**Chapter IV**

**Using Methods and Parameters**

**Chapter IV Topics**

4.1 Introduction

4.2 A Brief History of Program Design

4.3 OOP, a Gentle First Exposure

4.4 Using the Math Class

4.5 Using Java Graphics Methods

4.6 Observing the GridWorld Start

4.7 Examining the Grid Environment

4.8 GridWorld Objects Behavior

4.9 Summary

**4.1 Introduction**

In Chapter I you were introduced to a history of program languages. In that brief history of various languages you were told that C++ is a language with extensive *Object Oriented Programming* (OOP) features, but C++ can be written without using any type of OOP. Java on the other hand revolves around OOP. This means that you need to learn OOP to write correct Java programs and OOP needs to be introduced early in the computer course.

OOP can easily overwhelm the new computer science student with a sizable vocabulary of obscure words along with even more obscure definitions for those words. Words like *encapsulation*, *inheritance*, *composition* and *polymorphism* are not intuitive household words for the average person. It is the aim of Exposure Java to give you a very thorough treatment of all the various OOP concepts. However, these concepts will be introduced one small chunk at a time. OOP may be a formidable challenge to comprehend, but all challenges can be broken down into small manageable parts. This short chapter is but one small stage along the road to master computer science with the Java programming language. You will also find that future chapters will repeat prior concepts frequently to reinforce what was learned earlier.

Object Oriented Programming is now an established computer science style of programming that is making the demands of today's computer programs possible. You will also find OOP in future college courses as well as in the computer application programs that surround us. Thinking about objects and using all types of objects has infiltrated practically every area of technology.

**4.2 A Brief History of Program Design**

You are learning computer science with a strong emphasis on *Object Oriented Programming (OOP)*. It is not possible to write any substantial Java program without understanding OOP. You are probably very excited about learning OOP. It is just too bad that you do not have a clue what OOP is, where it came from and what its purpose is in your space-time-continuum.

Answering the *What is OOP?*question will take this whole course and the next one. Right now, you can learn where OOP comes from with very little complexity. Consider these four program design stages.

|  |
| --- |
| **The Four Stages of Program Design** |
| **• Cryptic Programming Stage**  **• Unstructured, Spaghetti-Programming Stage**  **• Structured Programming Stage**  **• Object Oriented Programming Stage** |

The four stages explained here are somewhat of an over-simplification of the evolution of programming style, but this will do nicely to give you a brief over- view of where computer science has been and where we are today.

**Cryptic Programming Stage**

In the very early days of programming, computers were incredibly expensive and programmers were very cheap. Computers cost more than one million dollars and a programmer could be hired for $5,000 a year. Programs were written without any thought about design. As a matter of fact, many programs were written intentionally in a very cryptic style that only the creator of the program could understand. This style of programming assisted a programmer with job security and it also stroked egos as only the programmer, the exulted mastermind, could fix or alter the program at the awe of other lesser-blessed mortals.

This style of programming did not last long. Computers became cheaper and programmers became more expensive. Programs required frequent debugging and updating, and the cryptic style usually meant starting a program from scratch if the creator left the company. Software development companies could not afford this wasteful approach.

**Unstructured, Spaghetti-Programming Stage**

The next stage made attempts at being less cryptic. An effort was started to use better, meaningful identifiers, comment program code to assist in updating and provide external documentation for future programmers. But a major problem lurked at this stage in the form of the **goto** statement. Programs then, and now, rarely worked immediately. Program errors were found and needed to be fixed and sometimes bugs showed up continuously as the program aged. A program requires some type of control sequence, which guides the flow of program execution and in large programs this program flow would frequently be faulty.

A quick fix was to use a **goto** statement to put the program back on track. Unfortunately, as programs grew so did the number of **goto** statements and large programs were mazes with many program statements linked to many other parts in the programs. People unfamiliar with the design of such programs would often use a pencil and draw the program flow on a printout of the source code. Such drawings resulted in a spaghetti appearance of crisscrossing lines and the term *spaghetti programming* was born.

**Structured Programming Stage**

Programs in the past and today have one common goal that allows no compromise, which is **reliability**. A program must work correctly. It is nice if a program executes rapidly, but no amount of speed can excuse a program that is unreliable. The Denver International Airport that opened in the late Nineties was delayed considerably by its computerized luggage handling system. Originally, the system proved, as it was hoped, to be a marvel of speed. Luggage was handled in a manner never thought possible. It was true computers added speed not possible with the simple human process approach. Initially, there was one complaint though. The system was fast, but the luggage ended up on the wrong plane. Never in the history of aviation was luggage lost more rapidly. It took a computer program to achieve that goal.

Spaghetti programming is a major drawback to reliability. The many **goto** statements are basically a quick Band-Aid for a flawed program and once there are so many Band-Aids, the whole structure becomes weak and very unreliable. It is no longer possible to have any type of overview of the program execution.

A new style of programming was developed called *structured programming*. It is not easy to explain exactly what that kind of programming involves, but you can realize this. **Goto** statements received the boot and were no longer used. Program flow was strictly controlled with one entrance and one exit and program statements with a common purpose were grouped together in special modules. The philosophy was *one-task-one-module*. Structured programming greatly improved the creation, debugging and updating of programs. Reliability was increased along the way.

**Object Oriented Programming Stage**

Structured programming did just fine in the earlier world of text-style output. Programs were simpler and shorter than the later windows-style programs. The incredible complexity of current programs brought on a new challenge in reliability. Program users expected programs to look attractive, provide on-line help, be user-friendly, and yes they still needed to be reliable. The length of programs grew incredibly. Many programs in the late Eighties and early nineties could fit on three to five floppy diskettes. That changed dramatically as today's software is lucky to fit on one CD. Somehow with this new magnitude of programming, structured programming could not keep up. A level of program complexity had arrived that required a new approach.

This chapter will give a brief introduction to Object Oriented Programming. Please accept right now that OOP has many features specifically designed to make a program easier to develop, simpler to debug and faster to test with confidence. Program development with OOP has brought a new level of program reliability.

This also explains why universities have embraced Java as an introductory programming language. C++ used to be a popular program language in computer science classes. C++ has all the powerful OOP features and C++ was and still is an extremely important language in the computer science community and software industry. The key problem with using C++ at the introductory level is that C++ can be used without OOP at all. In a world where students need to be thinking about OOP from the beginning, C++ can be problematic. Java, on the other hand, is pure OOP. You are not asked if you like OOP or want to do OOP. Oh no, if you program in Java you will be up to your nostrils in OOP.

**4.3 OOP, a Gentle First Exposure**

Textbooks can be so helpful to the young, eager, but quite confused computer science student. You, the eager student, open a textbook excited to learn about *Object Oriented Programming*. Your textbook just oozes the wisdom of the ages, or at least the last few computer science decades. You just know this wisdom will be uploaded to your brain by osmosis or some other, equally mysterious process, which is known as studying in some circles. Actually, you feel pretty good because you do understand most of the introductory material that you have been reading. You have heard about OOP from some people, but you do not have a clue what the hype is all about. Maybe your textbook will clarify. In a pretty box you now see the following definition:

|  |
| --- |
| **Object Oriented Programming (OOP) is a style of programming that incorporates the three features of *encapsulation*, *class interaction* and *polymorphism*.** |

Well what could be clearer? Encapsulation has always been one of your hobbies. class interaction sounds like social networking with your fellow students and polymorphism is a regular topic around the dinner table. Why this kind of vocabulary? Simple. Professions generate vocabulary, which is known only to members of the profession to help identify its own members. The outsider people then need to pay big bucks to acquire the services of the inside professionals. Doctors, lawyers, accountants, educators and yes computer scientists all have a set of vocabulary to add prestige and economic benefits to the profession. Think I am kidding? Consider the following invoice by a network technician who fixes a computer network problem.

*Repaired layer-1 radical disruption to the layer-2 NIC, which resulted in a failure of any frame to identify the appropriate MAC address of the destination host, while simultaneously disallowing the NIC to perform its usual role of CSMA/CD of the Ethernet protocol.*

Translation: *the network cable was not plugged in*. Now how can you charge $500.00 by stating on an invoice that you plugged in a cable? OK, back to OOP, and the whole point of this tangent is to explain that unpleasant vocabulary needs to be accepted. It comes with any profession. Do not think that a concept is complex because the vocabulary sounds intimidating. Professional vocabulary is meant to be intimidating. Once you work with the language it is no longer a big deal. So, now that you have accepted that *encapsulation*, *class interaction* and *polymorphism* are a way of computer science life, let us see what this really means.

Forget computer science right now and think **objects**. What comes to mind? It probably depends on your age, culture, gender and upbringing but I bet that every object you think about has **nouns** and **verbs**. What do I mean? Well, let us look at cars. Cars have many objects. They have seats, doors, lights, seat belts, wheels, radios, engines, transmissions and many other objects that make up the entire car. Now each of these objects has nouns and verbs. Sounds weird? Consider a radio. A radio has buttons and/or knobs, which can be pushed or turned. A radio also has a display screen. A knob, a button, a display is a noun, and each one of these nouns does something. A knob turns to the right frequency. A button turns the power on and off. The display shows the current frequency or the CD that is played. The actions performed by the nouns are the verbs. Everywhere you look, you are surrounded by many objects. Object Oriented Programming simulates real life by using a program style that treats a program as a group of objects. Now in OOP we do not use the term nouns and verbs. You may hear many different terms being used but currently the more popular terms are **attributes** and **methods**. Attributes store program information, which is like a noun. Methods perform some actions on the information, which is like a verb. For example, imagine that you have a **student** object. In this object the attributes are all the student records of information. The object also contains the ability to add, delete, alter, sort and display these student records. Adding, deleting, altering, sorting and displaying the records are methods.

You have already encountered some methods in Java. Both **print** and **println** are methods. Methods are program modules that perform some task. In the case of **print** the task is to display information in a text window.

The significance of Object Oriented Programming is that attributes and methods are packaged in the same container, the same object or the same capsule. This accounts for the name *encapsulation*. The earlier, non OOP, languages would keep the attributes, the information stored by a program, separate from the functions, procedures, actions or methods that process the information.

Do not even try to think about how all this is achieved. The details come later. This is meant to be an introduction of general concepts. Where is the reliability in all this encapsulation business? Imagine that you want to have a reliable lumber- yard delivery system. Your job is to drop off large quantities of lumber at a customer's site. Now this lumber is very heavy. At the lumberyards you have forklifts to load the lumber. How about unloading the lumber at the customer location? Perhaps the customer is at a construction site with a forklift. Perhaps the forklift is in use, is too large or is too small or maybe the customer does not have a forklift at the job site at all.

If we keep the lumber (attribute) separate from the forklift (method) that handles the lumber we may have serious problems delivering the lumber efficiently and reliably.

The solution is to attach a forklift to the back-end of the delivery truck. We now have a delivery object. The object stores the lumber, moves the lumber and unloads the lumber. Everything required to manage the lumber is contained in the same object. The containment or encapsulation of materials or information and the processes that access the materials is the corner stone of OOP. It increases reliability.

In this chapter no real attempt will be made to explain polymorphism and class interaction. Polymorphism means *many forms* and that is a topic that is right now not really possible to introduce. Have patience, you need some more tools in your programming kit before we can return to that topic.

Class interaction means that existing classes work together by interacting with each other and with newly created classes. This interaction is done with *inheritance* and *composition*. Inheritance in computer science is very similar to the meaning of the English word you know. Something already exists that you acquire for some reason. Inheritance increases reliability tremendously in programming. Imagine that you have spent many hours or possibly weeks creating a program that accomplishes a certain task like a cube that moves in 3-D space and can be controlled with a mouse. It does not stop with a cube. You want to move on and control an airplane that moves in 3-D space. It is not necessary to start from scratch because much of what is necessary for the plane was already done with the cube. In this case you start with the capabilities of the cube, you inherit them, and then move on to add the requirements of the plane. The inheritance concept is used a lot in mathematical definitions, especially in Geometry. Consider a statement like: *a square is a rectangle with four equal sides.* That definition is very short, but only, because we *inherited* all the properties of the rectangle to define the square.

Composition also uses existing classes, but not in an *is-a* relationship, like you see with inheritance. You can say that a car *has-an* engine, which is not the same as stating that a car *is-an* engine. You can create a Student class first and test that class thoroughly. This is followed by a **Course** class, which has many **Student** class objects.

All these concepts will be repeated many times and you will be surprised how comfortable the vocabulary and the concepts behind the vocabulary will become to you before this course is finished. You will not get a complete OOP treatment in this chapter. There will be many separate chapters devoted to different OOP concepts. OOP is not trivial computer science, but it is very manageable. We will start early in the course with a comfortable introduction where you learn to *use classes and objects*. In later chapters you will learn how to design and create your own classes, how to use inheritance and play around with polymorphism. At every stage you will appreciate more and more why Object Oriented Programming is such a powerful computer science concept.

**4.4 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. Computers exist to make life simpler and as programming 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.

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 OOP. Modules that perform a related set of functions are grouped together in a special program type or container, called a **class**.

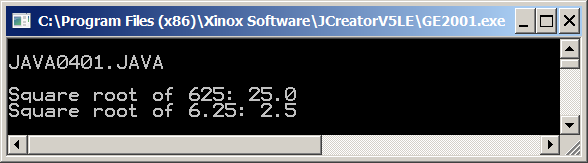
You will learn more about the organization of all these classes at a later time. Right now take a look at the Java **Math** class. The **Math** class contains many useful program modules, which compute a wide variety of mathematical functions.

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.

This action, procedure, function, task, behavior, subroutine, module or whatever you may wish to call it shall in Java be called a **method**. Classes contain **data** or **attributes** and classes contain **methods**. Java has two types of methods, which are *class* methods and *object* methods. Program **Java0401.java**, in figure 4.1, starts with the **sqrt** class method, which is an abbreviation for **square root**.

**Figure 4.1**

|  |
| --- |
| // 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 in the use of a class method. Object methods will be explained in a later chapter.

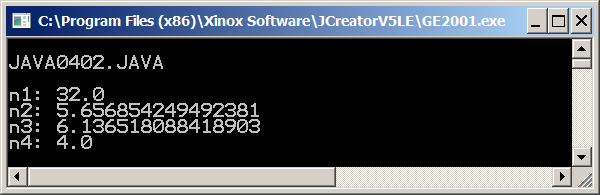
|  |
| --- |
| **Class Method Syntax** |
| **Math.sqrt(n1)**  1. **Math** is the class identifier, which contains the methods you call.  2. **•** separates the class identifier from the method identifier  3. **sqrt** is the method identifier  4. **(n1)** n1 is the argument or parameter passed to the 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 4.2, 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 4.2**

|  |
| --- |
| // 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** stores any  non-negative number (**int** or **double**), but not if **x** stores  a **String** value like **"aardvark"**. |

NOTE: If you attempt to take the square root of a negative number in a Java program, the computer will give you a result of **NaN** which means *Not a Number*.

**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.

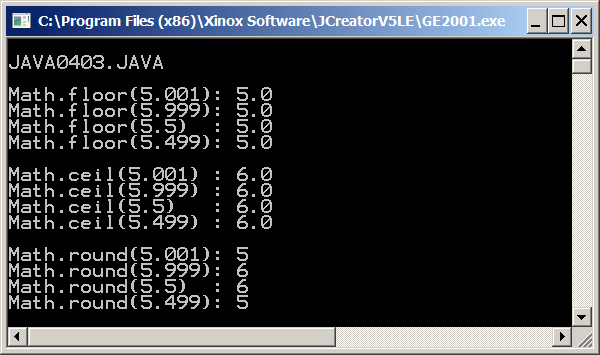
The name of the method gives a good clue as to its purpose. The **floor** method returns the next lowest whole number. Think **floor is down**. The **ceil** method returns the next higher whole number. Think **ceil**ing **is up**. The **round** method is the traditional rounding approach, which *rounds-up*,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

Program **Java0403.java**, in figure 4.3 on the next page, demonstrates each one of the three different "rounding" methods. Carefully compare 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 result.

Remember if you alter a program - no matter how slight - 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 4.3**

|  |
| --- |
| // 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();  }  } |

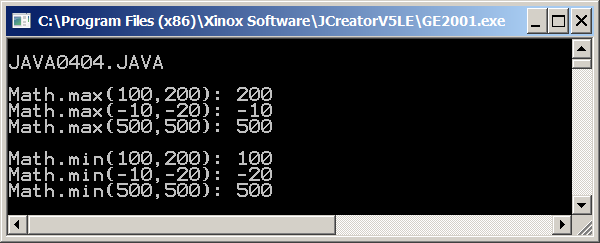


**max and min Methods**

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 4.4.

**Figure 4.4**

|  |
| --- |
| // 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();  }  } |

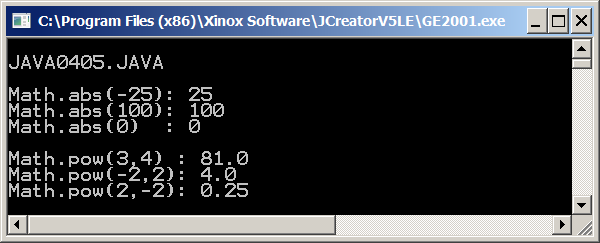


**abs and pow Methods**

The **abs** and **pow** methods are not related, but I 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 4.5.

**Figure 4.5**

|  |
| --- |
| // 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();  }  } |

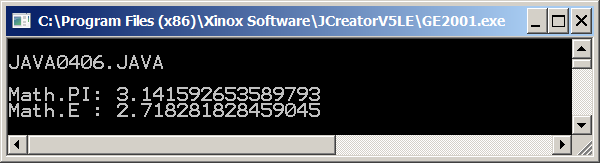


**PI and E Math Fields**

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 4.6, 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 fields of the class directly.

**Figure 4.6**

|  |
| --- |
| // 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 my intention in this chapter to give a full treatment of the **Math** class. My purpose here is to show you how to use the methods of a provided class.

|  |
| --- |
| **AP Computer Science Examination Alert** |
| The **Math** class has many methods.  Only the **abs**, **pow**, **sqrt** and **random** methods will be tested on the AP Computer Science Examination.  You will learn about **Math.random()** in chapter 6. |

**4.5 Using Java Graphics Methods**

Chapter II introduced the first set of programs. All these earlier programs were application programs that were 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. |

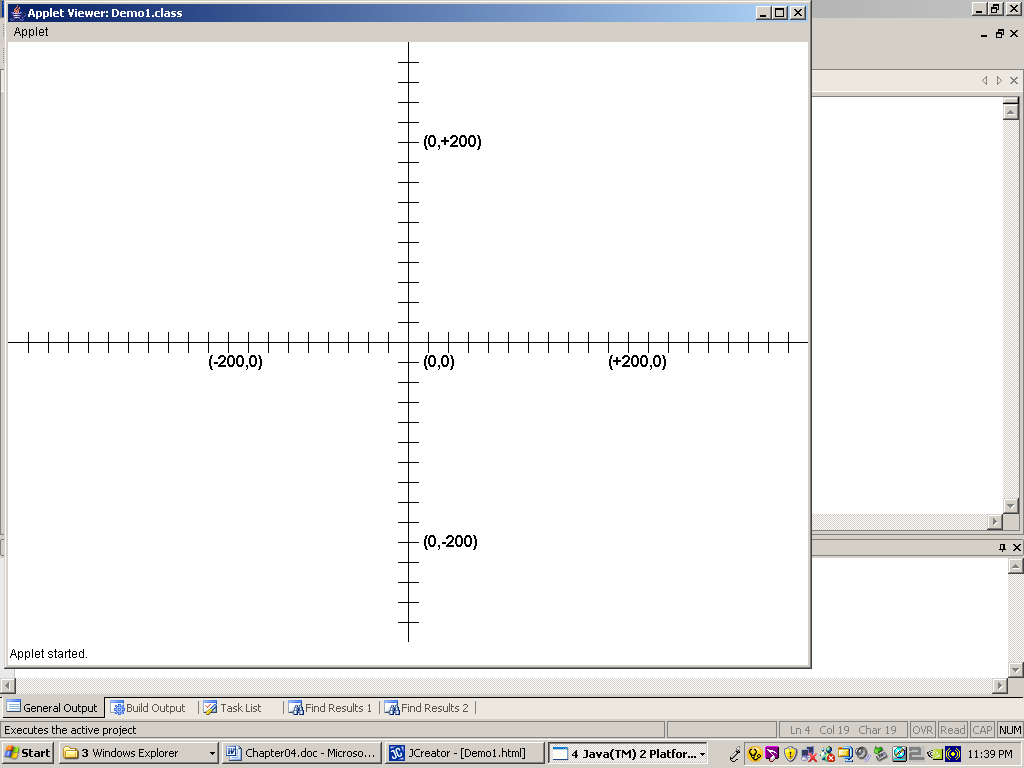
**Graphics and Coordinate Geometry**

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.

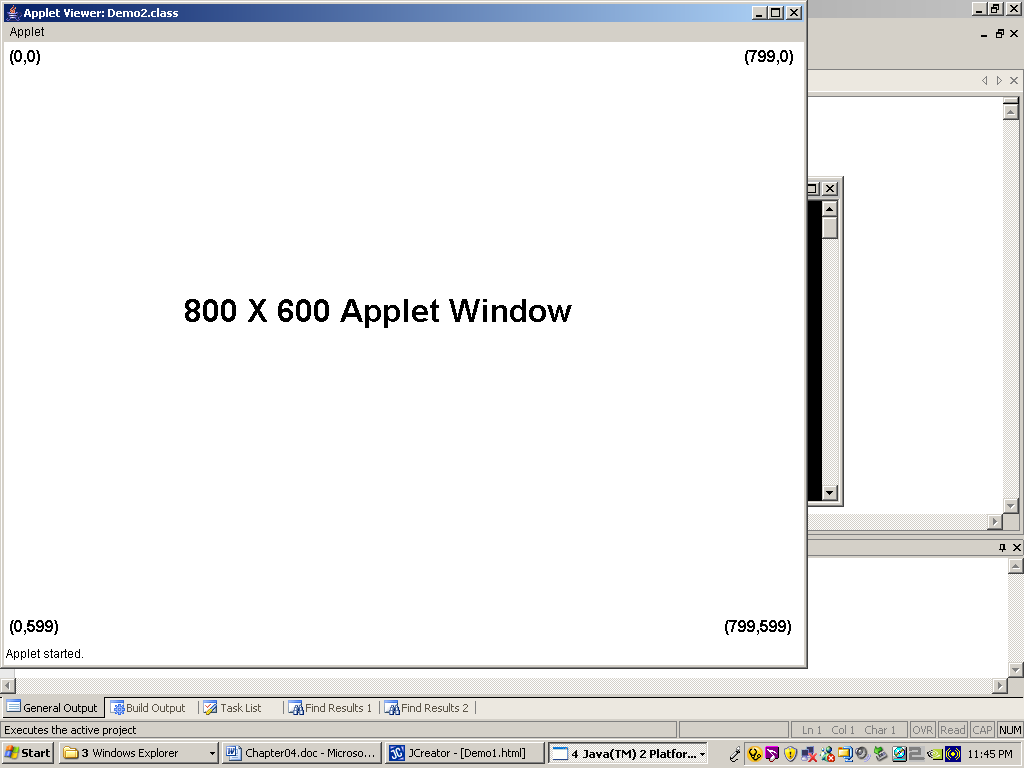
A graphics window uses a system of (**x,y**) coordinates in a manner similar to the use of coordinates that you first learned in your math classes. Figure 4.7 shows an example of the Cartesian coordinate system. In particular, note that the Cartesian system has four quadrants with the (0,0) coordinate located in the center of the grid where the X-Axis and the Y-Axis intersect.

**Figure 4.7**



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 mathematical system, but has some important fundamental differences. Figure 4.8 shows an applet window dimensioned to **800 x 600** pixels. The window is 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 4.8**



|  |
| --- |
| **Executing Java Applet Programs** |
| All the graphics programs examples that follow in this section are created as ***Java applets***. Make sure that you remember to compile the Java source code file, ***and then switch to some small web page file for execution.*** It is possible to create the same exact displays with Java applications. |

**Drawing Lines with method drawLine**

Look at **Java0407.java**, in figure 4.9, which will follow a format that will be seen in all the remaining graphics program examples in this section. Method **main** is not visible. The **main** method controls program execution in an application program. In an applet the program execution sequence is controlled by the web browser. The **main** method is not required. You do require a **paint** method with a **Graphics** variable, **g** in this case, which is used to access all the necessary methods in the **Graphics** class. It will appear that method **paint** takes over the responsibilities of the **main** method. Four lines will be drawn with four **drawLine** statements.

This approach is different from method calls in the previous program examples. When you accessed methods of the **Math** class like **sqrt**, **abs**, and **pow** you started with the class name, **Math** followed by a period and then a method name, such as **Math.sqrt(16)**. The methods of the **Graphics** class will not follow this same pattern. It is necessary to create a variable object, like **g** of the **Graphics** class and then use the variable identifier with the method calls. You will learn in a later chapter when to use class identifiers and when to use object variable identifiers.

Program **Java0407.java**, in figure 4.9, and the remainder of program examples in this section 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.

**Figure 4.9**

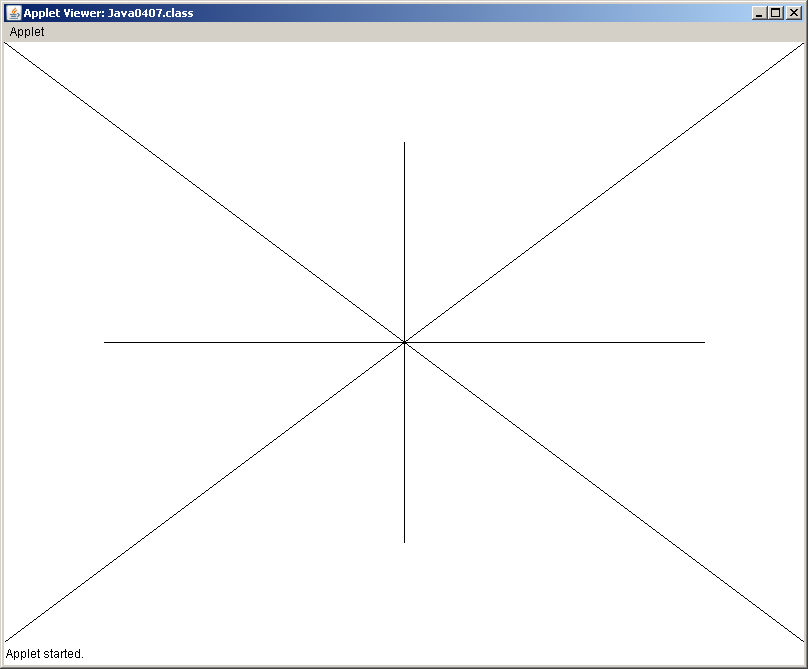
|  |
| --- |
| // Java0407.java  // This program demonstrates how to draw lines.  // Lines are drawn from (X1,Y1) to (X2,Y2) with drawLine(X1,Y1,X2,Y2).  import java.awt.\*;  import java.applet.\*;  public class Java0407 extends Applet  {  public void paint(Graphics g)  {  g.drawLine(0,0,800,600);  g.drawLine(0,600,800,0);  g.drawLine(100,300,700,300);  g.drawLine(400,100,400,500);  }  } |

**Figure 4.10**

|  |
| --- |
| <APPLET CODE = "Java0407.class" WIDTH=800 HEIGHT=600>  </APPLET> |

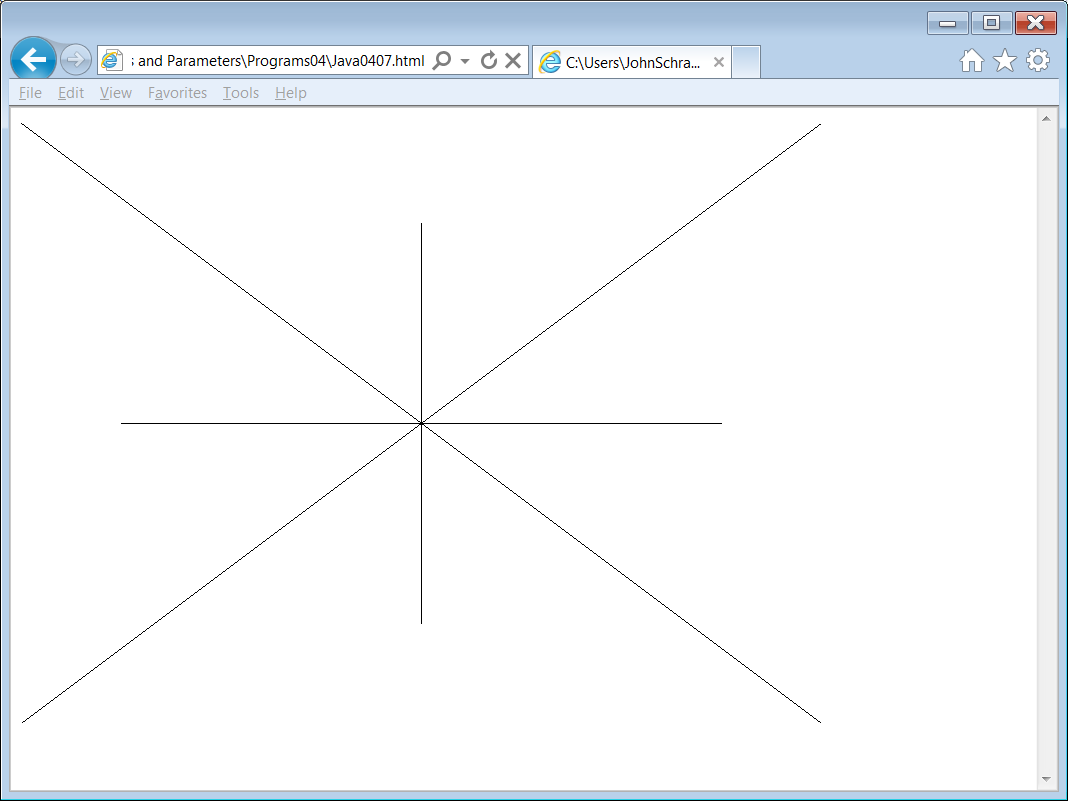
First you need to compile the **Java0407.java** program, which will create a bytecode file called **Java0407.class**. Then you need to load the **Java0407.html** file, shown in figure 4.10. JCreator will automatically display all the program source code files that end with **.java**. You will need to change the *types of files* window in JCreator to looking for **.html** files. Then you will see the html files and you can load **Java0407.html**. Do not try to compile this small file. It is not a Java program. It is a very small web page, which includes the Java applet called **Java0407.class**. Execute this program in JCreator and JCreator will automatically call the **Appletviewer.exe** program to execute the applet. You will get the display shown in figure 4.11 when you execute the small web page.

**Figure 4.11**



The program can also be executed by loading the html file in a web browser like *Google Chrome* or *Internet Explorer*. An example is shown below in figure 4.12. 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 be shown in **appletviewer** windows rather than a web browser because it is much faster, especially for lab lectures.

**Figure 4.12**



|  |
| --- |
| **drawLine Method Class: Graphics** |
| drawLine(int x1, int y1, int x2, int y2)  Draws a line from coordinate (x1,y1) to coordinate (x2,y2) |

**Drawing Rectangles with method drawRect**

A line is one-dimensional; it only has length. Many **Graphics** methods are two-dimensional and have the option to draw an outline only or to draw and fill the outline. For instance, the **drawRect** method will draw a rectangle of specified dimensions at a specified location. The **fillRect** method, with the exact same parameters will draw a rectangle at the same location, with the same dimensions and then also fill the rectangle. **Java0408.java**, in figure 4.13 will show the two types of rectangles using both of the **Rect** methods displayed in figure 4.14.

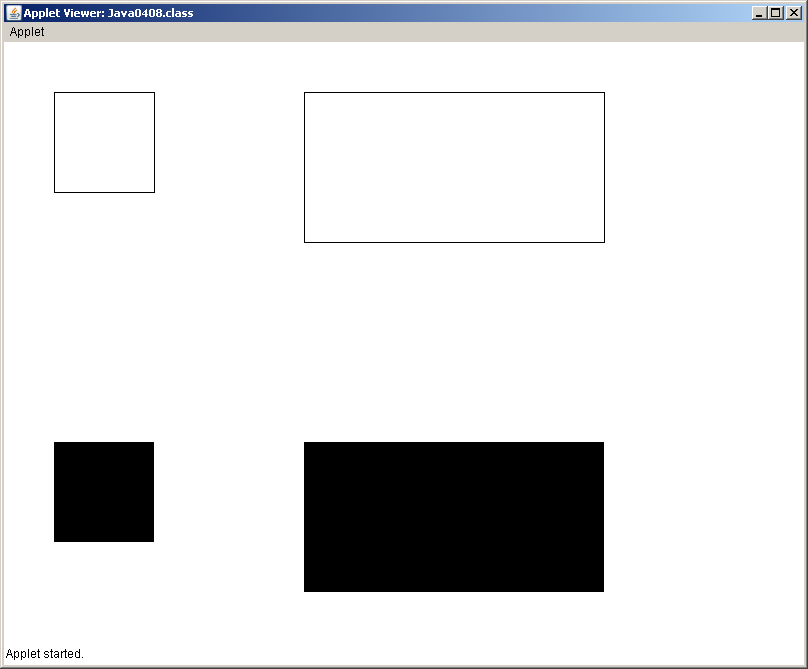
**Figure 4.13**

|  |
| --- |
| // Java0408.java  // This program introduces the rectangle command. A rectangle is drawn from  // the top-left (X,Y) coordinate of a rectangle followed by Width and Height using  // <drawRect(X,Y,Width,Height)>.  // The <fillRect> command draws a rectangle filled with solid pixels.  import java.awt.\*;  import java.applet.\*;  public class Java0408 extends Applet  {  public void paint(Graphics g)  {  g.drawRect(50,50,100,100);  g.drawRect(300,50,300,150);  g.fillRect(50,400,100,100);  g.fillRect(300,400,300,150);  }  } |

Methods **drawRect** and **fillRect** require four parameters. The first two parameters are the (x,y) coordinates of the top-left corner of the rectangle. The third parameter is the **width** of the rectangle and the fourth parameter is the **height** of the rectangle.

Right now all the graphics programs display in white and black only. This section will finish by showing how to display graphics designs in different colors.

**Figure 4.14**



You should have noted that it is not necessary to create a special **drawSquare** method. A *square is a rectangle* and if the third and fourth parameters are the same size, a square will automatically be displayed.

|  |
| --- |
| **drawRect and fillRect Methods Class: Graphics** |
| drawRect(int x, int y, int width, int height)  Draws a rectangle with top-left corner at coordinate (x,y) using width and height dimensions.    fillRect uses identical parameters, but fills in the rectangle. |

**Drawing Ovals with method drawOval**

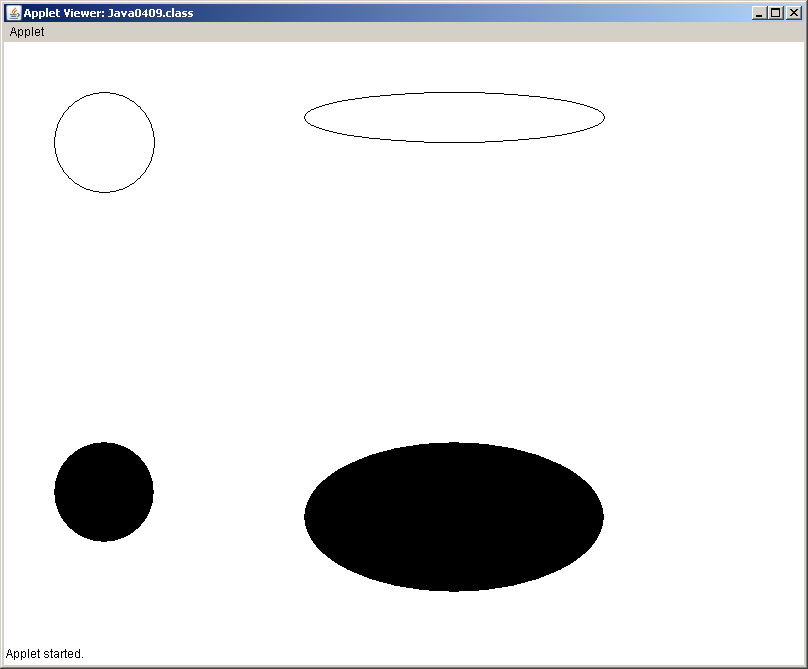
Drawing ovals and circles is completely identical to drawing rectangles and squares. Yes, I know that squares and circles are different, but the logic and the parameters required are identical. Java once again wants to know the top-left corner along with the width and height of the graphics object. The object is now an oval and the dimensions that are provided are for the rectangle that will outline the oval. This means that the oval is inscribed inside the rectangle or that the rectangle circumscribes the oval.

The first two parameters provided, the (x,y) coordinates of the top-left corner, are actually not any part of the oval object. You must imagine the rectangle that will circumscribe the oval and then provide the correct parameters. Program **Java0409.java**, in figure 4.15 shows how to use the **drawOval** and **fillOval** commands. Figure 4.16 demonstrates the output of this program.

**Figure 4.15**

|  |
| --- |
| // Java0409.java  // This program uses the <drawOval> method to draw ovals and circles.  // The four parameters of the <drawOval> method are identical to the parameters of  // the <drawRect> method. With <drawOval(X,Y,Width,Height)> (X,Y) is the  // coordinate of the top-left corner of the rectangle that circumscribes the oval.  // It also shows that the Graphics variable does not have to be "g".  import java.awt.\*;  import java.applet.\*;  public class Java0409 extends Applet  {  public void paint(Graphics screen)  {  screen.drawOval(50,50,100,100);  screen.drawOval(300,50,300,50);  screen.fillOval(50,400,100,100);  screen.fillOval(300,400,300,150);  }  } |

**Figure 4.16**

****

|  |
| --- |
| **drawOval and fillOval Methods Class: Graphics** |
| drawOval(int x, int y, int width, int height)  Draws an oval that is circumscribed by the rectangle with  top-left corner at coordinate (x,y) using width and height dimensions.    fillOval uses identical parameters, but fills in the oval. |

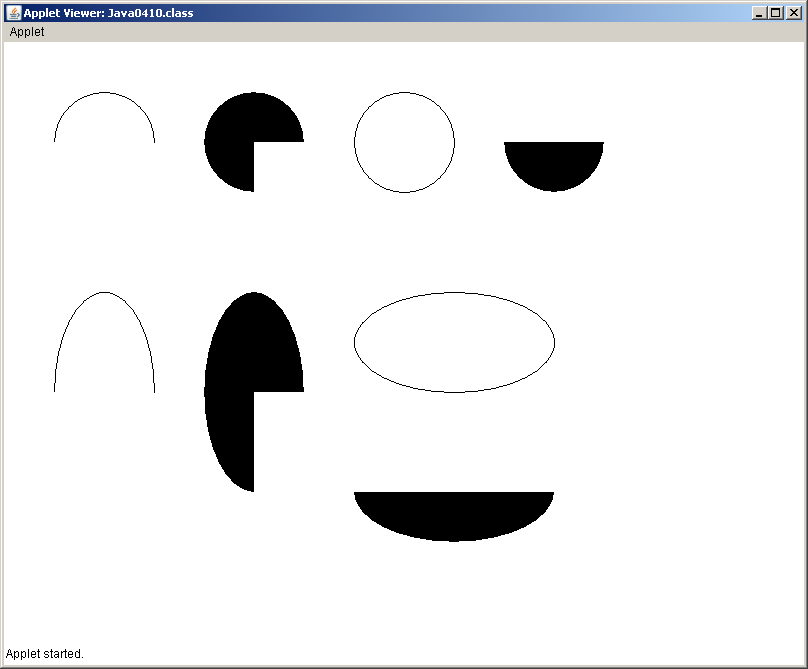
**Drawing Arcs with method drawArc**

Part of an oval or circle is called an arc. Methods **drawArc** and **fillArc** start with the logic provided for drawing ovals. The first four parameters are the same parameters used by the **drawOval** method, which means you describe the rectangle that will circumscribe the oval. That is correct, you need to think of the complete oval and provide rectangle information for such an oval. After the rectangle is determined you will need to provide two additional parameters that indicate the size of the arc. Program **Java0410.java**, in figure 4.17 demonstrates the arc method. Figure 4.18 shows the output.

**Figure 4.17**

|  |
| --- |
| // Java0410.java  // This program uses the <drawArc> and <fillArcs> methods.  // Method <drawArc(X,Y,Width,Height,Start,Degrees)> uses the first four  // parameters in the same manner as the <drawOval> method. Start is the  // degree value of the arc-start and Degrees is the number of degrees the arc travels.  // Start (0 degrees) is at 3:00 o'clock and positive degrees travel counter-clockwise.  import java.awt.\*;  import java.applet.\*;  public class Java0410 extends Applet  {  public void paint(Graphics g)  {  g.drawArc(50,50,100,100,0,180);  g.fillArc(200,50,100,100,0,270);  g.drawArc(350,50,100,100,0,360);  g.fillArc(500,50,100,100,0,-180);  g.drawArc(50,250,100,200,0,180);  g.fillArc(200,250,100,200,0,270);  g.drawArc(350,250,200,100,0,360);  g.fillArc(350,400,200,100,0,-180);  }  } |

**Figure 4.18**



Take a close look at the output of the arc program. The tricky part with **drawArc** is to determine where the starting location is and understand the meaning of the degrees. The 5th parameter is the start of the arc in degrees. Zero degrees is at the 3:00 o'clock position. The 6th parameter is the amount of degrees of the arc. It is not the end position. A positive value for the 6th parameter creates an arc that rotates counter-clockwise. A negative value rotates clockwise.

|  |
| --- |
| **drawArc and fillArc Methods Class: Graphics** |
| drawArc(int x, int y, int width, int height, int start, int degrees)  Draws an arc that is circumscribed by the rectangle with top-left corner at coordinate (X,Y) using Width and Height dimensions. Start indicates the degree location of the beginning of the arc and *degrees* indicates the degrees traveled by the arc.  **0** **degrees** is at the **3:00 o’clock** position and increases counter-clockwise to **360 degrees**. |

**Watch Your Parameter Sequence**

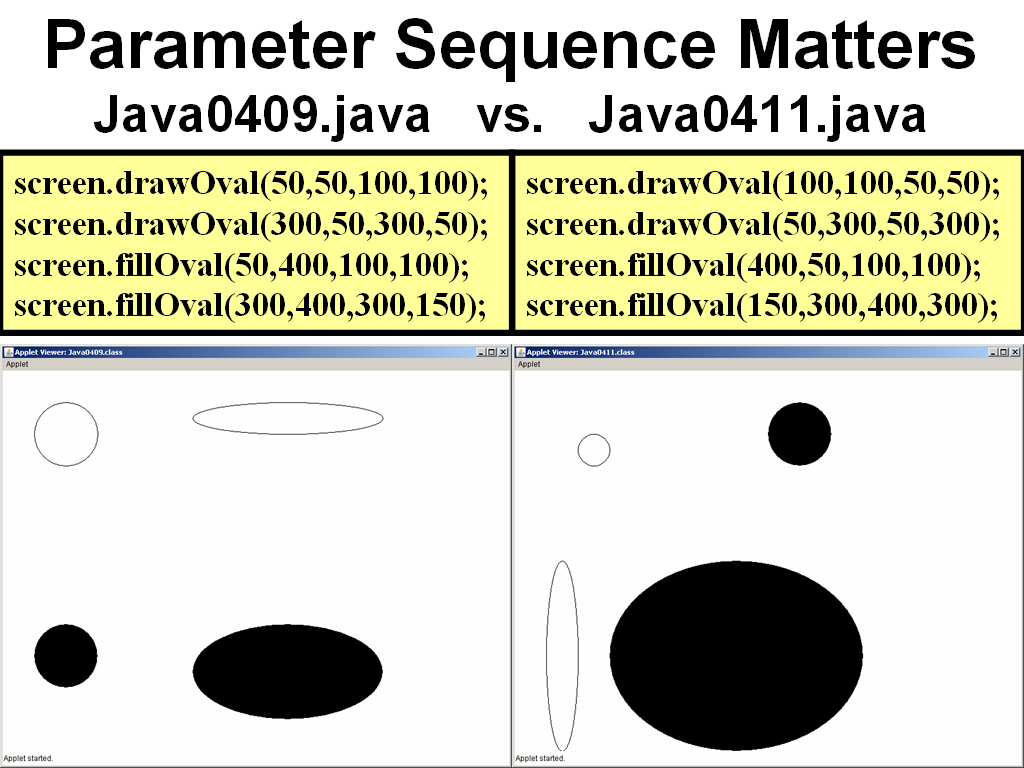
Students enjoy working with graphic objects. It is not very difficult to determine the information that is needed by graphics methods. Problems can occur when the sequence of parameters is ignored or misunderstood. Program **Java0411.java**, in figure 4.19, is pretty much the same program as **Java0409.java** with rearranged parameters. Look at figure 4.20 and see how different the graphics objects appear.

**Figure 4.19**

|  |
| --- |
| // Java0411.java  // This program demonstrates the significance of using parameters in the correct  // sequence Java0411.java is very similar to Java0409,java with rearranged parameters.  import java.awt.\*;  import java.applet.\*;  public class Java0411 extends Applet  {  public void paint(Graphics screen)  {  screen.drawOval(100,100,50,50);  screen.drawOval(50,300,50,300);  screen.fillOval(400,50,100,100);  screen.fillOval(150,300,400,300);  }  } |

Figure 4.20 compares the parameters and outputs of **Java0411.java** and of **Java0409.java**. Both programs use correct syntax and both programs will compile and execute. This is precisely the problem. When you use parameters incorrectly it may not always be apparent that the program has some logic error. Be careful with parameters. Use them correctly in the correct sequence.

**Figure 4.20**

****

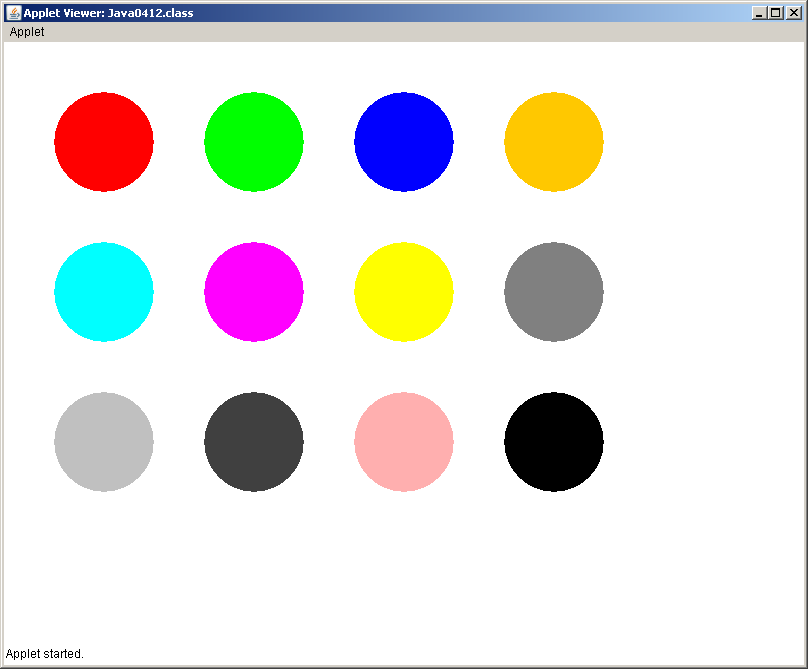
**Controlling Color with method setColor**

Every graphics objects displayed in each one of the graphics programs has been a display of dull looking black and white objects. The time has come to create some color. The **Graphics** class has a **setColor** method that can be called prior to calling any **draw** or **fill** method. The result will be a display of objects according to the selected color. The **Color** class has thirteen pre-set colors. Twelve of these are used by program **Java0412.java**, in figure 4.21 and 4.22. The thirteenth is **white**, which is not displayed because a while circle will not show up on a white background.

**Figure 4.21**

|  |
| --- |
| // Java0412.java  // This program demonstrates how to control the output display color with  // the <Color> class and the <setColor> method.  import java.awt.\*;  import java.applet.\*;  public class Java0412 extends Applet  {  public void paint(Graphics g)  {  g.setColor(Color.red);  g.fillOval(50,50,100,100);  g.setColor(Color.green);  g.fillOval(200,50,100,100);  g.setColor(Color.blue);  g.fillOval(350,50,100,100);  g.setColor(Color.orange);  g.fillOval(500,50,100,100);  g.setColor(Color.cyan);  g.fillOval(50,200,100,100);  g.setColor(Color.magenta);  g.fillOval(200,200,100,100);  g.setColor(Color.yellow);  g.fillOval(350,200,100,100);  g.setColor(Color.gray);  g.fillOval(500,200,100,100);  g.setColor(Color.lightGray);  g.fillOval(50,350,100,100);  g.setColor(Color.darkGray);  g.fillOval(200,350,100,100);  g.setColor(Color.pink);  g.fillOval(350,350,100,100);  g.setColor(Color.black);  g.fillOval(500,350,100,100);  }  } |

**Figure 4.22**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **setColor Method Class: Graphics** | | | | |
| **Color.constant Class: Color** | | | | |
| setColor(Color.constant)  Sets the graphics display color of the following graphics output to the specified constant of the **Color** class. There are 13 color constants listed below. | | | | |
| red  magenta pink | green  yellow  black | blue  gray  white | orange  lightGray | cyan  darkGray |
| NOTE: You are not limited to only these 13 colors.  By combining different amounts of red, green, and blue  values you can create any of over 16 million different colors.  You will be shown how to do this in a later chapter. | | | | |

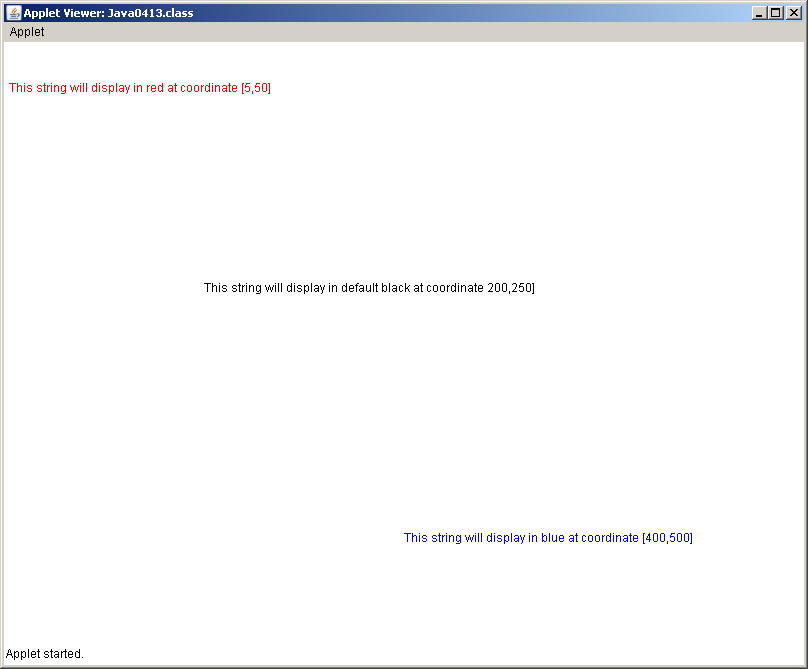
**Drawing Graphics Strings with method drawString**

This graphics section will conclude by displaying strings. You may be surprised to read that. You have already seen many examples with program statements like **System.out.println("How are you today?");** The **print** statements do produce string output, but it is done in a text window and not a graphics window. If you want to add some type of letters or words with your graphics programs you need to use the **drawString** method. Program **Java0413.java**, in figure 4.23, displays three different graphics strings. The approach is easy to do. Start by providing a string of characters followed by a set of coordinates to indicate the starting position of the string on the monitor. Figure 4.24 shows the strings. They are in three different colors, which may be difficult to see if you are looking at a printed version of this chapter.

**Figure 4.23**

|  |
| --- |
| // Java0413.java  // This program demonstrates the <drawString> method.  // With <drawString("Hello World",x,y)>, the string Hello World  // will be displayed starting at the [x,y] pixel coordinate.  import java.awt.\*;  import java.applet.\*;  public class Java0413 extends Applet  {  public void paint(Graphics g)  {  g.drawString("This string will display in default black at coordinate 200,250]",200,250);  g.setColor(Color.red);  g.drawString("This string will display in red at coordinate [5,50]",5,50);  g.setColor(Color.blue);  g.drawString("This string will display in blue at coordinate [400,500]",400,500);  }  } |

**Figure 4.24**



|  |
| --- |
| **drawString Method Class: Graphics** |
| drawString(String s, int x, int y)  Draws a string s starting at the coordinate (x,y). |

|  |
| --- |
| **PreAP and AP Graphics Alert** |
| If you used the **Expo** class in a PreAP Computer Science course, be aware that you will strictly use Java commands that are part of Java standard libraries.  Using the **Expo** class was like riding a bike with training wheels.  Now that you are in **AP** Computer Science, the time has come to remove those training wheels. |

**C:\Documents and Settings\JohnSchram\Local Settings\Temporary Internet Files\Content.IE5\2WY1E5QY\MCj02310490000[1].wmf**

****

**4.6 Observing the GridWorld Start**

The GridWorld Case Study was introduced in Chapter III and you will see it pop up in many chapters. Not every chapter, but plenty. There will not be some introduction every time GridWorld is used to tell you *well here we are again back to using the GridWorld Case Study.* Starting with the next time somewhere in the chapter a new concept will be introduced and it may well be that using the GWCS provides some good examples. It is also possible that specific features of the GridWorld's program are explained in detail. In other words, you are learning computer science and to learn computer science you need to start by learning programming. Learning programming will be done with the language Java and learning Java will be done with the help of the GridWorld Case Study.

Using GridWorld here is quite appropriate since you have just finished looking at many graphics programs. All the previous graphics programs in this chapter were written as applets. You were required to compile the java program and then switch to the web page file ending in **.html** to execute the applet program. The GridWorld programs are also graphics programs, but they are meant to be executed as application programs and you will notice that there is a **main** method.

You have already compiled and executed a GridWorld program and you have observed the execution of the program using **Step** and **Run**. Now let us take a closer look. In this chapter you are not doing a lab experiment. That means you will not be creating a GridWorld project and observing its execution. The work of compiling and executing is done for you. Sometimes the GWCS is presented with text and pictures and at other times you are guided along with some lab experiment.

On the next page you will see four executions of a GridWorld program. Make sure that you understand the difference between executing the GridWorld program from your Integrated Development Environment (**IDE**), like JCreator and some other IDE. This first execution gets you a static environment of GridWorld objects that do nothing. There is a second execution that occurs when you click on the **Run** button of the GridWorld display.

The four displays shown by figure 4.25 are the result of four separate execution starts from your IDE. Each time the execution was closed and restarted. Observe the output and what do you notice?

**Figure 4.25**

|  |  |
| --- | --- |
|  |  |
|  |  |

You should observe that each execution is both the same and different. The executions are the same in that you see one **Bug**, one **Rock**, one **Flower** and one **Actor** (mask) object. The executions are different in that each one of the four GridWorld objects starts in a different location. This is one of many observations about a GridWorld program where you wonder what is going on. In this chapter you will only observe. Soon enough you will get many explanations.

**4.7 Examining the Grid Environment**

Altering the behavior of the GridWorld objects and creating new GridWorld objects can be challenging and a concern for future chapters. Manipulating the GridWorld environment is not much of a problem. Figure 4.26 shows a typical GridWorld display. The objects on the grid may change in appearance, location and quantity, but the GridWorld environment will have the same features available.

**Figure 4.26**

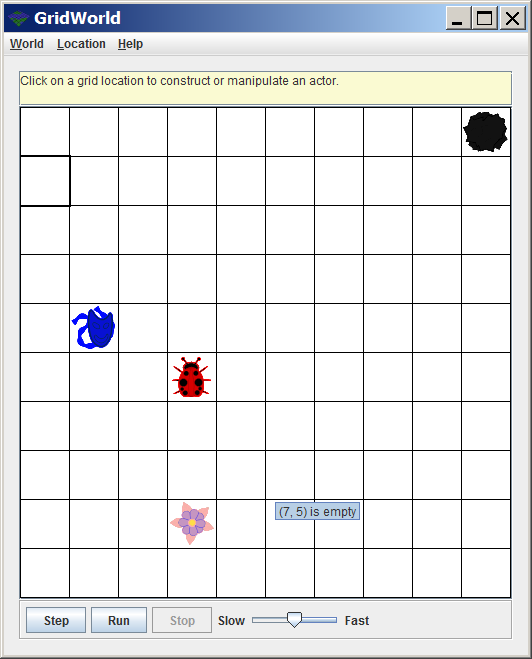
****

Figure 4.27 shows the **World** selection at the top-left corner. This feature lets the user select between a *bounded* grid of a specified number of rows and columns, or an *unbounded* grid that has no boundaries. The majority of the early GridWorld examples will use a 10 x 10 bounded grid. This is the default grid size and default is what you get when you do not specify.

**Figure 4.27**

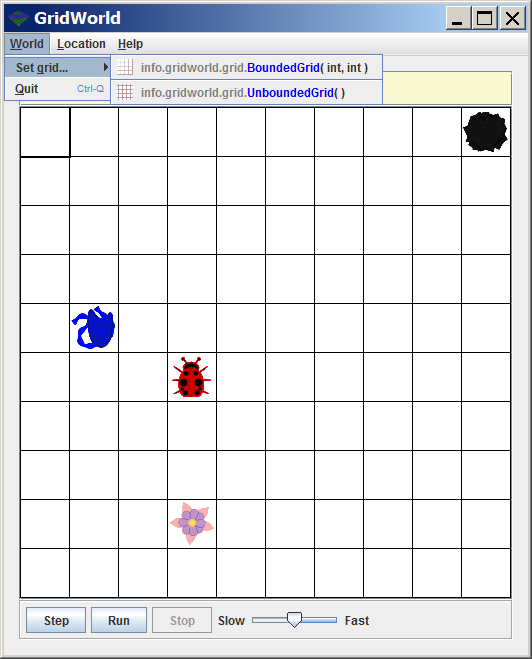
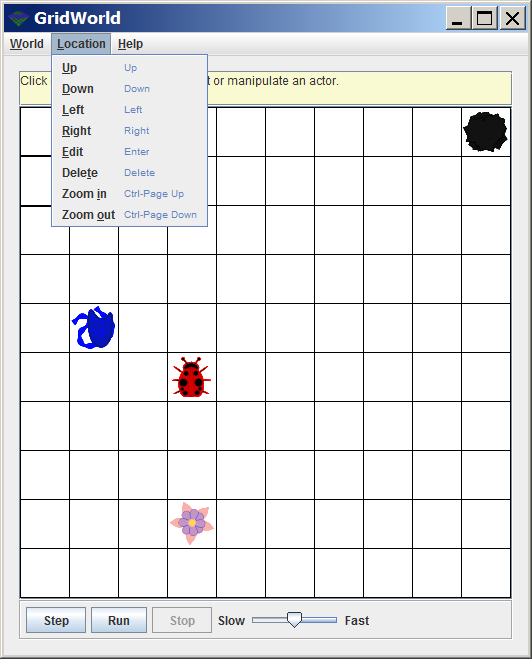
****

Figure 4.28 outlines the **Location** button. You may note a highlighted cell on the grid and you can make this cell location move up or down, left or right. The *Zoom* feature, shown figure 4.29 allows small cells to be viewed easier.

**Figure 4.28**

****

**Figure 4.29**

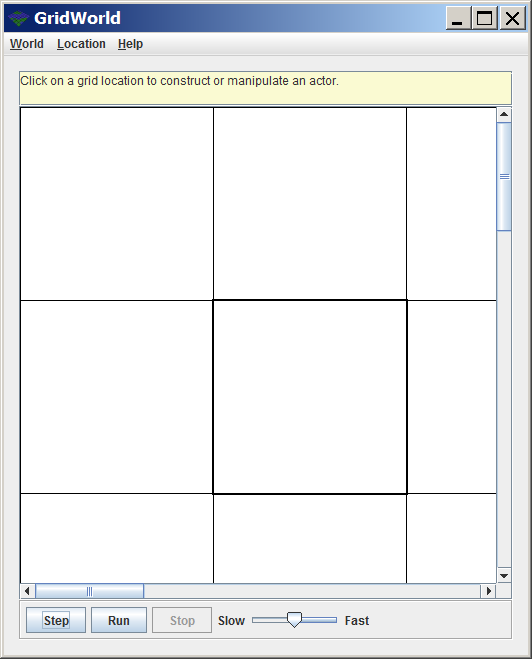
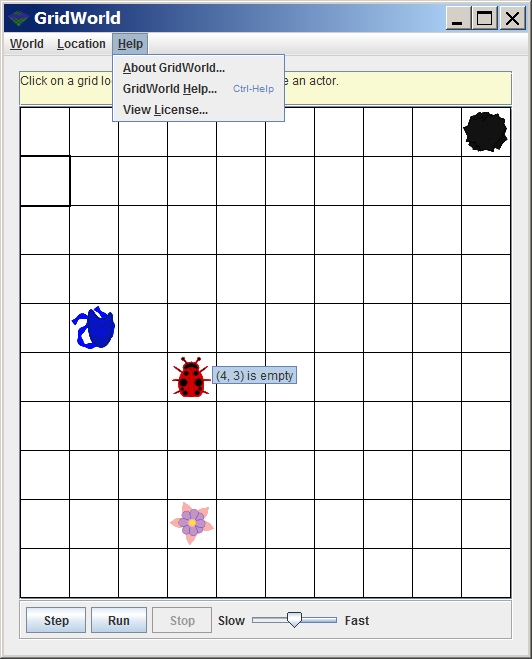
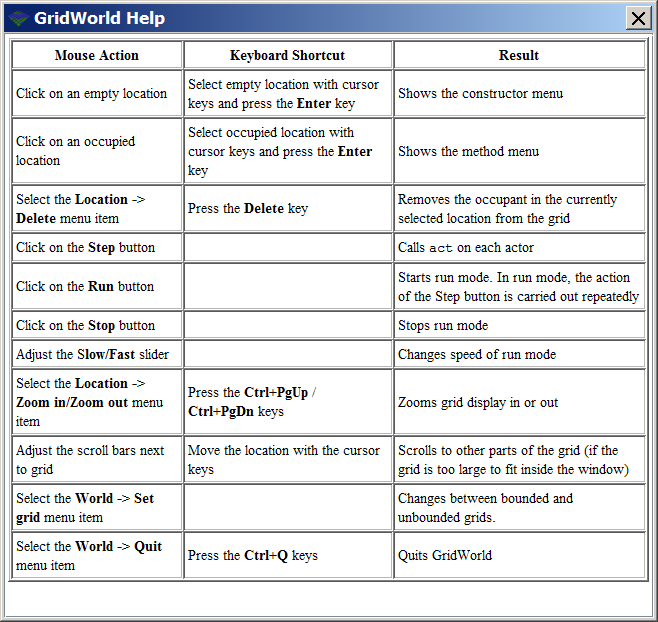
****

Figure 4.30 has the **Help** button. This gives you copyright information about the GridWorld Case Study and also provides an online help feature. This is not an Internet-connected help feature, but it does bring up the **GridWorld Help** matrix that you see displayed in figure 4.31.

**Figure 4.30**

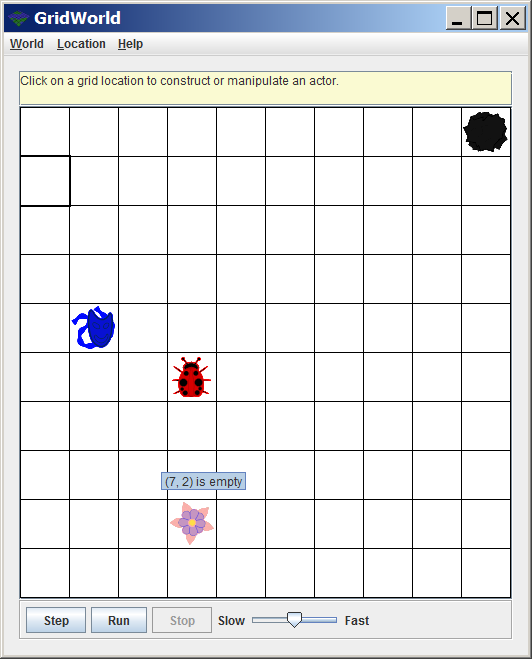
****

**Figure 4.31**

****

At the bottom of the Grid is figure 4.32, which has four execution features. You can use **Step**, which makes each object on the Grid act once. You can select **Run**, which makes all the objects act continuously at a speed determined by the speed slide. You can click **Stop**, which stops the execution of the Grid objects.

**Figure 2.8**

****

**4.8 GridWorld Objects Behavior**

We will now take a closer look at the behavior of different GridWorld objects. Figure 4.33 shows six consecutive steps of GridWorld from left-to-right and top-to-bottom. Now look at one object at a time and check the six displays. Start with the blue **Actor** object. One thing is sure, the actor does not move from its location. However, it does move in place. It starts right-side-up and then turns up-side-down and keeps flipping 180 degrees.

Check out the black **Rock** next. The rock does not move, does not change color, and simply sits doing nothing. This is logical behavior for a rock.

The **Bug** object moves one cell forward in the direction that it faces and drops a flower in its path. However, when it comes to another object, it makes a 45 degree, clock-wise turn to find a new path. It is not positive yet if this will happen with every object encountered by a bug.

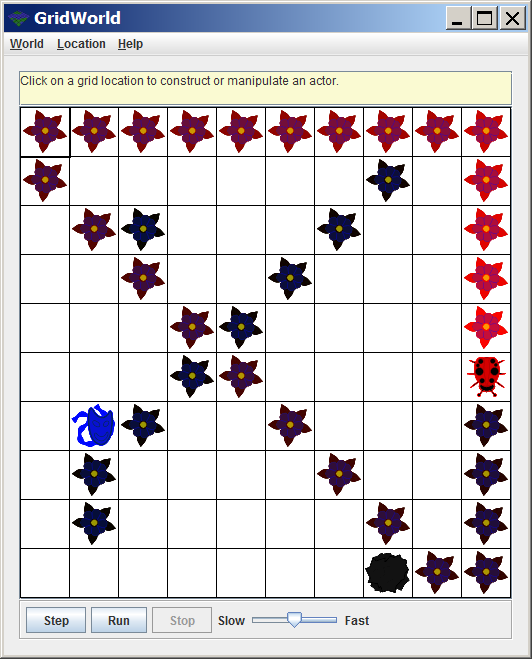
Six steps may not be enough for the **Flower** behavior to observe. Check it during the lab assignment. You will see the flowers turning darker with each step.

**Figure 4.33**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Figure 4.34 shows the GridWorld after more than 40 steps. With that many steps you can observe more detailed behavior. With a colored display you should be able to tell that the flowers wilt with time, changing from red to black. There is evidence that the bug not only avoids actors, but also rocks. However, the bug has no problem running over its very own flowers and then drops some new ones.

**Figure 2.10**



There are more objects to study with the GWCS, but for now this is the initial list. There are more objects that are already created for the GWCS and you also have the option to create your very own objects that can have its own behavior. All of this will be explored and learned in due time.

**4.9 Summary**

This chapter introduced Object Oriented Programming, known as **OOP**. The biggest concern to professional programmers is program **reliability**. The evolution to *Object Oriented Programming* has increased the reliability of large, complex programs considerably. It will take many more chapters to appreciate why Object Oriented Programming makes programs more reliable and makes programs easier to write, debug and enhance as well.

The importance of OOP makes it necessary to start thinking about objects and working with objects early on in the course. Learning OOP starts early, but it will take the entire course. A gradual, steady introduction of new OOP topics throughout the course will make Object Oriented Programming manageable, rather than bewildering. You will not only find new OOP topics introduced in many chapters to come, but you will also see frequent repetitions of topics that were already introduced.

The key point introduced here is the notion that objects have capabilities that are neatly wrapped inside a container. Both the data and any special actions that need to access the data are placed inside the same container or capsule. This is known as **encapsulation**.

A general container with data and actions, known as **methods**,is called a **class**. A class is a data type. One particular variable of a **class** is 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 all the **Math** methods.

This chapter also introduced methods of the **Graphics** class. Each one of the program examples created an applet, which can only be viewed with a small web page that includes the byte code file of the applet. Methods like **drawLine**, **drawRect** and other **Graphics** methods are called by using a variable object identifier of the **Graphics** class. In the program examples this variable normally was **g**, but it can be any name. Access to **Graphics** methods and **Applet** features requires that certain Java libraries are "imported" into the program. Using statements at the top of the program like:

**import java.awt.\*;**

**import java.applet.\*;**

give the Java compiler the required information to use the methods in the program and create an applet for a web browser.

The majority of the graphics programs in this chapter were applet programs and required a special web page **\*.html** file to execute the program in a web browser or in **AppletViewer**.

The GridWorld programs, that followed are also graphics programs, but they are application programs with a **main** method and do not run in a web browser.

The GridWorld Case Study was presented again and you observed features about the GridWorld environment and the behavior of the **Actor**, **Rock**, **Bug** and **Flower** objects.