**Chapter VI**

**Using Object Methods**

**Chapter VI Topics**

6.1 Introduction

6.2 Classes and Objects

6.3 Using Object Methods

6.4 Using the Random Class

6.5 Using the Math.random Method

6.6 Using the DecimalFormat Class

6.7 Working with Graphics Objects

6.8 Combining Random and Graphics Objects

6.9 Using the Scanner Class

6.10 Using the Integer Class

6.11 Summary

**6.1 Introduction**

Chapter IV started a gentle introduction to *Object Oriented Programming*. You learned a healthy quantity of new vocabulary used by OOP. After this brief OOP introduction, the chapter concentrated on using *class methods*. It really was fuzzy why you were using class methods rather than some other type of methods. This chapter aims to make the fog less dense. In particular, you will learn the difference between classes and objects as well as class methods and object methods.

Part of the problem is that the word *object* has two meanings. There is the meaning of object in the general sense as the word *object* in *Object Oriented Programming* (OOP). There is also a more technical meaning which helps to distinguish between a *class* and an *object*, and furthermore between a *class method* and an *object method*.

Right now your understanding about this object business should be that an object is some type of structure, or container that holds information like integers, characters, strings or more complex information. We can call this information data. An object stores more than data, because data by itself lacks all functionality. We want to add data, search data, alter data, display data and sort data. All these actions on the object data are performed by other object members, called *methods*, which have the ability to process data. The corner stone of objects is this business of packing both the data and various modules that process the data inside the same container.

In Chapter IV you used a variety of methods that processed data. You saw methods capable of computing the square root, absolute value, and other mathematical computations. All these methods were called *class methods.* Somehow this implies that a *class* can also hold action modules, just like an object.

It probably all sounds very confusing. I was quite confused at the first introduction to OOP and all the new vocabulary words took some time before I found myself on comfortable ground with all the different aspects of OOP. This chapter will start by clarifying the difference between a class and an object and then continue to show you how to use object methods.

**6.2 Classes and Objects**

The word *class* implies a collection of some category. In high school there may be a French class, a Geometry class, an Art class and a Computer Science class. A French class is not the same as a Geometry class, but you would expect first period Geometry to be quite similar to second period Geometry.

You can think of a class as a category, as a blue print, as a general description. You are surrounded by *classes*. A *cat*, a *dog*, a *car*, a *student*, a *soldier*, a *physician*, a *teacher*, a *mother*, a *baby* are just a few of the many classes that exist in your world. Now notice that none of these classes are specific. We all know what a cat is, but it is a general statement unlike *Fluffy*, which is a specific cat you can see and pet. In this example *cat* is a *class* and *Fluffy* is an *object.*

Figure 6.1 shows one example at the start of a GridWorld execution. Every object on this grid is in the same category. You see five objects, but they are all objects of the same **Bug** class.

|  |  |
| --- | --- |
| **Figure 6.1** | Figure 6.1 has:  1 Class  5 Objects  or  1 **Bug** class  5 **Bug** objects |

|  |  |
| --- | --- |
| **Figure 6.2** | Figure 6.2 has:  2 Classes  8 Objects  or  1 **Bug** class  1 **Rock** class  5 **Bug** objects  3 **Rock** objects |
| **Figure 6.3** | Figure 6.3 has:  4 Classes  4 Objects  or  1 **Bug** class  1 **Rock** class  1 **Actor** class  1 **Flower** class  1 **Bug** object  1 **Rock** object  1 **Actor** object  1 **Flower** object |

|  |  |
| --- | --- |
| **Figure 6.4** | Figure 6.4 has:  4 Classes  12 Objects  or  1 **Bug** class  1 **Rock** class  1 **Actor** class  1 **Flower** class  3 **Bug** objects  3 **Rock** objects  3 **Actor** objects  3 **Flower** objects |

You may realize that a class is a category and that an object is one example or object of that category. There is a **Student** category or class and **kathy** is one specific example of a **Student** object. Make sure that you make the distinction between classes and objects.

|  |
| --- |
| **Classes and Objects** |
| A class is a category.  An object is one example of a category.  You can have more objects than classes.  You can never have more classes than objects.  By Java convention all class identifiers start with upper-case and all object identifiers start with lower-case  . |

Now you are no stranger to working with general concepts (*classes)* and specific cases (*objects)*. You have written programs that used **int**. The **int** data type is general and cannot be used by itself so you create variables like **number1** and **number2**, each being a variable of the **int** type. This means that we are not interested in working with just one integer, but with as many different integers as our program requires. Does this mean that **int** is a class and **number1** is an object? No, it does not and **number1** can only store a single integer value without any module available to process the integer value. The example on **int** and **number1** is used to illustrate the concept of a general item like **int** and one specific kind of **int**, which is **number1**.

Now let us look at a class you have seen used in several previous program examples, the **Math** class. Figure 6.5 is an illustration of the **Math** classes with a small set of the actual members that are found in this class.

**Figure 6.5**

|  |
| --- |
| **Math Class** |
| **round()**  **min()**  **E**  **sqrt()**  **PI**  **abs()**  **floor()**  **pow()**  **ceil()**  **max()** |

Figure 6.5 shows a **Math** class with ten members. The class is not complete, but that does not matter. Consider this question. How many different copies do you need of the **sqrt** method, or the **round** method? Do you need to store different values of **PI** for different situations? Hardly, you only need one copy of each one of the class members. The data **PI** and **E** will always be the same and the functions **sqrt**, **abs**, **pow**, **min**, **max**, **floor**, **ceil** and **round** have no reason to change.

So what is the point? The point is that we can function very nicely with **one** copy of a **Math** class. We have no need to store a variety of different types of data. When we speak about an integer, we need to know which integer we are using. Do you need to know, which square root or which PI is being used? No they are always the same. In those cases where data does not change, where action modules always behave in the same manner there is only one copy needed and modules or *methods* in such a case are called *class methods*. This is precisely what you did with the **Math** class.

Perhaps it makes sense that there are classes with data that does not change. The **Math** class did a good job to illustrate that point. Now what about another type of class that stores data, which can change during program execution?

Consider a class to handle banking operations, appropriately called the **Bank** class. You will see such a class actually being used later in this chapter. Let us put some data and methods in the **Bank** class. Data can include **checkingBalance**, **savingsBalance**, **loanBalance** and **interestRate**. Methods for a **Bank** class can be **getChecking**, **getSavings**, **getLoan**, **makeDeposit**, **makeWithdrawal** and many other banking functions.

Now a question pops up immediately. In the **Bank** class is a data attribute called **savingsBalance**. If you call method **getSavings**, whose savings account are you accessing for a balance? This is totally different from using **Math.PI**. There is not a concern about which **PI** is used. The value of PI is the same yesterday, today and tomorrow. A savings balance is different for different individuals.

This means it is possible to call **Math.sqrt(100)** and the square root of **100** will be computed. It is not possible to call **Bank.makeDeposit(1000)**. One thousand dollars need to be deposited, but in whose account? Now it makes sense to say something like **tom.makeDeposit(1000)** or **sue.makeDeposit(1000)**. You can deposit money in Tom's account or Sue's account. So what happened to the **Bank** identifier? **Bank** becomes just like **int** and **tom** and **sue** are specific variables or **objects** of the **Bank** class. This also means that the methods of the **Bank** class are *object methods.*

You will find that object methods are far more common than class methods. You may wonder why the more common methods are not first explained. The simple reason is that class methods are easier to use. You start with a class name, like **Math**, you add a period • and follow this with a method identifier, like **sqrt**. Keep in mind that the method will probably require a parameter and then you the completed method call of . . .

**Math.sqrt(100)**

You will find that obnject methods are more involved and require the creation of an object before an object method can be called. So, there has been enough conversation without program examples. This section served as a brief theoretical discussion on *class methods* and *object methods*. We are now ready to look at the actual process to creating objects and using them in various program examples.

**6.3 Using Object Methods**

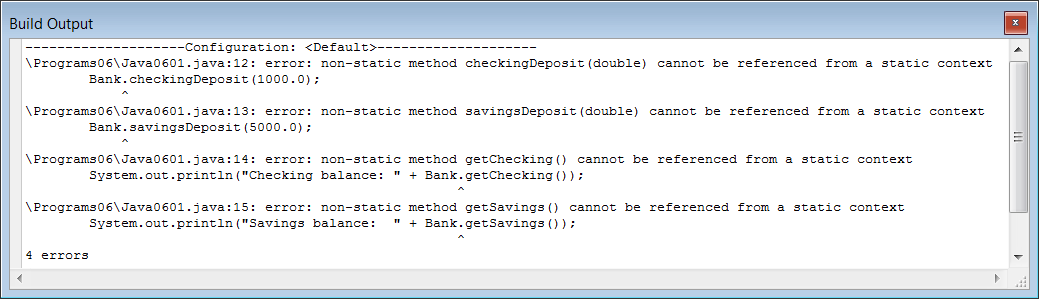
Chapter IV started with the **Math** class and ended with an introduction of several graphics classes and methods. The primary motivation to introduce graphics programming in Chapter IV was to enhance your understanding of method calls and parameter passing.

Program **Java0601.java**, in figure 6.6, tries to use the **Bank** class in the same manner you saw demonstrated with the **Math** class. The program tries to make deposits to the checking and savings accounts with the **checkingDeposit** and **savingsDeposit** methods. After the deposits are made, the **getChecking** and **getSavings** methods try to display the balances of the accounts in the **Bank** class.

With the **Math** class nobody worries about *whose square root* you are using and what might the value of **Math.PI** be today? It is different with a **Bank** class since there is fundamental question about whose *checking account* or whose *savings account* is altered? If this all seems weird, do not be concerned. This program will not even compile and the output box shows a variety of error messages indicating serious unhappiness with the programmer.

**Figure 6.6**

|  |
| --- |
| // Java0601.java  // This program demonstrates that the methods of a class are not always  // accessible, like they were with the <Math> class. In this case an  // attempt is made to use methods of the <Bank> class without success.  public class Java0601  {  public static void main (String args[])  {  System.out.println("\nJAVA0601.JAVA\n");  Bank.checkingDeposit(1000.0);  Bank.savingsDeposit(5000.0);  System.out.println("Checking balance: " + Bank.getChecking());  System.out.println("Savings balance: " + Bank.getSavings());  System.out.println();  }  } |

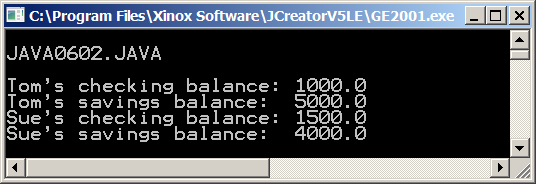


The error messages make many statements about *non-static methods*. This is a logical error message, but right now you do not know about *static* or *non-static* methods. At this stage please accept the fact that if you treat object methods like class methods, error messages will be plentiful.

Program **Java0602.java**, in figure 6.7, shows how to use *object methods* correctly. For each new customer a new **Bank** object is created, and for convenience sake, I have selected to identify each new object with the name of a bank customer.

**Figure 6.7**

|  |
| --- |
| // Java0602.java  // This program creates two Bank objects, called tom and sue. Each object stores its own Bank information.  public class Java0602  {  public static void main (String args[])  {  System.out.println("\nJAVA0602.JAVA\n");  Bank tom;  tom = new Bank();  Bank sue;  sue = new Bank();  tom.checkingDeposit(1000.0);  tom.savingsDeposit(5000.0);  sue.checkingDeposit(1500.0);  sue.savingsDeposit (4000.0);    System.out.println("Tom's checking balance: " + tom.getChecking());  System.out.println("Tom's savings balance: " + tom.getSavings());  System.out.println("Sue's checking balance: " + sue.getChecking());  System.out.println("Sue's savings balance: " + sue.getSavings());  System.out.println();  }  } |



There are two differences between the **Math** class and the **Bank** class. First, there are some statements that were never used before with any of the **Math** sample programs. These are the statements shown in figure 6.8.

**Figure 6.8**

|  |
| --- |
| **Bank tom;**  **tom = new Bank();**  **Bank sue;**  **sue = new Bank();** |

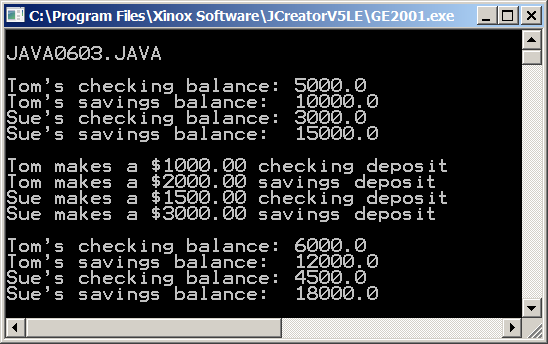
The statements **Bank tom;** and **Bank sue;** should not be very surprising. They are identical to the declarations that you have seen in previous chapters. It is a *data type* followed by a *variable identifier* format, like **int num;** the data type is **Bank** and the variables are **tom** and **sue**. This should make sense. We have one data type with two variables of the **Bank** data type or as we say the **Bank** class. This is precisely the distinction between a class and an object mentioned in the previous section. A class is a data type and an object is a variable.

Objects are more complicated variables than **int**, **double, boolean** or **char** variables. After you declare some identifier to be an object of a certain class, you also need to make sure that the new object is properly *constructed* to assume its new duties as an object. The statements **tom = new Bank();** and **sue = new Bank();** allocate the necessary space in memory for the new object and construct the new objects ready for business. What does it mean to *construct* a new object? It can mean many things, but right now think of constructing as a combination of allocating space for all the object data and properly initializing each data member of the object. You can think of the constructor method as a special method to open new accounts with the **Bank** class. The previous program opened two new accounts, one for **tom** and one for **sue**. All accounts were opened with a zero balance. After the two objects were constructed with the **new** operator, calls to **checkingDeposit** and **savingsDeposit** added money to the new accounts. The program finished with **getChecking** and **getSavings** to display the account balances for **tom** and **sue**.

It might make sense that two new accounts were opened for **tom** and **sue** with two new objects, but both accounts started with a zero balance. Objects are always created with some initial value and unless there is a way to specify an initial value, it will become some value like 0.0. Java allows constructor methods to come in different flavors. You will learn in a later chapter how to write your own classes and your own methods, but I will tell you right now that the special constructor method, which creates a new object, has the same identifier as the class identifier. Check out program **Java0603.java**, in figure 6.9, and see if you recognize which identifier is the **Bank** class and which identifier is the **Bank** constructor method. You will note that there are two types of *Bank* identifiers: one without and another one with parentheses. The **Bank** without parentheses is the class identifier and the **Bank(5000.0,10000.0)** is the constructor method. The new constructor has two parameters. The first parameter is the initial checking account balance and the second parameter is the initial savings account balance.

# Figure 6.9

|  |
| --- |
| // Java0603.java  // This program demonstrates how an object can be constructed with a specified initial balance in checking and  // savings. Most Java classes have multiple constructors to create objects for multiple situations.  public class Java0603  {  public static void main (String args[])  {  System.out.println("\nJAVA0603.JAVA\n");  Bank tom;  tom = new Bank(5000.0,10000.0);  Bank sue;  sue = new Bank(3000.0,15000.0);    System.out.println("Tom's checking balance: " + tom.getChecking());  System.out.println("Tom's savings balance: " + tom.getSavings());  System.out.println("Sue's checking balance: " + sue.getChecking());  System.out.println("Sue's savings balance: " + sue.getSavings());  System.out.println();    System.out.println("Tom makes a $1000.00 checking deposit");  tom.checkingDeposit(1000.0);  System.out.println("Tom makes a $2000.00 savings deposit");  tom.savingsDeposit(2000.0);  System.out.println("Sue makes a $1500.00 checking deposit");  sue.checkingDeposit(1500.0);  System.out.println("Sue makes a $3000.00 savings deposit");  sue.savingsDeposit(3000.0);  System.out.println();    System.out.println("Tom's checking balance: " + tom.getChecking());  System.out.println("Tom's savings balance: " + tom.getSavings());  System.out.println("Sue's checking balance: " + sue.getChecking());  System.out.println("Sue's savings balance: " + sue.getSavings());  System.out.println();  }  } |



Let us think back to Chapter III. You were shown statements that define and initialize variables like:

**int x;**

**x = 5;**

But then, you were also shown that it is better to define and initialize your variables in one statement like this:

**int x = 5;**

The same thing applies to creating objects. The previous program first declared an object, and then constructed it, with 2 separate statements:

**Bank tom;**

**tom = new Bank(5000.0,10000.0);**

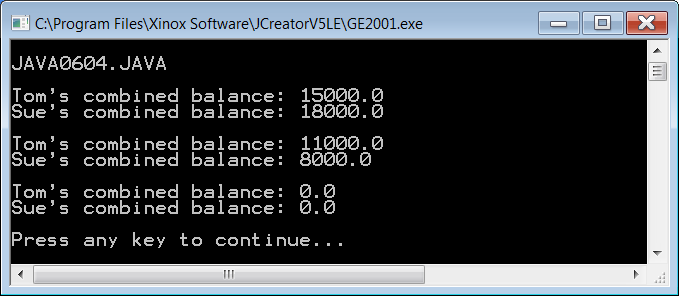
As before, it is better to combine these 2 statements into one. We can declare and construct the object in one statement like this:

**Bank tom = new Bank(5000.0,10000.0);**

Program **Java0604.java**, in Figure 6.10 demonstrates this concept. It also shows 2 *withdrawing* methods of the **Bank** class as well as the **getCombined** method, **closeChecking** methodand the **closeSavings** method. It was not mentioned before, but the **Bank** class is not part of any Java class library. I created this class to demonstrate how to use the object methods of a short, simple class. Your programs compiled and executed because the **Bank.class** file is in the same directory as your other programs.

**Figure 6.10**

|  |
| --- |
| // Java0604.java  // This program demonstrates how an object can be declared and defined all in one statement.  // It also demonstrates the <checkingWithdrawal> and <savingsWithdrawal> methods of the <Bank> class.  // It also demonstrates other <Bank> class methods like <getCombined>.  // This method returns the combined checking and savings balance.  // It also shows how to close bank accounts with the <closeChecking> and the <closeSavings> methods.  public class Java0604  {  public static void main (String args[])  {  System.out.println("\nJAVA0604.JAVA\n");  Bank tom = new Bank(5000.0,10000.0);  Bank sue = new Bank(3000.0,15000.0);  System.out.println("Tom's combined balance: " + tom.getCombined());  System.out.println("Sue's combined balance: " + sue.getCombined());  System.out.println();  tom.checkingWithdrawal(4000);  sue.savingsWithdrawal(10000);  System.out.println("Tom's combined balance: " + tom.getCombined());  System.out.println("Sue's combined balance: " + sue.getCombined());  System.out.println();  tom.closeChecking();  tom.closeSavings();  sue.closeChecking();  sue.closeSavings();  System.out.println("Tom's combined balance: " + tom.getCombined());  System.out.println("Sue's combined balance: " + sue.getCombined());  System.out.println();  }  } |



**6.4 Using the Random Class**

Generating random values is a surprisingly big deal in many ways. Billions of dollars are spent on advertising, which is largely based on the surveys of a population sample picked at random. Politicians are continuously "polling" their constituents, which requires a random survey sample . Video games have become big business and no video game can survive without a true random behavior of events. So how do we make something random? Java does have a **random** method, which belongs to the **Math** class with limited capabilities. There is also a **Random** class with a variety of methods to satisfy many random needs.

Program **Java0605.java**, in figure 6.11, introduces the **Random** class. I mentioned earlier that the actual Java language is not so large. However, Java is supported by a huge set of class libraries. Related methods and data form one class. Related classes are organized together in one location, called a package, and related packages are placed in even larger packages.

At the top of this hierarchy is the **java** package. The most important package is the **java.lang** package. This language package is automatically loaded for you. Other packages need to be "imported" for proper access to a class.

In figure 6.11 you will see the statement **import java.util.Random**. The lower-case identifiers are package names, and the upper-case **Random** is the class identifier. This means that the **Random** class is contained inside the **util** package, which in turn resides inside the **java** package.

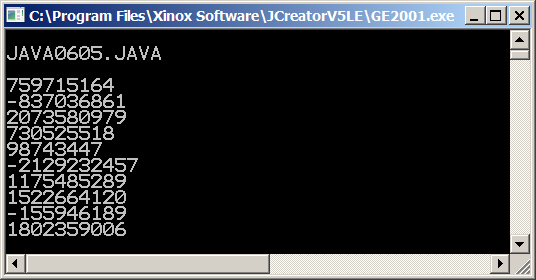
Program **Java0605.java** constructs a **rand** object of the **Random** class. Remember that **Random** is the data type (called a class) and **rand** is the variable (called an object). The **nextInt** method generates the next random integer value.

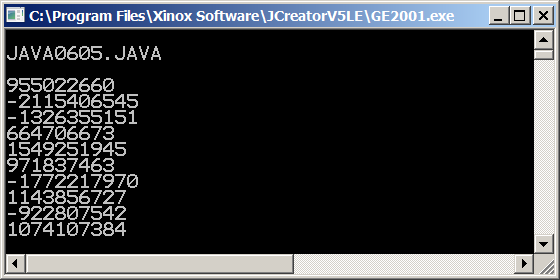
**Figure 6.11**

|  |
| --- |
| // Java0605.java  // This program introduces the <Random> class, which is part of the <java.util>  // package. The <nextInt> method returns a random integer value. Note that the  // values change each time that you execute the program.  import java.util.Random;  public class Java0605  {  public static void main (String args[])  {  System.out.println("\nJAVA0605.JAVA\n");  Random rand = new Random();  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println();  }  } |

Look carefully at the output of the **Java0605.java**. The program is executed twice. Do you expect the output to be the same set of numbers for each program execution or do you expect the output to be different each time?

**Figure 6.11 Continued**





There are two separate outputs for the same program. Note that each output is different and if you run the program your output will be different as well. This should make sense if your aim is to create a true set of random numbers. You will probably agree that these numbers are very random indeed.

It is also possible that you want some control over random numbers. As a teacher I may wish to see if everybody generates the same "random" results. How can I control that? Random number generators perform some fancy calculations to generate the next random number. These calculations can be controlled by specifying a starting seed. This seed insures that every execution will display the same sequence of random numbers. Is that random? It is the very first time that you run the program. You will not have a clue what the first, second or third number will be. However, after repeated executions, you will start to remember the pattern.

Program **Java0606.java**, in figure 6.8, is quite similar to the previous program, but this time the **rand** object is constructed with the **12345** parameter. The **12345** parameter is the "seed" of the new object. Watch the results as you look at three consecutive executions of this program.

**Figure 6.8**

|  |
| --- |
| // Java0606.java  // This program "seeds" the Random object with a specific starting seed.  // Multiple executions will now display the same random numbers.  import java.util.Random;  public class Java0606  {  public static void main (String args[])  {  System.out.println("\nJAVA0606.JAVA\n");  Random rand = new Random(12345);  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println(rand.nextInt());  System.out.println();  }  } |

**Figure 6.8 Continued**

****





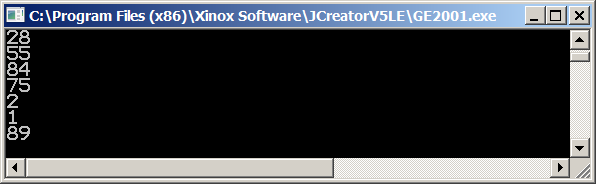
It is no secret what the difference is between the current program and the last program. You are looking at three identical executions, courtesy of constructing a **Random** object with a starting seed. You can experiment with different seeds. Different seeds will generate different numbers, but in each case the same set of numbers will be generated with the same starting seed.

You have seen that the special constructor method can be called with and without parameters. Methods with multiple capabilities are called *overloaded* and this is not limited to constructor methods. The last program example introduced the **nextInt** method, which did not use any parameter. Program **Java0607.java**, in figure 6.9, uses the **nextInt** method again, but now an integer parameter is used. In this case the number **100** is the parameter. The result is that the next number generated will be a random integer in the range of **[0..99]**. This feature of the **nextInt** method is very practical, because in most cases random numbers need to be generated in a special range. For instance a computer card game needs to generate numbers in the **[1. . .4]** range to determine the Clubs, Diamonds, Hearts or Spades suit. Additionally, a number in the **[1. . .13]** range will determine the value of the card within each suit.

The rule for using **nextInt** with an integer parameter **n** is that the next random integer **x** will be generated, such that **0 <= x < n**. If you check the output you will notice that all numbers fall in this range.

**Figure 6.9**

|  |
| --- |
| // Java0607.java  // This program demonstrates the second "overloaded" <nextInt(n)> method,  // which returns an integer x, such that 0 <= x < n.  import java.util.Random;  public class Java0607  {  public static void main (String args[])  {  System.out.println("\nJAVA0607.JAVA\n");  Random rand = new Random(12345);  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println();  }  } |



The **Random** class has a **setSeed** method with the ability to control the random sequence during program execution. It has the same impact as the random seed that is provided when a new **Random** object is constructed. Program **Java0608.java**, in figure 6.10, uses **12345** as a parameter for the new **rand** object as well as the parameter of the **setSeed** method.

**Figure 6.10**

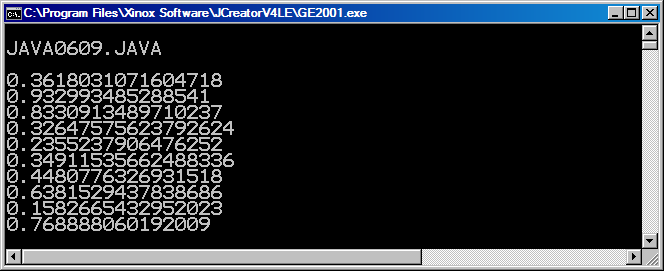
|  |
| --- |
| // Java0608.java  // This program introduces the <setSeed> method. This method allows you to  // control the "randomness" and repeat the same sequence.  import java.util.Random;  public class Java0608  {  public static void main (String args[])  {  System.out.println("\nJAVA0608.JAVA\n");  Random rand = new Random(12345);  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println();  rand.setSeed(12345);  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println();  }  } |



Randomness is not limited to integers. Figure 6.11 shows that it is also possible to generate random real numbers. Substitute **nextInt** with **nextDouble** and you will generate random real numbers **x**, such that **0 <= x < 1**.

**Figure 6.11**

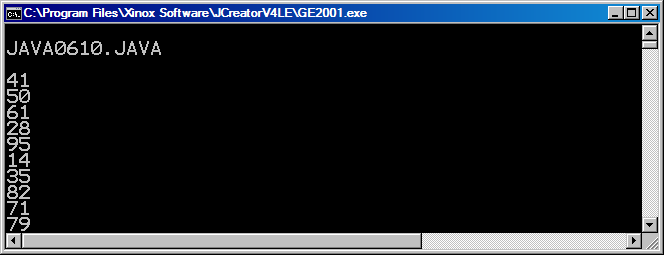
|  |
| --- |
| // Java0609.java  // This program demonstrates the <nextDouble> method,  // which returns a real x, such that 0 <= x < 1.  import java.util.Random;  public class Java0609  {  public static void main (String args[])  {  System.out.println("\nJAVA0609.JAVA\n");  Random rand = new Random(12345);  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println();  }  } |



With program **Java0610.java**, in figure 6.12, you provide an integer parameter with **nextInt** to control the upper bound of the random integer, but the smallest number always starts with **0**. You can alter this restriction by adding a "smallest" number. In the next program example you will see that all the numbers generate integers in the **[10. . .99]** range.

**Figure 6.12**

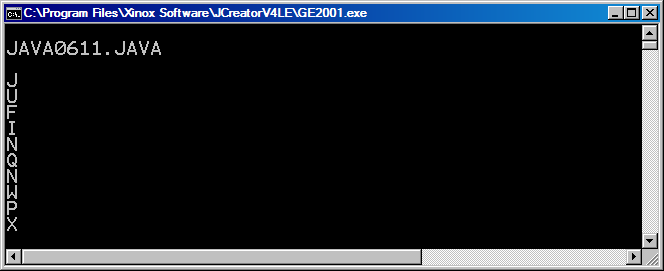
|  |
| --- |
| // Java0610.java  // This program demonstrates how to control an Random class object so that it  // generates integers in a desired range. In this example the range is [10..99].  import java.util.Random;  public class Java0610  {  public static void main (String args[])  {  System.out.println("\nJAVA0610.JAVA\n");  Random rand = new Random(12345);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println(rand.nextInt(90) + 10);  System.out.println();  }  } |



You must be impressed by the capabilities of the humble **Random** class. You have seen both random integers and random real numbers, but is it possible to generate random values that are not numbers, like characters? The answers is no and yes. Directly, the **Random** classwill not generate random characters. However, you remember that characters have special numerical code numbers. Upper-case **A** has code **65**, **B** has **66** and so on. If you generate random integers in the **65..90** range, followed by type casting to **(char)**, you will be in business. Sounds weird? Check program **Java0611.java**, in figure 6.13, to see the details.

**Figure 6.13**

|  |
| --- |
| // Java0611.java  // This program demonstrates how to control an Random class object so that it  // generates random upper-case letters. Note how "type casting" with (char) is  // used to change random integers in the [65..90] range to [A..Z] letters.  import java.util.Random;  public class Java0611  {  public static void main (String args[])  {  System.out.println("\nJAVA0611.JAVA\n");  Random rand = new Random(12345);  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println( (char) (rand.nextInt(26) + 65) );  System.out.println();  }  } |



|  |
| --- |
| **AP Examination Alert** |
| **The Random class and its methods are not tested on the AP Exam.**  **However, the random method of the Math class, Math.random(), will be tested.** |

**6.5 Using the Math.random Method**

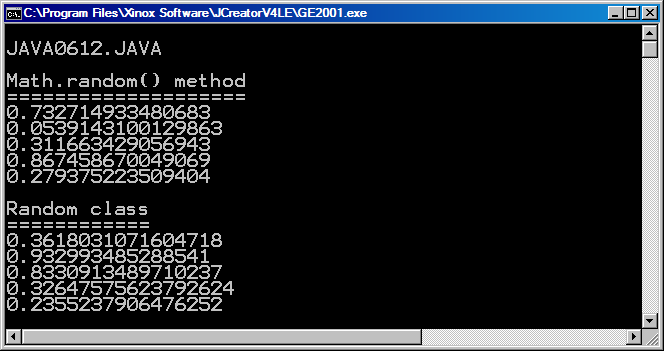
Just when you thought you understood the complexities of the **Random** class, here comes the **Math.random** method. You might be wondering why they even created the **Random** class if there was a perfectly good **random** method in the **Math** class. **Math.random** does not have all of the features of the **Random** class. How could it? **Math.random** is just one method. **Random** is an entire class full of methods. The next 4 programs will demonstrate **Math.random** and the **Random** class doing the same thing.

Program **Java0612.java**, in figure 6.14, shows that the **Math.random** method essentially does the same thing as the **nextDouble** method of the **Random** class.

**Figure 6.14**

|  |
| --- |
| // Java0612.java  // This program demonstrates the Math.random method.  // Notice that it behaves just like the nextDouble method of the Random class.  // Both will return a random real number n such that 0 <= n < 1  import java.util.Random; // needed for Random class, but not needed for Math.random method  public class Java0612  {  public static void main (String args[])  {  System.out.println("\nJAVA0612.JAVA\n");  System.out.println("Math.random() method");  System.out.println("====================");  System.out.println(Math.random());  System.out.println(Math.random());  System.out.println(Math.random());  System.out.println(Math.random());  System.out.println(Math.random());  System.out.println();  System.out.println("Random class");  System.out.println("============");  Random rand = new Random(12345);  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println(rand.nextDouble());  System.out.println();  }  } |

**Figure 6.14 Continued**

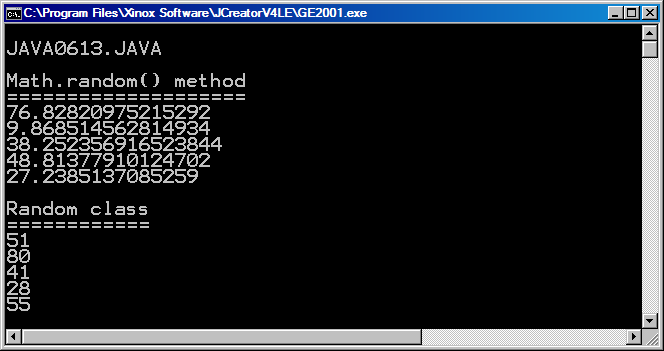


We see here that both the **Math.random** method and the **nextDouble** method of the **Random** class will return a random real number between 0.0 and 0.9999999... If you are provided with a random real number between 0.0 and 0.9999999, you can get any range of random numbers you wish. Program **Java0613.java**, in figure 6.15, shows the first example.

**Figure 6.15**

|  |
| --- |
| // Java0613.java  // This program demonstrates the Math.random method being used to display  // random real numbers in the range of 0 to 99.  // Note the comparison with the nextInt method of the Random class.  import java.util.Random;  public class Java0613  {  public static void main (String args[])  {  System.out.println("\nJAVA0613.JAVA\n");  System.out.println("Math.random() method");  System.out.println("====================");  System.out.println(Math.random() \* 100);  System.out.println(Math.random() \* 100);  System.out.println(Math.random() \* 100);  System.out.println(Math.random() \* 100);  System.out.println(Math.random() \* 100);  System.out.println();  System.out.println("Random class");  System.out.println("============");  Random rand = new Random(12345);  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println();  }  } |

**Figure 6.15 Continued**



At first glance the output generated by the **Math.random** method and the **nextInt** method of the **Random** class may look totally different. But look closely. Both sets of numbers are between 0 and 99. The **nextInt** method should be old news by now. You simply provide some integer as a parameter, call it *n*, and the result is you get a random number in the range of *0* through *n-1*.

How then is the same thing -- or close to the same thing -- accomplished with the **Math.random** method? Consider the same integer *n*. We now multiply this number by a random real number that is between 0.0 and 0.9999999... What will the result be? Let us look the examples in figure 6.16.

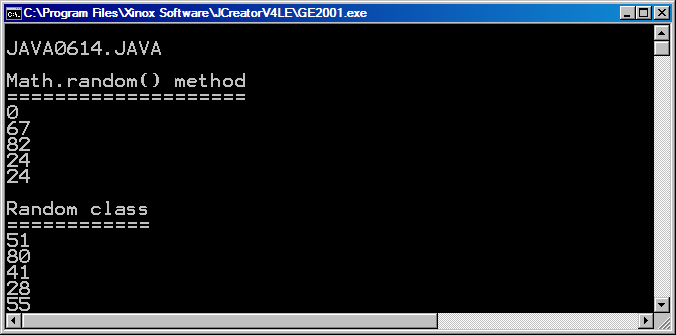
**Figure 6.16**

|  |  |  |
| --- | --- | --- |
| **n = ?** | **Math.random returns**  **smallest value of 0.0** | **Math.random returns**  **largest of 0.999999...** |
| **20** | **20 \* 0.0 = 0** | **20 \* 0.99999... = 19.99999...** |
| **50** | **50 \* 0.0 = 0** | **50 \* 0.99999... = 49.99999...** |
| **100** | **100 \* 0.0 = 0** | **100 \* 0.99999... = 99.99999...** |

Whatever *n* is, when it is multiplied by a real number between 0.0 and 0.9999999. The result will be a number between *0* and *n-1*. You probably noticed there is one important difference between the **nextInt** method of the **Random** class and the **Math.random** method. **nextInt** returns an integer. Program **Java0614.java**, in figure 6.17, shows how *typecasting* can help.

**Figure 6.17**

|  |
| --- |
| // Java0614.java  // This program demonstrates the Math.random method being used to display  // random integers in the range of 0 to 99 by using typecasting.  // Note the comparison with the nextInt method of the Random class.  import java.util.Random;  public class Java0614  {  public static void main (String args[])  {  System.out.println("\nJAVA0614.JAVA\n");  System.out.println("Math.random() method");  System.out.println("====================");  System.out.println( **(int)** (Math.random() \* 100) );  System.out.println( **(int)** (Math.random() \* 100) );  System.out.println( **(int)** (Math.random() \* 100) );  System.out.println( **(int)** (Math.random() \* 100) );  System.out.println( **(int)** (Math.random() \* 100) );  System.out.println();  System.out.println("Random class");  System.out.println("============");  Random rand = new Random(12345);  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println(rand.nextInt(100));  System.out.println();  }  } |



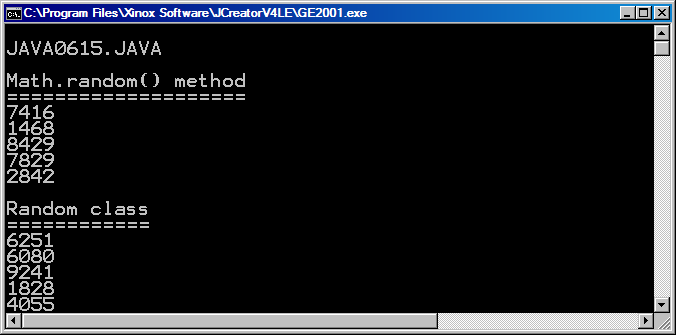
If our goal is to use **Math.random** to give us all of the capabilities of the **Random** class, then we are almost there. The only thing left is the fact that the range of numbers you want might not always start at *0*. What you are about to see is an example of a very important fact in computer science. While the *syntax* of doing something may differ from language to language -- or in this case, from the **Random** class to the **Math.random** method. The concept is still the same.

Program **Java0615.java**, in figure 6.18, shows several random integers in a range of 1000 to 9999 being displayed. If you look closely, you should see that while the syntax is different in the two parts of the program, the logic of what is being done is exactly the same.

**Figure 6.18**

|  |
| --- |
| // Java0615.java  // This program demonstrates the Math.random method being used to display  // random integers in the range of 1000 to 9999 by using typecasting.  // Note the comparison with the nextInt method of the Random class.  import java.util.Random;  public class Java0615  {  public static void main (String args[])  {  System.out.println("\nJAVA0615.JAVA\n");  System.out.println("Math.random() method");  System.out.println("====================");  System.out.println( (int) (Math.random() \* 9000) + 1000);  System.out.println( (int) (Math.random() \* 9000) + 1000);  System.out.println( (int) (Math.random() \* 9000) + 1000);  System.out.println( (int) (Math.random() \* 9000) + 1000);  System.out.println( (int) (Math.random() \* 9000) + 1000);  System.out.println();  System.out.println("Random class");  System.out.println("============");  Random rand = new Random(12345);  System.out.println(rand.nextInt(9000) + 1000);  System.out.println(rand.nextInt(9000) + 1000);  System.out.println(rand.nextInt(9000) + 1000);  System.out.println(rand.nextInt(9000) + 1000);  System.out.println(rand.nextInt(9000) + 1000);  System.out.println();  }  } |

**Figure 6.18 Continued**



To truly understand random numbers, whether you are using the **Random** class or the **Math.random** method, you need to understand that these 3 variables come into play:

L = Largest possible random number you want

S = Smallest possible random number you want

R = Range of all possible different random numbers

or put in mathematical terms:

R = L - S + 1

If we want a range of numbers between 1000 and 9999, then

L = 9999

S = 1000

R = 9999 - 1000 + 1 = 9000

If you look at program **Java0615.java**, you will see that the numbers 9000 and 1000 are used both with the **Random** class and the **Math.random** method. The *range* is used as the parameter for the **nextInt** method, and it is also the number, which multiplies the value returned by the **Math.random** method. And in both cases, the *smallest* number is what is added at the end. The only thing you do not see in either case is the *largest* number. It is used for calculating the *range*, but it is not used in the program itself. A very common mistake is to use the *largest* number instead of the *range*.

**6.6 Using the DecimalFormat Class**

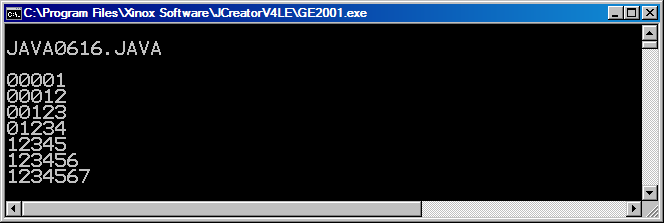
Hopefully, you are getting a feel for using class methods and object methods, but you need to explore one more class to solidify the concept in your mind. Java has a clever class, called **DecimalFormat**. It is a class that is found in the **text** package, which in turn is part of the **java** package. This means that you need to use the statement **import java.text.DecimalFormat;** to access this class. With the **DecimalFormat** class you can control the output appearance of numbers.

We will start the ball rolling with program **Java0616.java**, in figure 6.19. This program will "right justify" the output of all the numbers. This appearance is achieved with the **output** object of the **DecimalFormat** class. Notice how the **output** object is constructed with the **("00000")** parameter of five zeroes. This means that any number displayed with the **format** method will be shown with padded leading zeroes for any number with less than five digits. When a number is greater than five digits, the output format is ignored.

**Figure 6.16**

|  |
| --- |
| // Java0616.java  // This program demonstrates how to "right justify" integers with an object  // of the <DecimalFormat> class and the <format> method.  import java.text.DecimalFormat;  public class Java0616  {  public static void main (String args[])  {  System.out.println("\nJAVA0616.JAVA\n");  DecimalFormat output = new DecimalFormat("00000");  System.out.println(output.format(1));  System.out.println(output.format(12));  System.out.println(output.format(123));  System.out.println(output.format(1234));  System.out.println(output.format(12345));  System.out.println(output.format(123456));  System.out.println(output.format(1234567));  System.out.println();  }  } |

**Figure 6.19 Continued**



Program **Java0617.java**, in figure 6.20, demonstrates that it is possible to display large numbers with the traditional comma displayed every third digit.

**Figure 6.20**

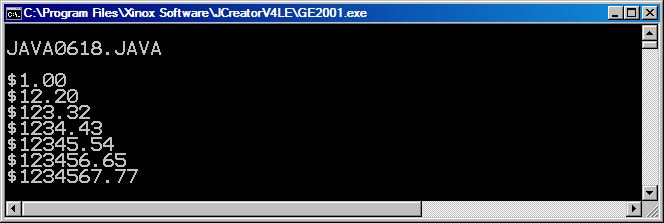
|  |
| --- |
| // Java0617.java  // This program demonstrates how to insert commas in numerical output  // with a <DecimalFormat> object.  import java.text.DecimalFormat;  public class Java0617  {  public static void main (String args[])  {  System.out.println("\nJAVA0617.JAVA\n");  DecimalFormat output = new DecimalFormat("0,000,000");  System.out.println(output.format(1));  System.out.println(output.format(12));  System.out.println(output.format(123));  System.out.println(output.format(1234));  System.out.println(output.format(12345));  System.out.println(output.format(123456));  System.out.println(output.format(1234567));  System.out.println();  }  } |



With the **DecimalFormat** class it is also possible to add zeroes on the right side of a number after the decimal point. Program **Java0618.java**, shown in figure 6.21, displays every number in a standard "US currency" format with a dollar sign and two digits after the decimal point. There is something else that you should notice. Several examples use the identifier **output** for the **DecimalFormat** object, but this object does not have to be called **output**. It is an object, and like any object it can have any name – preferably one than makes sense. In this case, I found it more appropriate to name the object **money**.

**Figure 6.21**

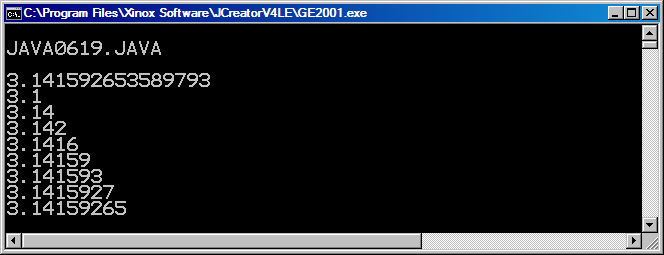
|  |
| --- |
| // Java0618.java  // This program demonstrates how to display US currency amounts.  // Additionally note how the <format> methods rounds off to the nearest penny.  // It also shows that the <DecimalFormat> object does not need to be called <output>.  import java.text.DecimalFormat;  public class Java0618  {  public static void main (String args[])  {  System.out.println("\nJAVA0618.JAVA\n");  DecimalFormat money = new DecimalFormat("$0.00");  System.out.println(money.format(1));  System.out.println(money.format(12.2));  System.out.println(money.format(123.32));  System.out.println(money.format(1234.432));  System.out.println(money.format(12345.543));  System.out.println(money.format(123456.654));  System.out.println(money.format(1234567.7651));  System.out.println();  }  } |



The next program example in this chapter shows how it is possible to use **DecimalFormat** objects to round off a real number to a required number of digits beyond the decimal point. The value of PI is used by **Java0619.java**, in figure 6.22, to show an increasing precision that starts with one digit beyond the decimal point up to eight digits.

**Figure 6.22**

|  |
| --- |
| // Java0619.java  // This program demonstrates how to control rounding off to a specified  // number of digits beyond the decimal point.  import java.text.DecimalFormat;  public class Java0619  {  public static void main (String args[])  {  System.out.println("\nJAVA0619.JAVA\n");  DecimalFormat output1 = new DecimalFormat("0.0");  DecimalFormat output2 = new DecimalFormat("0.00");  DecimalFormat output3 = new DecimalFormat("0.000");  DecimalFormat output4 = new DecimalFormat("0.0000");  DecimalFormat output5 = new DecimalFormat("0.00000");  DecimalFormat output6 = new DecimalFormat("0.000000");  DecimalFormat output7 = new DecimalFormat("0.0000000");  DecimalFormat output8 = new DecimalFormat("0.00000000");    System.out.println(Math.PI);  System.out.println(output1.format(Math.PI));  System.out.println(output2.format(Math.PI));  System.out.println(output3.format(Math.PI));  System.out.println(output4.format(Math.PI));  System.out.println(output5.format(Math.PI));  System.out.println(output6.format(Math.PI));  System.out.println(output7.format(Math.PI));  System.out.println(output8.format(Math.PI));  System.out.println();  }  } |



**6.7 Working with Graphics Objects**

Graphics programs are more interesting with color. You did learn that you may use the **setColor** method with some preset colors. The color variety is quite good, but still, preset color constants provide little variety. In this chapter you did learn how to create your own objects with the **new** operator. Put this knowledge to use and you can create some **Color** objects by providing three parameter values.

Do you realize how many different colors can be created? Each color is a combination of red, green and blue shades. Each one of the *red*, *green, blue* parameters has a color value in the [0..255] range. This means that there are *256 \* 256 \* 256* different combinations for a total of **16,777,216** different colors. It is true that many of these colors have very tiny shade differences, but the point is that we can do bunches better than the preset color constants, like **Color.red**, provided by Java. Program **Java0620.java**, in figure 6.23 only creates three new colors. The point here is not to create many colors, but to explain the process necessary to create your own **Color** objects.

**Figure 6.23**

|  |
| --- |
| // Java0620.java  // This program shows how you can create your own display colors by  // creating color objects with different (Red, Green, Blue) values.  // RGB values are in the [0..255] range.  import java.awt.\*;  import java.applet.\*;  public class Java0620 extends Applet  {  public void paint(Graphics g)  {  g.setColor(new Color(255,0,255));  g.fillRect(0,0,800,200);    g.setColor(new Color(0,255,255));  g.fillRect(0,200,800,200);    g.setColor(new Color(100,100,100));  g.fillRect(0,400,800,200);  }  } |

**Figure 6.23 Continued**

****

Program **Java0621.java**, in figure 6.24 uses three loop structures to iterate through every possible *red*, *green* and *blue* shade that can possibly be created. Each color starts with black and then gradually changes to the most intense red or green or blue.

**Figure 6.24**

|  |
| --- |
| // Java0621.java  // This program shows all the shades of Red, Green and Blue using the <setColor> method.  import java.awt.\*;  import java.applet.\*;  public class Java0621 extends Applet  {  public void paint(Graphics g)  {  for (int red = 0; red <= 255; red++)  {  g.setColor(new Color(red,0,0));  g.drawLine(red,0,red,600);  }    for (int green = 0; green <= 255; green++)  {  g.setColor(new Color(0,green,0));  g.drawLine(green+255,0,green+255,600);  }  for (int blue = 0; blue <= 255;blue++)  {  g.setColor(new Color(0,0,blue));  g.drawLine(blue+510,0,blue+510,600);  }  }  } |

**Figure 6.24 Continued**

****

The objects of the **Bank** class, the **Random** class and the **DecimalFormat** class all had names. In all these program examples you saw statements such as

**Bank tom = new Bank(2000);**

**Bank** is the class identifier. **tom** is the object identifier and **new Bank(2000)** calls the **Bank** constructor. This was the pattern with classes that use object methods. You construct a variable object with the **new** operator.

Did you take a close look at programs **Java0620.java** and **Java0621.java**? Was there something peculiar or different? In particular, how were the **Color** objects created? Let me reprint one of the statements that constructs a **Color** object.

**g.setColor(new Color(255,0,255));**

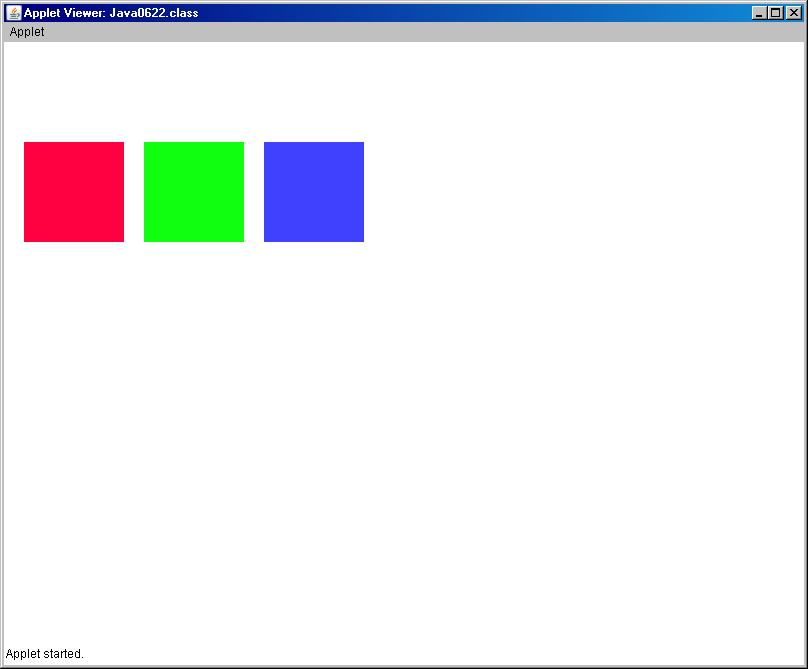
Do you see an object anywhere or more accurately do you see an object identifier in that statement? You are looking at a special type of object, called *anonymous object*. You see a situation where a parameter in a method requires an object. You can create an object name, but it is not necessary.

This *anonymous object* business may be quite confusing. It may help to look at **Java0622.java**, in figure 6.25. In that program actual named **Color** objects are created and used as parameters in the **setColor** method. Three different squares are displayed and each square has its own color, using a *named* object.

**Figure 6.25**

|  |
| --- |
| // Java0622.java  // This program draws three squares with user-defined <Color> objects.  import java.awt.\*;  import java.applet.\*;  public class Java0622 extends Applet  {  public void paint(Graphics g)  {  Color myRed = new Color(255,0,64);  Color myGreen = new Color(16,255,16);  Color myBlue = new Color(64,64,255);    g.setColor(myRed);  g.fillRect(20,100,100,100);  g.setColor(myGreen);  g.fillRect(140,100,100,100);  g.setColor(myBlue);  g.fillRect(260,100,100,100);  }  } |

**Figure 6.25 Continued**

****

|  |
| --- |
| **Anonymous Objects** |
| Situations exist where an object identifier is not necessary when creating a new object. The majority of objects have an object identifier like these two examples:  **Bank tom = new Bank(2500);**  **Color myRed = new Color(255,0,64);**  There are also objects that are used as parameters in a method call like:  **g.setColor(new Color(100,100,100));**  A new **Color** object is created, but not identified. Such objects are called *anonymous* object. |

**Drawing Polygons**

Java has some special features to draw polygons. The **Graphics** class provides two methods, **drawPolygon** and **fillPolygon**, but these methods only work with a **Polygon** object. Before you draw any polygon, you must first create an object of the **Polygon** class. The **Polygon** class has an **addPoint** method. The secret in drawing polygons is to use the following three steps:

1. Construct a **Polygon** object.

2. Add coordinates to the object using the **addPoint** method.

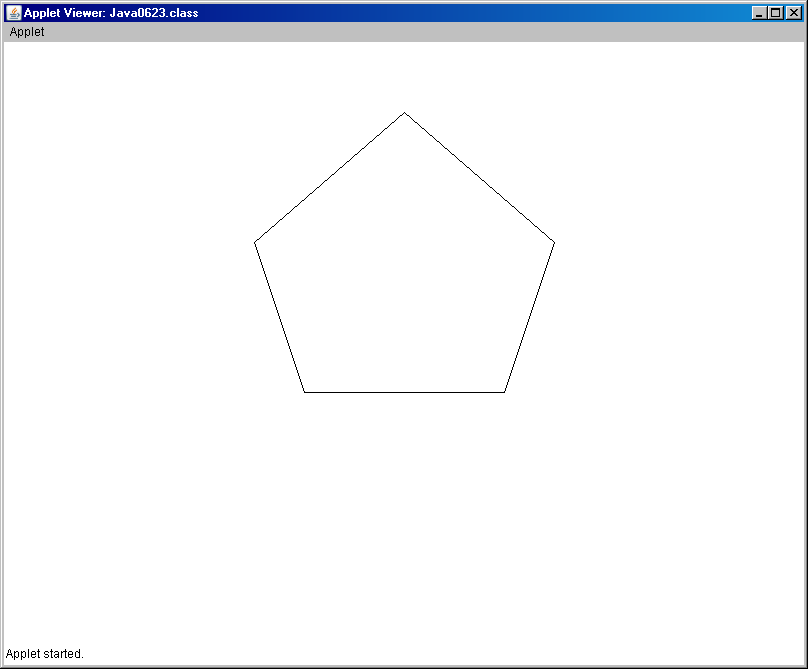
3. Draw the polygon with **drawPolygon**.

Program **Java0623.java**, in figure 6.26, draws a pentagon. The code starts by creating a **penta** object using the standard *object construction* approach with **new**. Then store five coordinates in the **penta** object with the **addPoint** method. You do need to be sure that you add the points in the proper sequence. Java needs to know how to connect the points and it uses the sequence that the coordinates are added to the **Polygon** object.

**Figure 6.26**

|  |
| --- |
| // Java0623.java  // This program draws a pentagon with the <drawPolygon> method.  // Methods <drawPolygon> and <fillPolygon> take a Polygon object as  // parameter. The polygon object stores a set of points. Points can  // be added to a Polygon object with the <addPoint> method.  import java.awt.\*;  import java.applet.\*;  public class Java0623 extends Applet  {  public void paint(Graphics g)  {  Polygon penta = new Polygon();  penta.addPoint(400,70);  penta.addPoint(550,200);  penta.addPoint(500,350);  penta.addPoint(300,350);  penta.addPoint(250,200);  g.drawPolygon(penta);  }  } |

**Figure 6.26 Continued**

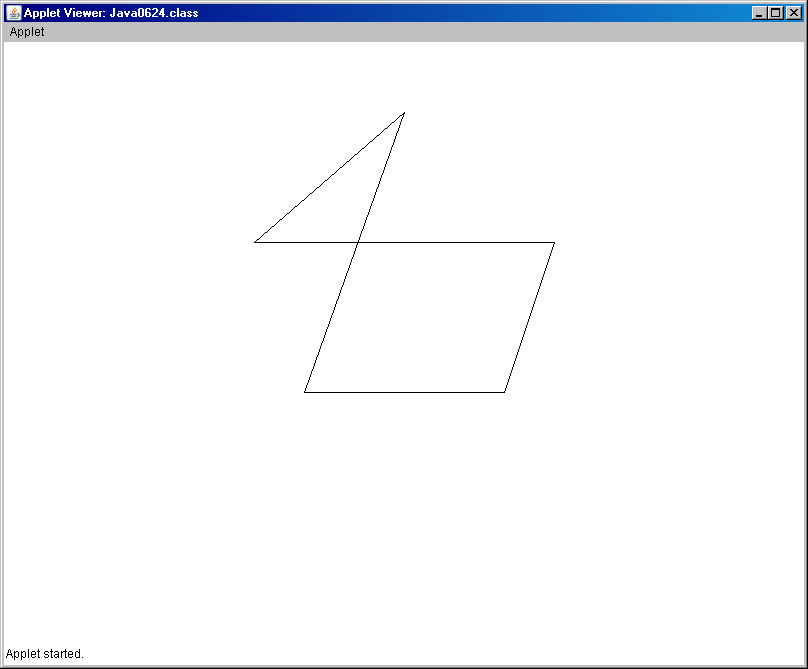
****

Program **Java0624.java**, in figure 6.27, drives home the point about the coordinate point sequence. This program is very similar to the previous program and the same exact coordinates are added to the **penta** object. The only difference is the sequence of the point addition. You will note that the pentagon looks very different now.

**Figure 6.27**

|  |
| --- |
| // Java0624.java  // This program demonstrates that the sequence of adding points is  // significant. The same coordinates of the previous program are used  // in the different sequence. The display is very different.  import java.awt.\*;  import java.applet.\*;  public class Java0624 extends Applet  {  public void paint(Graphics g)  {  Polygon penta = new Polygon();  penta.addPoint(300,350);  penta.addPoint(400,70);  penta.addPoint(250,200);  penta.addPoint(550,200);  penta.addPoint(500,350);  g.drawPolygon(penta);  }  } |

**Figure 6.27 Continued**



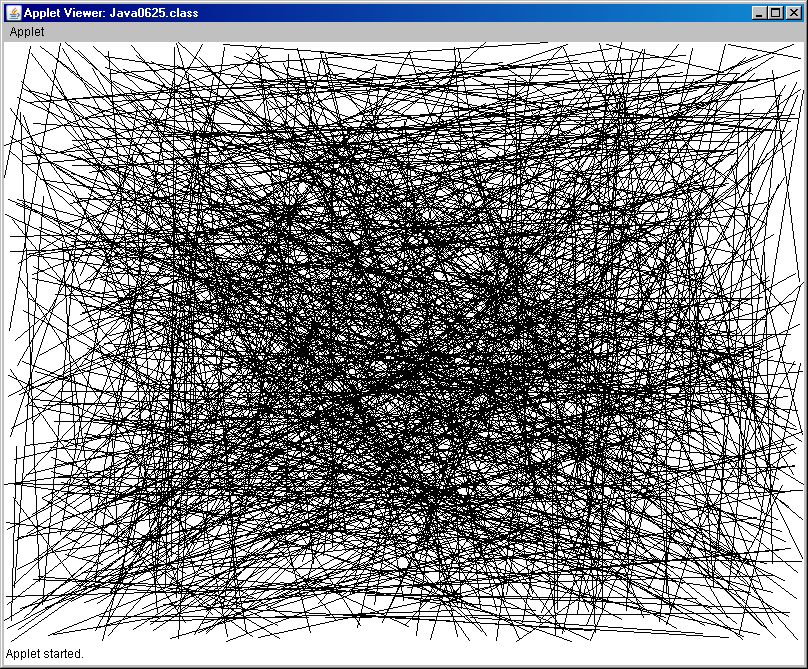
**6.8 Random Graphics Objects**

There are two Java concepts you have learned that combine very nicely. You have learned to create a variety of graphic images with methods of the **Graphics** and **Color** classes. In this chapter you have also learned how to generate random numbers of a specific range. When you combine these two features, you can create some interesting programs.

Program **Java0625.java**, in figure 6.28, generates random lines. These lines are all colored black. The program is meant for an applet window that is 800 by 600 pixels in size. You will see that the random numbers generate integers in the entire range of the applet window.

**Figure 6.28**

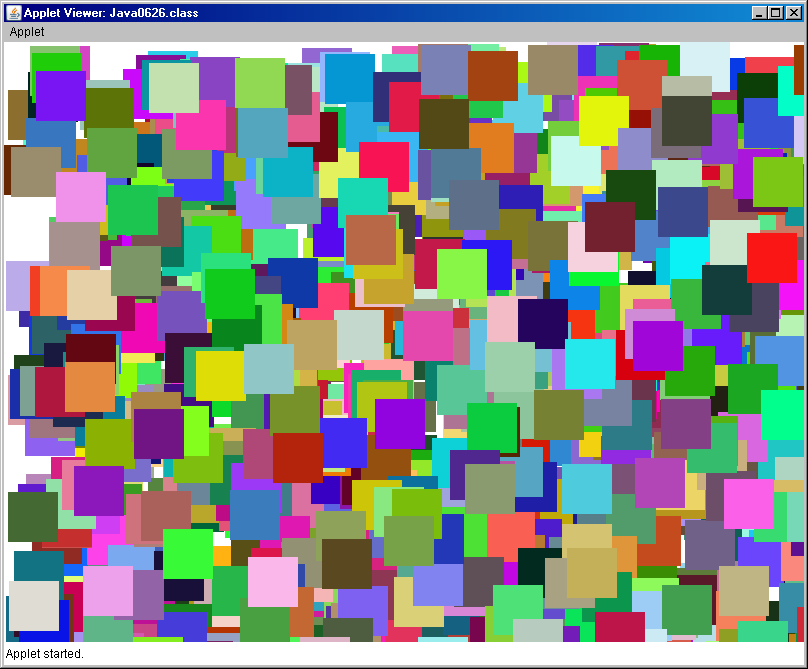
|  |
| --- |
| // Java0625.java  // This program displays 1000 random lines.  import java.awt.\*;  import java.applet.\*;  import java.util.Random;  public class Java0625 extends Applet  {  public void paint(Graphics g)  {  Random rndInt = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int x1 = rndInt.nextInt(800);  int y1 = rndInt.nextInt(600);  int x2 = rndInt.nextInt(800);  int y2 = rndInt.nextInt(600);  g.drawLine(x1,y1,x2,y2);  }  }  } |

****

The next program example in this chapter will combine using random values for the location of squares along with random color objects. Program **Java0626.java**, in figure 6.29, generates 1000 squares of a random color.

**Figure 6.29**

|  |
| --- |
| // Java0626.java  // This program combines random squares with random color objects.  import java.awt.\*;  import java.applet.\*;  import java.util.Random;  public class Java0626 extends Applet  {  public void paint(Graphics g)  {  Random rndInt = new Random(12345);  for (int k = 1; k <= 1000; k++)  {  int x = rndInt.nextInt(800);  int y = rndInt.nextInt(600);  int red = rndInt.nextInt(256);  int green = rndInt.nextInt(256);  int blue = rndInt.nextInt(256);  g.setColor(new Color(red,green,blue));  g.fillRect(x,y,50,50);  }  }  } |

****

**6.9 Using the Scanner Class**

In the previous chapter you learned how to enter data from the keyboard. Two points are made in that chapter. First, it is necessary to provide program input during program execution. Without program interaction, most programs are not very interesting. Second, it is possible to use certain program features without knowing specific details. In the last chapter you had not yet learned how to create or instantiate new objects of various classes. You used three **Scanner** methods without explanation about creating objects. At the end of this chapter you should now be more comfortable with the **new** operator and using keyboard input.

|  |
| --- |
| **Java 5.0 Alert** |
| The **Scanner** class is a feature that was released with JDK 5.0 (Java Development Kit) by Sun Micro Systems.  The **Scanner** class will not be recognized by any earlier editions of the Java language. |

Program **Java0627.java**, in figure 6.30, demonstrates how to use a **Scanner** object. First consider the statement that instantiates a **Scanner** object:

**Scanner input = new Scanner(System.in);**

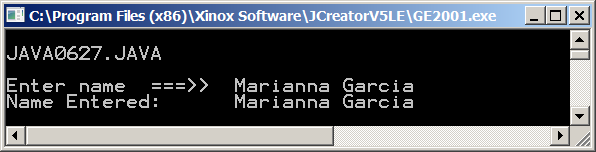
**Scanner** is the class. **input** is the object and may be any identifier. Normally, I have the habit to use identifiers **input** or **keyboard** for this purpose. The object is instantiated with the **System.in** parameter. You have used **System.out** since the first program example, which directs program output to the monitor. **System.in** receives program input from the keyboard. It is significant to specify that the input comes from the keyboard. Right now we are only using keyboard input, but in later chapters you will find that data input can also come from external files, which are stored on the hard drive. When information comes from a hard drive file, you will use a different parameter than **System.in**. The second statement, shown below, of concern actually transfers data from the keyboard to the variable **String** object **name**.

**String name = input.nextLine();**

The **nextLine** method is quite literal. It means that one entire line is entered at the keyboard until the **Enter** key is pressed. This makes **nextLine** a very practical method for entering characters of string information. String information not only includes alpha-numeric characters, but also spaces, punctuation and any other type of information, which can be entered with a keyboard.

**Figure 6.30**

|  |
| --- |
| // Java0627.java  // This demonstrates how to instantiate a <Scanner> object.  // The <input> object is used with the <nextLine> method to enter  // string values from the keyboard.  import java.util.Scanner;  public class Java0627  {  public static void main (String args[])  {  Scanner input = new Scanner(System.in);  System.out.println("\nJAVA0627.JAVA\n");  System.out.print("Enter name ===>> ");  String name = input.nextLine();  System.out.println("Name Entered: " + name);  System.out.println();  }  } |

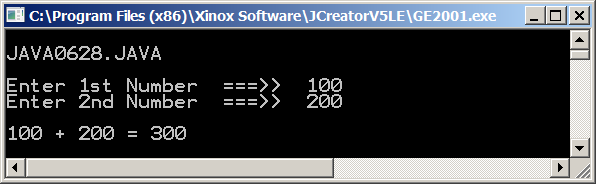


Program **Java0628.java**, in figure 6.31, instantiates the **Scanner** object **input** in the same manner as the previous program. This time the **nextInt** method is used to enter integer information from the keyboard.

**Figure 6.31**

|  |
| --- |
| // Java0628.java  // This program uses the <nextInt> method to enter integers from the keyboard.  // The arithmetic addition proves that the entered values are integers.  import java.util.Scanner;  public class Java0628  {  public static void main (String args[])  {  System.out.println("\nJAVA0628.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter 1st Number ===>> ");  int number1 = input.nextInt();  System.out.print("Enter 2nd Number ===>> "); int number2 = input.nextInt();  int sum = number1 + number2;  System.out.println();  System.out.println(number1 + " + " + number2 + " = " + sum);  System.out.println();  }  } |

**Figure 6.26 Continued**

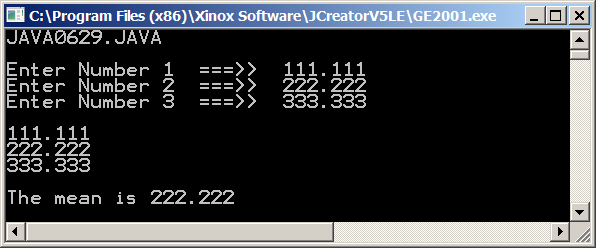


Program **Java0629.java**, in figure 6.32, instantiates the **Scanner** object **input** in the same manner as the two previous programs. This time the **nextDouble** method is used to enter double information from the keyboard.

**Figure 6.32**

|  |
| --- |
| // Java0629.java  // This program demonstrates how to use the <nextDouble> method for three separate  // double keyboard inputs, which are used to display the mean.  import java.util.Scanner;  public class Java0629  {  public static void main (String args[])  {  System.out.println("\nJAVA0629.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter Number 1 ===>> ");  double n1 = input.nextDouble();  System.out.print("Enter Number 2 ===>> ");  double n2 = input.nextDouble();  System.out.print("Enter Number 3 ===>> ");  double n3 = input.nextDouble();  System.out.println();  System.out.println(n1);  System.out.println(n2);  System.out.println(n3);  double mean = (n1+n2+n3)/3;  System.out.println();  System.out.println("The mean is " + mean);  System.out.println();  }  } |

**Figure 6.26 Continued**



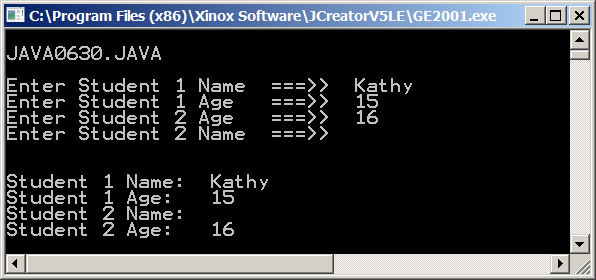
The **Scanner** class provides a convenient keyboard input. Just remember that this class is not available with Java JDK version earlier than **1.5.0** or **5.0** for short. This should not be a problem since starting in 2007 version **1.6.0** (or **6.0** for short) has become available for download.

We are not yet finished with this nifty new feature. There is a potential problem with using the **Scanner** class that you must be clearly understand, otherwise you will think that you entered the programming twilight zone. Program **Java0630.java**, in figure 6.33, enters student information from the keyboard. The first student enters the name first followed by the age. The second student reverses the order and enters age followed by name. When you execute this program, you will find that it is not possible to enter the second name. It appears that the program takes control and jumps ahead to the display without information for the second name.

**Figure 6.33**

|  |
| --- |
| // Java0630.java  // This program demonstrates an interesting problem with the <Scanner> class.  // It appears that order matters. Enter a string first, followed by a number, and  // everything is fine. In reverse order there is no opportunity to enter a name.  import java.util.Scanner;  public class Java0630  {  public static void main (String args[])  {  System.out.println("\nJAVA0630.JAVA\n");  Scanner input = new Scanner(System.in);  System.out.print("Enter Student 1 Name ===>> ");  String name1 = input.nextLine();  System.out.print("Enter Student 1 Age ===>> ");  int age1 = input.nextInt();    System.out.print("Enter Student 2 Age ===>> ");  int age2 = input.nextInt();  System.out.print("Enter Student 2 Name ===>> ");  String name2 = input.nextLine();  System.out.println("\n\n");  System.out.println("Student 1 Name: " + name1);  System.out.println("Student 1 Age: " + age1);  System.out.println("Student 2 Name: " + name2);  System.out.println("Student 2 Age: " + age2);  System.out.println();  }  } |

**Figure 6.33 continued**

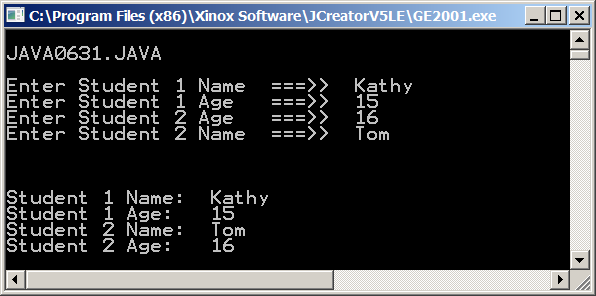


When characters are entered from the keyboard, they first go to a temporary memory location called the *keyboard buffer*. The manner in which methods take characters from the keyboard buffer is different for different **Scanner** methods.

The problem is caused by the behavior of the **nextLine** method. This method enters a string of characters until the *carriage-return/line-feed* (**crlf**)character is encountered. This character is also called the *end-of-line* character or the *Enter* key. This means that all characters, including the **crlf** character are removed from the keyboard buffer. Methods **nextInt** and **nextDouble** behave differently. These two methods are only interested in numerical digits and the **crlf** character is not a numerical digit. The result is that a **crlf** character is left in the keyboard buffer after using **nextInt** or **nextDouble**. Now there are no difficulties until you use the **nextLine** method. Method **nextLine** is meant to wait for keyboard input until the **crlf** character is encountered in the keyboard buffer. Result? The **crlf** character left by the preceding method call causes method **nextLine** to be finished before getting started. There is a solution for this problem. Look at program **Java0631.java**, in figure 6.34, and you notice a **dummy** variable. The mission of the **String** variable is to remove the leftover **crlf** character from the keyboard buffer and makes the following **nextLine** method call work correctly.

**Figure 6.34**

|  |
| --- |
| // Java0631.java  // This program cures the <Scanner> class problem with a "dummy" variable.  // The dummy variable removes the <crLf> character from the buffer.  import java.util.Scanner;  public class Java0631  {  public static void main (String args[])  {  System.out.println("\nJAVA0626.JAVA\n");  Scanner input = new Scanner(System.in);      System.out.print("Enter Student 1 Name ===>> ");  String name1 = input.nextLine();  System.out.print("Enter Student 1 Age ===>> ");  int age1 = input.nextInt();    System.out.print("Enter Student 2 Age ===>> ");  int age2 = input.nextInt();  String dummy = input.nextLine(); // <<<================ removes CRLF  System.out.print("Enter Student 2 Name ===>> ");  String name2 = input.nextLine();    System.out.println("\n\n");  System.out.println("Student 1 Name: " + name1);  System.out.println("Student 1 Age: " + age1);  System.out.println("Student 2 Name: " + name2);  System.out.println("Student 2 Age: " + age2);  System.out.println();  }  } |

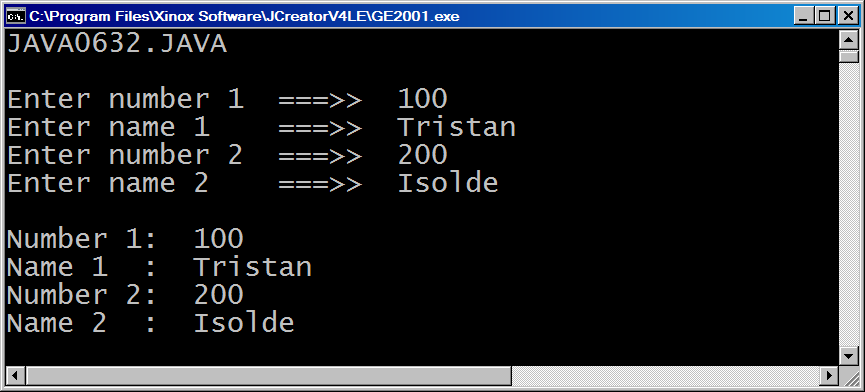


The "dummy" variable approach in the previous program works fine and it may help to explain the problem with the keyboard buffer. There is a simpler solution to this problem that may be preferred.

Program **Java0632.java**, in figure 6.35, shows the totally different approach. When you work with **Scanner** objects there are not problems as long as you only enter text values (**String** or **char**) or you only enter number values (**int** or **double**). Problems happens when the same program enters both. This program shows that another solution exists by creating two **Scanner** objects, one for numbers and one for text.

**Figure 6.35**

|  |
| --- |
| // Java0632.java  // This program example shows a second solution to the buffer problem.  // In this case two <Scanner> objects are constructed: One for number input and one for  // text input. It is not necessary to clear the buffer with a "dummy" variable.  import java.util.Scanner;  public class Java0632  {  public static void main (String args[])  {  System.out.println("\nJAVA0632.JAVA\n");  **Scanner numberInput = new Scanner(System.in);**  **Scanner textInput = new Scanner(System.in);**  System.out.print("Enter number 1 ===>> ");  int number1 = numberInput.nextInt();  System.out.print("Enter name 1 ===>> ");  String name1 = textInput.nextLine();  System.out.print("Enter number 2 ===>> ");  int number2 = numberInput.nextInt();  System.out.print("Enter name 2 ===>> ");  String name2 = textInput.nextLine();  System.out.println("\n\n");  System.out.println("Number 1: " + number1);  System.out.println("Name 1 : " + name1);  System.out.println("Number 2: " + number2);  System.out.println("Name 2 : " + name2);  System.out.println();  }  } |



|  |
| --- |
| **Scanner Class Notes** |
| With the release of Java JDK 5.0 the **Scanner** class is available for interactive, text-style, program input.  The **Scanner** class is part of the **util** package.  Keyboard input requires a **Scanner** object that is instantiated with the **System.in** parameter, like:  **Scanner keyboard = new Scanner(System.in)**  The **nextLine()** methodis used to enter **String** information.  The **nextInt()** methodis used to enter **int** information.  The **nextDouble()** methodis used to enter **double** information. |
| Methods **nextInt** and **nextDouble** do not remove the **crlf** character from the buffer, like the **nextLine** method. This means that a dummy **String** variable must be used like:  **String dummy = input.nextLine();**  There also exists an alternative solution.  Create two Scanner objects:  One for number input and one for text input. |

|  |
| --- |
| **AP Examination Alert** |
| Keyboard input is extremely important for efficient program testing that avoids hard-coding variable values.  However, the **Scanner** class is not tested on the AP Computer Science Examination. |

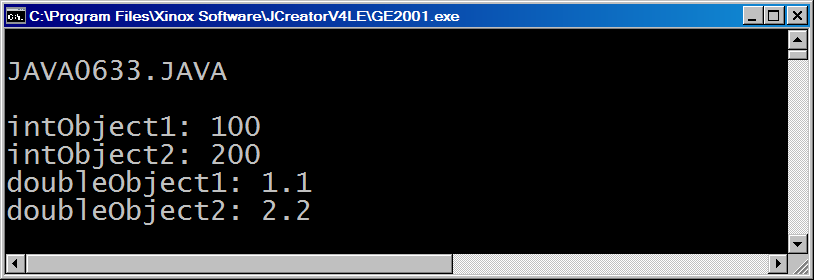
**6.10 Using the Integer Class**

Java has some special *wrapper* classes. A *wrapper* class creates objects capable of storing primitive data values like **int** and **double**. This feature is important, because many data type will be introduced that are only capable of storing objects. A wrapper class makes it possible to store primitive, simple data values when only objects can be stored.

Program **Java0633.java**, in figure 6.36, demonstrates how to store integer values and real number values in **Integer** and **Double** values. It is unlikely that you will get confused between the **int** simple data type and the **Integer** class. It is easier to mix up the **double** simple data type with lower-case **d** and the **Double** class with upper-case **D**. Methods **intValue** and **doubleValue** convert back to simple types.

**Figure 6.36**

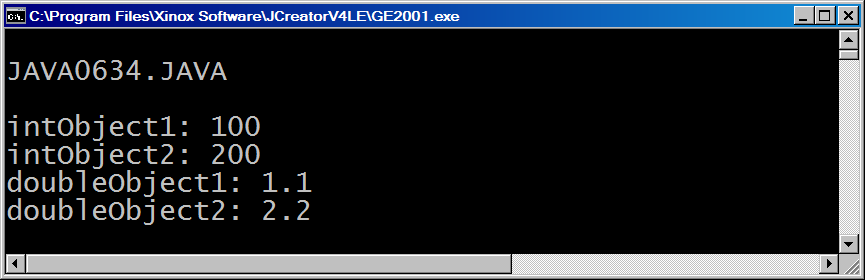
|  |
| --- |
| // Java0633.java  // This program introduces the <Integer> class.  // An <Integer> object is an "object" which can store an <int> value.  // The <Integer> class is used in this program as it was required  // in the Java versions before Java 5.0.  // This is also the manner in which AP Exam questions will appear.  import java.util.Scanner;  public class Java0633  {  public static void main (String args[])  {  System.out.println("\nJAVA0633.JAVA\n");  Integer intObject1 = new Integer(100);  Integer intObject2 = new Integer(200);  Double doubleObject1 = new Double(1.1);  Double doubleObject2 = new Double(2.2);    System.out.println("intObject1: " + intObject1.intValue());  System.out.println("intObject2: " + intObject2.intValue());  System.out.println("doubleObject1: " + doubleObject1.doubleValue());  System.out.println("doubleObject2: " + doubleObject2.doubleValue());  System.out.println();  }  } |



The wrapper classes became easier with the introduction of Java Version 5.0, as you will see in program **Java0634.java**, figure 6.37. The need to construct an **Integer** or **Double** object with **new Integer(100)** or **new Double(2.2)** is handled automatically with a feature called *auto-boxing*. This means that Java automatically *wraps* or *boxes* the simple data types. This also eliminates the need for **intValue** and **doubleValue**.

**Figure 6.37**

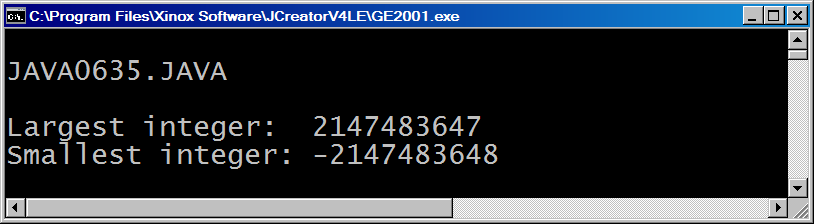
|  |
| --- |
| // Java0634.java  // This program demonstrates the exact same processing as the  // previous program. This processing is accomplished with less  // program code. These shortcut features came available with  // Java version 5.0 and later versions.  import java.util.Scanner;  public class Java0634  {  public static void main (String args[])  {  System.out.println("\nJAVA0634.JAVA\n");    Integer intObject1 = 100;  Integer intObject2 = 200;  Double doubleObject1 = 1.1;  Double doubleObject2 = 2.2;    System.out.println("intObject1: " + intObject1);  System.out.println("intObject2: " + intObject2);  System.out.println("doubleObject1: " + doubleObject1);  System.out.println("doubleObject2: " + doubleObject2);  System.out.println();  }  } |



The **Integer** class has two constant values that can be accessed called **Integer.MAX\_VALUE** and **Integer.MIN\_VALUE**, which are the largest and smallest integer values. The size is based on the storage available for an integer, which is four bytes. Program **Java0635.java**, in figure 6.38, shows how to use these Java constants. Note that capital letters are used as this is the common convention for constant values in programming languages.

**Figure 6.38**

|  |
| --- |
| // Java0635.java  // <int> values are stored in four bytes.  // This program uses the <Integer.MAX\_VALUE> and <Integer.MIN\_VALUE>  // constants to display the largest and smallest <int> values.  import java.util.Scanner;  public class Java0635  {  public static void main (String args[])  {  System.out.println("\nJAVA0635.JAVA\n");    System.out.println("Largest integer: " + Integer.MAX\_VALUE);  System.out.println("Smallest integer: " + Integer.MIN\_VALUE);  System.out.println();  }  } |

****

**6.11 Summary**

This chapter continued the Object Oriented Programming introduction started in the last chapter with class methods. This chapter continued with an introduction to using object methods.

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**.

There are **class methods** and there are **object methods**. This chapter is only concerned with using existing methods from existing classes, not creating any new classes and or new methods. The first methods, introduced in the last chapter, were the class methods of the **Math** class. Methods of a utilitarian style, like mathematical functions, are class methods and they are called by using the class identifier, like **Math.sqrt(16)**.

Most classes require that memory is allocated for each variable or object of the class that is constructed. The **Bank** class was used to demonstrate that there is need for one object to store account information for each customer of the **Bank** class. Objects are constructed with the **new** operator. The **Bank** class is a simple class that I created personally. It is not part of the large Java library of classes.

The **Random** class is part of the Java library and belongs with a group of other classes to the **util** package, which in turn belongs to the **java** package. Packages organize classes for easy access. You need to use the reserved word **import** to access a desired package. This is not necessary with the **java.lang.\*** package, which is automatically loaded by Java. The **Math** class belongs to the **java.lang** package. It is one of the most important Java packages.

The **Random** class contains methods, which generate random integers and random real numbers. It is possible to generate truly random numbers, which will be different with each program execution. It is also possible to construct **Random** objects with a starting *seed,* which will guarantee the same "random" sequence each time the program executes. Random numbers can also be generated with the **Math.random** method.

Output format can be controlled with methods of the **DecimalFormat** class. This class is part of the **java.text** package. With **DecimalFormat** and the **format** method it is possible to format the output of a number. You can pad zeroes in front and behind a number. It is also possible to insert commas and round off a real number to a desired number of digits beyond the decimal point.

This chapter also introduced some graphics classes. Polygons can be created with objects of the **Polygon** class and the **addPoint** method. The **drawPolygon** method or **fillPolygon** method of the **Graphics** class actually draws the polygon after the object is created.

Random objects can be combined with graphics objects to display objects at random location with random colors.

With the release of Java JDK 5.0 the **Scanner** class is available for interactive, text-style, program input. The **Scanner** class is part of the **util** package.

Keyboard input requires a **Scanner** object that is instantiated with the **System.in** parameter, like:

**Scanner keyboard = new Scanner(System.in)**

The **nextLine()** methodis used to enter **String** information.

The **nextInt()** methodis used to enter **int** information.

The **nextDouble()** methodis used to enter **double** information.

The keyboard entry method retrieves information from the keyboard buffer. Methods **nextInt** and **nextDouble** do not retrieve the carriage-return-life-feed character, like the **nextLine** method. Using a dummy **String** variable to remove the **crlf** character solves this problem.