diagonal line running from the upper left corner to the lower right corner:



Pull the applet viewer a little bigger to see the whole line. You should also be able to load the HTML page into any modern browser like Netscape Navigator or Microsoft Internet Explorer and see approximately the same thing.

You have successfully created your first program!!!

Understanding What Just Happened

So what just happened? First, you wrote a piece of code for an extremely simple **Java applet**. An applet is a Java program that can run within a Web browser, as opposed to a **Java application**, which is a stand-alone program that runs on your local machine (Java applications are slightly more complicated and somewhat less popular, so we will start with applets). We compiled the applet using **javac**. We then created an extremely simple Web page to "hold" the applet. We ran the applet using **appletviewer**, but you can just as easily run it in a browser.

The program itself is about 10 lines long:

```
import java.awt.Graphics;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        g.drawLine(0, 0, 200, 200);
    }
}
```

This is about the simplest Java applet you can create. To fully understand it you will have to learn a fair amount, particularly in the area of **object oriented programming techniques**. Since I am assuming that you have zero programming experience, what I would like you to do is focus your attention on just one line in this program for the moment:

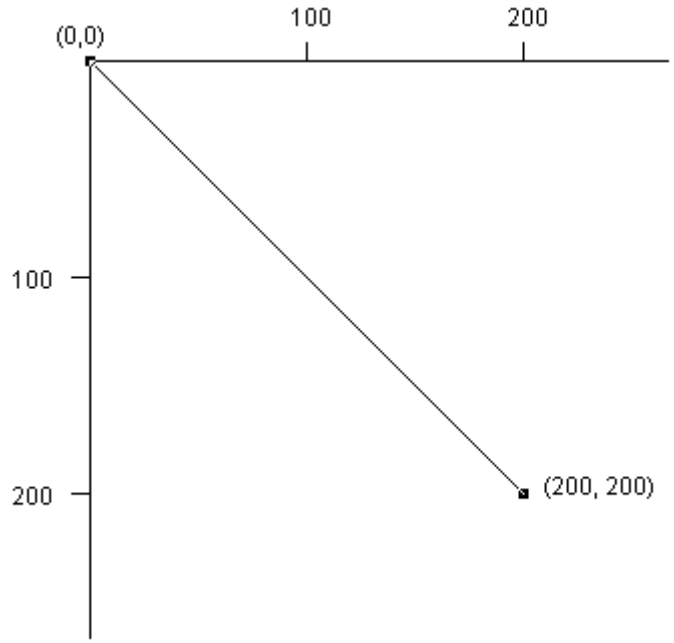
```
        g.drawLine(0, 0, 200, 200);
```

This is the line in this program that does the work. It draws the diagonal line. The rest of the program is scaffolding that supports that one line, and we can ignore the scaffolding for the moment. What happened here was that we told the computer to draw one line from the upper left hand corner (0,0) to the bottom right hand corner (200, 200). The computer drew it just like we told it to. That is the essence of computer programming!

(Note also that in the HTML page, we set the size of the applet's window in step 5 above to have a width of 200 and a height of 200.)

In this program, we called a **method** (a.k.a. **function**) called **drawLine** and we passed it four **parameters** (0, 0, 200, 200). The line ends in a semicolon. The semicolon acts like the period at the end of the sentence. The line begins with **g.**, signifying that we want to call the method named **drawLine** on the specific object named **g** (which you can see one line up is of the class **Graphics** -- we will get into classes and methods of classes in much more detail later in this article).

A method is simply a command -- it tells the computer to do something. In this case, **drawLine** tells the computer to draw a line between the points specified: (0, 0) and (200, 200). You can think of the window as having its 0,0 coordinate in the upper left corner, with positive X and Y axes extending to the right and down. Each dot on the screen (each **pixel**) is one increment on the scale.



Try experimenting by using different numbers for the four parameters. Change a number or two, save your changes, recompile with **javac** and rerun after each change in **appletviewer**, and see what you discover.

What other functions are available besides **drawLine**? You find this out by looking at the documentation for the **Graphics** class. When you installed the Java development kit and unpacked the documentation, one of the files unloaded in the process is called **java.awt.Graphics.html**, and it is on your machine. This is the file that explains the **Graphics** class. On my machine, the exact path to this file is D:\jdk1.1.7\docs\api\java.awt.Graphics.html. On your machine the path is likely to be slightly different, but close -- it depends on exactly where you installed things. Find the file and open it. Up toward the top of this file there is a section called "Method Index." This is a list of all of the methods this class supports. The **drawLine** method is one of them, but you can see *many* others. You can draw, among other things:

- Lines
- Arcs
- Ovals
- Polygons
- Rectangles
- Strings
- Characters

Read about and try experimenting with some of these different methods to discover what is possible. For example, try this code:

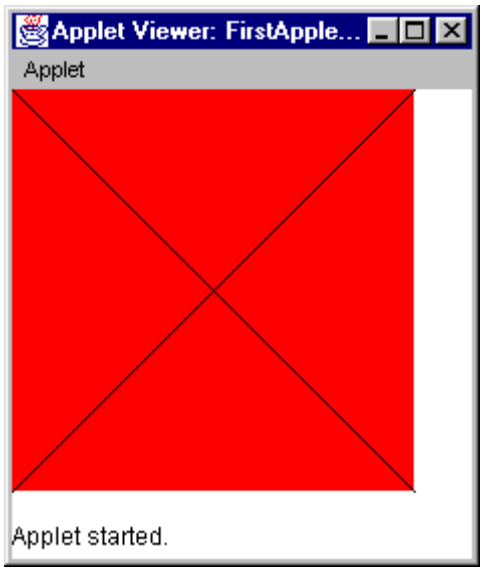
```
g.drawLine(0, 0, 200, 200);
g.drawRect(0, 0, 200, 200);
g.drawLine(200, 0, 0, 200);
```

It will draw a box with two diagonals (be sure to pull the window big enough to see the whole thing). Try drawing other shapes. Read about and try changing the color with the **setColor** method. For example:

```
import java.awt.Graphics;
import java.awt.Color;
```

```
public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.fillRect(0, 0, 200, 200);
        g.setColor(Color.black);
        g.drawLine(0, 0, 200, 200);
        g.drawLine(200, 0, 0, 200);
    }
}
```

Note the addition of the new **import** line in the second line of the program. The output of this program looks like this:



One thing that might be going through your head right now is, "How did he know to use **Color.red** rather than simply **red**, and how did he know to add the second **import** line?" You learn things like that by example. Because I just showed you an example of how to call the **setColor** method, you now know that whenever you want to change the color you will use **Color.** followed by a color name as a parameter to the **setColor** method, and you will add the appropriate **import** line at the top of the program. If you look up **setColor**, it has a link that will tell you about the **Color** class, and in it is a list of all the valid color names along with techniques for creating new (unnamed) colors. You read that information, you store it in your head and now you know how to change colors in Java. That is the essence of becoming a computer programmer -- you learn techniques and remember them for the next program you write. You learn the techniques either by reading an example (as you did here) or by reading through the documentation or by looking at example code (as in the demo directory). If you have a brain that likes exploring and learning and remembering things, then you will love programming!

In this section, you have learned how to write linear, sequential code -- blocks of code that consist of method calls starting at the top and working toward the bottom (try drawing one of the lines *before* you draw the red rectangle and watch what happens -- it will be covered over by the rectangle and made invisible. The order of lines in the code sequence is important). Sequential lines of code form the basic core of any computer program. Experiment with all the different drawing methods and see what you can discover.

Bugs and Debugging

One thing that you are going to notice as you learn about programming is that you tend to make a fair number of mistakes and assumptions that cause your program to either: 1) not compile, or 2) produce output that you don't expect when it executes. These problems are referred to as **bugs**, and the act of removing them is called **debugging**. About half of the time of any programmer is spent debugging.

You will have plenty of time and opportunity to create your own bugs, but to get more familiar with the possibilities let's create a few. In your program, try erasing one of the semicolons at the end of a line and try compiling the program with **javac**. The compiler will give you an error message. This is called a **compiler error**, and you have to eliminate all of them before you can execute your program. Try misspelling a function name, leaving out a "{" or eliminating one of the **import** lines to get used to different compiler errors. The first time you see a certain type of compiler error it can be frustrating, but by experimenting like this -- with known errors that you create on purpose -- you can get familiar with many of the common errors.

A bug, also known as an execution (or run-time) error, occurs when the program compiles fine and runs, but then does not produce the output you planned on it producing. For example, this code produces a red rectangle with two diagonal lines across it:

```
g.setColor(Color.red);
g.fillRect(0, 0, 200, 200);
g.setColor(Color.black);
g.drawLine(0, 0, 200, 200);
g.drawLine(200, 0, 0, 200);
```

The following code, on the other hand, produces just the red rectangle (which covers over the two lines):

```
g.setColor(Color.black);
g.drawLine(0, 0, 200, 200);
g.drawLine(200, 0, 0, 200);
g.setColor(Color.red);
g.fillRect(0, 0, 200, 200);
```

The code is almost exactly the same but looks completely different when it executes. If you are expecting to see two diagonal lines, then the code in the second case contains a bug.

Here's another example:

```
g.drawLine(0, 0, 200, 200);
g.drawRect(0, 0, 200, 200);
g.drawLine(200, 0, 0, 200);
```

This code produces a black outlined box and two diagonals. This next piece of code produces only one diagonal:

```
g.drawLine(0, 0, 200, 200);
g.drawRect(0, 0, 200, 200);
g.drawLine(0, 200, 0, 200);
```

Again, if you expected to see two diagonals, then the second piece of code contains a bug (look at the second piece of code until you understand what went wrong). This sort of bug can take a long time to find because it is subtle.

You will have plenty of time to practice finding your own bugs. The average programmer spends about half of his or her time tracking down, finding and eliminating bugs. Try not to get frustrated when they occur -- they are a normal part of programming life.

Variables

All programs use **variables** to hold pieces of data temporarily. For example, if at some point in a program you ask a user for a number, you will store it in a variable so that you can use it later.

Variables must be **defined** (or **declared**) in a program before you can use them, and you must give each variable a specific type. For example, you might declare one variable to have a type that allows it to hold numbers, and another variable to have a type that allows it to hold a person's name. (Because Java requires you to specifically define variables before you use them and state the type of value you plan to store in a variable, Java is called a **strongly typed** language. Certain languages don't have these requirements. In general, when creating large programs, strong typing tends to reduce the number of programming errors that you make.)

```
import java.awt.Graphics;
import java.awt.Color;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        int width = 200;
        int height = 200;
        g.drawRect(0, 0, width, height);
        g.drawLine(0, 0, width, height);
        g.drawLine(width, 0, 0, height);
    }
}
```

In this program, we have declared two variables named **width** and **height**. We have declared their type to be **int**. An **int** variable can hold an integer (a whole number such as 1, 2, 3). We have **initialized** both variables to 200. We could just as easily have said:

```
        int width;
        width = 200;
        int height;
        height = 200;
```

The first form is simply a bit quicker to type.

The act of setting a variable to its first value is called **initializing** the variable. A common programming bug occurs when you forget to initialize a variable. To see that bug, try eliminating the initialization part of the code (the "= 200" part) and recompile the program to see what happens. What you will find is that the compiler complains about this problem. That's a very nice feature, by the way. It will save you lots of wasted time.

There are two types of variables in Java -- simple (**primitive**) variables and **classes**.

The **int** type is simple. The variable can hold a number. That is all that it can do. You declare an **int** , set it to a value and use it. **Classes**, on the other hand, can contain multiple parts and have methods that make them easier to use. A good example of a straightforward class is the **Rectangle** class, so let's start with it.

One of the limitations of the program we have been working on so far is the fact that it assumes the window is 200 by 200 pixels. What if we wanted to ask the window, "How big are you?," and then size our rectangle and diagonals to fit? If you go back and look on the documentation page for the **Graphics** class (java.awt.Graphics.html -- the file that lists all the available drawing functions), you will see that one of the functions is called **getClipBounds**. Click on this function

name to see the full description. This function accepts no parameters but instead **returns** a value of type **Rectangle**. The rectangle it returns contains the width and height of the available drawing area. If you click on **Rectangle** in this documentation page you will be taken to the documentation page for the **Rectangle** class (java.awt.Graphics.html). Looking in the Variable Index section at the top of the page, you find that this class contains four variables named x, y, width and height, respectively. What we want to do, therefore, is get the clip boundary rectangle using **getClipBounds** and then extract the width and height from that rectangle and save the values in the **width** and **height** variables we created in the previous example, like this:

```
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Rectangle;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        int width;
        int height;
        Rectangle r;

        r = g.getClipBounds();
        width = r.width - 1;
        height = r.height - 1;

        g.drawRect(0, 0, width, height);
        g.drawLine(0, 0, width, height);
        g.drawLine(width, 0, 0, height);
    }
}
```

When you run this example, what you will notice is that the rectangle and diagonals exactly fit the drawing area. Plus, when you change the size of the window, the rectangle and diagonals redraw themselves at the new size automatically. There are five new concepts introduced in this code, so let's look at them:

1. First, because we are using the Rectangle class we need to import **java.awt.Rectangle** on the third line of the program.
2. We have declared three variables in this program. Two (**width** and **height**) are of type **int** and one (**r**) is of type Rectangle.
3. We used the **getClipBounds** function to get the size of the drawing area. It accepts no parameters so we passed it none ("()"), but it **returns** a Rectangle. We wrote the line, "r = g.getClipBounds();" to say, "Please place the returned rectangle into the variable **r**."
4. The variable **r**, being of the class Rectangle, actually contains four variables -- x, y, width, and height (you learn these names by reading the documentation for the Rectangle class). To access them you use the "." (dot) operator. So the phrase "r.width" says, "Inside the variable **r** retrieve the value named **width**." That value is placed into our local variable called **width**. In the process, we subtracted 1. Try leaving the subtraction out and see what happens. Also try subtracting five instead and see what happens.
5. Finally, we used **width** and **height** in the drawing functions.

One question commonly asked at this point is, "Did we really need to declare variables named **width** and **height**?" The answer is, "No." We could have typed **r.width - 1** directly into the drawing function. Creating the variables simply makes things a little easier to read, and it's therefore a good habit to fall into.

Java supports several simple variable types. Here are three of the most common:

- **int** - integer (whole number) values (1, 2, 3...)
- **float** - decimal values (3.14159, for example)
- **char** - character values (a, b, c...)

You can perform math operations on simple types. Java understands **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division) and several others. Here's an example of how you might use these operations in a program. Let's say that you want to calculate the volume of a sphere with a diameter of 10 feet. The following code would handle it:

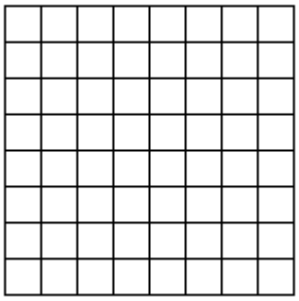
```
float diameter = 10;
float radius;
float volume;

radius = diameter / 2.0;
volume = 4.0 / 3.0 * 3.14159 * radius * radius * radius;
```

The first calculation says, "Divide the value in the variable named **diameter** by 2.0 and place the result in the variable named **radius**." You can see that the "=" sign here means, "Place the result from the calculation on the right into the variable named on the left."

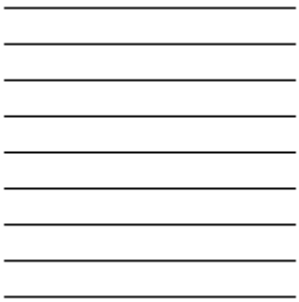
Looping

One of the things that computers do very well is perform repetitive calculations or operations. In the previous sections, we have seen how to write "sequential blocks of code," so the next thing we should discuss is the techniques for causing a sequential block of code to occur repeatedly.

For example, let's say that I ask you to draw the following figure:



A good place to start would be to draw the horizontal lines, like this:



One way to draw the lines would be to create a sequential block of code:

```
import java.awt.Graphics;

public class FirstApplet extends java.applet.Applet
```

```
{

    public void paint(Graphics g)
    {
        int y;
        y = 10;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
        y = y + 25;
        g.drawLine(10, y, 210, y);
    }
}
```

(For some new programmers, the statement "y = y + 25;" looks odd the first time they see it. What it means is, "Take the value currently in the variable **y**, add 25 to it and place the result back into the variable **y**." So if y contains 10 before the line is executed, it will contain 35 immediately after the line is executed.)

Most people who look at this code immediately notice that it contains the same two lines repeated over and over. In this particular case the repetition is not so bad, but you can imagine that if you wanted to create a grid with thousands of rows and columns, this approach would make program-writing very tiring. The solution to this problem is a **loop**, as shown below:

```
import java.awt.Graphics;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        int y;
        y = 10;
        while (y <= 210)
        {
            g.drawLine(10, y, 210, y);
            y = y + 25;
        }
    }
}
```

When you run this program, you will see that it draws nine horizontal lines 200 pixels long.

The **while** statement is a looping statement in Java. The statement tells Java to behave in the following way: At the **while** statement, Java looks at the expression in the parentheses and asks, "Is **y** less than or equal to 210?"

- If the answer is yes, then Java enters the block of code bracketed by braces -- "{" and "}". The looping part occurs at the end of the block of code. When Java reaches the ending brace, it loops back up to the **while** statement and asks the question again. This looping sequence may occur many times.
- If the answer is no, it skips over the code bracketed by braces and continues.

So you can see that when you run this program, initially **y** is 10. Ten is less than 210, so Java enters the block in braces, draws a line from (10,10) to (210, 10), sets **y** to 35 and then goes back up to the **while** statement. Thirty-five is less than 210, so Java enters the block in braces, draws a line from (10,35) to (210, 35), sets **y** to 60 and then goes back up to the **while** statement. This sequence repeats until **y** eventually gets to be greater than 210. Then the program quits.

We can complete our grid by adding a second loop to the program, like this:

```
import java.awt.Graphics;

public class FirstApplet extends java.applet.Applet
{

    public void paint(Graphics g)
    {
        int x, y;
        y = 10;
        while (y <= 210)
        {
            g.drawLine(10, y, 210, y);
            y = y + 25;
        }
        x = 10;
        while (x <= 210)
        {
            g.drawLine(x, 10, x, 210);
            x = x + 25;
        }
    }
}
```

You can see that a **while** statement has three parts:

- There is an **initialization step** that sets y to 10.
- Then there is an **evaluation step** inside the parentheses of the while statement.
- Then, somewhere in the while statement there is an **increment step** that increases the value of y.

Java supports another way of doing the same thing that is a little more compact than a **while** statement. It is called a **for** statement. If you have a **while** statement that looks like this:

```
        y = 10;
        while (y <= 210)
        {
            g.drawLine(10, y, 210, y);
            y = y + 25;
        }
```

then the equivalent **for** statement looks like this:

```
        for (y = 10; y <= 210; y = y + 25)
        {
```

```
            g.drawLine(10, y, 210, y);
        }
```

You can see that all the **for** statement does is condense the initialization, evaluation and incrementing lines into a short, single line. It simply shortens the programs you write, nothing more.

While we are here, two quick points about loops:

- In many cases, it would be just as easy to initialize **y** to 210 and then decrement it by 25 each time through the loop. The evaluation would ask, "Is **y** greater than or equal to 10?" The choice is yours. Most people find it easier to add than subtract in their heads, but you might be different.
- The increment step is very important. Let's say you were to accidentally leave out the part that says "y = y + 25;" inside the loop. What would happen is that the value of **y** would never change -- it would always be 10. So it would never become greater than 210 and the loop would continue forever (or until you stop it by turning off the computer or closing the window). This condition is called an **infinite loop**. It is a bug that is pretty common.

To get some practice with looping, try writing programs to draw the following figures: