Chapter IV

Java Program Organization

Chapter IV Topics

4.1 Introduction

This book is intentionally written in an informal, conversational style. I explained that in the first chapter. Now with Chapter IV you will see *I* changing to *we*. Why? There is another Schram computer science teacher. He is my son and he is very actively involved in the creation of this brand-new *PreAP* version of *Exposure Java*. The first three chapters are mostly unchanged. Starting with Chapter IV this book is very different from Exposure Java, the AP edition. John Schram, the son, has now joined Leon Schram, the father, in writing this book. Hence, from this point forward *I will no longer use I. We, the combined Schrams, will be using we*.

Students new to computer programming ask a fundamental, and very reasonable, question. Why do we write programs in a language like Java? What is wrong with programming in English? This question was first stated in the first chapter?

> *So why not simply write your programs in English? Is that not a matter of creating some kind of translating program that takes English instructions and creates a machine code file for the computer? This is certainly what has been attempted for many years, but translating human languages has been very elusive. Consider the following example. In the sixties, computer scientists tried to write a program that would translate English into Russian and Russian into English. This is really the same problem of properly understanding the meaning of human language. The scientists decided to test their program by entering an English sentence. The resulting Russian sentence was then entered back into the computer and the final result should be the original English sentence. Should, that is, if the computer could translate correctly in both directions. The original sentence entered was:*
>
> ***The spirit is willing but the flesh is weak***
>
> *We do not have a clue what the Russian result was, but we do know the final English result. Our computer scientists were quite surprised with the response of*
>
> ***The Vodka is great but the meat is rotten***
>
> *This little experiment showed the major problem with human languages. Human languages like English are idiomatic. We use all these idioms and proverbs, and slang and special nuances that are meaningless to the computers. How can computers figure out human language, when humans are confused? The bottom line is that programming requires a restricted language. This language can have human type words, but the words selected, the symbols and punctuation used all must have a very precise meaning. The manner in which the program language structures its statements is called syntax. Program languages must have very precise syntax. Compilers first check to see if a program has correct syntax. Only after the syntax checks out, is the next step of translating into binary code performed.*

All programming languages require a very precise organization. In this chapter you will learn about the basic organization of the Java programming language. There is a pleasant surprise. There are no exceptions in Java. This makes Java easier to learn than a human language like Spanish, German or Mandarin, but it still will be a foreign language. Actually, you are lucky, because Java is based on the English language. This makes learning Java simpler for you than the students learning Java in a country that does not speak English.

4.2 Java Program Components

All languages have organization and organization starts with components. Consider writing a book in English. You combine words into a sentence. Then multiple sentences combine to form a paragraph. A group of paragraphs form a chapter. Finally, many chapters create a book. Words, sentences, paragraphs and chapters are components of the English language. As you write your book you are also careful to spell the words correctly, use proper grammar in your sentence structure and utilize the correct punctuation. There exists a similar logic in a Java program. Java has special keywords that have meaning in Java. You have already seen a fair amount of keywords. Examples are **public**, **main**, **System**, **int**, **double**, **print**, **void** and there will be many more you will learn during this course. Now consider the following program segment in figure 4.1.

Figure 4.1

```
int a = 100;
int b = 200;
int sum = a + b;
System.out.println(sum);
```

The program segment looks correct, but it will neither compile nor execute. Java has a very precise program structure and the first rule is that Java requires the use of containers. Now Java does not call these components containers, but rather *classes* and *methods*. The English-to-Java comparison chart in figure 4.2 explains the different components in English and how they compare to Java components.

Figure 4.2

English		Java	
Component	**Example**	**Component**	**Example**
word	tiger	**keyword**	public
sentence	The tiger is big.	**program statement**	System.out.print("The tiger is big.");

paragraph	My sister and I went to the zoo. We saw many animals. The tigers were very scary. They were large, very loud and they smelled bad. We liked the funny monkeys better.	**method**	public static void main(String args[])
			{
			int a = 100;
			int b = 200;
			int sum = a + b;
			System.out.println(sum);
			}

chapter or essay	Our Trip to the Zoo
	
	Opening paragraph
	Middle paragraphs
	Closing paragraph
		class	public class Demo
			{
			public static void main(String args[])
			{
			System.out.println("Hello");
			}
			}

The chart, shown in figure 4.2, is a good start. Java is very organized and we can now make some rules that need to be observed. Any violation of Java language rules, also known as syntax rules, will result in error messages. A program with syntax errors will not compile, and a program that does not compile will not create a *byte code* (class) file. You must have a byte code file to execute a Java program.

Fundamental Java Syntax Rules

- **All program statements end with a semi-colon.**
- **All program statements are contained in a method.**
- **All methods are contained in a class.**
- **Each method must have a heading.**
- **Each class must have a heading.**
- **Class and method containers start with a { brace.**
- **Class and method containers end with a } brace.**
- **Method headings and class heading are not program statements and they do not get a semi-colon.**
- **All keywords in Java are case-sensitive. This means that System and system are two different words.**
- **Comments, which are not program statements, start with two slashes and do not require a semi-colon.**

Examples of these rules are shown by the program in figure 4.3

Figure 4.3

```
public class Example                          //  class, called Example, heading
{                                             //  start of the Example class container
    public static void main (String args[])   //  method, called main, heading
    {                                         //  start of the main method container
        int a = 10;                           //  program statement
        int b = 25;                           //  program statement
        System.out.println();                 //  program statement
        System.out.println(a);                //  program statement
        System.out.println(b);                //  program statement
        System.out.println();                 //  program statement
    }                                         //  end of the main method container
}                                             //  end of the Example class container
```

For several chapters you will only be concerned with placing program statements between the braces of the **main** method. How exactly you create your own methods and classes will be explained in future chapters. Right now the class container will be provided for you and so will the single method, called **main**. The **main** method heading will always be the same, like:

```
public static void main(String args[])
```

It is too early in the course to explain the purpose of keywords like **public**, **static**, **void** and **main**. All you need to comprehend in this chapter is that your programs require a **main** method and that **main** method has a precise heading. Your job is to place a logical set of program statements in between the braces **{ }** of the **main** method container.

Most students take foreign language classes. You know that communication in a foreign language is not possible without vocabulary. The meaning of words in any language is fundamental to comprehending a language. The same is true in Java. In this chapter, and all the chapters that follow, you will continuously be introduced to new keywords in Java. You must learn the meanings of these words and please do not cram the last day before an exam; that will not work. The list of Java keywords is reasonable, and the names are similar to English in many cases.

Besides special keywords, you will also need to learn the punctuation of Java. So far you know that every program statement ends with a semi-colon. Additionally, you have seen that the equal sign = is an assignment operator in a statement like **int number = 5000;** It is normal to feel confused and overwhelmed in the beginning. Relax, students have learned programming for many decades now and they have all done very nicely. Do your reading, complete the exercises, complete the lab assignments and you will do well.

4.3 Using the Math Class

Most students reading this book have little experience with other programming languages. Java is considered a relatively *small and simple* programming language. This means that the sum total of its reserved words in Java and the rules of its syntax are not very large and relatively easy to learn. This only tells part of the story.

Computers exist to make life simpler and as the programming of computers evolves, it makes sense that an effort is made to simplify programming. How does programming become simpler? For starters, programming is simpler when programs are written in a human-type language rather than binary machine code.

This process has already been accomplished for a number of decades. It is in the nature of human ingenuity to make tools to simplify our lives. It is possible to make large structures with very few tools, but somebody decided that it would be simpler to constructs some cranes to help in construction. Cranes are only a small part because we now have a huge number of tools for every conceivable construction job.

It is not very different in the computer science world. It is possible to start each program by writing code that accomplishes certain required tasks. It is also possible to create libraries of special program modules that perform a variety of potentially necessary tasks. Each one of these modules was designed and written at some prior time, but once written, the module is now available for future use.

It is possible to create a disorganized set of modules that are scattered around a number of library files. Such is not the nature of Java. Modules that perform a related set of functions are grouped together in a special type of container, called a **class**. Java has an absolutely enormous set of classes for every conceivable purpose.

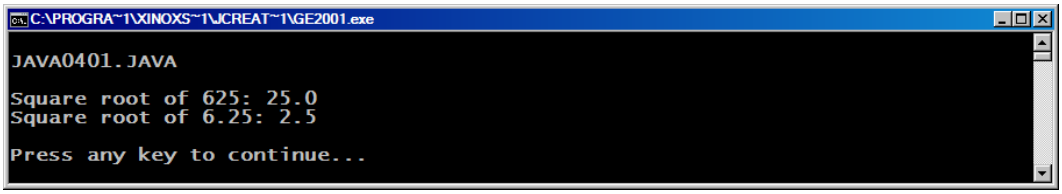
You will learn more about the organization of all these classes at a later time. It may help to consider a class to be a *toolkit* and consider the methods in the class to be its *tools*. The first Java toolkit, or class, to examine is the **Math** class. The **Math** class contains many useful program modules, which compute a wide variety of mathematical functions.

Now you need to understand something. You have already seen program examples that use simple data types. Manipulating variables of simple data types requires only the use of the variable identifier and nothing else. Such is not the case with classes. You must first specify the identifier of the class and then use the identifier of the action within the class. A hammer is a tool, but it is not sufficient to ask for a hammer. You need to specify *roofing hammer* or *carpentry hammer* or *sheetrock hammer*.

This tool, action, procedure, function, task, behavior, subroutine, module or whatever you may wish to call it shall in Java be called a **method**. Classes contain one or more *methods* and methods contain one or more *program statements*. Program **Java0401.java**, in figure 4.4, starts with the **sqrt** class method, which is an abbreviation for **square root**.

Figure 4.4

```
// Java0401.java
// This program shows how to use the <sqrt> method of the Math
// class.  The Math class is part of the java.lang package, which is
// automatically loaded (imported) by the compiler.
// Math.sqrt returns the square root of the argument.


public class Java0401
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0401.JAVA\n");
        int n1 = 625;
        double n2 = 6.25;
        System.out.println("Square root of " + n1 + ": " + Math.sqrt(n1));
        System.out.println("Square root of " + n2 + ": " + Math.sqrt(n2));
        System.out.println();
  }
}
```



This first program example displays the square root of two different numbers, **n1** and **n2**. You need to understand the four important components required in the use of a method.

Method Syntax

Math.sqrt(n1)

1. **Math** is the class identifier, which contains the methods you call.
2. • separates the class identifier from the method identifier
3. **sqrt** is the method identifier
4. **(n1)** n1 is the argument or parameter passed to the method

Maybe after only one program example the four components do not make much sense. Do not worry. There are many more examples that will help to clarify this concept. In particular, you need to understand the concept of a **parameter**, which is used to provide necessary information to a method.

Suppose you say only **Math.sqrt**? Can Java digest such a statement? Can you? What do you say when your math teacher walks up to you and says: *"Give me the mathematical square root!"* You will probably be a little perplexed because the computation of a mathematical problem requires some type of information. This information is passed to the method, which can then provide the requested response.

In the next program example, shown in figure 4.5, you will see that parameters can be passed in different formats. It is possible to pass a numerical constant, a variable, an expression or even another method.

Figure 4.5

```
// Java0402.java
// This program shows different arguments that can be used with the <sqrt> method.
// Note how a method call can be the argument of another method call.

public class Java0402
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0402.JAVA\n");
        double n1, n2, n3, n4;
        n1 = Math.sqrt(1024);              // constant argument
        n2 = Math.sqrt(n1);                // variable argument
        n3 = Math.sqrt(n1 + n2);           // expression argument
        n4 = Math.sqrt(Math.sqrt(256));  // method argument
        System.out.println("n1: " + n1);
        System.out.println("n2: " + n2);
        System.out.println("n3: " + n3);
        System.out.println("n4: " + n4);
        System.out.println();
    }
}
```

Method Arguments or Parameters

The information, which is passed to a method is called an **argument** or a **parameter**.

Parameters are placed between parentheses immediately following the method identifier.

Parameters can be constants, variables, expressions or they can be methods. The only requirement is that the correct data type value is passed to the method. In other words, **Math.sqrt(x)** can compute the square root of **x**, if **x** is any correct number, but not if **x equals "aardvark"**.

floor, ceil and round Methods

The Math class has three related methods, **floor**, **ceil** and **round**. You are most likely familiar with rounding to the nearest integer. Java gives you several choices of "rounding" numbers.

The name of the method gives a good clue to its purpose. The **floor** method returns the next lowest whole number. Think **floor is down**. The **ceil** method returns the next higher whole number. Think **ceil**ing **is up**. The **round** method is the traditional rounding approach, which *rounds-up*, if the fraction is 0.5 or greater and *rounds-down* otherwise.

Program **Java0403.java**, in figure 4.6 on the next page, demonstrates each one of the three different "rounding" methods. Carefully compare all the program statements with the output. It will also help to alter the values of the arguments and then recompile and re-execute the program to see the results.

Remember if you alter a program - no matter how slightly - and then execute without recompiling, you will be using the bytecode file that was created with an older version of the program. When in doubt re-compile the program.

Figure 4.6

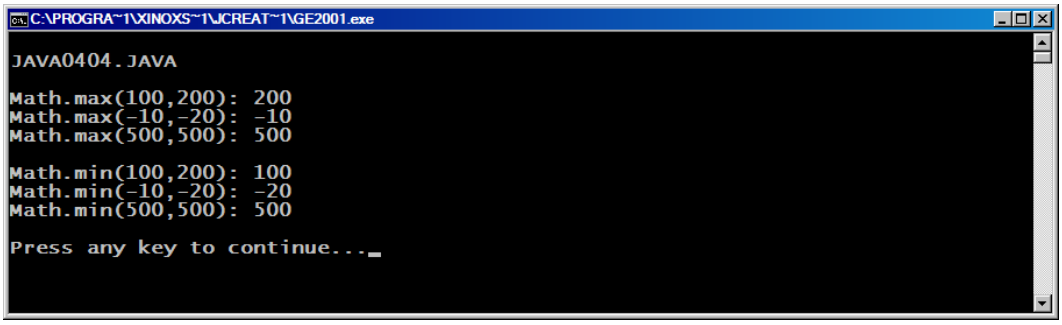
```
// Java0403.java
// This program demonstrates the <floor> <ceil> and <round> methods.
// The <floor> method returns the truncation down to the next lower integer.
// The <ceil> method returns the next higher integer.
// The <round> method rounds the argument and returns the closest integer.


public class Java0403
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0403.JAVA\n");
        System.out.println("Math.floor(5.001): " + Math.floor(5.001));
        System.out.println("Math.floor(5.999): " + Math.floor(5.999));
        System.out.println("Math.floor(5.5)  :  " + Math.floor(5.5));
        System.out.println("Math.floor(5.499): " + Math.floor(5.499));
        System.out.println();

        System.out.println("Math.ceil(5.001) : " + Math.ceil(5.001));
        System.out.println("Math.ceil(5.999) : " + Math.ceil(5.999));
        System.out.println("Math.ceil(5.5)   :  " + Math.ceil(5.5));
        System.out.println("Math.ceil(5.499) : " + Math.ceil(5.499));
        System.out.println();

        System.out.println("Math.round(5.001): " + Math.round(5.001));
        System.out.println("Math.round(5.999): " + Math.round(5.999));
        System.out.println("Math.round(5.5)  :  " + Math.round(5.5));
        System.out.println("Math.round(5.499): " + Math.round(5.499));
        System.out.println();
    }
}
```

max and min Methods

The first three program examples can easily give the impression that methods use a single parameter, or at least it appears that methods of the Math class use a single parameter. In mathematics there are many examples where only a single argument is required. Functions like square root, absolute value, rounding, etc. can be computed with a single value. There are also some examples where multiple arguments or parameters are used. Many area and volume computations involve multiple arguments, like the area of a rectangle, which requires **length** and **width**. The Java Math class has two methods, which both require two parameters, **max** and **min**. The **max** method returns the larger of the two parameters and the **min** method returns the smaller of the two arguments. Both methods are shown in figure 4.7.

Figure 4.7

```
// Java0404.java
// This program demonstrates the <max> and <min> methods.
// Math.max returns the largest value of the two arguments.
// Math.min returns the smallest value of the two arguments.


public class Java0404
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0404.JAVA\n");
        System.out.println("Math.max(100,200): " + Math.max(100,200));
        System.out.println("Math.max(-10,-20): " + Math.max(-10,-20));
        System.out.println("Math.max(500,500): " + Math.max(500,500));
        System.out.println();
        System.out.println("Math.min(100,200): " + Math.min(100,200));
        System.out.println("Math.min(-10,-20): " + Math.min(-10,-20));
        System.out.println("Math.min(500,500): " + Math.min(500,500));
        System.out.println();
    }
}
```

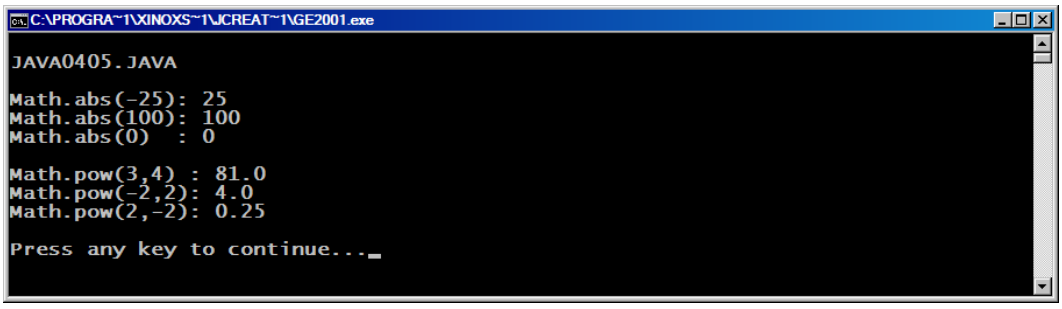


abs and pow Methods

The **abs** and **pow** methods are not related, but we did not want to write a program example with a single new method in it, besides the original **sqrt** method. The **abs** method returns the absolute value of the argument. The **pow** method returns a value, which is computed by raising the first parameter to the power of the second parameter, as shown in figure 4.8.

Figure 4.8

```
// Java0405.java
// This program demonstrates the <abs> and <pow> methods.
// Math.abs returns the absolute value of the argument.
// Math.pow returns the first argument raised to the power
// of the second argument.


public class Java0405
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0405.JAVA\n");
        System.out.println("Math.abs(-25): " + Math.abs(-25));
        System.out.println("Math.abs(100): " + Math.abs(100));
        System.out.println("Math.abs(0)  : " + Math.abs(0));
        System.out.println();

        System.out.println("Math.pow(3,4) : " + Math.pow(3,4));
        System.out.println("Math.pow(-2,2): " + Math.pow(-2,2));
        System.out.println("Math.pow(2,-2): " + Math.pow(2,-2));
        System.out.println();
    }
}
```

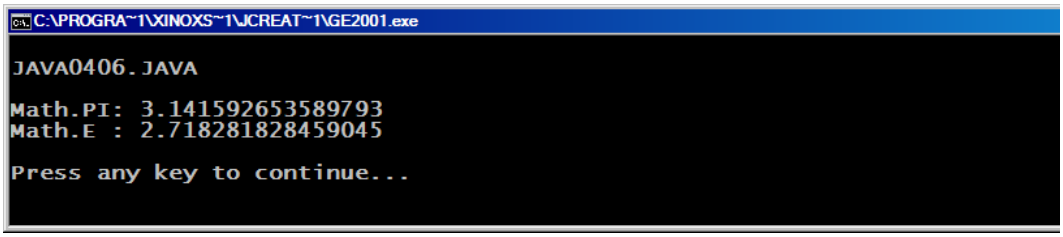


PI and E Math Fields

In the majority of cases the only class members that you use in a program are the class methods. Most classes do contain data, but such data is normally processed by methods. There are a few exceptions and the Math class has two examples. The values of **PI** and **E** are constant mathematical values that never change. These values are accessed not with a method call, but something similar, which looks like a method call without parameters. Look at program **Java0406.java**, in figure 4.9, and notice the missing parameters and parentheses. Now think logically, here. Is a parameter value necessary to compute **PI** or **E**? The values never change since they are constant or **final** as Java calls it. In this case you are not making a method call, but accessing the data field of the class directly.

Figure 4.9

```
// Java0406.java
// This program demonstrates the <PI> and <E> fields of the Math class.
// Both <PI> and <E> are "final" attributes of the <Math> class.
// <PI> and <E> are not methods.  Note there are no parentheses.


public class Java0406
{
    public static void main (String args[])
    {
        System.out.println("\nJAVA0406.JAVA\n");
        System.out.println("Math.PI: " + Math.PI);
        System.out.println("Math.E : " + Math.E);
        System.out.println();
    }
}
```



This section introduced the **Math** class and showed a partial selection of the available methods. There are many more **Math** methods that were not shown, such as methods to handle trigonometric calculations, logarithmic calculations and conversions between degrees and radians. It is not our intention in this chapter to give a full treatment of the **Math** class. Our purpose here is to show you how to use the methods of a provided class.

4.5 Introduction to the Expo Class

This next section introduces something new for the 2008-2009 school year that we are really excited about. This is the **Expo** class. The **Expo** class has many methods, just like the **Math** class. The first thing you must realize is the **Expo** class is not part of the Java programming language. It is a special user-defined class created by John Schram, the son.

This is where some eyebrows might go up. *Not part of Java?* What are the Schrams up to now? Well here is what we have observed. In all programming languages, there are simple concepts, and there are complex concepts. Now, in many program languages, the simple concepts have simple commands, and the complex concepts have complex commands. The commands are complex only because the concept is complex and there is no simple one-step approach to do it. The designers of most computer languages seem to understand this. The designers of Java seem to be of a different breed. There exist many topics in Java, which could have simple commands, but they have complicated syntax. At least the syntax is complicated to the brand-new student. Many teenagers first learn to drive a car on the parking lot of a shopping mall, early on a Sunday morning. Empty parking lot driving is not real world driving, but it is an excellent way to get started. It is also possible that you first learned to ride a bicycle with training wheels. Once again, a bike with training wheels is not the real thing, but it does teach you how to ride a bicycle. When you are ready, the training wheels come off. This is precisely the nature of the **Expo** class. You will learn programming in a simpler, gentler environment. All the concepts you learn will be valid and help you to understand the logic of computer programming. In the next course, if you continue to AP Computer Science, the training wheels come off, and you will be surprised how easy the transition will be after the simpler introduction.

4.6 Introduction to Cartesian Graphics

Chapter II introduced the first set of programs. All of these earlier programs were application programs that displayed in a text window. It was mentioned that Java can also create special graphics programs that display in a web page. You will now see some examples of graphics programs with *applets*. Graphics programming is far more interesting than simple text output. You will be pleased to know that many program examples will be used with graphics and there will be

many lab assignments, including for this chapter, that require that you create a graphics program. Actually, right now the biggest reason for using graphics examples is not that it is more interesting. Graphics methods use many parameters and working with graphics methods helps to understand how to use parameters very well. Methods of the **Math** class are fine, but they have mostly a single parameter and at most two parameters. You will see that graphics methods have many parameters.

Learning Graphics Programming

Learning graphics programming is not simply a fun issue. You will learn many sophisticated computer science concepts by studying graphics programs.

Some of the most sophisticated programs are video games. Only very dedicated and knowledgeable programmers can write effective video games.

After the video game statement you may get excited and think that video games are next on the agenda. Well no. The summary box also mentioned the phrase *very dedicated and knowledgeable programmers*. You may be very dedicated, but right now the knowledge part is lacking. You will learn graphics early in the computer science course, but it will start with a gentle introduction of simple graphics routines.

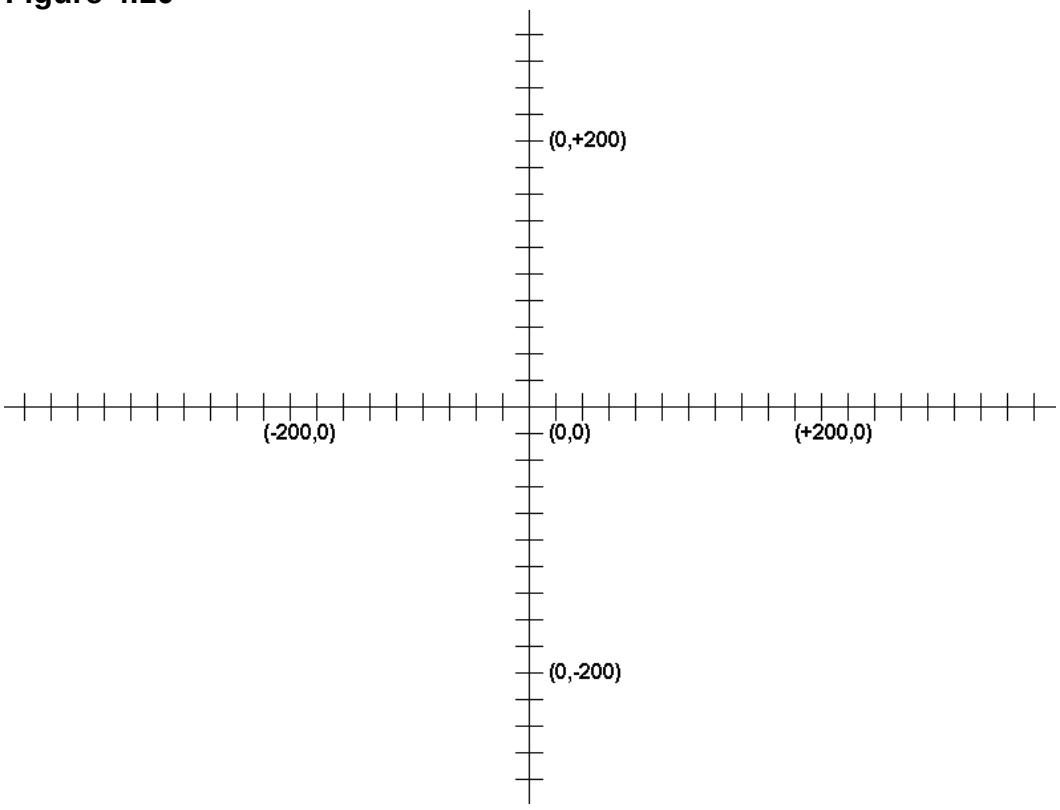
Graphics and Coordinate Geometry

Graphics images are not solid pictures, but images created with hundreds or thousands of individual colored dots. Individual graphics dots are called pixels. Pictures with a larger quantity of smaller pixels are said to have higher resolution and the quality of the image is better. The more difficult it is to see the individual pixels, the better the graphics image. Keep in mind that no matter how good the image quality is, every graphics picture, when enlarged, will display the pixels.

Graphics methods will draw a large variety of shapes on the monitor. The name of the method and the values of the provided parameters will determine the details of the graphics picture. The minimum requirement of any call to a graphics method is information about the location or coordinates for the picture. Selecting the correct coordinates requires an understanding of the coordinate system used by your computer in graphics.

A graphics window uses a system of (**X,Y**) coordinates in a manner similar to the use of coordinates that you first learned in your math classes. Figure 4.10 shows an example of the Cartesian coordinate system. In particular, note that the Cartesian system has four quadrants with the (0,0) coordinate located in the center of the grid where the X-Axis and the Y-Axis intersect.

Figure 4.10



The first set of *Expo* class methods will be referred to as *graphing* methods. They were specifically designed to help students with their Algebra skills, performing some of the same functions as a graphing calculator.

This first example shows the *drawGrid* method. This method has 4 parameters. The first parameter is ***g***. ***g*** is a **Graphics** variable. Any method that will do anything graphical will require this **Graphics** variable ***g***. The next 2 parameters

indicate the size of the grid. This should match the size of the applet window. You are about to compile and execute your first applet. This is different from compiling and executing applications. How do we know that program **Java0407.java** is an applet and not an application? One clue is the statement:

```
public class Java0407 extends Applet
```

Another clue is that there is no **main** method. Applications always have a **main** method. Applets do not. Instead, they have a **paint** method. Program **Java0407.java**, in figure 4.11, and the remainder of program examples in this chapter have some other features of note. In particular note the statements:

```
import java.awt.*;
import java.applet.*;
```

These statements are new and placed prior to the normal first program statement. The actual Java programming language is not that large, but hundreds of libraries with many classes and even more methods have been created for every conceivable program purpose. Computers do not have enough space in RAM to store all these libraries and no program requires access to every library. Java's solution is simple and used by many program languages. Take the standard libraries and store them on the hard drive at some known location. When any libraries are needed, use the **import** keyword followed by the library name. For these graphics programs you need to use classes of the **awt** (abstract windows tools) and **applet** libraries.

If we get back to the issue of compiling and executing there is another important difference between applications and applets. With applications, which are all the files you have used so far, you compile and execute the same file. With applets, you compile the .**java** file, and you execute the .**html** file. Remember that applets are designed to execute inside a webpage. This is why an .**html** file is required. In the example below, the .**java** file is shown first and the .**html** file is shown after that. In JCreator, you will need to load both .**java** and .**html** files in order to compile and execute.

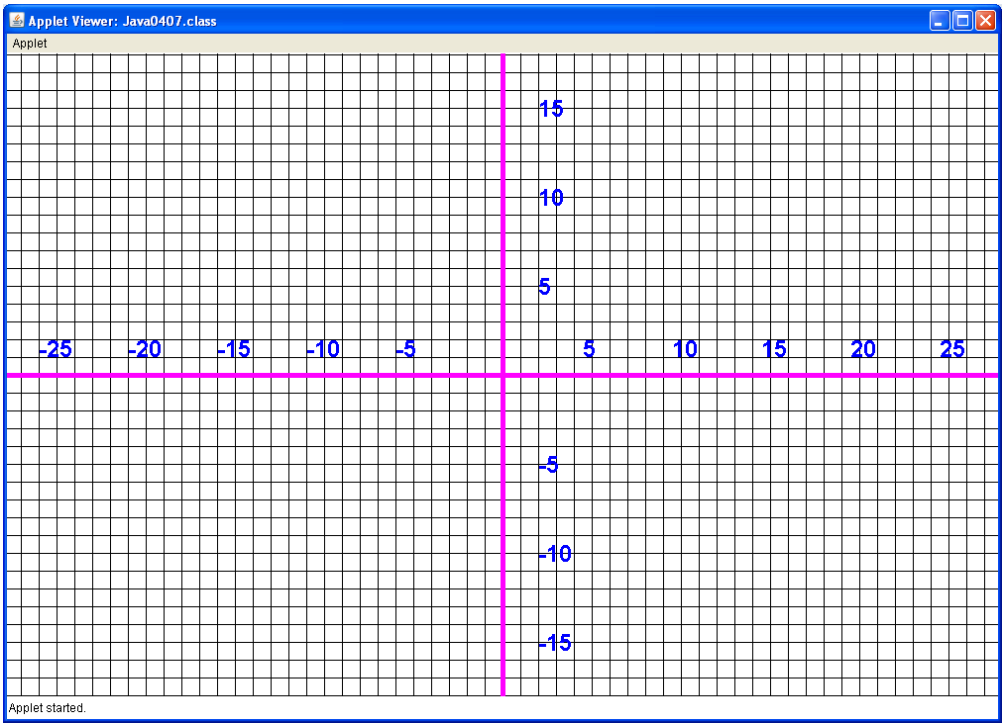
Figure 4.11

```
// Java0407.java
// This demonstrates the drawGrid method of the Expo class.
// This method must be called at the beginning of any program using
// Expo graphing methods.

import java.awt.*;
import java.applet.*;
```

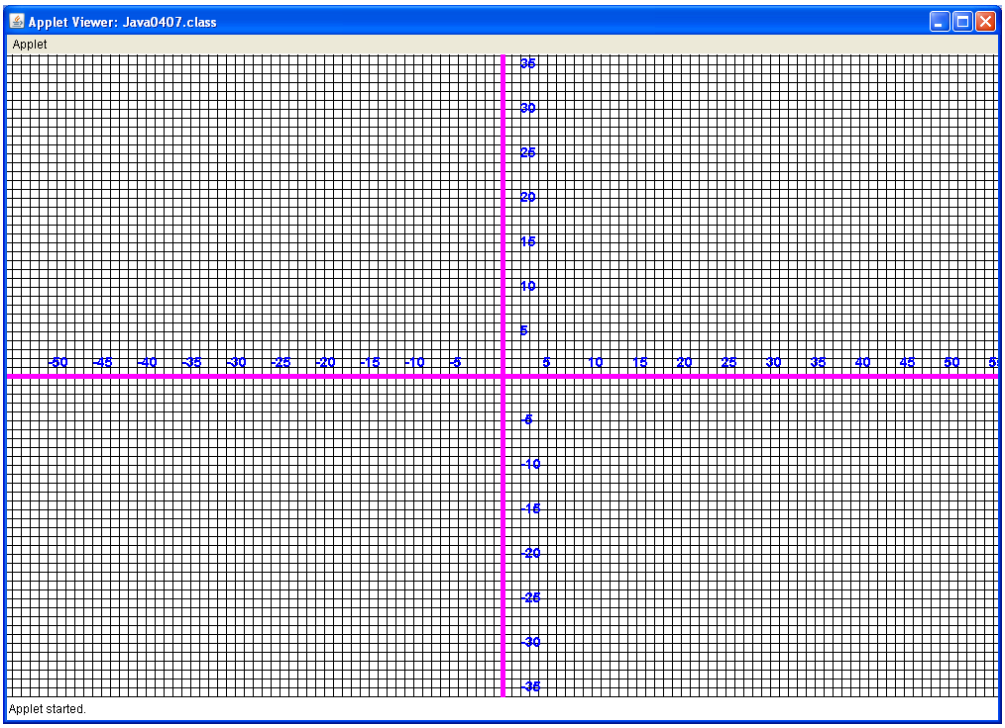
```
public class Java0407 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawGrid(g,1000,650,18);
    }
}
```

```
<APPLET CODE = "Java0407.class" WIDTH=1000 HEIGHT=650>
</APPLET>
```

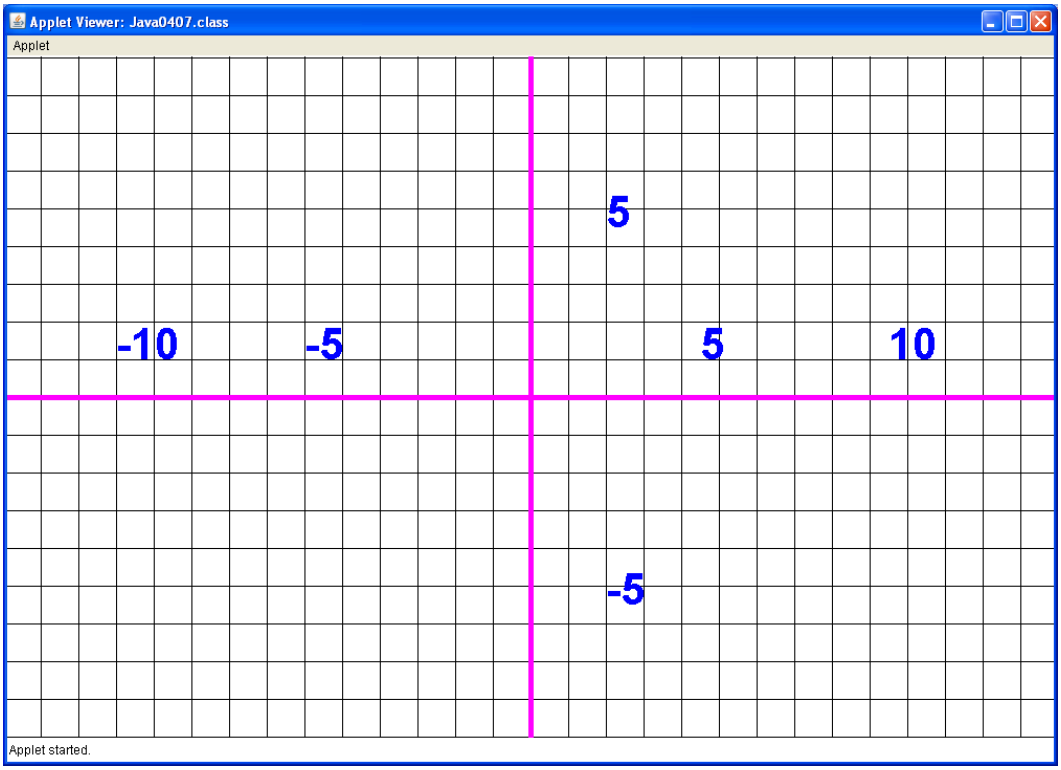
In the .**html** file you will notice that not only do we specify what file is to be executed, we also specify the width and height of the applet window. This is why we have a 1000 and a 650 as parameters in **drawGrid**. We will talk about the last parameter in a little bit, but first let us look at the program's output.



The last parameter, 18, indicates the scale of the graph. Change that number to 9, recompile, and re-execute. Remember that you must first compile the .**java** file and execute the .**html** file.



You should see that everything is half the size as it was before. Change the size to 36 and you will see that the grid cells are much bigger.



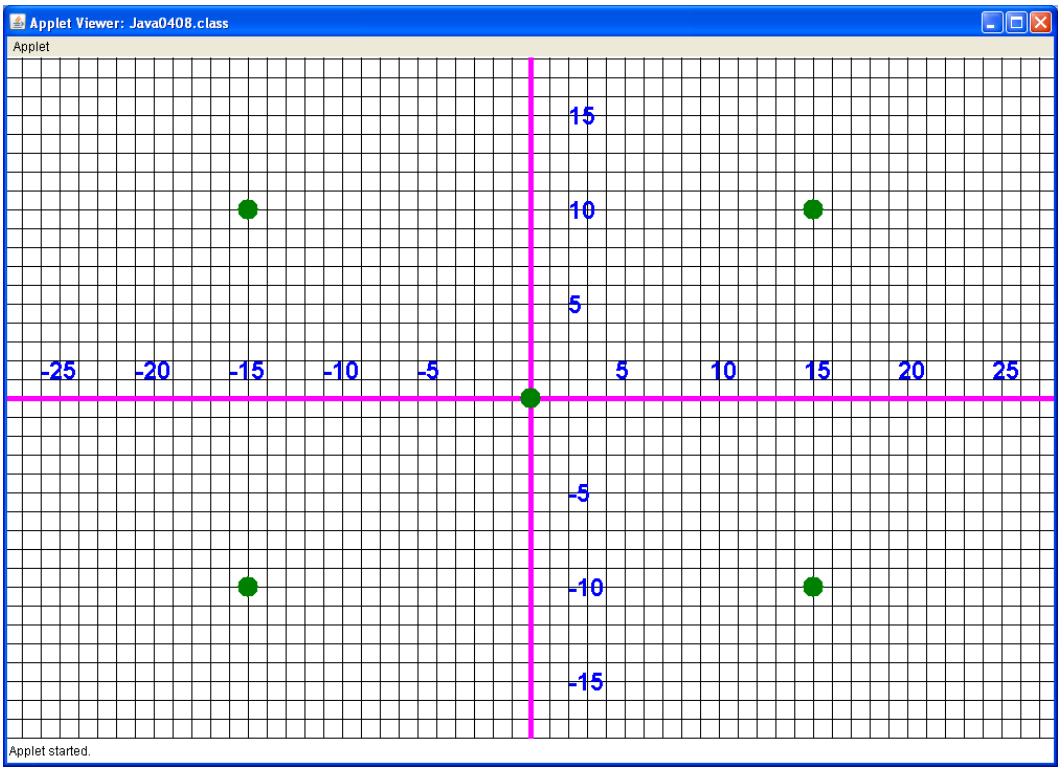
Graphing Points and Lines

Students in Algebra frequently are given points to plot from a provided equation. Program **Java0408.java**, in figure 4.12, demonstrates **graphPoint**. You will notice the same ***g*** parameter at the front. There are 2 other parameters. They are the X and Y location of the point.

Figure 4.12

```
// Java0408.java
// This demonstrates the graphPoint method of the Expo class.
// Remember, drawGrid must be called first.

import java.awt.*;
import java.applet.*;

public class Java0408 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawGrid(g,1000,650,18);

        Expo.graphPoint(g,15,10);
        Expo.graphPoint(g,15,-10);
        Expo.graphPoint(g,-15,10);
        Expo.graphPoint(g,-15,-10);
        Expo.graphPoint(g,0,0);
    }
}
```

```
<APPLET CODE = "Java0408.class" WIDTH=1000 HEIGHT=650>
</APPLET>
```



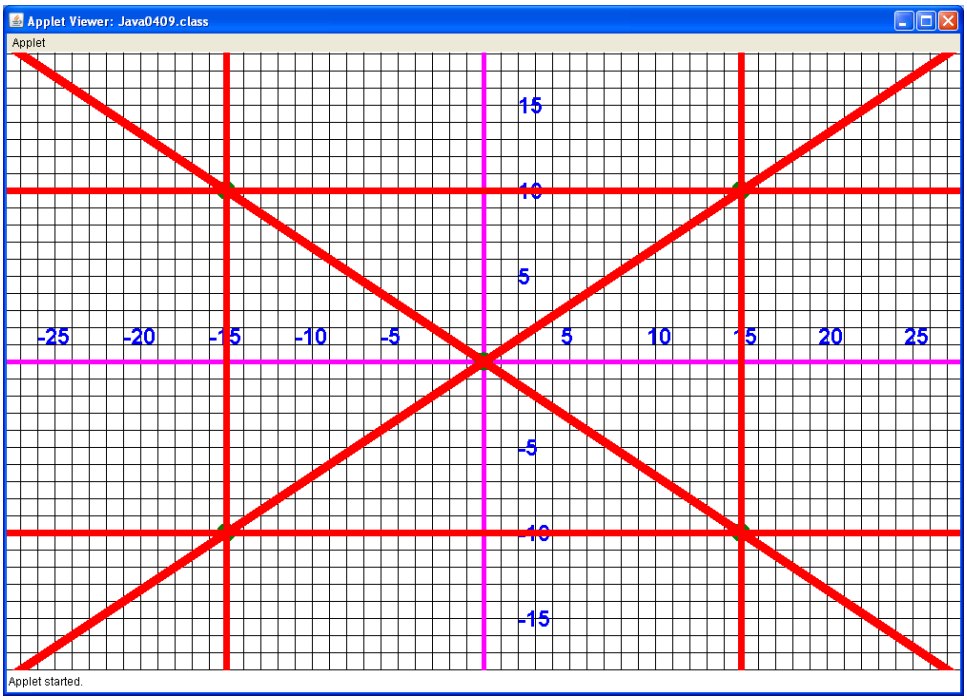
Program **Java0409.java**, in figure 4.13, adds lines. To graph a line you need 2 coordinate points. That means an X and Y value for the first point, and another X and Y value for the second point. This is usually documented as X1, Y1, X2, Y2. Keep in mind that these are *lines* and not *line segments*. A *line segment* would start at one point and end at another. A *line* goes forever in both directions, which you should see in the example on the next page.

Figure 4.13

```
// Java0409.java
// This demonstrates the graphLine method of the Expo class.
// The graphPoint method calls from the previous program are included intentionally.
// Remember, drawGrid must be called first.

import java.awt.*;
import java.applet.*;

public class Java0409 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawGrid(g,1000,650,18);

        Expo.graphPoint(g,15,10);
        Expo.graphPoint(g,15,-10);
        Expo.graphPoint(g,-15,10);
        Expo.graphPoint(g,-15,-10);
        Expo.graphPoint(g,0,0);
```

```
        Expo.graphLine(g,15,10,15,-10);
        Expo.graphLine(g,15,10,-15,10);
        Expo.graphLine(g,15,10,-15,-10);
        Expo.graphLine(g,-15,-10,15,-10);
        Expo.graphLine(g,-15,-10,-15,10);
        Expo.graphLine(g,-15,10,15,-10);
    }
}
```

```
<APPLET CODE = "Java0409.class" WIDTH=1000 HEIGHT=650>
</APPLET>
```

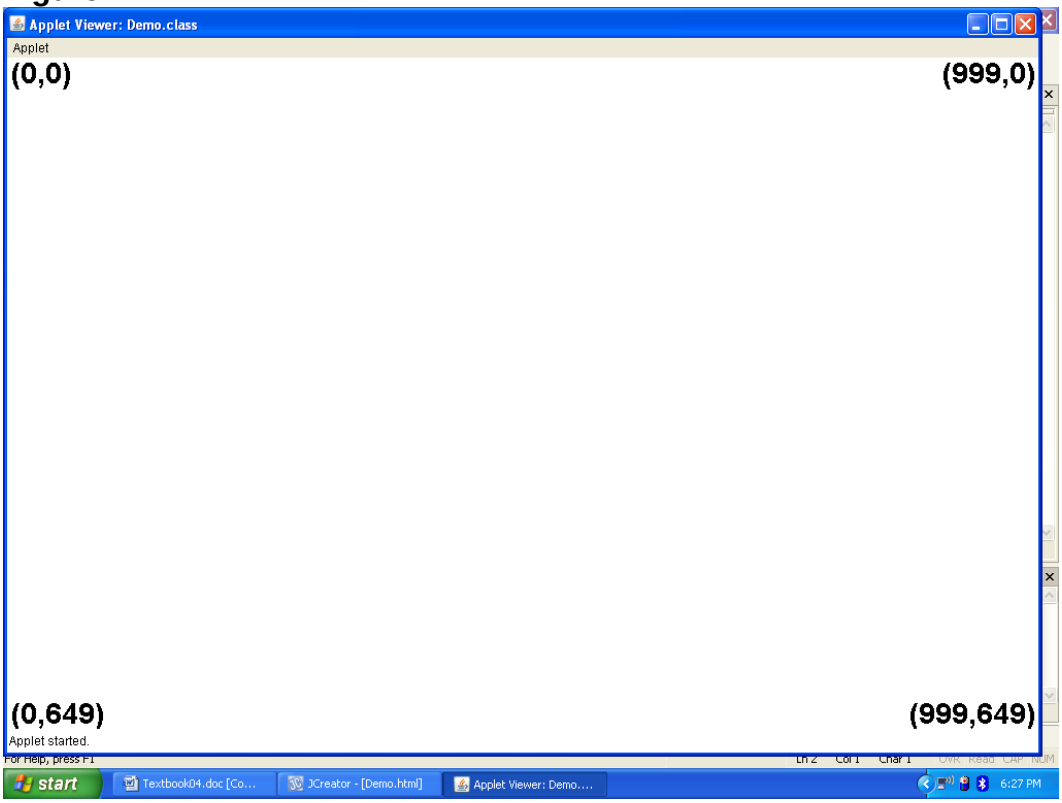


4.7 Introduction to Computer Graphics

Assuming now that you are comfortable with the Coordinate Geometry you learned in Algebra and Geometry classes, you must realize that computer graphics uses a coordinate system that is logically the same as the Cartesian system, but has some important fundamental differences. Figure 4.14 shows an applet window

dimensioned to **1000 x 650** pixels. The window is a little smaller than the actual resolution of the monitor display, which is **1024 x 768** pixels. The coordinate values at the four corners of the applet window are displayed. There are two significant differences to observe. The (**0,0**) coordinate is located in the left-top corner of the applet window. The graphics window behaves as if it were just one of the four quadrants of the Cartesian system. The second difference is with the behavior of the **Y** coordinate values. In a Cartesian system **Y**-values increase from the bottom to the top. In a computer graphics system **Y**-values increase from the top down to the bottom. The **X**-values in both coordinate systems increase from left to right.

Figure 4.14



Like the *graphing* methods, the *graphics* programs that follow are created as Java applets. It is possible to create the same exact display with Java applications, but graphics applications are actually more complex than Java applets. Since all examples are applets you need to make sure that you remember to compile the Java source code file, and then switch to some small web page file for execution.

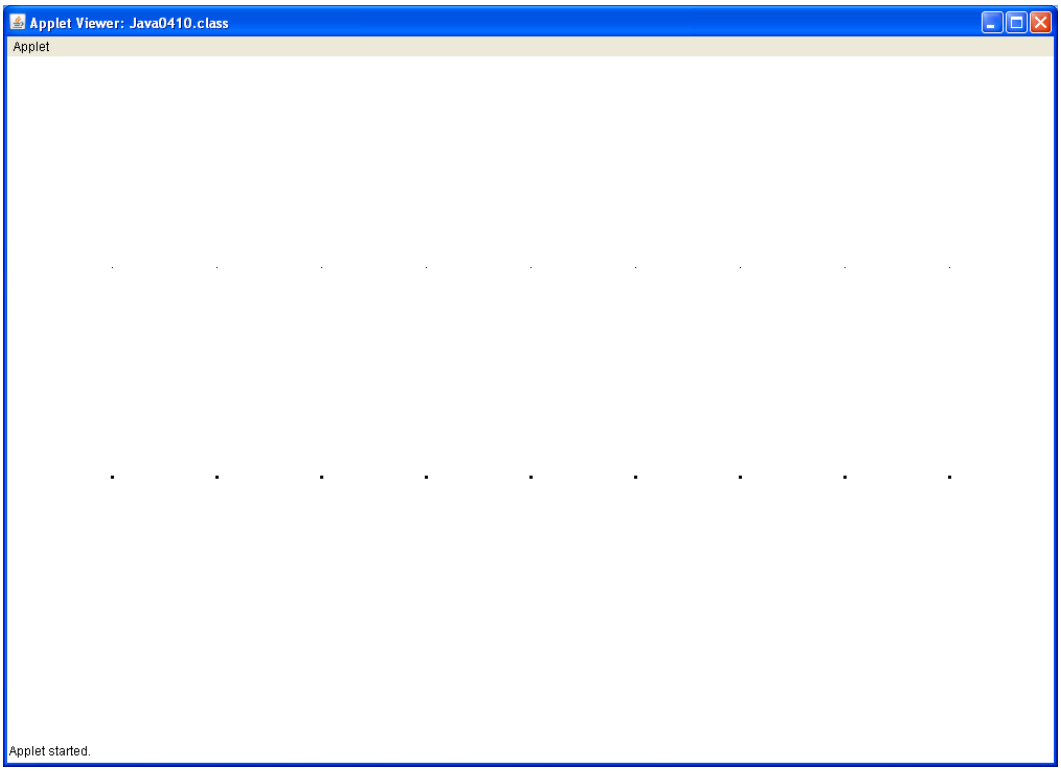
Drawing Pixels and Points

Program **Java0410.java**, in figure 4.15, will demonstrate two very simple methods of the **Expo** class. Both of these methods will require the same parameters. They need the **Graphics** variable, also called object ***g***, as well as the X and Y value of the pixel/point.

Figure 4.15

```
// Java0410.java
// This demonstrates the drawPixel and drawPoint methods of the Expo class.

import java.awt.*;
import java.applet.*;

public class Java0410 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawPixel(g,100,200);
        Expo.drawPixel(g,200,200);
        Expo.drawPixel(g,300,200);
        Expo.drawPixel(g,400,200);
        Expo.drawPixel(g,500,200);
        Expo.drawPixel(g,600,200);
        Expo.drawPixel(g,700,200);
        Expo.drawPixel(g,800,200);
        Expo.drawPixel(g,900,200);

        Expo.drawPoint(g,100,400);
        Expo.drawPoint(g,200,400);
        Expo.drawPoint(g,300,400);
        Expo.drawPoint(g,400,400);
        Expo.drawPoint(g,500,400);
        Expo.drawPoint(g,600,400);
        Expo.drawPoint(g,700,400);
        Expo.drawPoint(g,800,400);
        Expo.drawPoint(g,900,400);
    }
}
```

```
<APPLET CODE = "Java0410.class" WIDTH=1000 HEIGHT=650>
</APPLET>
```



You might have a difficult time seeing the pixels that were plotted by the **drawPixel** commands. There are 9 evenly spaced pixels between the top and middle of the screen. If you do not see them, then simply trust me that they are there. Part of what makes modern digital images so sharp is that they are made up of millions of tiny pixels. The smaller the pixel, the sharper the image. This might explain why it is hard to see just one.

Along that line, we have the **drawPoint** method. This works just like **drawPixel** in terms of parameters. The difference is in the output. **drawPixel** only draws one pixel. **drawPoint** actually draws a 3 by 3 "square" of 9 pixels. The center of the square is the X and Y coordinate you specified. The purpose for **drawPoint** is to have something a little more visible than **drawPixel**.

4.8 Drawing Methods

The majority of the methods in the **Expo** class are graphics method. A large portion of those are *drawing* methods. The next several programs will be examples of these *drawing* methods.

Also, the previous 4 programs showed the .**html** file along with the .**java** file. We think by now we have established that an applet requires an .**html** file. The remainder of the applet programs will show only the .**java** file.

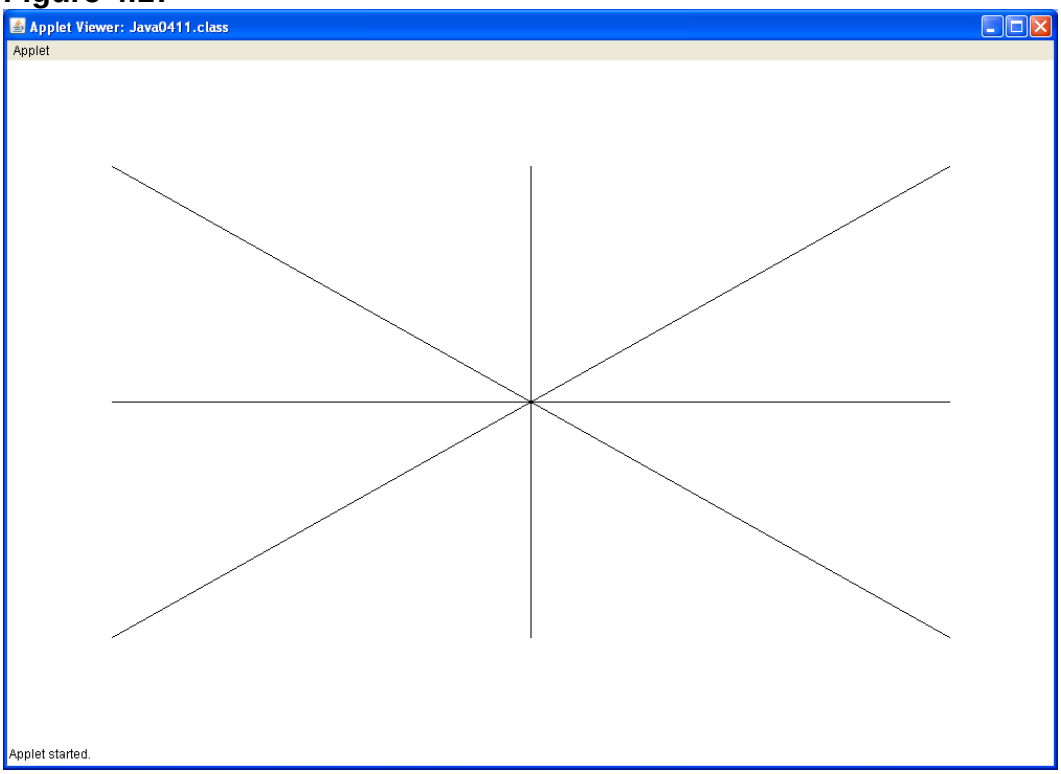
Drawing Lines

This may seem very similar to the **graphLine** method you saw earlier. The parameters are even the same. There are a couple important differences. First, **graphLine** is used on a Cartesian graph while **drawLine** is used on an applet window. Second, **graphLine** drew lines which went on forever in both directions. **drawLine** draw *line segments* which start at one point and finish at another. Program **Java0411.java**, in figure 4.16 demonstrates **drawLine**.

Figure 4.16

```
// Java0411.java
// This program demonstrates the drawLine method of the Expo class.
// Lines are drawn from (X1,Y1) to (X2,Y2) with drawLine(g,X1,Y1,X2,Y2).

import java.awt.*;
import java.applet.*;

public class Java0411 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawLine(g,100,100,900,550);
        Expo.drawLine(g,100,550,900,100);
        Expo.drawLine(g,100,325,900,325);
        Expo.drawLine(g,500,100,500,550);
    }
}
```

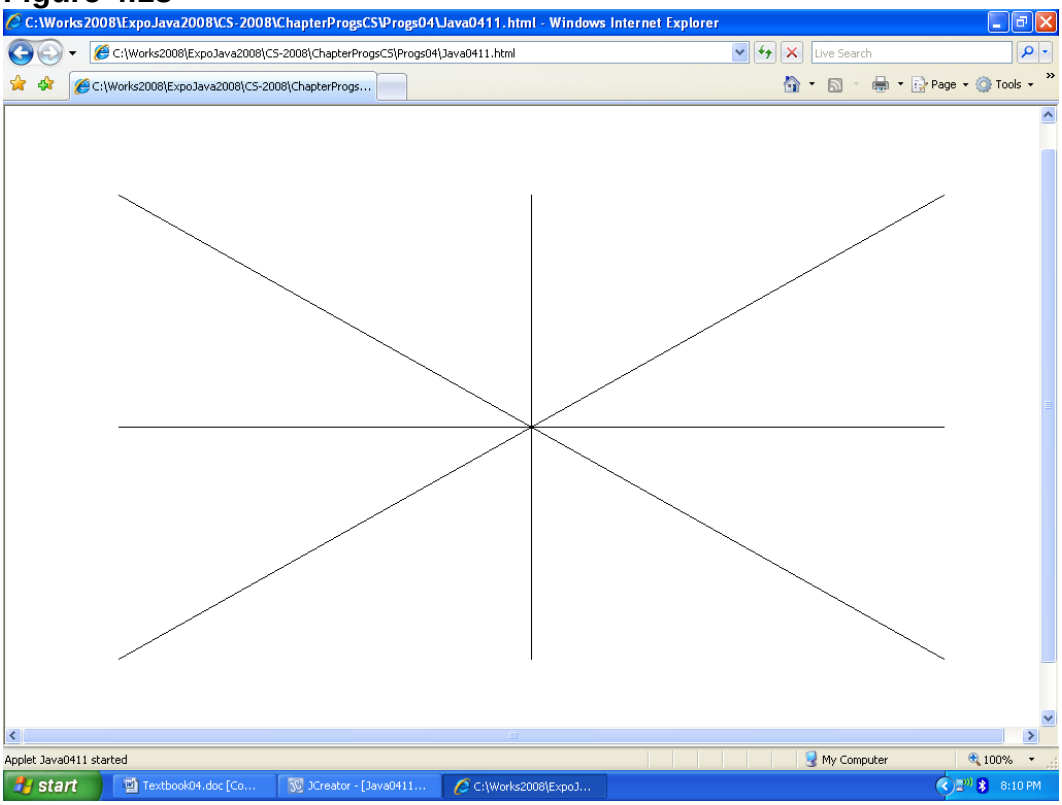
Figure 4.17



In figure 4.17 above you see the output as you have in earlier programs. We can see that we do in fact have line segments because the lines have both a beginning point and an ending point.

On a completely different note, do you remember that we said applet programs are designed to execute inside a webpage? The file we execute is an .**html** file after all. While JCreator has the ability to execute applets like the previous program, applets can also be executed by loading the **html** file directly into a web browser like *Netscape Navigator* or *Internet Explorer*. An example is shown on the next page in figure 4.18. Remember that the whole point in creating an applet is to have a program that can execute as part of a web page. In spite of this web browser capability, all graphics examples will executed from JCreator rather than a web browser because it is much faster, especially for lab lectures.

Figure 4.18



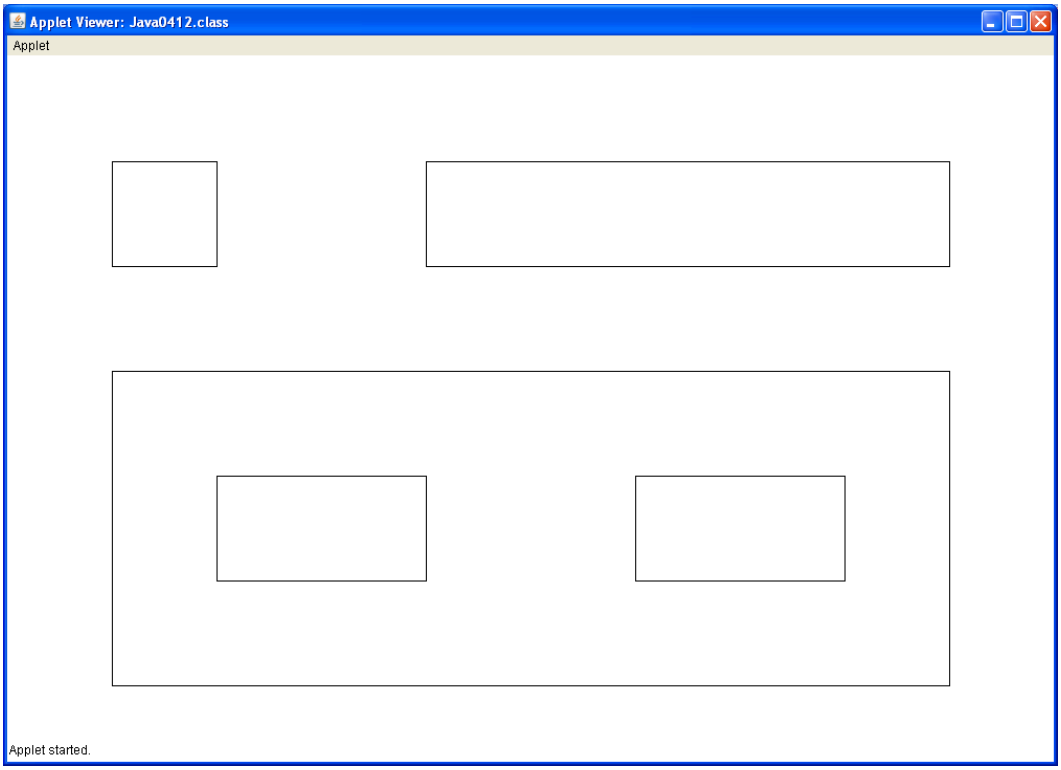
It is possible that when you do this, you might get an annoying "blocked content" message in a yellow bar at the top of the window. Just right click it and select "Allow blocked content."

Drawing Rectangles

A line is one-dimensional; it only has length. Many of the graphics methods of the **Expo** class are two-dimensional and have the option to draw different shapes. **drawRectangle** is demonstrated in program **Java0412.java**, in figure 4.19. The parameters for **drawRectangle** are actually the same as **drawLine** with one important difference. In **drawLine**, the 2 coordinate point you specify (as in X1, Y1, X2, Y2) are the starting point and the stopping point of the line. In **drawRectangle**, you also specify 2 coordinate points, but they represent the upper-left-hand coordinate (X1,Y1) and the lower-right-hand coordinate (X2,Y2).

Figure 4.19

```
// Java0412.java
// This program demonstrates the drawRectangle method of the Expo class.
// Rectangles are drawn from the upper-left-hand corner(X1,Y1) to the
// lower-right-hand corner(X2,Y2) with drawRectangle(g,X1,Y1,X2,Y2).

import java.awt.*;
import java.applet.*;

public class Java0412 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawRectangle(g,100,100,200,200);
        Expo.drawRectangle(g,400,100,900,200);
        Expo.drawRectangle(g,100,300,900,600);
        Expo.drawRectangle(g,200,400,400,500);
        Expo.drawRectangle(g,600,400,800,500);
    }
}
```

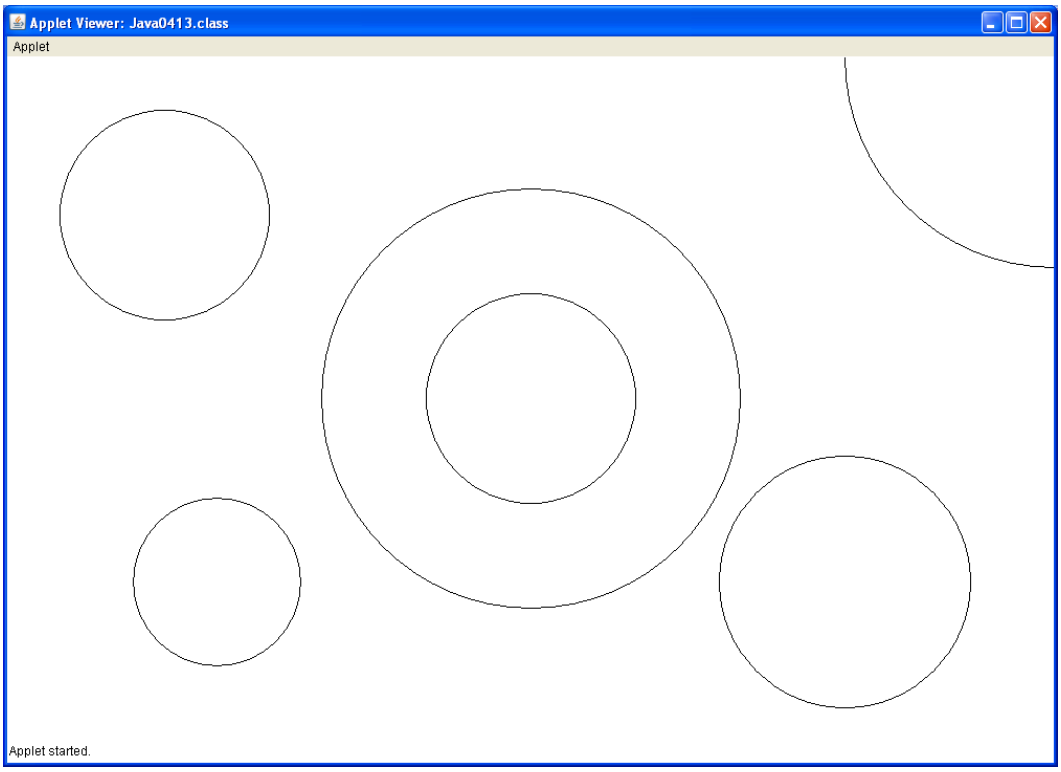


Drawing Circles

The mathematical definition of a circle is *a set of points in a plane that are equidistant from a given point*. The given point is the X and Y coordinate of the center of the circle. The distance from the center of the circle to any point on the edge of the circle itself is called the *radius*. No matter how you draw a line from the circle's center to its edge, the distance will be the same (hence the word "equidistant" earlier). It should come as no surprise that the parameters for *drawCircle* will involve the X and Y coordinate of the center, as well as the radius. Program **Java0413.**java, in figure 4.20, draws several circles.

Figure 4.20

```
// Java0413.java
// This program demonstrates the drawCircle method of the Expo class.
// Circles are drawn from their center (X,Y) with a particular radius
// with drawCircle(g,X,Y,radius).

import java.awt.*;
import java.applet.*;

public class Java0413 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawCircle(g,150,150,100);
        Expo.drawCircle(g,1000,0,200);
        Expo.drawCircle(g,500,325,100);
        Expo.drawCircle(g,500,325,200);
        Expo.drawCircle(g,200,500,80);
        Expo.drawCircle(g,800,500,120);
    }
}
```



Drawing Ovals

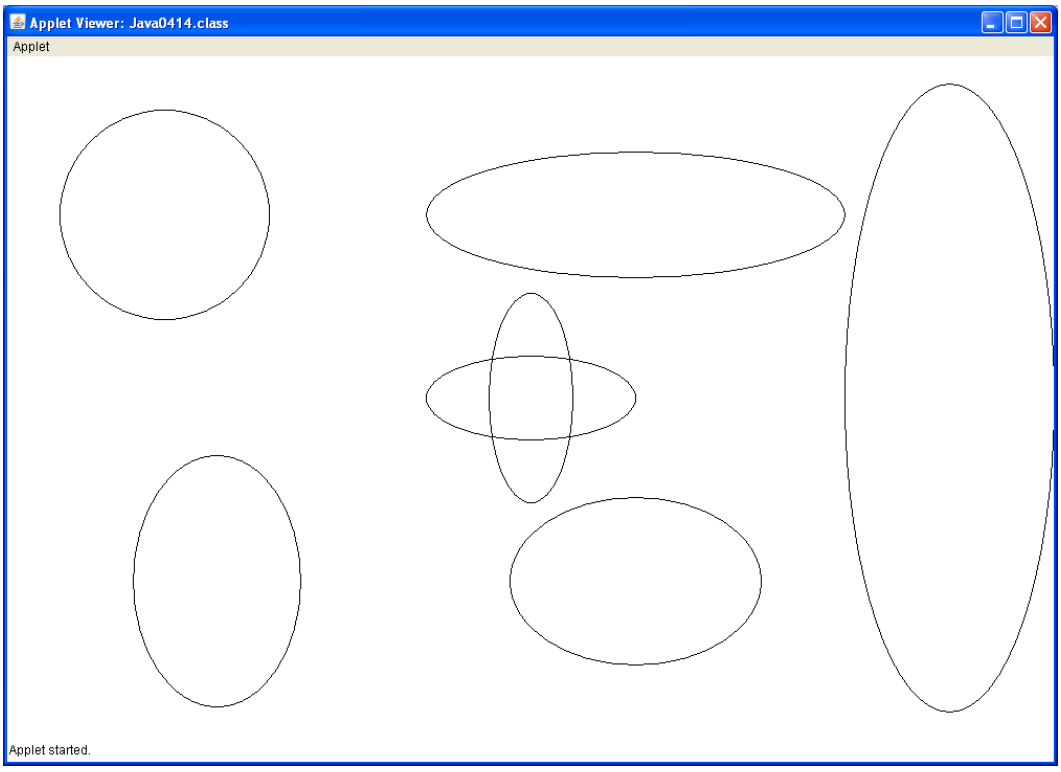
Drawing an oval is very similar to drawing a circle. You still have to specify the X and Y coordinate of the center. What is different is now you do not have one radius. You instead have 2 *radii* (plural for radius). There is a horizontal radius, and a vertical radius. In the same way, the parameters for **drawOval** are very similar to **drawCircle**. The only difference is instead of only one radius parameter, you have a parameter for the horizontal radius, followed by another parameter for the vertical radius. Program **Java0414.**java, in figure 4.21, draws several ovals.

Figure 4.21

```
// Java0414.java
// This program demonstrates the drawOval method of the Expo class.
// Ovals are drawn from their center (X,Y) with a horizontal radius (hr)
// and a vertical radius (vr) with drawCircle(g,X,Y,hr,vr).

import java.awt.*;
import java.applet.*;
```

```
public class Java0414 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawOval(g,150,150,100,100);
        Expo.drawOval(g,900,325,100,300);
        Expo.drawOval(g,600,150,200,60);
        Expo.drawOval(g,500,325,40,100);
        Expo.drawOval(g,500,325,100,40);
        Expo.drawOval(g,200,500,80,120);
        Expo.drawOval(g,600,500,120,80);
    }
}
```

When you look at the output on the next page, notice the *circle* near the upper left hand corner. You may wonder how a circle was drawn with the *drawOval* command. Well look at the first **drawOval** command in the program. You will see the same value is passed for the horizontal radius, and the vertical radius. When these 2 radii are equal, you get a circle. You should also notice that some of the ovals are tall and thin, while others are short and fat. If the horizontal radius is greater than the vertical radius, you get a short, fat oval. However, if the vertical radius is greater, the oval is tall and thin.



Drawing Arcs

The first thing you need to understand about drawing an arc is *an arc is a piece of an oval*. Because of this, you will notice that the first 5 parameters of **drawArc** are identical to **drawOval**. You still have to use the **Graphics** object *g*, followed by the X and Y coordinate value of the center, and the 2 radii. For an arc you need 2 additional parameters. To understand what they mean, look at the clock below:

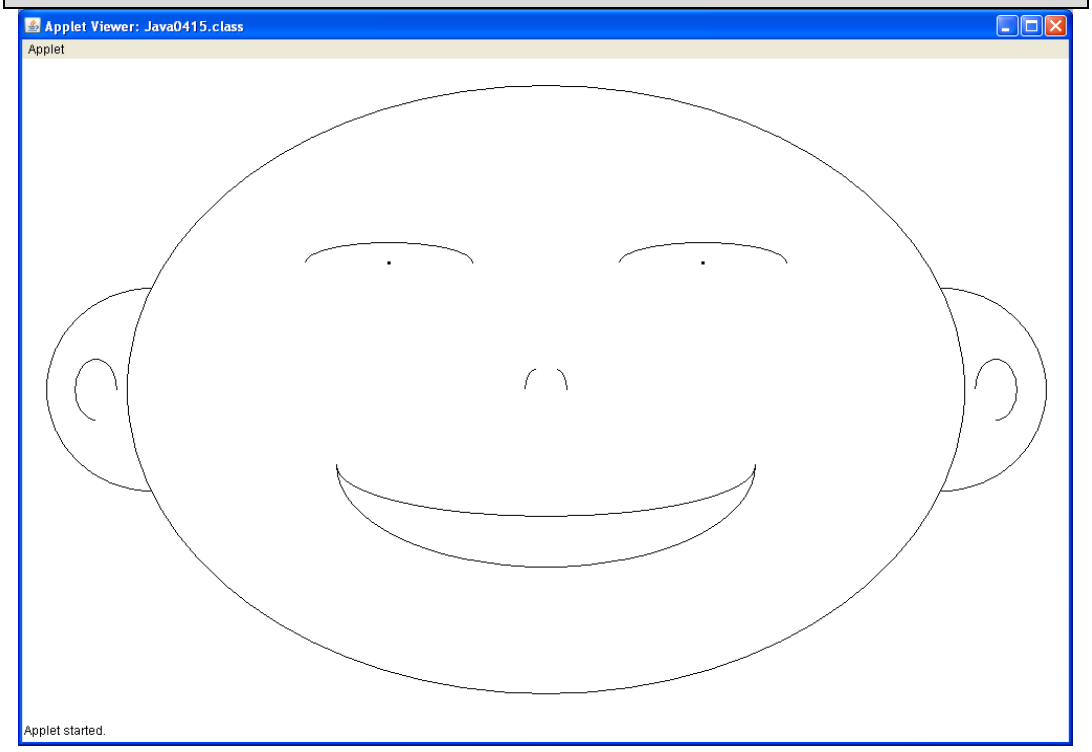
Compare the clock to a circle. In math classes you learned that a circle has 360 degrees. To draw an *arc*, you need to specify the *starting degree value*, and the *stopping degree value*. On a clock, the 0 degree position is at 12:00. The 90 degree position is at 3:00. 180 degrees is at 6:00. 270 degrees is at 9:00. And 360 degrees is back at 12:00.

If you want to draw the bottom half of a circle – possibly to draw a smiley face ☺ – you need to use 90 and 270 as the last 2 parameters of the **drawArc** command. This draws a partial oval which starts at 90 degrees (3:00) and goes clockwise to 270 degrees (9:00). If the 90 and 270 are reversed, you get the top half of the oval, because it will start at 270 degrees (9:00) and end at 90 degrees (3:00).

Figure 4.22 shows program **Java0415.java** which draws several arcs. Unlike several of the previous programs, which just drew several shapes, this program combines the arcs into a picture. With the exception of the 2 points used for the eyes, everything you see in the output is done with an arc. The head may look a lot like an oval, but it is actually an arc that is starting at 0 degrees (12:00) and going 360 degrees completely around the oval.

Figure 4.22

```
// Java0415.java
// This program demonstrates the drawArc method of the Expo class.
// An "arc" is a piece of an "oval".
// like ovals, arcs are drawn from their center (X,Y) with a horizontal radius (hr)
// and a vertical radius (vr).  Arcs also require a starting and stopping degree
// value.  This is done with drawArc(g,X,Y,hr,vr,start,stop).

import java.awt.*;
import java.applet.*;

public class Java0415 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawArc(g,500,325,400,300,0,360);      // complete oval
        Expo.drawArc(g,500,400,200,50,90,270);      // bottom half of an oval
        Expo.drawArc(g,500,400,200,100,90,270);
        Expo.drawArc(g,350,200,80,20,270,90);       // top half of an oval
        Expo.drawArc(g,650,200,80,20,270,90);
        Expo.drawArc(g,123,325,100,100,180,0);      // left half of an oval
        Expo.drawArc(g,878,325,100,100,0,180);      // right half of an oval
        Expo.drawArc(g,490,325,10,20,270,360);      // top-left 1/4 of an oval
        Expo.drawArc(g,510,325,10,20,0,90);         // top-right 1/4 of an oval
        Expo.drawArc(g,70,325,20,30,180,90);        // 3/4 of an oval
        Expo.drawArc(g,930,325,20,30,270,180);      // different 3/4 of an oval
        Expo.drawPoint(g,350,200);
        Expo.drawPoint(g,650,200);
    }
}
```



Watch Your Parameter Sequence

Students enjoy working with graphics. It is not very difficult to determine the information that is needed by graphics methods. Problems can occur when the sequence of parameters is ignored or misunderstood. Program **Java0416.java**, in figure 4.23, is pretty much the same program as **Java0415.java**. The difference is that program **Java0416.java** has rearranged the parameters. Look at figure 4.24 and see how simply changing the order of the parameters affects the output.

Figure 4.23

```
// Java0416.java
// This repeats the previous program which drew the smiley face.
// The program demonstrates what happens parameters are put in the wrong order.
// The program might compile and execute, but the results are not what you expect.


import java.awt.*;
import java.applet.*;

public class Java0416 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawArc(g,325,500,400,300,0,360);
        Expo.drawArc(g,500,400,50,200,90,270);
        Expo.drawArc(g,400,500,200,100,270,90);
        Expo.drawArc(g,200,350,20,80,270,90);
        Expo.drawArc(g,650,200,80,20,90,270);
        Expo.drawArc(g,123,325,100,100,0,180);
        Expo.drawArc(g,878,325,100,100,180,0);
        Expo.drawArc(g,490,325,10,20,270,360);
        Expo.drawArc(g,325,510,10,20,90,0);
        Expo.drawArc(g,325,70,20,30,90,270);
        Expo.drawArc(g,930,325,30,20,270,180);
        Expo.drawPoint(g,200,350);
        Expo.drawPoint(g,650,200);
    }
}
```

Figure 4.24

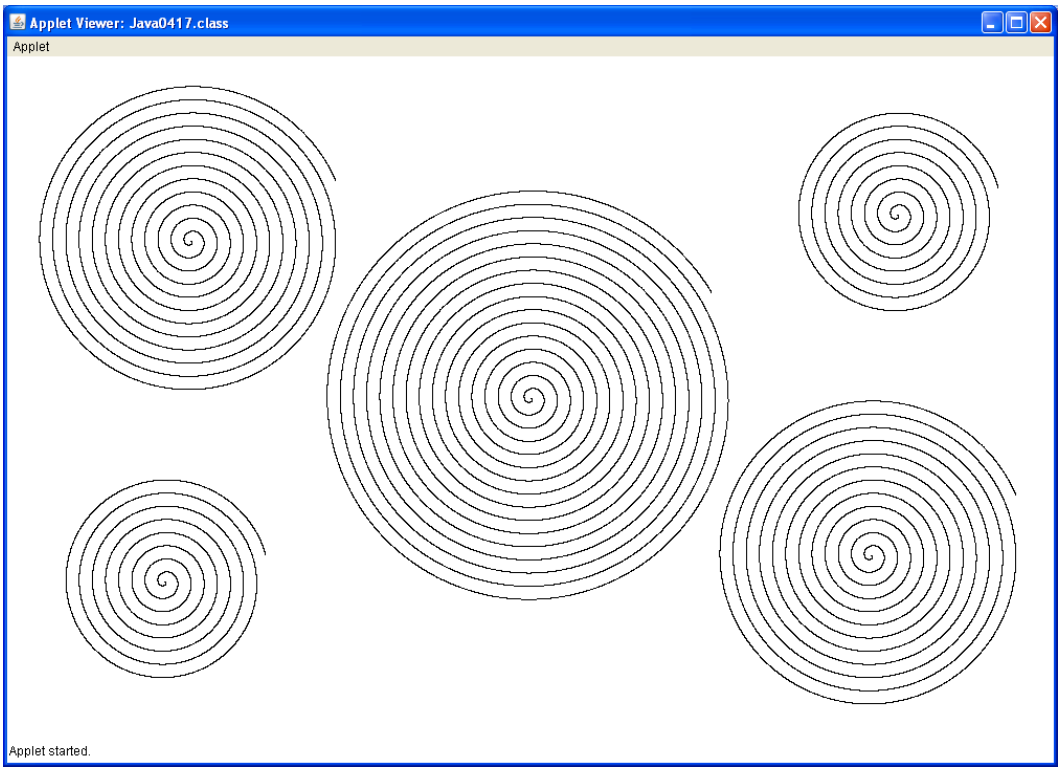
Drawing Spirals

We now have the basic shapes behind us. Most graphics packages have lines, rectangles, ovals, and arcs. The **Expo** class has added some extra methods. One of these is the **drawSpiral** method. The parameters of **drawSpiral** are identical to **drawCircle**. Just like a circle, a spiral has a center with an X and Y coordinate. A spiral also has a radius. The way it is drawn is it starts in the center and draws outward in a "spiral-like" pattern until the desired radius is reached. Program **Java0417.java**, in figure 4.25, shows 5 spirals being drawn.

Figure 4.25

```
// Java0417.java
// This program demonstrates the drawSpiral method of the Expo class.
// Spirals are drawn from their center (X,Y) with a certain radius
// with drawSpiral(g,x,y,radius).

import java.awt.*;
import java.applet.*;
```

```
public class Java0417 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawSpiral(g,500,325,200);
        Expo.drawSpiral(g,175,175,150);
        Expo.drawSpiral(g,850,150,100);
        Expo.drawSpiral(g,150,500,100);
        Expo.drawSpiral(g,825,475,150);
    }
}
```

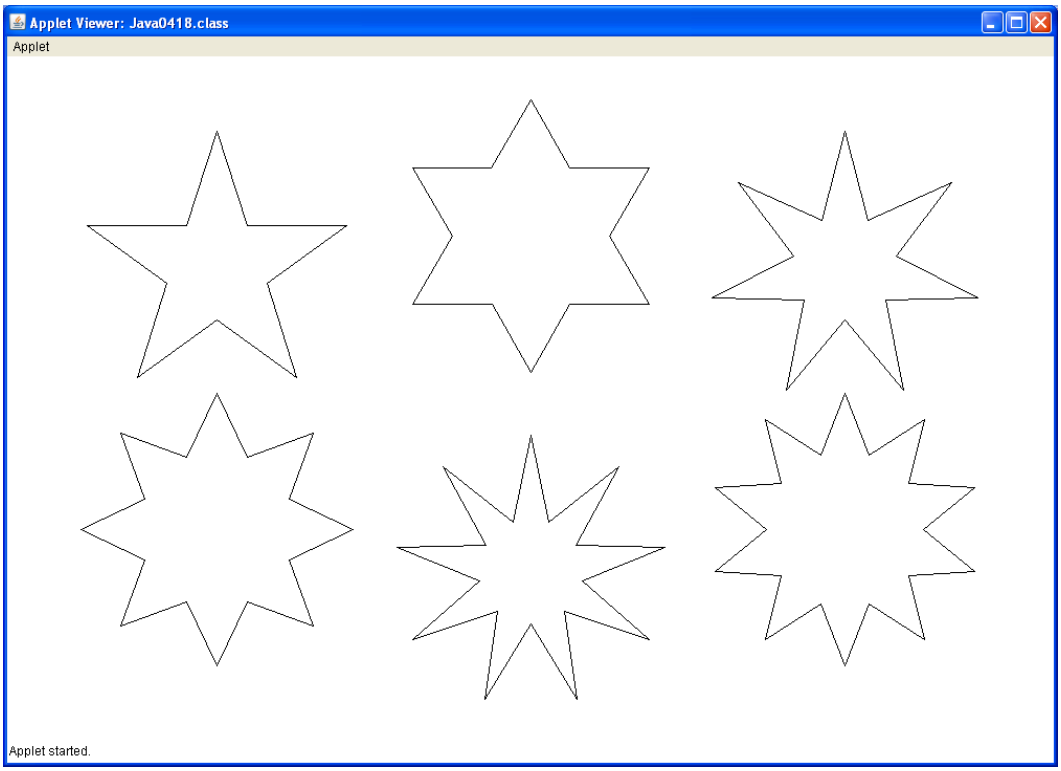


Drawing Stars

Like spirals, drawing stars is very similar to draw circles. A star also has a center, so there is an X and Y parameter for the center as before. The distance from the center of a star to any point is the *radius* of the star. So there will be a radius parameter as well. The only difference is there is an extra parameter for the number of points you want on the star. Program **Java0418.java**, in figure 4.26, draws a 5, 6, 7, 8, 9 and 10 pointed stars. Every star is a *regular* star, which means that the distance between points on the star are all the same.

Figure 4.26

```
// Java0418.java
// This program demonstrates the drawStar method of the Expo class.
// Stars are drawn from their center (X,Y) with a certain radius
// and a certain number of points with drawStar(g,x,y,radius,numPoints).

import java.awt.*;
import java.applet.*;

public class Java0418 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawStar(g,200,200,130,5);
        Expo.drawStar(g,500,170,130,6);
        Expo.drawStar(g,800,200,130,7);
        Expo.drawStar(g,200,450,130,8);
        Expo.drawStar(g,500,490,130,9);
        Expo.drawStar(g,800,450,130,10);
    }
}
```



4.9 Fill and Thick Methods

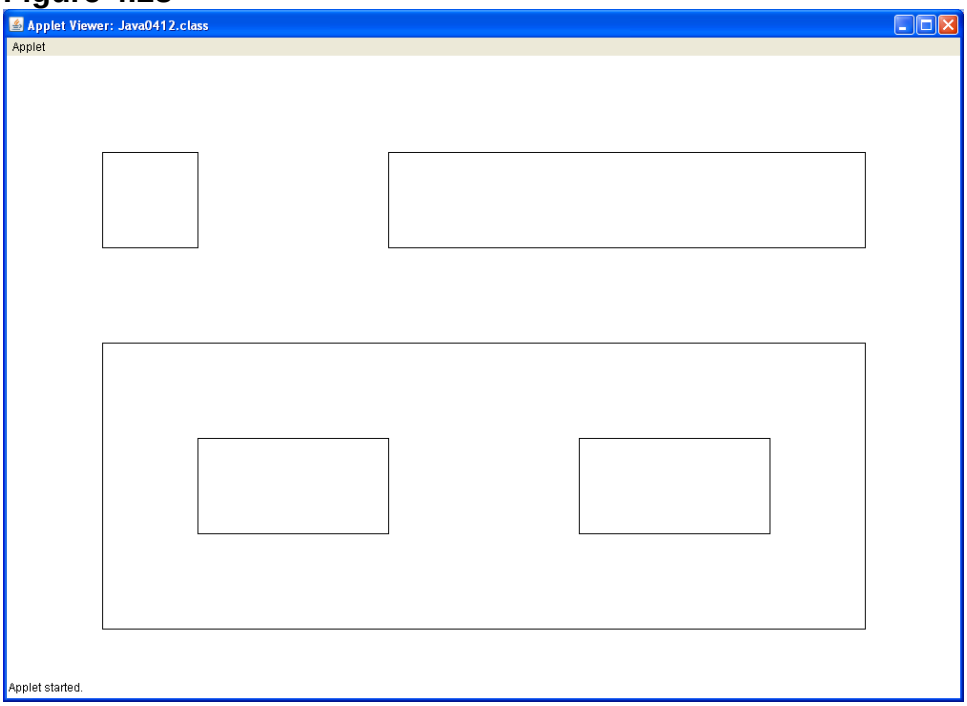
You have been shown several *drawing* methods of the **Expo** class. There are actually more *drawing* methods in the **Expo** class that you will be shown in other chapters. For now, we are going to look at *fill* methods. Most of the *drawing* methods, except for **drawLine** and **drawSpiral**, have a corresponding *fill* method. The *fill* methods have the exact same parameters, with the exact same meaning as the corresponding *draw* methods. For example, program **Java0419.java**, in figure 4.27, repeats the **drawRectangle** example from program **Java0412.java**. That program drew five *open* rectangles. In figure 4.27 the program simply changes the word **draw** to **fill**. Now five *solid* or *filled in* rectangles are drawn.

Figure 4.27

```
// Java0419.java
// This program demonstrates the fillRectangle method of the Expo class.
// The parameters are the same as drawRectangle.
// Even though 5 solid rectangles are drawn, only 3 show up on the screen.
// Where are the other 2?

import java.awt.*;
import java.applet.*;

public class Java0419 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.fillRectangle(g,100,100,200,200);
        Expo.fillRectangle(g,400,100,900,200);
        Expo.fillRectangle(g,100,300,900,600);
        Expo.fillRectangle(g,200,400,400,500);
        Expo.fillRectangle(g,600,400,800,500);
    }
}
```

When you look at the output, you might be surprised to see that only three rectangles show up? Where did the other two go? Let us look at the output of program **Java0412.java**, in figure 4.28, again. You do see five rectangles there.

Figure 4.28



Do you see the two small rectangles inside the one big one? Well, when all of these are solid, filled in, rectangles the two small solid black rectangles are drawn on top of a bigger solid, filled in, rectangle which is also black. The result is the two smaller rectangles are not visible. The only way to make them visible is to change their color. Program **Java0420.java**, in figure 4.29, will take care of that.

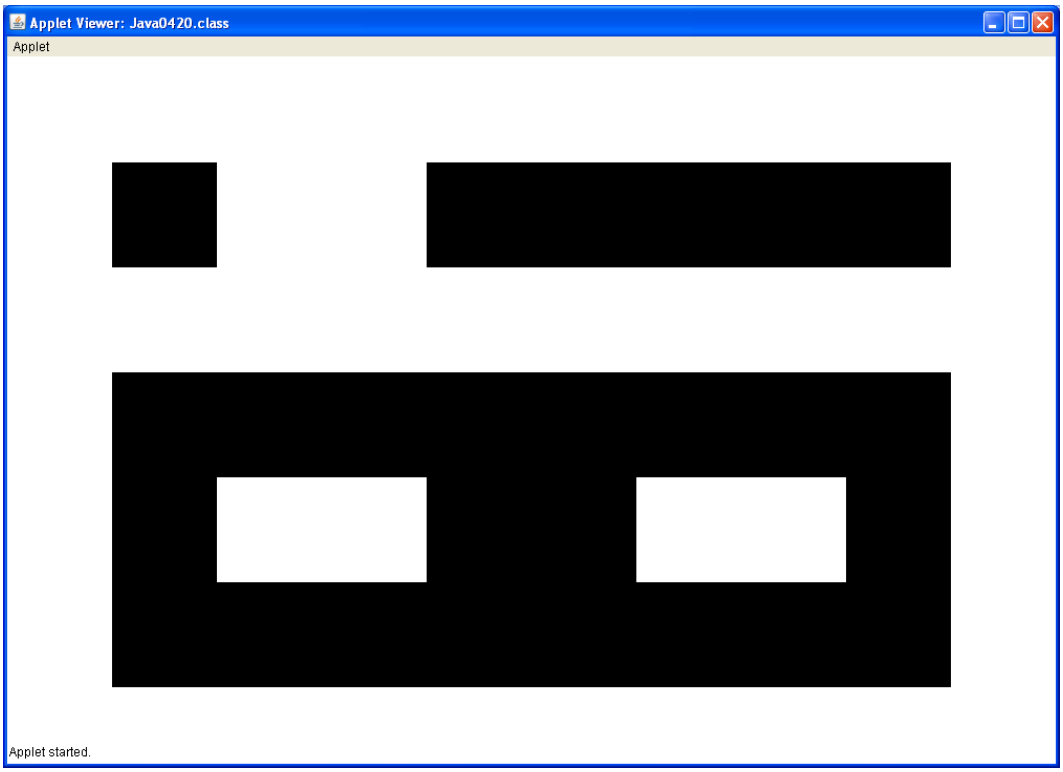
Changing Colors

When you start an applet, the background is white and the drawing/filling color is **black** by default. The **setColor** command allows you to change colors. Using this, we can fix the problem of the previous program.

Figure 4.29

```
// Java0420.java
// This program demonstrates the setColor method of the Expo class.

import java.awt.*;
import java.applet.*;

public class Java0420 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.fillRectangle(g,100,100,200,200);
        Expo.fillRectangle(g,400,100,900,200);
        Expo.fillRectangle(g,100,300,900,600);
        Expo.setColor(g,Expo.white);
        Expo.fillRectangle(g,200,400,400,500);
        Expo.fillRectangle(g,600,400,800,500);
    }
}
```

Now the two small solid rectangles are drawn in white on top of the big black rectangle. That makes them visible. Remember, any time you draw a small object and a big object that you must draw the big object first. Draw the small object second, in a different color, to make it visible.



Something else needs to be pointed out here which is very important. Method **setColor** has two parameters. We have the **Graphics** object ***g*** that all graphics methods need. We also have an object for the color. **Expo.white** is a **Color** object that stores the color **white**. It is similar to **Math.PI** and **Math.E**. These are not methods. They are *attributes* of their respective classes. The **Expo** class has 25 different **Color** attributes built in. All of these colors, except for **white**, are demonstrated in the next program. Program **Java0421.java**, in figure 4.30, also demonstrates **fillCircle** which has the same parameters as **drawCircle**.

Figure 4.30

```
// Java0421.java
// This program demonstrates 24 of the built in colors from the Expo class.
// The only color not demonstrated is Expo.white since that would not show up
// on a white background.
// The program also demonstrates the fillCircle method, which uses the same
// parameters as drawCircle.


import java.awt.*;
import java.applet.*;

public class Java0421 extends Applet
{
```

```
    public void paint(Graphics g)
    {
        int radius = 100;
        Expo.setColor(g,Expo.red);
        Expo.fillCircle(g,100,100,radius);
        Expo.setColor(g,Expo.green);
        Expo.fillCircle(g,250,100,radius);
        Expo.setColor(g,Expo.blue);
        Expo.fillCircle(g,400,100,radius);
        Expo.setColor(g,Expo.orange);
        Expo.fillCircle(g,550,100,radius);
        Expo.setColor(g,Expo.cyan);
        Expo.fillCircle(g,700,100,radius);
        Expo.setColor(g,Expo.magenta);
        Expo.fillCircle(g,850,100,radius);
        Expo.setColor(g,Expo.yellow);
        Expo.fillCircle(g,100,250,radius);
        Expo.setColor(g,Expo.gray);
        Expo.fillCircle(g,250,250,radius);
        Expo.setColor(g,Expo.lightGray);
        Expo.fillCircle(g,400,250,radius);
        Expo.setColor(g,Expo.darkGray);
        Expo.fillCircle(g,550,250,radius);
        Expo.setColor(g,Expo.pink);
        Expo.fillCircle(g,700,250,radius);
        Expo.setColor(g,Expo.black);
        Expo.fillCircle(g,850,250,radius);
        Expo.setColor(g,Expo.brown);
        Expo.fillCircle(g,100,400,radius);
        Expo.setColor(g,Expo.purple);
        Expo.fillCircle(g,250,400,radius);
        Expo.setColor(g,Expo.violet);
        Expo.fillCircle(g,400,400,radius);
        Expo.setColor(g,Expo.lightRed);
        Expo.fillCircle(g,550,400,radius);
        Expo.setColor(g,Expo.darkRed);
        Expo.fillCircle(g,700,400,radius);
        Expo.setColor(g,Expo.lightGreen);
        Expo.fillCircle(g,850,400,radius);
        Expo.setColor(g,Expo.darkGreen);
        Expo.fillCircle(g,100,550,radius);
        Expo.setColor(g,Expo.lightBlue);
        Expo.fillCircle(g,250,550,radius);
        Expo.setColor(g,Expo.darkBlue);
        Expo.fillCircle(g,400,550,radius);
        Expo.setColor(g,Expo.gold);
        Expo.fillCircle(g,550,550,radius);
        Expo.setColor(g,Expo.tan);
        Expo.fillCircle(g,700,550,radius);
        Expo.setColor(g,Expo.turquoise);
        Expo.fillCircle(g,850,550,radius);
    }
}
```

Figure 4.30 Continued

Expo class built-in Color attributes				
setColor(g, Expo.*constant*) Sets the graphics display color of the following graphics output to the specified color constant of the **Expo** class. There are 25 color constants listed below.				
red	**green**	**blue**	**orange**	**cyan**
magenta	**yellow**	**gray**	**lightGray**	**darkGray**
pink	**black**	**white**	**brown**	**purple**
violet	**lightRed**	**darkRed**	**lightGreen**	**darkGreen**
lightBlue	**darkBlue**	**gold**	**tan**	**turquoise**
NOTE: You are not limited to only these 25 colors. By combining different amounts of red, green, and blue values, you can create any of over 16 million different colors. At a later time, you will learn how to create more colors.				

Drawing Pie Wedges and Pacman

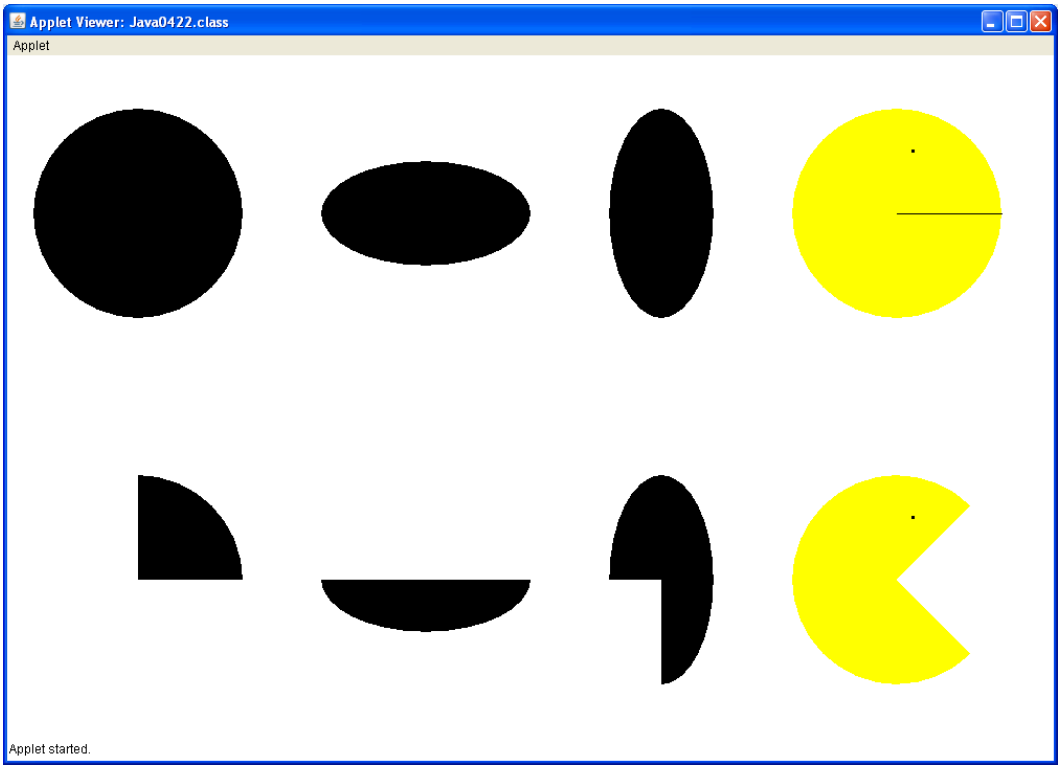
The next program demonstrates **fillOval** and **fillArc**. Now a **fillOval** method might have been expected, but **fillArc** may seem a little strange. Arcs are not enclosed like rectangles, circles, ovals, and stars. When you *fill* an arc, you get a *piece* of a solid circle or oval. Imagine a piece of pie. The pie is the solid circle. The piece looks like a wedge. What is left over looks something like Pacman.

The parameters for **fillOval** are the same as **drawOval**. The parameters for **fillArc** are the same as **drawArc**. Program **Java0422.**java, in figure 4.31, will draw solid ovals followed by a partial oval, or "filled arc", of the same size.

Figure 4.31

```
// Java0422.java
// This program demonstrates fillOval and fillArc.

import java.awt.*;
import java.applet.*;

public class Java0422 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.fillOval(g,125,150,100,100);
        Expo.fillArc( g,125,500,100,100,0,90);
        Expo.fillOval(g,400,150,100,50);
        Expo.fillArc( g,400,500,100,50,90,270);
        Expo.fillOval(g,625,150,50,100);
        Expo.fillArc( g,625,500,50,100,270,180);
        Expo.setColor(g,Expo.yellow);
        Expo.fillOval(g,850,150,100,100);
        Expo.fillArc( g,850,500,100,100,135,45);
```

```
        Expo.setColor(g,Expo.black);
        Expo.drawLine(g,850,150,950,150);
        Expo.drawPoint(g,865,90);
        Expo.drawPoint(g,865,440);
    }
}
```



Drawing "Thick" Shapes

We mentioned earlier that most of the *drawing* methods in the *Expo* class have a corresponding *fill* method. Well, there is more to this story. All of the *drawing* methods also have a corresponding **drawThick** method. Some graphics drawings can be hard to see. You might want a thicker – easier to see – line. You might also want circles, ovals, rectangles, and even stars to show up in a more vivid fashion. This is the purpose of the **drawThick** methods. The **drawThick** methods are almost identical to their corresponding *draw* method. There are two differences. First, you need to insert the word "Thick" after the word "draw". Second, all **drawThick** methods have an extra parameter at the end. This parameter is an integer that specifies the thickness. The bigger the number, the thicker the shape. Program **Java0423.java**, in figure 4.32, uses variable **t**, with value **8**, to specify the thickness of the graphics lines. By changing this value, you can make the shapes thicker or thinner.

Figure 4.32

```
// Java0423.java
// This program demonstrates various "drawThick" methods.
// "drawThick" methods have the same parameters as their corresponding
// "draw" methods with one exception:
// "drawThick" methods have an extra parameter at the end for the thickness.

import java.awt.*;
import java.applet.*;

public class Java0423 extends Applet
{
    public void paint(Graphics g)
    {
        int t = 8; // thickness
        Expo.setColor(g,Expo.darkRed);
        Expo.drawThickLine(g,100,80,900,80,t);
        Expo.drawThickRectangle(g,50,30,950,630,t);
        Expo.drawThickCircle(g,500,360,240,t);
        Expo.drawThickOval(g,400,340,20,30,t);
        Expo.drawThickStar(g,400,340,110,5,3*t);
        Expo.drawThickOval(g,600,340,30,40,t);
        Expo.drawThickStar(g,600,340,90,6,3*t);
        Expo.drawThickArc(g,500,460,100,50,90,270,t);
        Expo.drawThickSpiral(g,200,250,100,t);
        Expo.drawThickSpiral(g,805,250,100,t);
        Expo.drawThickLine(g,175,400,175,600,t);
        Expo.drawThickLine(g,75,500,275,500,t);
        Expo.drawThickLine(g,750,450,900,600,t);
        Expo.drawThickLine(g,900,450,750,600,t);
    }
}
```

4.9 Using Expo Class Web Page Documentation

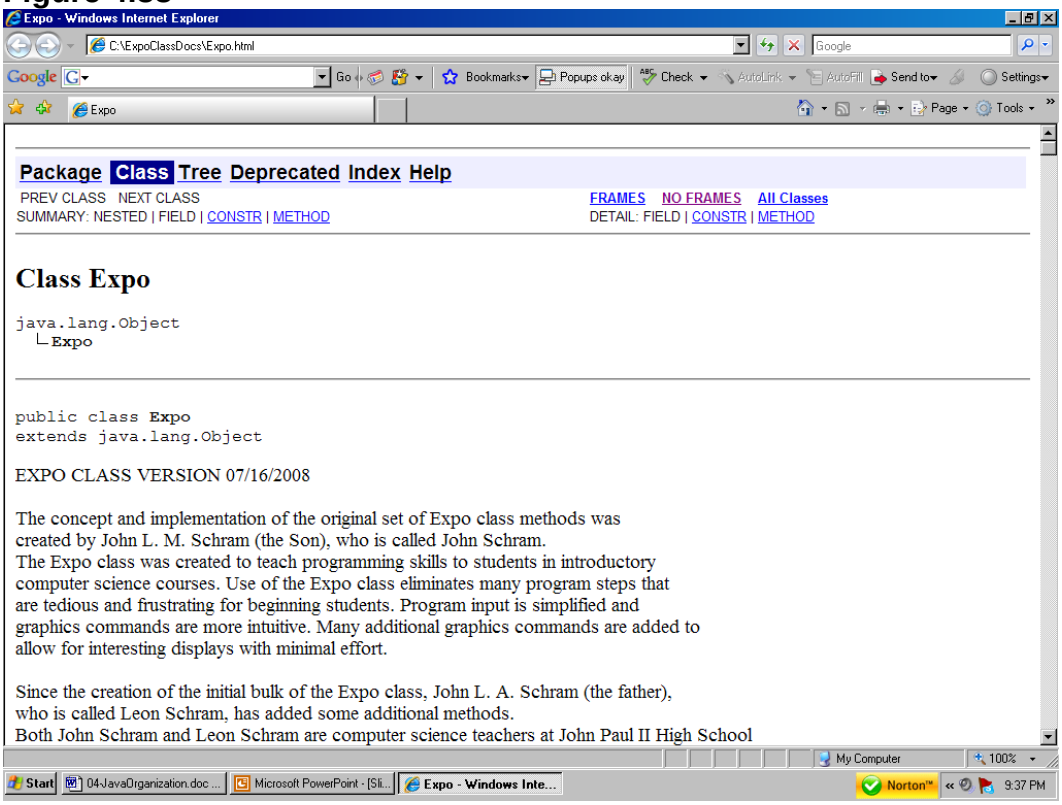
You have been shown several methods of the Expo class… and more will come in later chapters. There are a few important things you need to remember about the **Expo** class:

Important Facts to Remember about the Expo class

- **The *Expo* class is not part of any Java standard library.**
- **The class was created to simplify programming and allow students to focus on the logic of programming.**
- **In order to use the *Expo* class, the file `Expo.java` must be in the same folder/directory as the .java file that calls the *Expo* class methods.**
- **Students will NOT be required to memorize the methods of the *Expo* class. They will instead be provided with documentation to use during labs and tests.**

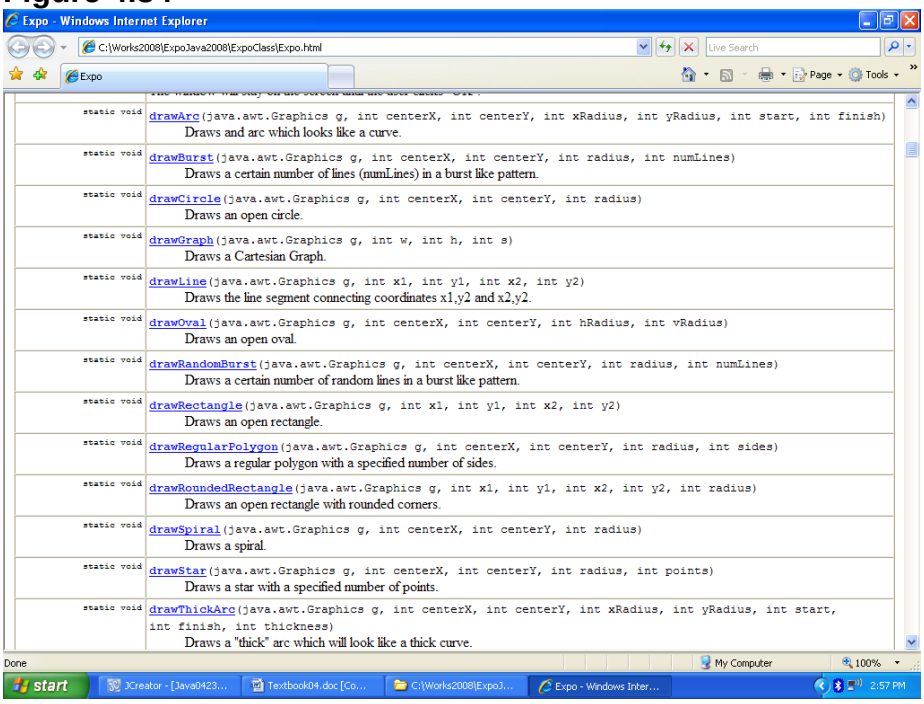
This next section deals with the last bullet on the list above. You do not need to memorize the **Expo** class. You will be provided with a fiolder called **ExpoClassDocs** or similar name. This folder contains many .**html** files. When the **index.html** file is loaded in a browser like *Internet Explorer* you will see the figure 4.33 display.

Figure 4.33



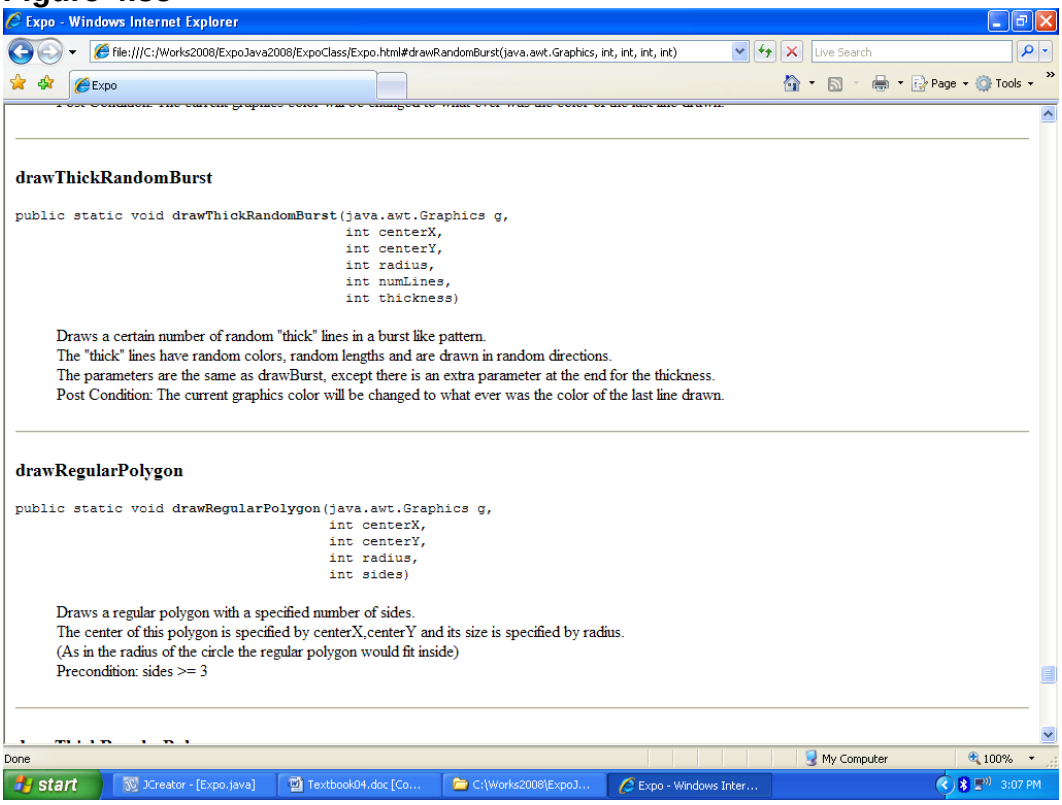
Initially you see a description of the class and a list of the 25 colors. The file is quite large so you need to scroll down a bit to observe the figure 4.34 display.

Figure 4.34



What you saw in figure 4.34 is a partial list of the methods of the *Expo* class. Some of the **Expo** class methods were covered in this chapter. You will also see several methods that we did not cover, which will be explained in future chapters. The point is this. You can scroll down through this alphabetized list and see some quick information on each method. First you see how the method name is spelled. Then you see the required list of parameters. Finally, you see a brief – one sentence – description of what the method does. Suppose that is not enough. You need more. A "one sentence" description does not cut it. In that case, click on the method name and it will link to a more details description. The example, in figure 4.35, assumes that you clicked the method, **drawThickRandomBurst**.

Figure 4.35



This links you to a part of the webpage that has more detailed information. Maybe it will have a few extra sentences of description. Maybe it will have an example. Sometimes, you will simply see the same one sentence description again if that was all the information that was provided. If you look closely at the two methods shown in figure 4.35, you will see something else. What are *preconditions* and *postconditions*? That topic is addressed in the next section.

Preconditions and Postconditions

Sometime when you look at the documentation of a method, it will include a *precondition*, *postcondition*, or both. It is important to understand what these mean. A *precondition* is like the disclaimer you see at the bottom of your televisions during a commercial. The commercial shows a toy soldier moving and talking. The disclaimer says, "*Toy does not actually move or talk*".

Look at the example in figure 4.29 again. The *drawRegularPolygon* method has a *precondition*, which that states: **Precondition: sides >= 3**

This is a method to draw polygons. A polygon cannot have less than 3 sides. The methods simply will not work if you have less than 3 sides. In the same example, look at the **drawThickRandomBurst** method. This method draws several random colored lines to simulate something like a firecracker. Because of this, when the method is finished, the current color will not be the same as it was before the method was called. This possibility is covered in its *postcondition*:

Postcondition: The current graphics color will be changed to whatever is the color of the last line drawn.

Something important should be realized about *preconditions* and *postconditions*. The *postcondition* will only happen *if the precondition is true*! You should have learned in your math class that you cannot take the square root of a negative number. **Math.sqrt** will ONLY return the proper square root if the parameter value is positive or 0. If the parameter value is a negative number, meaning the precondition is false, then there is no guarantee that the postcondition will happen. In the case of **Math.sqrt**, using a negative number will not return a number. Instead, it returns the result of **NaN** which means *Not a Number*.

Preconditions and Postcondition

A precondition describes the state of a method that must be true before the method is called.

A postcondition describes what must be true after the method is called ... provided ... the precondition(s) were satisfied.

Math.sqrt(x)
Precondition: **x must be a positive real number or 0.**
Postcondition: **The principal square root of x is returned.**

4.10 Implementing Mathematical Functions

There is a close connection between computer science and mathematics. You will find that many mathematical concepts will be practiced and reinforced this year. For instance, you may be learning about functions in your Algebra class right now or you learned algebraic functions in a previous math class. For example:

Given: **f(x) = 3x + 7**

Question: *What is f(10)?*

The 10 is the information or argument for the function **f(x)**. **3x + 7** is what we are supposed to compute. If **x = 10** then we have 3(10) + 7 = 30 + 7 = 37. This means **f(10) = 37**.

This exact same computation can be done in Java. We can create a method that *implements* the mathematical function **f(x)**. Look at this example.

```
public static double f(double x)
{
    return 3 * x + 7;
}
```

For now, ignore the entire method heading. We are only concerned that you understand the **return** statement in the method. This method will compute the function **f(x) = 3x + 7**. When you look at the **return** statement, do you see that that is exactly what we are doing? Program **Java0424**, in figure 4.36, uses this *user-created* method and computes **f(10)**, as well as **f(5)** and **f(1)**.

Figure 4.30

```
// Java0424.java
// This program implements the Mathematical Function:
// f(x) = 3x + 7
// NOTE: This is an application, not an applet.  There is no .html file.

public class Java0424
{
   public static void main (String args[])
   {
       System.out.println("\nJAVA0424.JAVA\n");
       System.out.println("f(10) = " + f(10) );
       System.out.println("f(5)   = " + f(5)  );
       System.out.println("f(1)   = " + f(1)  );
       System.out.println();
   }

   public static double f(double x)
   {
       return 3 * x + 7;
   }
}
```

We hope you realize that this short little method can be altered to many other mathematical functions. Nothing prevents you from creating of the following:

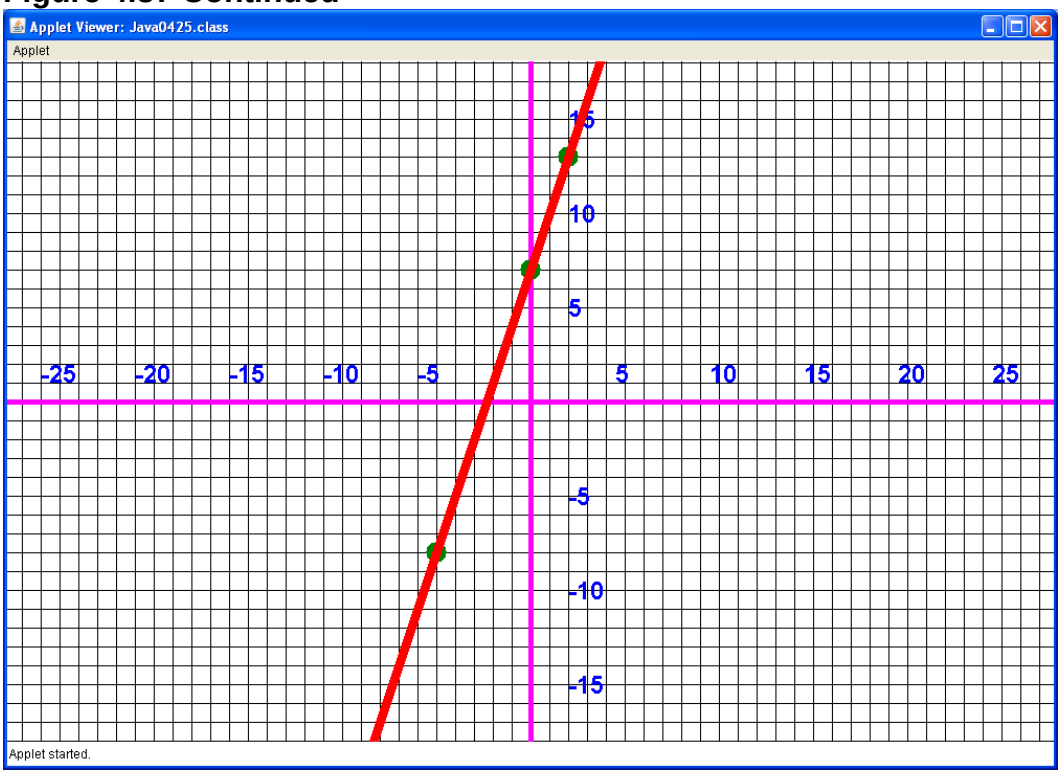
```
return 5 * x + 23;        //  f(x) = 5x + 23
return 7 * x – 14;        //  f(x) = 7x – 14
return 2 * x;             //  f(x) = 2x
return 4;                 //  f(x) = 4
```

The last program, **Java0425.java**, in figure 4.37, of this chapter will actually use the same method for **f(x)**. The difference will be in that this program is an applet, which will graph the line of **y = 3x + 7**, or **f(x) = 3x + 7**.

Figure 4.37

```
// Java0425.java
// This program implements the Mathematical Function:
// f(x) = 3x + 7
// and graphically displays it.


import java.awt.*;
import java.applet.*;

public class Java0425 extends Applet
{
    public void paint(Graphics g)
    {
        Expo.drawGraph(g,1000,650,18);
        Expo.graphPoint(g,2,f(2));
        Expo.graphPoint(g,0,f(0));
        Expo.graphPoint(g,-5,f(-5));
        Expo.graphLine(g,2,f(2),-5,f(-5));
    }

    public static double f(double x)
    {
        // This can be changed to graph any line.
        return 3 * x + 7;
    }
}
```

Figure 4.37 Continued

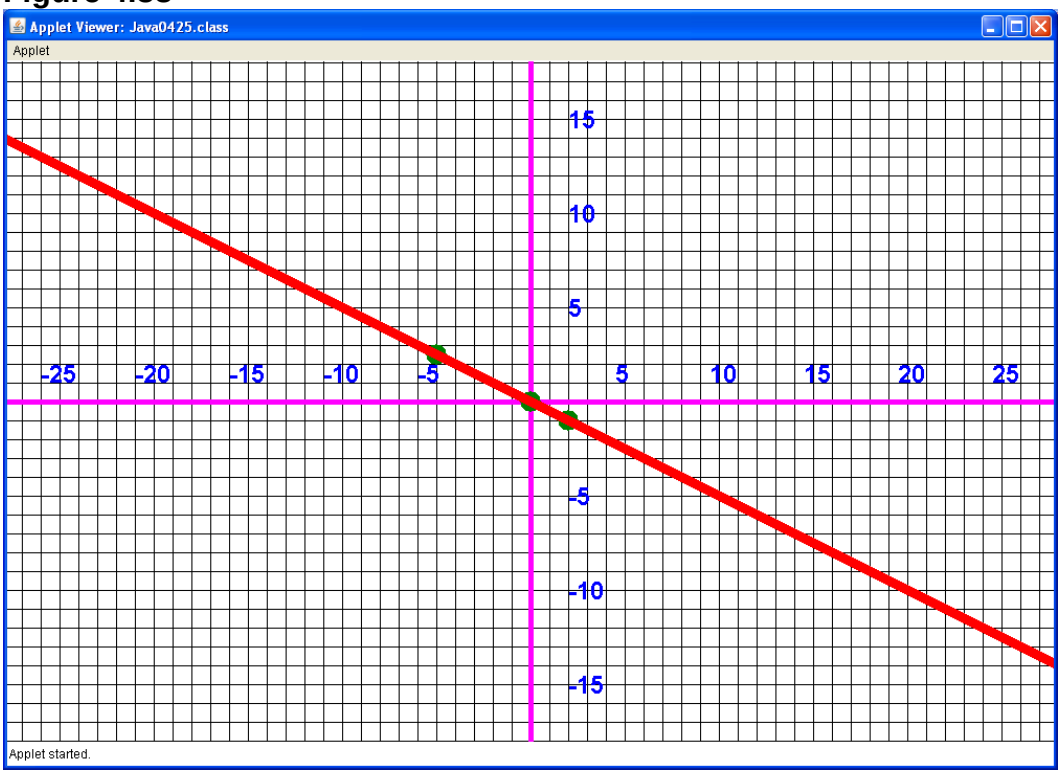


Now suppose that you want to graph the line **y = -½x** or **f(x) = -½x**. Change the **return** statement to say:

```
return –x / 2;
```

and then the output would look like the figure 4.38 display.

Figure 4.38



4.11 Summary

Chapter IV introduced the fundamental program organization of Java. The organization of a program is similar to the organization of an English essay, story or book. Java keywords combine to form program statements. A group of program statements, which accomplish some specific purpose, are placed together in a container called a method. A set of methods with similar functions are placed inside a larger container, called a *class*.
A *class* also contains data, which is used by the methods. A class is a data type. One particular variable of a **class** is called an **object**.

The first methods introduced were methods of the **Math** class. Access to methods is done by using the class identifier, followed by a period and then the method identifier, like **Math.sqrt(16)**. This approach is done with **Math** methods.

During this course you will be using a very large class, called **Expo**. This class is not part of the Java standard classes.

Things to Remember about the Expo class

- **The *Expo* class is not part of Java.**
- **The class was created to simplify programming and allow students to focus on the logic of programming.**
- **In order to use the *Expo* class, the file `Expo.java` must be in the same folder/directory as the .java file that calls the *Expo* class methods.**
- **Students will NOT be required to memorize the methods of the *Expo* class. They will instead be provided with documentation to use during labs and tests.**