**Chapter VII**

**Creating Class Methods**

**Chapter VII Topics**

7.1 Introduction

7.2 The Math Class Revisited

7.3 Modular Programming and

User-Created Methods

7.4 User-Declared Parameter Methods

7.5 Void Methods and Return Methods

7.6 Making a Utility Library Class

7.7 The Payroll Case Study

7.8 Creating a GridWorld Library

7.9 Observing Bug act Method

7.10 Creating new Bug Methods

7.11 Summary

**7.1 Introduction**

Chapter IV introduced a few Object Oriented Programming (OOP) concepts. In particular, emphasis was placed on *encapsulation*. You learned that an object is capable of containing both data, often called *attributes*, and action modules that can process data, called *methods*. The lion's share of the Chapter IV revolved around using methods of existing classes. You also learned the distinction between calling **class methods** with the *class* identifier and **object methods** with the *object* identifier. Class methods are normally utility methods that do not access changing data fields. The **Math** class is a good example of a class where data will not change. There are many methods in the **Math** class, but only two **data** fields, **PI** and **E**, which do not change. Methods in **Math** are class methods and must be accessed using the **Math** class identifier.

It is a different story when you have the need to use methods with different sets of data. A new and separate object needs to be constructed for each required variable. A class is a data type, which is capable of only storing information for one single occasion. This presents no problem for a utility class, which does not store any user provided data. However, most classes require variables for many different data storage situations. In the last chapter I showed you the **Bank** class. That is a good example of a class, which requires multiple objects, one for each customer of the **Bank** class.

In the statement **int num; int** is the *type* and **num** is the *variable*. Likewise in the statement **Bank tom;** the class **Bank** is the *type* and the object **tom** is the *variable*. There is a very important distinction between simple data types like **int**, **double**, **char**, and **boolean** and complex data types like **Math**, **Random** and **DecimalFormat**. Each of the simple data types stores only a single value. The variable objects of a class data type, on the other hand, can store many values. Additionally, class data types also contain methods, which can access data.

As you saw examples of class methods and object methods, you also learned that methods can have one or more parameters or arguments. Parameters provide information to methods for processing. Additionally, methods fall into two major categories, which are **return** methodsand **void** methods. Return methods return some requested value, like the **tom.getChecking();** method,which returns the checking account balance of the object **tom**. Void methods do not return any values, but frequently alter object data, like the **tom.changeChecking(2000.0);** method, which adds $2000.00 to the checking account balance of the **tom** object.

You were told that learning OOP will not happen in one section, one chapter or even a couple of chapters. It will happen throughout the entire course. In previous chapters you were introduced to some general concepts of Object Oriented Programming and then you learned how to use existing class methods and existing object methods. In this chapter you will learn how to write your own class methods and in the next chapter you will learn how to write your own object methods. You have already learned that there is a distinction in using or calling class methods and object methods. There is also a difference in writing class methods and object methods.

As your programs start to increase in size, it becomes important to consider some proper program design concepts. This chapter will introduce program design with the use of classes. It is not possible to create large, reliable programs without being very conscious of program design.

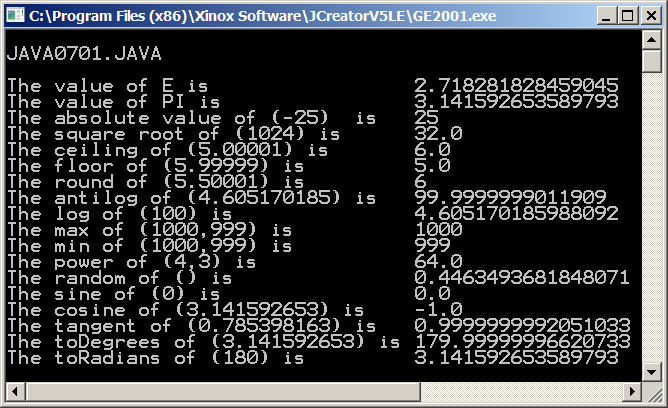
**7.2 The Math Class Revisited**

The **Math** class was used in the last chapter because students have familiarity with the methods of the **Math** class. Program **Java0701.java**, in figure 7.1, calls the majority of the **Math** methods in one program. This will review how to call class methods and it will also introduce various **Math** methods that were not shown in the earlier chapter. It is possible that some of these math functions are not familiar to you. You will learn about them in Algebra II and Pre-Calculus.

|  |
| --- |
| **Additional Math Methods (not shown in earlier chapter)** |
| Math.exp(p) returns the antilog of the p or ep  Math.log(p) returns the log (base e) of p  Math.sin(p) returns the trigonometric sine of p  Math.cos(p) returns the trigonometric cosine of p  Math.tan(p) returns the trigonometric tangent of p  Math.toDegrees(p) returns the number of degrees in p radians  Math.toRadians(p) returns the number of radians in p degrees |

**Figure 7.1**

|  |
| --- |
| // Java0701.java  // This program reviews using class methods and demonstrates most of the  // available <Math> class methods and data fields.  public class Java0701  {  public static void main (String args[])  {  System.out.println("\nJAVA0701.JAVA\n");  System.out.println("The value of E is " + Math.E);  System.out.println("The value of PI is " + Math.PI);  System.out.println("The absolute value of (-25) is " + Math.abs(-25));  System.out.println("The square root of (1024) is " + Math.sqrt(1024));  System.out.println("The ceiling of (5.00001) is " + Math.ceil(5.00001));  System.out.println("The floor of (5.99999) is " + Math.floor(5.99999));  System.out.println("The round of (5.50001) is " + Math.round(5.50001));  System.out.println("The antilog of (4.605170185) is " + Math.exp(4.605170185));  System.out.println("The log of (100) is " + Math.log(100));  System.out.println("The max of (1000,999) is " + Math.max(1000,999));  System.out.println("The min of (1000,999) is " + Math.min(1000,999));  System.out.println("The power of (4,3) is " + Math.pow(4,3));  System.out.println("The random of () is " + Math.random());  System.out.println("The sine of (0) is " + Math.sin(0));  System.out.println("The cosine of (3.141592653) is " + Math.cos(3.141592653));  System.out.println("The tangent of (0.785398163) is " + Math.tan(0.785398163));  System.out.println("The toDegrees of (3.141592653) is " + Math.toDegrees(3.141592653));  System.out.println("The toRadians of (180) is " + Math.toRadians(180));  System.out.println();  }  } |



**7.3 Modular Programming and**

**User-Created Methods**

Near the end of this chapter you will see an example of a very poorly written program, which is then slowly improved with many stages of program design. The title of this chapter is *Creating Class Methods*, but the creation of special modules, called *methods*, is motivated by program design. In other words, this chapter will introduce many introductory program design features.

One important program design feature is *modular programming*. It is possible to write a program with hundreds and even thousands of lines of program code that are all shoved into a single **main** or **paint** method. Such a program may work correctly, but making any fixes or changes to the program will be very difficult.

Imagine the following program. Your program draws a very beautiful horse. This horse requires 2500 program statements. Now suppose that you have a gorgeous horse, but the tail is wrong. Fixing the tail problem is difficult with many program statements bunched together. On the other hand, if you had created many small modules for each part of the horse, making changes is simple. If any group of related statements is placed in its own module, called a method in Java, then you have taken an important step in designing your program.

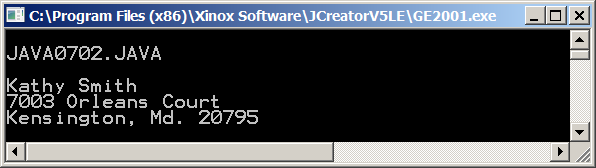
Another terrific benefit is that methods have a name or identifier. Perhaps the tail requires fifty lines of programming. Place each one of those statements into one module and call the module **drawTail**. Any future programming that needs to fix or improve the tail is now simpler. Look for the method called **drawTail** and you can get started without tedious searching through program statements. This business of combining common statements into their own modules is called *Modular Programming*. Years before *Object Oriented Programming* became the new hot programming style we used to say: *One task, one module*.

|  |
| --- |
| **Modular Programming** |
| Modular programming is the process of placing statements  that achieve a common purpose into its own module.  An old programming saying says it well  ***One Task, One Module*** |

Using modular programming will start with a very simple program that displays a mailing address. There is nothing complicated about this program and there is not much that demonstrates modular programming. Program **Java0702.java**, in figure 7.2, shows a program that will be used to teach the process of creating modules, called *methods*, in Java. You certainly have used classes and methods since Chapter 2, but all those methods already exist in Java. Now you will learn to create your very own classes and methods.

**Figure 7.2**

|  |
| --- |
| // Java0702.java  // This program displays a simple mailing address.  // It is used to demonstrate how to divide sections in  // the main method into multiple user-created methods.  public class Java0702  {    public static void main(String[] args)  {  System.out.println("\nJAVA0702.JAVA\n");  System.out.println("Kathy Smith");  System.out.println("7003 Orleans Court");  System.out.println("Kensington, Md. 20795");  System.out.println();  }  } |



It is nice to use other people’s tools. It can save time and make program writing much less tedious. You are ever so pleased that Java has cordially provided you with the **Math** class, **Random** class and **DecimalFormat** class and many other classes to make your programming life simpler. At the same time you should now start to try out your own wings. What if you want to create your very own class, and create your very own methods? Is that possible? Is that difficult? If it is possible and difficult is a matter of perspective. The classes shown in the next couple of program examples are not very impressive. You may wonder why I would bother create classes for the simplistic output that they create. That is fine. The mission is to learn OOP program design with a step-by-step approach and this introduction is more comfortable on the neurons than the avalanche approach.

Program **Java0703**.**java**, in figure 7.3, displays a rather unimpressive mailing address. The three **println** statements could all have been placed in the main method, as they were in the previous program. This time there are three additional program modules. The familiar **main** module is ever present and now there is also a module for **fullName**, **street** and **cityStateZip**. The three new modules appear very similar in syntax to the main module. They all start with **public static void** followed by a method identifier, like **main**.

Each module has opening and closing braces that contain a **println** statement. In the **main** method there are three statements with the **dot.method** notation. This class is not the earlier **Math** class or any other Java library class, but your very own class, **Java0703**.

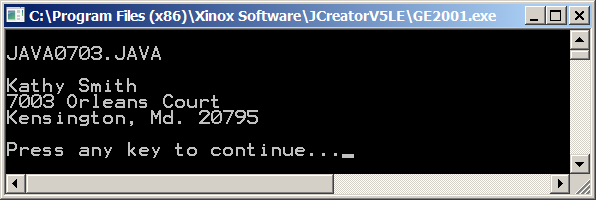
Please keep in mind that the next couple of program examples are designed to show the correct syntax for declaring a class with class methods. They are not good examples of Object Oriented Programming. Better OOP programs and more practical classes will be shown later in the chapter and later in the course.

Often students are curious why a certain new concept is introduced. There appears little justification for the new program feature. It is easy enough for an author to show a truly practical program that is now much simpler because of the new and improved program concept. Unfortunately, such practical programs tend to be very long and complex. The new idea, being introduced, is totally hidden in the complexity of hundreds of program statements.

**Figure 7.3**

|  |
| --- |
| // Java0703.java  // This program introduces user-created class methods.  // The three class methods are called with the same dot.method syntax  // as the methods of the Math class.  public class Java0703  {  public static void main(String[] args)  {  System.out.println("\nJAVA0702.JAVA\n");  Java0703.fullName();  Java0703.street();  Java0703.cityStateZip();  System.out.println();  }    public static void fullName()  {  System.out.println("Kathy Smith");  }  public static void street()  {  System.out.println("7003 Orleans Court");  }      public static void cityStateZip()  {  System.out.println("Kensington, Md. 20795");  }  } |

**Figure 7.3 Continued**



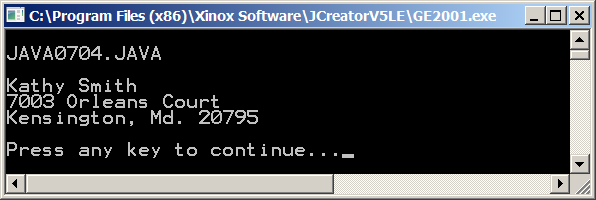
All your previous program examples always had a class. This single class has the same name as the program file. This class name, like **Java0703.fullName**, is used to call the newly created methods. Java requires that programming is done with classes and an application program requires a **main** method, placed inside a class. Now we have been shoving all the program statements in the main method and that can get crowded and very unreadable. It is much nicer to break up a program into manageable modules. Each module uses the same format as you have been using for the main method.

Now how about a little surprise? Program **Java0704.java**, in figure 7.4, is almost identical to the previous program, but now the class identifier is totally ignored and the method identifiers are called without concern about any class identifier, object identifier or anything else. How can that also be correct?

**Figure 7.4**

|  |
| --- |
| // Java0704.java  // This program example displays the same output as the previous program.  // This time the methods are called directly without using the class identifier.  // Omitting the class identifier is possible because all the methods are  // encapsulated in the same class, Java0704.  public class Java0704  {    public static void main(String[] args)  {  System.out.println("\nJAVA0704.JAVA\n");  fullName();  street();  cityStateZip();  System.out.println();  }    public static void fullName()  {  System.out.println("Kathy Smith");  }  public static void street()  {  System.out.println("7003 Orleans Court");  }  public static void cityStateZip()  {  System.out.println("Kensington, Md. 20795");  }    } |

**Figure 7.4 Continued**



There is Kathy Smith again and her address seems to indicate that the program compiled without any difficulties. So why bother with class identifiers when they appear to be extra baggage? It turns out that it is extra baggage in this particular example. The three new methods created are all members of the **Java0704** class.

It is not necessary to state the name when you are already in the same class. Consider this analogy. A letter needs to be given to Tom Jones, who is in room D116 of Royse City High School during third period. If this letter is delivered in the school’s office, some office aid is told to bring the letter to Tom Jones in room D116. That makes sense. Now suppose that I have a letter on my desk for Tom Jones and I am in room D116. I hand the letter to a student next to my desk and tell the student to give it to Tom Jones. I do not bother to add that Tom Jones is in room D116. I am in room D116. The student delivering the letter is in room D116 and Tom Jones is in room D116. It is not necessary to add the room location information to make the delivery possible.

|  |
| --- |
| **Using the Class Identifier** |
| Use of the class identifier is *optional* if a method is called from  a program statement in another method, which resides in the same class as the method being called.  Use of the class identifier is *required*if a method is called in  a program statement that resides outside the class of the  method that is being called. |

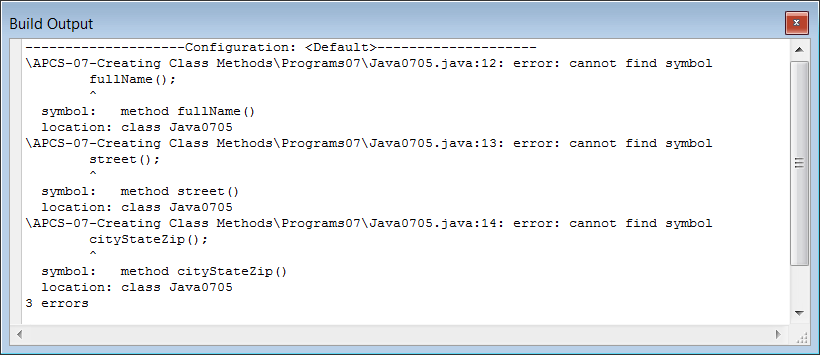
Proof about the class identifier statement made in figure 7.4 is provided with program **Java0705.java**, in figure 7.5. That program declares a second class, called **Address**. The methods of **fullName**, **street** and **cityStateZip** are declared as members of the **Address** class. These same methods are called, as before, from the **main** method of the **Java0705** class. This time **fullName**, **street** and **cityStateZip** are no longer members of the **Java0705** class and the program will not compile.

You have not seen a second class declaration in any previous program. Declaring a second class is not a problem. The syntax of a second class is almost identical to the primary class with one important distinction. A second and third class, placed in the same file, should not be declared as **public**. Only the primary class with the same name as the file is public. **Java0705.java**, in figure 7.5, does not compile because it does not know what to do with the method calls. Look at the many error messages to realize how confused the compiler is.

**Figure 7.5**

|  |
| --- |
| // Java0705.java  // This program demonstrates how to use a second class separate from the  // main program class. This program will not compile because the Name,  // Street and CityStateZip methods are no longer encapsulated in Java0705.  public class Java0705  {  public static void main(String args[])  {  System.out.println("\nJAVA0705.JAVA\n");  fullName();  street();  cityStateZip();  System.out.println();  }  }  class Address  {  public static void fullName()  {  System.out.println("Kathy Smith");  }  public static void street()  {  System.out.println("7003 Orleans Court");  }  public static void cityStateZip()  {  System.out.println("Kensington, Md. 20795");  }  } |

**Figure 7.5 Continued**

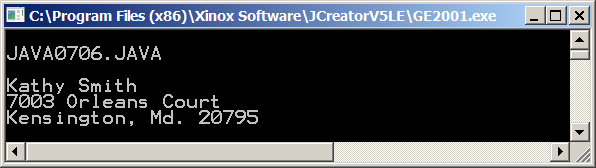


Program **Java0706.java**, in figure 7.6, solves the problem of the previous program by using the **Address** class identifier. More importantly, it demonstrates that you can really have multiple classes in one program. As your program grows in complexity, you will learn that it is customary to have only one class for one program file. Right now it is simple to demonstrate new concepts when all the concepts are in the same program file.

**Figure 7.6**

|  |
| --- |
| // Java0706.java  // The problem of Java0705.java is now fixed. It is possible to declare  // multiple classes in one program. However, you must use the dot.method  // syntax to call any of the <Address> class methods.  public class Java0706  {  public static void main(String args[])  {  System.out.println("\nJAVA0706.JAVA\n");  Address.fullName();  Address.street();  Address.cityStateZip();  System.out.println();  }  }  class Address  {  public static void fullName()  {  System.out.println("Kathy Smith");  }  public static void street()  {  System.out.println("7003 Orleans Court");  }  public static void cityStateZip()  {  System.out.println("Kensington, Md. 20795");  }  } |

**Figure 7.6 Continued**

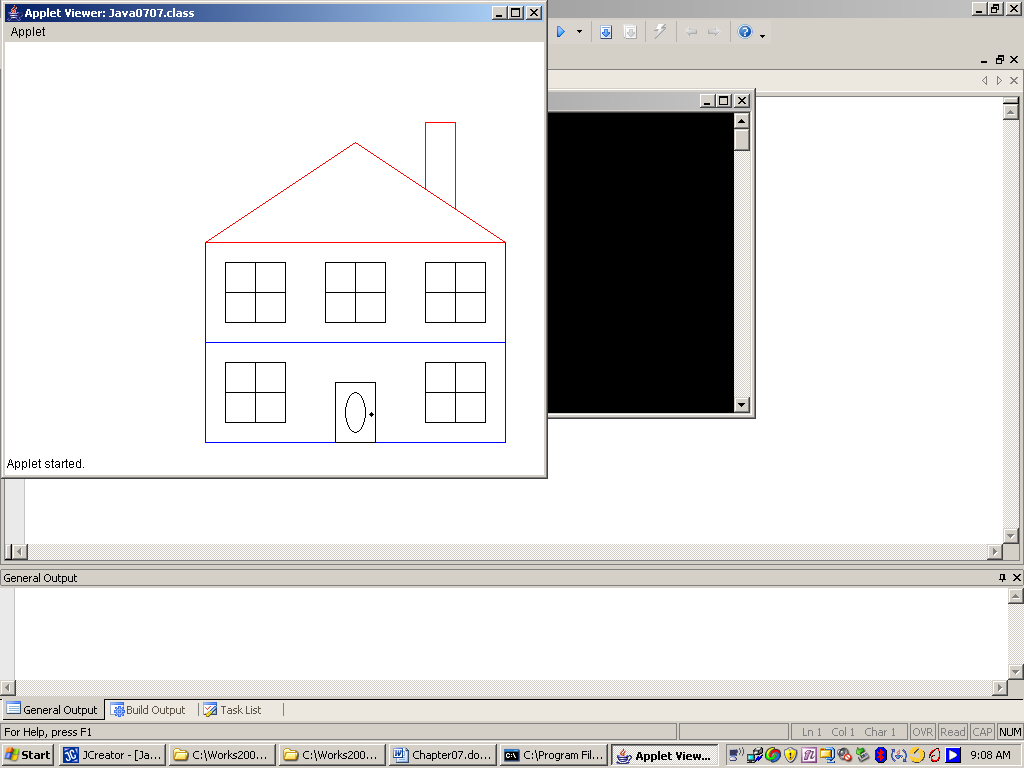


The batch of *Kathy-Smith-address-mailing-label-programs* do little to justify the creation of additional methods. The previous set of four programs was provided to demonstrate the proper syntax used to create your own methods. You will now see a repeat of the same type of methods done graphically. This time the program will be longer so that the benefits of modular programming are more apparent. Program **Java0707.java**, in figure 7.7, shows a house drawn in an applet. All the statements to draw this house are placed in the **paint** method.

When you see the program you may argue that the use of comments would be beneficial to help identify the different program segments. I have intentionally left out the comments to help demonstrate that a bunch of program statements placed inside a single module can be pretty confusion. Hopefully, this will motivate the reason for using modular programming.

**Figure 7.7**

|  |
| --- |
| // Java0707.java  // This program draws a house by placing all the necessary program statements in the <paint> method.  import java.awt.\*;  import java.applet.\*;  public class Java0707 extends Applet  {  public void paint(Graphics g)  {  g.setColor(Color.blue);  g.drawRect(200,200,300,100);  g.drawRect(200,300,300,100);  g.setColor(Color.red);  g.drawLine(200,200,350,100);  g.drawLine(500,200,350,100);  g.drawLine(200,200,500,200);  g.setColor(Color.red);  g.drawLine(420,146,420,80);  g.drawLine(420,80,450,80);  g.drawLine(450,80,450,166);  g.setColor(Color.black);  g.drawRect(330,340,40,60);  g.drawOval(340,350,20,40);  g.fillOval(364,370,5,5);  g.setColor(Color.black);  g.drawRect(220,220,60,60);  g.drawLine(220,250,280,250);  g.drawLine(250,220,250,280);  g.drawRect(420,220,60,60);  g.drawLine(420,250,480,250);  g.drawLine(450,220,450,280);  g.drawRect(320,220,60,60);  g.drawLine(320,250,380,250);  g.drawLine(350,220,350,280);  g.drawRect(220,320,60,60);  g.drawLine(220,350,280,350);  g.drawLine(250,320,250,380);  g.drawRect(420,320,60,60);  g.drawLine(420,350,480,350);  g.drawLine(450,320,450,380);  }  } |



I hope you understand that the house displayed in figure 7.7 can easily be made far more complex with hundreds of program statements. Just imagine if the house includes actual bricks, bushes, plants in the windows, numbers on the door, smoke coming from the chimney, Christmas lights, and kids playing in the yard. In no time a project as the one I described requires more than 1000 program statements. Placing all those statements in one single **paint** method is very poor program design. Making any changes becomes very tedious.

Sometimes when students work on a program that contains too many lines in one module, they argue that they know the purpose of each program statement. This is quite true when the program writing is fresh in your mind. It is a different story when time goes by. I mentioned in an earlier chapter that I wrote an Academic Decathlon program of about 12,000 lines. I was in a hurry to meet a deadline. The program lacked proper modular program design and comments were pretty much non-existent. In the heat of writing the program day after day, I stayed on top of everything. I received a wake-up call two years later. It was a program for Academic Decathlon data processing and there were major changes. I was horrified to realize how little I remembered and altering the program two years later was a nightmare.

Program **Java0708.java**, in figure 7.9, presents the exact same house. This time the program statements in the **paint** method are removed and divided into five methods. The **paint** method now looks very organized with a sequence of five method calls shown in figure 7.8.

**Figure 7.8**

|  |
| --- |
| **public void paint(Graphics g)**  **{**  **drawFloors(g);**  **drawRoof(g);**  **drawChimney(g);**  **drawDoor(g);**  **drawWindows(g);**  **}** |

There is a second reason why the house program is shown. Yes, it is more complex and hopefully it motivates the use of modular programming, but there is a special applet issue. The **paint** method controls the graphics display in the same manner that the **main** method controls the sequence in an application program. Output to the monitor requires the use of a **Graphics** object. You have pretty much seen the continuous use of **g** for the **Graphics** object. If you select to divide the graphics output into multiple modules make sure that you pass the **Graphics** object to the other modules.

**Figure 7.9**

|  |
| --- |
| // Java0708.java  // This program divides all the statements of <paint> in the previous program into five separate methods.  import java.awt.\*;  import java.applet.\*;  public class Java0708 extends Applet  {  public void paint(Graphics g)  {  drawFloors(g);  drawRoof(g);  drawChimney(g);  drawDoor(g);  drawWindows(g);  }    public static void drawFloors(Graphics g)  {  g.setColor(Color.blue);  g.drawRect(200,200,300,100);  g.drawRect(200,300,300,100);  }    public static void drawRoof(Graphics g)  {  g.setColor(Color.red);  g.drawLine(200,200,350,100);  g.drawLine(500,200,350,100);  g.drawLine(200,200,500,200);  }    public static void drawChimney(Graphics g)  {  g.setColor(Color.red);  g.drawLine(420,146,420,80);  g.drawLine(420,80,450,80);  g.drawLine(450,80,450,166);  }    public static void drawDoor(Graphics g)  {  g.setColor(Color.black);  g.drawRect(330,340,40,60);  g.drawOval(340,350,20,40);  g.fillOval(364,370,5,5);  }    public static void drawWindows(Graphics g)  {  g.setColor(Color.black);  g.drawRect(220,220,60,60);  g.drawLine(220,250,280,250);  g.drawLine(250,220,250,280);  g.drawRect(420,220,60,60);  g.drawLine(420,250,480,250);  g.drawLine(450,220,450,280);  g.drawRect(320,220,60,60);  g.drawLine(320,250,380,250);  g.drawLine(350,220,350,280);  g.drawRect(220,320,60,60);  g.drawLine(220,350,280,350);  g.drawLine(250,320,250,380);  g.drawRect(420,320,60,60);  g.drawLine(420,350,480,350);  g.drawLine(450,320,450,380);  }  } |

Removing statements from the **main** method or the **paint** method and placing common purpose statements in separate modules is good. It is good in the sense that modular programming is used. Object Oriented Design is not satisfied to place common statements in a module. We must continue and improve by placing common methods into a class. Program **Java0709.java**, in figure 7.10, takes the five house draw methods and places them all inside a **House** class. Take note that calling these five methods must now be preceded by using the **House** identifier.

**Figure 7.10**

|  |
| --- |
| // Java0709.java  // This program uses the better program design of creating a separate <House> class,  // which contains the five methods to draw the complete hour program.  import java.awt.\*;  import java.applet.\*;  public class Java0709 extends Applet  {  public void paint(Graphics g)  {  House.drawFloors(g);  House.drawRoof(g);  House.drawChimney(g);  House.drawDoor(g);  House.drawWindows(g);  }  }  class House  {    public static void drawFloors(Graphics g)  {  g.setColor(Color.blue);  g.drawRect(200,200,300,100);  g.drawRect(200,300,300,100);  }    public static void drawRoof(Graphics g)  {  g.setColor(Color.red);  g.drawLine(200,200,350,100);  g.drawLine(500,200,350,100);  g.drawLine(200,200,500,200);  }    public static void drawChimney(Graphics g)  {  g.setColor(Color.red);  g.drawLine(420,146,420,80);  g.drawLine(420,80,450,80);  g.drawLine(450,80,450,166);  }    public static void drawDoor(Graphics g)  {  g.setColor(Color.black);  g.drawRect(330,340,40,60);  g.drawOval(340,350,20,40);  g.fillOval(364,370,5,5);  }      public static void drawWindows(Graphics g)  {  g.setColor(Color.black);  g.drawRect(220,220,60,60);  g.drawLine(220,250,280,250);  g.drawLine(250,220,250,280);  g.drawRect(420,220,60,60);  g.drawLine(420,250,480,250);  g.drawLine(450,220,450,280);  g.drawRect(320,220,60,60);  g.drawLine(320,250,380,250);  g.drawLine(350,220,350,280);  g.drawRect(220,320,60,60);  g.drawLine(220,350,280,350);  g.drawLine(250,320,250,380);  g.drawRect(420,320,60,60);  g.drawLine(420,350,480,350);  g.drawLine(450,320,450,380);  }    } |

|  |
| --- |
| **Some Program Design Notes** |
| Programs should not be written by placing all the program statements in the main or paint methods.  Program statements that perform a specific purpose should be placed inside their own modules. This follows the *one-task, one-module* principle of earlier program design principles.  Object Oriented Design continues by placing modules of a common nature into a separate class.  In this chapter you are learning how to create class methods. The distinction between creating class methods and object methods will become clear in the next chapter. |

**7.4 User-Declared Parameter Methods**

This chapter started by showing many of the methods available in the **Math** class. The majority of those methods used arguments or parameters to perform the desired computation. You do know that some methods do not require a parameter, like **System.out.println();** and **int** **randomInt = rnd.nextInt();** but many methods require information for processing.

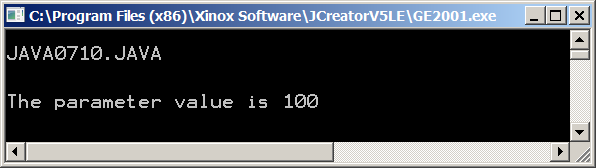
|  |
| --- |
| **Methods Calls With and Without Parameters** |
| **Parameter method examples:**  **double result1 = Math.sqrt(100);**  **double result2 = Math.pow(2,5);**  **g.drawLine(100,200,300,400);**  **g.fillPolygon(star);**  **Non-Parameter method examples:**  **Bug barry = new Bug( );**  **barry.move( );**  **barry.turn( );**  **Overloaded method examples:**  **System.out.println(“Hello World”);**  **System.out.println( );** |

Why is it that so many methods require parameters? This is very natural because methods perform some type of task. In most cases the task requires the processing of some type of data. There certainly are situations where the data to be processed already belongs to the class. In such situations, parameters are not necessary. There are also plenty of processes where external information needs to be processed and such information arrives to the method by parameter.

Program **Java0710.java**, in figure 7.11, has a small method. The purpose of the method is to display the information that is passed to the method. Appropriately, the method is called **displayParameter**. The key difference between creating no-parameter methods and parameter methods is the parameter declaration. All method declarations have an identifier followed by parentheses. If no information is required for the method, the parentheses stay empty. If information is required than the method heading lists one or more parameters inside the parentheses. Make sure that you include the data type of your parameter.

**Figure 7.11**

|  |
| --- |
| // Java0710.java  // This program introduces user-defined methods with parameters.  // The purpose of using parameters may be hard to tell, but at this  // stage concentrate on the mechanics and the manner in which information  // is passed from one program module to another program module.  public class Java0710  {  public static void main(String args[])  {  System.out.println("\nJAVA0710.JAVA\n");  displayParameter(100);  System.out.println();  }    public static void displayParameter(int number)  {  System.out.println();  System.out.println("The parameter value is " + number);  System.out.println();  }  } |



Before moving on we need to get used to some parameter vocabulary. There are two parameters, which work together to pass information to a method. The vocabulary for the calling parameter is *actual parameter* and the receiving parameter in the method heading is called the *formal parameter*.

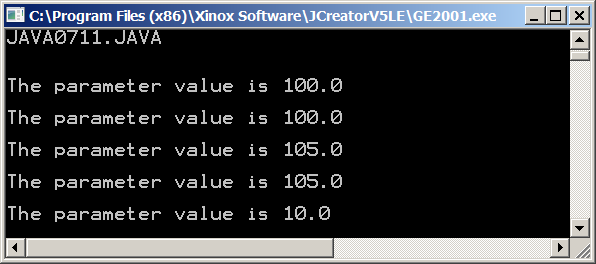
|  |
| --- |
| **Actual Parameters and Formal Parameters** |
| **showSum(10,15); // line 1**    **public static void showSum(int n1, int n2) // line 2**  **{**  **System.out.println(n1 + n2);**  **}**  The parameter in line 1 is called the *actual parameter*.  The parameter in line 2 is called the *formal parameter*. |

The formal parameter in the method heading indicates the data type that must be used by the actual parameter. The previous program example used a constant integer to pass the information. There are many ways to pass information and program **Java0711.java**, in figure 7.12, shows that the actual parameter can take five different formats. It is possible to use a constant only, a variable only, an expression with constants only, an expression with a variable and a constant, and a method call that returns the appropriate value.

**Figure 7.12**

|  |
| --- |
| // Java0711.java  // This program demonstrates that the calling parameter can be:  // a constant, like 100.  // a variable, like value  // an expression with only constants, like 10 + 5.  // an expression with a variable and a constant like value + 5.  // A call to a method, which returns a value, like Math.sqrt(100).  public class Java0711  {  public static void main(String args[])  {  System.out.println("\nJAVA0711.JAVA\n");  double value = 100;  displayParameter(100);  displayParameter(value);  displayParameter(100 + 5);  displayParameter(value + 5);  displayParameter(Math.sqrt(100));  System.out.println();  }    public static void displayParameter(double number)  {  System.out.println();  System.out.println("The parameter value is " + number);  }  } |

**Figure 7.12 Continued**

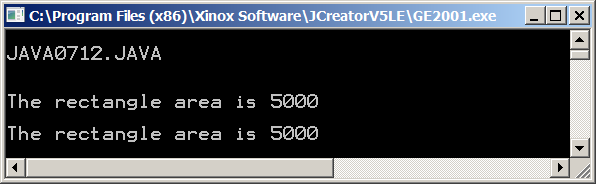


You know from using a variety of methods that some methods use more than one parameter. There is not much to worry about with two parameters. If a method wants two pieces of information, then by all means provide two values. Program **Java0712.java**, in figure 7.13, demonstrates a **showArea** method. Note also that the **showArea** method is intentionally called twice. In this particular example, the order of the actual parameters makes no difference. In either order the output of the **showArea** method is identical.

**Figure 7.13**

|  |
| --- |
| // Java0712.java  // This program demonstrates passing two parameters to a method.  // The <showArea> method is called twice. In this case reversing  // the sequence of the parameters is not a problem.  public class Java0712  {    public static void main(String args[])  {  System.out.println("\nJAVA0712.JAVA\n");  int width = 100;  int height = 50;  showArea(width,height);  showArea(height, width);  System.out.println();  }    public static void showArea(int w, int h )  {  System.out.println();  int area = w \* h;  System.out.println("The rectangle area is " + area);  }    } |

**Figure 7.13 Continued**

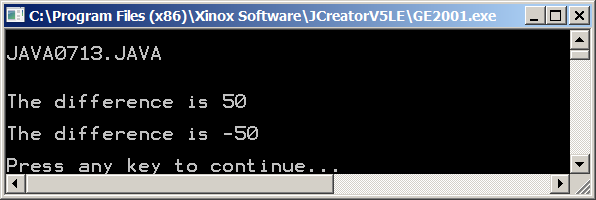


Do not start thinking that parameter sequence does not matter. Program **Java0713.java**, in figure 7.14, will prove otherwise. This time a **showDifference** method is used. You will now see that reversing the actual parameters can very much create a different method result.

**Figure 7.14**

|  |
| --- |
| // Java0713.java  // This program demonstrates that parameter sequence matters.  // In this example method <showDifference> will display different  // results when the calling parameters are reversed.  public class Java0713  {    public static void main(String args[])  {  System.out.println("\nJAVA0713.JAVA\n");  int num1 = 100;  int num2 = 50;  showDifference(num1,num2);  showDifference(num2,num1);  System.out.println();  }    public static void showDifference(int a, int b )  {  System.out.println();  int difference = a - b;  System.out.println("The difference is " + difference);  }    } |

**Figure 7.14 Continued**



|  |
| --- |
| **Actual Parameter Sequence Matters** |
| **The first actual parameter passes information to the first formal parameter.**  **The second actual parameter passes information to the second formal parameter.**  **Parameters placed out of sequence may result in compile errors or logic errors.** |

Parameters are tricky critters and students make a bunch of mistakes with parameters when they are first introduced. Program **Java0714.java**, in figure 7.15, demonstrates two common errors. Both errors will result in compile errors. In **line 1** two actual parameters **num1** and **num2** are declared inside the parentheses of the method call. That will not work. In **line 2** there are two formal parameters, which appear to be declared as **int**. This also does not compile. Formal parameters each need their own data type.

**Figure 7.15**

|  |
| --- |
| // Java0714.java  // This program demonstrates a common mistake made by students.  // Parameters are declared in the method heading, but may not be  // declared in the method call. This program will not compile.  public class Java0714  {    public static void main(String args[])  {  System.out.println("\nJAVA0714.JAVA\n");  showDifference(int num1, int num2); // line 1  System.out.println();  }    public static void showDifference(int a, b) // line 2  {  System.out.println();  int difference = a - b;  System.out.println("The difference is " + difference);  System.out.println();  }    } |

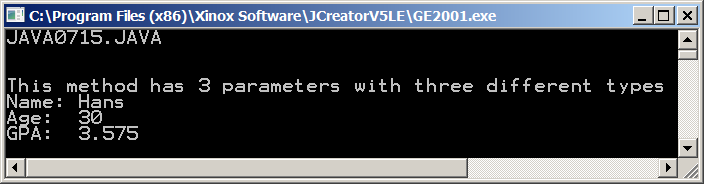
|  |  |
| --- | --- |
| **Common Parameter Mistakes** | |
| **Wrong** | **Correct** |
| **qwerty(int num1, int num2);** | **int num1 = 100;**  **int num2 = 200;**  **qwerty(num1,num2);** |
| **public static void qwerty(int a, b);** | **public static void qwerty(int a, int b)** |

You have seen several programs using methods with multiple parameters. In each case the multiple parameters were all the same data type. This is not a requirement. Program **Java0715.java**, in figure 7.16, demonstrates a **multiTypeDemo** method using three different parameter data types.

**Figure 7.16**

|  |
| --- |
| // Java0715.java  // This program demonstrates that multiple parameters may be different data types.  public class Java0715  {  public static void main(String args[])  {  System.out.println("\nJAVA0715.JAVA\n");  multiTypeDemo("Hans",30,3.575); // three different type parameters method call  System.out.println();  }  public static void multiTypeDemo(String studentName, int studentAge, double studentGPA)  {  System.out.println("\nThis method has 3 parameters with three different types");  System.out.println("Name: " + studentName);  System.out.println("Age: " + studentAge);  System.out.println("GPA: " + studentGPA);  }  } |

**Figure 7.16 Continued**



You can expect to use parameters in all your future programs. I have helped hundreds of students in FORTRAN, BASIC, Pascal, C++ and Java with their program problems. Mistakes that arise from incorrect parameter use certainly are at the top of the list for programs that do not compile or work logically.

This section will finish with a track analogy. Look at the diagrams carefully and you will find that they illustrate the correct way to use parameters. Parameters will simplify your life. Your programs will be better designed. Your programs will be much shorter by using the same method for multiple situations. At the same time, your programs can be become headaches with incorrect parameters.

**The Track Relay Analogy**

Let us summarize this parameter business, with a real life analogy that may help some students. The analogy that follows explains some of the parameter rules in a totally different manner. Imagine that you are at a track meet and you are watching a relay race. In this relay race the starters run 100 meters and then pass a baton to the next runner in their team.

In the first relay race example, the race official checks the track and announces that the race is not ready. A look at **Race-1** shows there are four starters ready in their lanes, but only three runners at the 100 meter *baton passing* mark. A runner from the Netherlands (NL) is missing.

**Race-1**

|  |  |
| --- | --- |
| **US** | **US** |
| **GB** | **GB** |
| **FR** | **FR** |
| **NL** |  |

**Race-2** presents another situation with a different problem. This time the number of runners is correct. There are four starters and there are also four runners at the 100 meter mark ready to receive a baton. However two runners at the 100 meter mark are standing in the wrong lane. The track official announces that the race cannot start unless the runners change lanes and are ready to receive the batons from their own countrymen.

**Race-2**

|  |  |
| --- | --- |
| **US** | **US** |
| **GB** | **GB** |
| **FR** | **NL** |
| **NL** | **FR** |

**Race3** is not a problem situation. This race demonstrates an analogy to help explain the naming of parameters. In **Race3**,runner *John* starts for the United States (US) and passes to *Greg.* *George* starts for Great Britain (GB) and passes to *Thomas*. *Gerard* starts for France (FR) and passes to *Louis*. Finally, *Hans* starts for the Netherlands and passes to another *Hans*.

**Race-3**

|  |  |
| --- | --- |
| **US (John)** | **US (Greg)** |
| **GB (George)** | **GB (Thomas)** |
| **FR (Gerald)** | **FR (Louis)** |
| **NL (Hans)** | **NL (Hans)** |

The point of this analogy is that the parameter names do not matter. What matters is that there are the same number of runners at the passing mark, as there are runners in the starting blocks. It also matters that everybody stays in their lanes and that the runners receiving batons are on the same team as the starters.

**The batons are passed not based on the names of the runners, but on the lanes they run in.**

|  |
| --- |
| **Important Rules About Using Methods With Parameters** |
| The number of parameters in the method call (actual parameters) must match the number of parameters in the method heading (formal parameters).  The corresponding actual parameters must be the same type as the formal parameters.  The sequence of the actual parameters must match the sequence of the formal parameters.  The actual parameter identifiers may be the same identifier or a different identifier as the formal parameters. |

**7.5 Void Methods and Return Methods**

All the user-declared methods in this chapter have been **void** methods. You have already been using both **return** and **void** methods in the previous chapter. With the **Bank** class you made a deposit with a method call, like:

**tom.checkingDeposit(1000.0);**

The **checkingDeposit** method performs some action, which in this case adds money to your checking account. There is no value returned to the method call. Such a method is called a **void** method. Void methods are called as "stand-alone" program statements.

Return methods always return a value. You do not use a return method call in a stand-alone statement. A return method call must be part of a program statement, which uses the value that is returned by the method, like:

**System.out.println(tom.getChecking());**

The name **return** method and **void** method will make more sense when you see the distinction between the two method declarations in an actual class. In the last chapter, all the class and method declarations were hidden. In this chapter you can see the code and this will help to motivate the naming conventions of the different methods.

Let us start with a **Calc** class. This is somewhat of a simplified version of the Java **Math** class. Furthermore, all the methods in this **Calc** class are **void** methods. You can identify **void** methods by the headings, which use the reserved word **void**. Observant students will now realize that the main method, which you have used since day one, is also a **void** method.

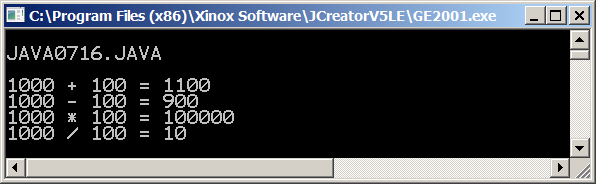
Our modest **Calc** class has only four methods for the four basic arithmetic operations. Each method requires two parameters, which pass the two operands for each one of the four binary operations. The methods compute the required result and then display the two parameters and the calculated result.

This amazing display of **Calc** class wizardry shown by program **Java0716.java**, in figure 7.17, is sure to astound the most discriminating Computer Science student or even AP Computer Science student.

**Figure 7.17**

|  |
| --- |
| // Java0716.java  // This program demonstrates how to create a four-function <Calc> class with void methods.  public class Java0716  {    public static void main(String args[])  {  System.out.println("\nJAVA0716.JAVA\n");  int number1 = 1000;  int number2 = 100;  Calc.add(number1,number2);  Calc.sub(number1,number2);  Calc.mul(number1,number2);  Calc.div(number1,number2);  System.out.println();  }    }  class Calc  {    public static void add(int n1, int n2)  {  int result = n1 + n2;  System.out.println(n1 + " + " + n2 + " = " + result);  }    public static void sub(int n1, int n2)  {  int result = n1 - n2;  System.out.println(n1 + " - " + n2 + " = " + result);  }    public static void mul(int n1, int n2)  {  int result = n1 \* n2;  System.out.println(n1 + " \* " + n2 + " = " + result);  }    public static void div(int n1, int n2)  {  int result = n1 / n2;  System.out.println(n1 + " / " + n2 + " = " + result);  }    } |

**Figure 7.17 Continued**

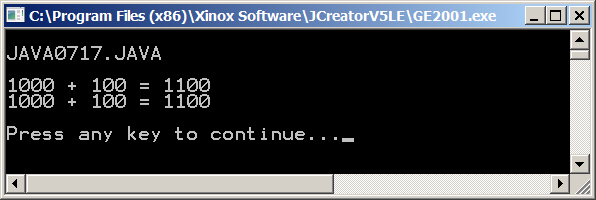


Program **Java0717.java**, in figure 7.18, demonstrates the syntactical difference between writing void methods and return methods. Two versions of a **sum** method are used. **sum1** is a **void** method and **sum2** is a return method. The **void** method, **sum1**, uses the reserved word **void** and displays the result of computing the sum of the two provided parameter values. The **return** method **sum2** uses **int** in the method heading in place of **void**, which indicates the data type of the return value. Additionally, you will note the reserved word **return**, which is a required statement at the end of a return method to indicate which value is returned.

**Figure 7.18**

|  |
| --- |
| // Java0717.java  // This program demonstrates the difference between a  // void <Sum1> method and a return <Sum2> method.  // There are two differences.  // Void and return methods are declared differently.  // Void and return methods are also called differently.  public class Java0717  {    public static void main(String args[])  {  System.out.println("\nJAVA0717.JAVA\n");  int nbr1 = 1000;  int nbr2 = 100;  sum1(nbr1,nbr2);  System.out.println(nbr1 + " + " + nbr2 + " = " + sum2(nbr1,nbr2));  System.out.println();  }    public static void sum1(int n1, int n2)  {  int sum = n1 + n2;  System.out.println(n1 + " + " + n2 + " = " + sum);  }    public static int sum2(int n1, int n2)  {  int sum = n1 + n2;  return sum;  }    } |

**Figure 7.18 Continued**

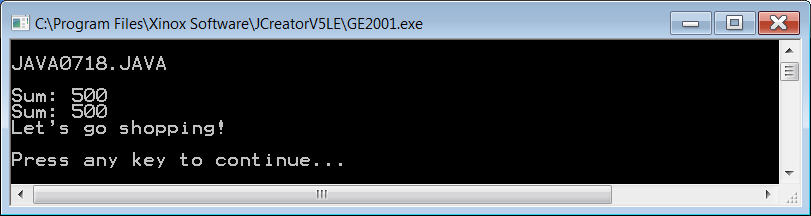


Program **Java0718.java**, in figure 7.19, demonstrates a variety of ways to call a return method. In this case **sum** is called with an output display using **println**, with an assignment statement and finally with a boolean condition in a selection statement. In each example you will note the value returned by the **sum** method is used in the program statement.

**Figure 7.19**

|  |
| --- |
| // Java0718.java  // This program reviews different ways to call a return method.  public class Java0718  {  public static void main(String args[])  {  System.out.println("\nJAVA0718.JAVA\n");  System.out.println("Sum: " + sum(200,300));  int sum = sum(200,300);  System.out.println("Sum: " + sum);  int checking = 200;  int savings = 300;  if (sum(checking,savings) <= 0)  System.out.println("You are broke!");  else  System.out.println("Let's go shopping!");  System.out.println();  }  public static int sum(int n1, int n2)  {  int sum = n1 + n2;  return sum;  }  } |

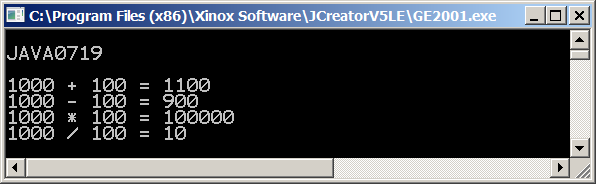
**Figure 7.19 Continued**



This section concludes by returning to the exciting **Calc** class. This time the methods are altered from **void**methods to **return** methods. For a calculation class, return methods are more commonly used. Program **Java0719.java**, in figure 7.20, shows the altered, and improved, version of the **Calc** class.

**Figure 7.20**

|  |
| --- |
| // Java0719.java  // This program demonstrates how to create a four-function <Calc> class with return methods.  public class Java0719  {    public static void main(String args[])  {  System.out.println("\nJAVA0719\n");  int nbr1 = 1000;  int nbr2 = 100;  System.out.println(nbr1 + " + " + nbr2 + " = " + Calc.add(nbr1,nbr2));  System.out.println(nbr1 + " - " + nbr2 + " = " + Calc.sub(nbr1,nbr2));  System.out.println(nbr1 + " \* " + nbr2 + " = " + Calc.mul(nbr1,nbr2));  System.out.println(nbr1 + " / " + nbr2 + " = " + Calc.div(nbr1,nbr2));  System.out.println();  }    }  class Calc  {  public static int add(int n1, int n2)  {  return n1 + n2;  }    public static int sub(int n1, int n2)  {  return n1 - n2;  }    public static int mul(int n1, int n2)  {  return n1 \* n2;  }    public static int div(int n1, int n2)  {  return n1 / n2;  }    } |



**7.6 Making a Utility Library Class**

The **Math** class has many practical methods available for the programmer. It is a true example of a class with class methods that simplify writing programs. This chapter showed several small versions of a **Calc** class to demonstrate the syntax of class methods. Make no special efforts to save the **Calc** class. You will benefit far more from the methods available in the **Math** class.

There are a variety of tasks that we can put in a special utility class to simplify our job. This chapter will finish with a few program examples that demonstrate an **Util**, short for Utility, class. Do not get too excited, the **Util** is not all that interesting, but it does demonstrate the purpose of class methods very nicely. Program **Java0720.java**, in figure 7.21, shows a program with methods that can center text, right justify text and skip lines. Like I said, not very exciting, but it is a comfortable start.

A method like **skip** is very practical. With a convenient parameter you can specify the number of lines that need to be skipped. You may argue that skipping lines is not that difficult and you are right. How about centering text or right justifying text? That is more complicated and a method, designed for those actions, certainly simplifies matters very much here.

Now that you have had an introduction to classes, I will let you in on a secret. **String** is not a simple data type, it is a class. Did you notice that **int**, **char** and **double** start with lower-case letters and **String** starts with an upper-case letter, like all other classes. Well, it is a class and in the methods of the **Util** class I use the **length** method of the **String** class to help compute how to center text and right-justify text.

**Figure 7.21**

|  |
| --- |
| // Java0720.java  // This program demonstrates a user-declared <Util> class with a variety  // of methods that can be used in any program.  public class Java0720  {  public static void main(String args[])  {  System.out.println("\nJAVA0720\n");  Util.skip(2);  System.out.println("This message is left justified");  Util.skip(2);  Util.center("This message is centered");  Util.skip(2);  Util.rightJustify("This message is right justified");  Util.skip(1);  }  }  class Util  {  public static void skip(int n)  {  for (int k = 1; k <= n; k++)  System.out.println();  }    public static void center(String str)  {  int len = str.length();  int tab = (80 - len) / 2;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }  public static void rightJustify(String str)  {  int len = str.length();  int tab = 80 - len;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }  } |

**Figure 7.21 Continued**

****

Can we do better than the previous **Util** class? Certainly, we can. How about adding a clever **heading** method that automatically puts your name, lab assignment, due date and point version inside one neat little box. That actually is fairly tricky to do and if your instructor requires such a box, it will take extra time for each lab assignment. Now with the **heading** method it is only a matter of providing the necessary parameters. Program **Java0721.java**, in figure 7.22, demonstrates the improvement.

**Figure 7.22**

|  |
| --- |
| // Java0721.java  // This program adds the <heading> method to the <Util> class. Note the use of the <spaces> helper method.  public class Java0721  {  public static void main(String args[])  {  Util.heading("Leon Schram","Java0721.Java","10-24-06","100 Points");  System.out.println("This message is left justified");  Util.skip(2);  Util.center("This message is centered");  Util.skip(2);  Util.rightJustify("This message is right justified");  Util.skip(1);  }  }  class Util  {  public static void skip(int n)  {  for (int k = 1; k <= n; k++)  System.out.println();  }    public static void center(String str)  {  int len = str.length();  int tab = (80 - len) / 2;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }  public static void rightJustify(String str)  {  int len = str.length();  int tab = 80 - len;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }    public static String spaces(int n)  {  String temp = "";  for (int k = 1; k <= n; k++)  temp += " ";  return temp;  }    public static void heading(String name, String lab, String date, String points)  {  int nameTab = 28 - name.length();  int labTab = 28 - lab.length();  int dateTab = 28 - date.length();  int pointsTab = 28 - points.length();  Util.skip(2);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Student Name: " + name + spaces(nameTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Lab Assignment: " + lab + spaces(labTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Date Due: " + date + spaces(dateTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Point Version: " + points + spaces(pointsTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Util.skip(2);  }  } |

**Figure 7.22 Continued**



Do you like that nifty **heading** method? If you do not like it, my response is simple. Make your own methods, create your own classes, do what is necessary to impress yourself with your computer skills.

I will finish this chapter by showing you the more correct way to work with a utility class, or any other user-declared class for that matter. You can access the methods of a class, as long as that class is in the same directory as your driving program. The driving program is the program with the **main** method. There are other approaches, such as creating a package, which can be used with **import** and there is an approach called *creating a project*. Yes there are multiple ways to work with multiple files. Until further notice your mission is simple, put all the files that need to work together in one directory. Check out **Java0722.java**, in figure 7.23, you will see that it is now a small program with one method. It uses the **Util** class, which is in a separate file, shown in figure 7.24. Both files are in the **Programs07** folder and everything will compile just nicely without a problem.

**Figure 7.23**

|  |
| --- |
| // Java0722.java  // This program is identical to Java0718.java with the <Util> class not included in this file.  public class Java0722  {  public static void main(String args[])  {  Util.heading("Leon Schram","Lab0722","03-15-12","100 Points");  System.out.println("This message is left justified");  Util.skip(2);  Util.center("This message is centered");  Util.skip(2);  Util.rightJustify("This message is right justified");  Util.skip(1);  }  } |

**Figure 7.24**

|  |
| --- |
| // Util.java  // This file is the <Util> class. This file can compile by itself, but  // it cannot execute. It requires the Java0713.java driver program to  // test the <Util> class.  class Util  {  public static void skip(int n)  {  for (int k = 1; k <= n; k++)  System.out.println();  }    public static void center(String str)  {  int len = str.length();  int tab = (80 - len) / 2;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }  public static void rightJustify(String str)  {  int len = str.length();  int tab = 80 - len;  for (int k = 1; k <= tab; k++)  System.out.print(" ");  System.out.println(str);  }    public static String spaces(int n)  {  String temp = "";  for (int k = 1; k <= n; k++)  temp += " ";  return temp;  }  public static void heading(String name, String lab, String date, String points)  {  int nameTab = 28 - name.length();  int labTab = 28 - lab.length();  int dateTab = 28 - date.length();  int pointsTab = 28 - points.length();    Util.skip(2);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Student Name: " + name + spaces(nameTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Lab Assignment: " + lab + spaces(labTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Date Due: " + date + spaces(dateTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\* Point Version: " + points + spaces(pointsTab) + "\*\*");  System.out.println("\*\* \*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Util.skip(2);  }  } |

**7.7 The Payroll Case Study**

You are about to study eight stages of a case study. This is the first of many case studies that will appear in your textbook. What is a case study? The words itself imply the meaning. There is some special case that will be studied. Now what exactly does that mean in computer science? Program examples are used to present complete programs. The programs may be small, they may be large or any size in between, but basically you are presented with fully functional programs. Now that programs are starting to become more complex and your knowledge of computer science and Java is growing, you will find there are times when the impact of a complex program can be quite overwhelming. You may benefit more from an approach where the first example is not a complete program at all, but a simple starting stage. From this beginning each program adds some feature or features and slowly but surely an entire program develops.

This is only a partial explanation of case studies. Nobody writes a complete program instantaneously. Furthermore, many programmers make mistakes along the way of creating a complete and reliable program. A case study can also be an excellent tool in program development. Stages in the case study may intentionally be quite wrong, but they reflect a natural development process. In other words, it is hardly sufficient to teach computer science by simply showing a few Java keywords with some program examples to use the keywords. You also need to learn how to develop your own programs.

The case study that follows will demonstrate the development of a payroll program. It is the intention of the case study to present the first set of information on program design. Program design is not an easy topic to teach or learn. There is quite a subjective side to program design. The Java compiler cares little about program design. On the other hand if you need to enhance your program six months later, you will greatly appreciate if you followed some fundamental rules of design. Likewise if you leave the job and somebody else take over from a program that you started, the new programmer will appreciate if your program is developed in a manner that makes debugging and enhancement manageable.

With the popularity of Object Orient Programming a new and popular computer science field emerged called *Object Oriented Design*. Program design does have some chicken and egg problems. The computer science community is concerned that students learn computer science and will form bad programming habits if design is not taught immediately. At the same time program design, especially in object oriented programming area, makes little sense unless you have some OOP knowledge.

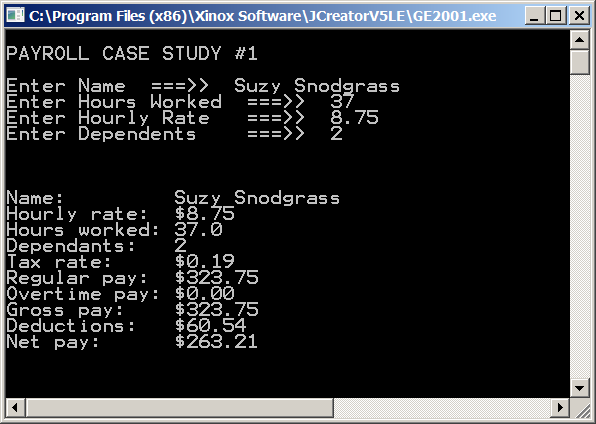
Not everybody agrees on the best approach to this design dilemma. My personal opinion is that you have now learned a sufficient amount of computer science to start investigating issues. The program design treatment in this chapter is not complete; it is a start. As you learn additional computer science concepts, the design concern will return at regular intervals.

**Payroll Case Study, Stage #1**

Stage #1 is a very bad example of program design. I have never seen a student or anybody else program in this manner. The only reason why this stage is presented is to make a point. Students frequently complain about the need to use proper indentations or proper comments. Look at program **Jav0723.java**, in figure 7.25. This program actually compiles and executes correctly. All the source code runs together, there are no meaningful identifiers and no comments. This type of program is impossible to debug or enhance, if it were a large program.

**Figure 7.25**

|  |
| --- |
| // Java0723.java  // Payroll Case Study #1  // The first stage of the Payroll program has correct syntax and logic.  // However, there is no concern about any type of proper program design,  // even to the degree that there is no program indentation. This program is totally unreadable. import java.util.Scanner;import java.text.\*; public class Java0723 { public static void main (String args[]) { Scanner keyboard = new Scanner(System.in); String a; double b,c,e,f,g,h,i,j,k; int d; DecimalFormat output = new DecimalFormat("$0.00"); System.out.println("\nPAYROLL CASE STUDY #1\n"); System.out.print("Enter Name ===>> "); a = keyboard.nextLine(); System.out.print("Enter Hours Worked ===>> "); b = keyboard.nextDouble(); System.out.print("Enter Hourly Rate ===>> "); c = keyboard.nextDouble(); System.out.print ("Enter Dependents ===>> "); d = keyboard.nextInt(); if (b > 40) { e = b - 40; k = 40 \* c; j = e \* c \* 1.5; } else { k = b \* c; j = 0; } g = k + j; switch (d) { case 0 : f = 0.295; break; case 1 : f = 0.249; break; case 2 :  f = 0.187; break; case 3 : f = 0.155; break; case 4 : f = 0.126; break; case 5 : f = 0.100; break; default: f = 0.075; } i = g \* f; h = g - i; System.out.println("\n\n"); System.out.println("Name: " + a); System.out.println("Hourly rate: " + output.format(c)); System.out.println("Hours worked: " + b); System.out.println("Dependents: " + d); System.out.println("Tax rate: " + output.format(f)); System.out.println("Regular pay: " + output.format(k)); System.out.println("Overtime pay: " + output.format(j)); System.out.println("Gross pay: " + output.format(g)); System.out.println("Deductions: " + output.format(i));  System.out.println("Net pay: " + output.format(h)); System.out.println("\n\n"); } } |



**Payroll Case Study, Stage #2**

Program **Java0724.**java, in figure 7.26, makes a small improvement. Every program statement is written on a separate line. Block structure also uses indentation to indicate which statement will be executed. At the same time program statements within the same block are indented the same amount. This stage is a long way from a well-designed program, but even this first improvement helps the program significantly for any future program enhancement. The program output is identical to the first stage, as will be the case for each stage. Future stages will not include any more program outputs. Each stage along the way features some improvement, but it does not alter the fundamental logic of the program.

**Figure 7.26**

|  |
| --- |
| // Java0724.java  // Payroll Case Study #2  // The second stage does use indentation, but it is still very poor program design.  // All the program logic is contained in the <main> method and there are no program  // comments anywhere, nor are the identifiers self-commenting.  import java.util.Scanner;  import java.text.\*;  public class Java0724  {  public static void main (String args[])  {  Scanner keyboard = new Scanner(System.in);  String a;  double b,c,e,f,g,h,i,j,k;  int d;  DecimalFormat output = new DecimalFormat("$0.00");  System.out.println("\nPAYROLL CASE STUDY #2\n");  System.out.print("Enter Name ===>> ");  a = keyboard.nextLine();  System.out.print("Enter Hours Worked ===>> ");  b = keyboard.nextDouble();  System.out.print("Enter Hourly Rate ===>> ");  c = keyboard.nextDouble();  System.out.print("Enter Dependents ===>> ");  d = keyboard.nextInt();    if (b > 40)  {  e = b - 40;  k = 40 \* c;  j = e \* c \* 1.5;  }  else  {  k = b \* c;  j = 0;  }  g = k + j;  switch (d)  {  case 0 : f = 0.295; break;  case 1 : f = 0.249; break;  case 2 : f = 0.187; break;  case 3 : f = 0.155; break;  case 4 : f = 0.126; break;  case 5 : f = 0.100; break;  default: f = 0.075;  }  i = g \* f;  h = g - i;  System.out.println("\n\n");  System.out.println("Name: " + a);  System.out.println("Hourly rate: " + output.format(c));  System.out.println("Hours worked: " + b);  System.out.println("Dependents: " + d);  System.out.println("Tax rate: " + output.format(f));  System.out.println("Regular pay: " + output.format(k));  System.out.println("Overtime pay: " + output.format(j));  System.out.println("Gross pay: " + output.format(g));  System.out.println("Deductions: " + output.format(i));  System.out.println("Net pay: " + output.format(h));  System.out.println("\n\n");  }    } |

**Payroll Case Study, Stage #3**

Stage #3 makes a large step forward to improving the program. The single-letter, meaningless identifiers of the previous stages, are now replaced with very readable self-commenting identifiers. Programs should have useful comments at strategic locations in the program, but the first step in commenting is to select good identifiers. With self-commenting identifiers, like **Java0725.java**, in figure 7.27, you know anywhere in the program what the purpose of a variable should be. Identifiers like **hoursWorked**, **grossPay** and **netPay** provide an immediate clarification for the variable.

**Figure 7.27**

|  |
| --- |
| // Java0725.java  // Payroll Case Study #3  // Stage 3 improves program readability by using meaningful identifiers.  import java.util.Scanner;  import java.text.\*;  public class Java0725  {  public static void main (String args[])  {  Scanner keyboard = new Scanner(System.in);  String employeeName;  double hoursWorked;  double hourlyRate;  int numDependents;  double overtimeHours;  double regularPay;  double overtimePay;  double taxRate;  double grossPay;  double taxDeductions;  double netPay;  DecimalFormat output = new DecimalFormat("$0.00");    System.out.println("\nPAYROLL CASE STUDY #3\n");  System.out.print("Enter Name ===>> ");  employeeName = keyboard.nextLine();  System.out.print("Enter Hours Worked ===>> ");  hoursWorked = keyboard.nextDouble();  System.out.print("Enter Hourly Rate ===>> ");  hourlyRate = keyboard.nextDouble();  System.out.print("Enter Dependents ===>> ");  numDependents = keyboard.nextInt();    if (hoursWorked > 40)  {  overtimeHours = hoursWorked - 40;  regularPay = 40 \* hourlyRate;  overtimePay = overtimeHours \* hourlyRate \* 1.5;  }  else  {  regularPay = hoursWorked \* hourlyRate;  overtimePay = 0;  }  grossPay = regularPay + overtimePay;    switch (numDependents)  {  case 0 : taxRate = 0.295; break;  case 1 : taxRate = 0.249; break;  case 2 : taxRate = 0.187; break;  case 3 : taxRate = 0.155; break;  case 4 : taxRate = 0.126; break;  case 5 : taxRate = 0.100; break;  default: taxRate = 0.075;  }    taxDeductions = grossPay \* taxRate;  netPay = grossPay - taxDeductions;    System.out.println("\n\n");  System.out.println("Name: " + employeeName);  System.out.println("Hourly rate: " + output.format(hourlyRate));  System.out.println("Hours worked: " + hoursWorked);  System.out.println("Dependents: " + numDependents);  System.out.println("Tax rate: " + output.format(taxRate));  System.out.println("Regular pay: " + output.format(regularPay));  System.out.println("Overtime pay: " + output.format(overtimePay));  System.out.println("Gross pay: " + output.format(grossPay));  System.out.println("Deductions: " + output.format(taxDeductions));  System.out.println("Net pay: " + output.format(netPay));  System.out.println("\n\n");  }    } |

**Payroll Case Study, Stage #4**

Program **Java0726.java**, in figure 7.28, provides two improvements. Stage #4 adds comments and also separates the program into segments to help identify the purpose of a program segment. I remember about ten years ago there was a student in my class who made a beautiful horse for her graphics project. Everything looked great on the horse except for the tail. The tail was totally wrong and not attached anywhere on the horse. She asked for help and much to my surprise this student could not tell me the segment of her program responsible for drawing the tail. She had added program statement after program statement in one continuous, giant block of code. Debugging or enhancing such a program becomes a nightmare.

**Figure 7.28**

|  |
| --- |
| // Java0726.java  // Payroll Case Study #4  // Stage 4 separates the program statements in the main method with spaces and comments  // to help identify the purpose for each segment. This helps program debugging and updating.  // Note that this program does not prevents erroneous input.  import java.util.Scanner; // provides access to the input methods of the Scanner class.  import java.text.\*; // used for text output with <DecimalFormat> class.  public class Java0726  {  public static void main (String args[])  {  /////////////////////////////////////////////////////////////////////////////////////////////////////  // Program variables  //  String employeeName; // employee name used on payroll check  double hoursWorked; // hours worked per week  double hourlyRate; // employee wage paid per hour  int numDependents; // number of dependents declared for tax rate purposes  double overtimeHours; // number of hours worked over 40  double regularPay; // pay earned for up to 40 hours worked  double overtimePay; // pay earned for hours worked above 40 per week  double taxRate; // tax rate, based on declared dependents, used for deductions  double grossPay; // total pay earned before deductions  double taxDeductions; // total tax deductions  double netPay; // total take-home pay, which is printed on the check  //////////////////////////////////////////////////////////////////////////////////////////////////////      /////////////////////////////////////////////////////////////////////////////////////////////////////  // Program objects  //  Scanner keyboard = new Scanner(System.in);  // keyboard is used for interactive keyboard input  DecimalFormat output = new DecimalFormat("$0.00");  // output is used to display values in monetary format  ////////////////////////////////////////////////////////////////////////////////////////////////////      ///////////////////////////////////////////////////////////////////////////////////////////////////  // Program input  //  System.out.println("\nPAYROLL CASE STUDY #3\n");  System.out.print("Enter Name ===>> ");  employeeName = keyboard.nextLine();  System.out.print("Enter Hours Worked ===>> ");  hoursWorked = keyboard.nextDouble();  System.out.print("Enter Hourly Rate ===>> ");  hourlyRate = keyboard.nextDouble();  System.out.print("Enter Dependents ===>> ");  numDependents = keyboard.nextInt();  //////////////////////////////////////////////////////////////////////////////////////////////////      //////////////////////////////////////////////////////////////////////////////////////////////////  // Program computation  //  if (hoursWorked > 40) // qualifies for overtime pay  {  overtimeHours = hoursWorked - 40;  regularPay = 40 \* hourlyRate;  overtimePay = overtimeHours \* hourlyRate \* 1.5;  }  else // does not qualify for overtime pay  {  regularPay = hoursWorked \* hourlyRate;  overtimePay = 0;  }    grossPay = regularPay + overtimePay;  // total pay earned before any deductions      switch (numDependents)  // compute proper tax rate based on declared dependents  // everybody gets 0.075 tax rate if dependents are greater than 5  {  case 0 : taxRate = 0.295; break;  case 1 : taxRate = 0.249; break;  case 2 : taxRate = 0.187; break;  case 3 : taxRate = 0.155; break;  case 4 : taxRate = 0.126; break;  case 5 : taxRate = 0.100; break;  default: taxRate = 0.075;  }  taxDeductions = grossPay \* taxRate;    netPay = grossPay - taxDeductions;  // computes actual take-home-pay, which is printed on the paycheck    /////////////////////////////////////////////////////////////////////////////////////////////  // Output display, which simulates the printing of a payroll check  //  System.out.println("\n\n");  System.out.println("Name: " + employeeName);  System.out.println("Hourly rate: " + output.format(hourlyRate));  System.out.println("Hours worked: " + hoursWorked);  System.out.println("Dependents: " + numDependents);  System.out.println("Tax rate: " + output.format(taxRate));  System.out.println("Regular pay: " + output.format(regularPay));  System.out.println("Overtime pay: " + output.format(overtimePay));  System.out.println("Gross pay: " + output.format(grossPay));  System.out.println("Deductions: " + output.format(taxDeductions));  System.out.println("Net pay: " + output.format(netPay));  System.out.println("\n\n");  /////////////////////////////////////////////////////////////////////////////////////////////  }  } |

**Payroll Case Study, Stage #5**

Stage #5 with program **Java0727.java**, in figure 7.29, uses modular programming. The previous program segments, which had been separated by comments, are now each placed inside methods. This is better program design, but something else must happen. Previously, the variables were declared in the **main** method. This worked since all the program statements were also in the **main** method. Now the variables are declared as static, class variables, which makes them accessible to the entire class.

**Figure 7.29**

|  |
| --- |
| // Java0727.java  // Payroll Case Study #5  // Stage #5 creates variables that can be used anywhere in the class  // and the program segments are now placed in modules.  import java.util.Scanner;  import java.text.\*;  public class Java0727  {  static String employeeName;  static double hoursWorked;  static double hourlyRate;  static int numDependents;  static double overtimeHours;  static double regularPay;  static double overtimePay;  static double taxRate;  static double grossPay;  static double taxDeductions;  static double netPay;  public static void main (String args[])  {  System.out.println("\nPAYROLL CASE STUDY #5\n");  enterData();  computeGrosspay();  computeDeductions();  computeNetpay();  printCheck();  }    public static void enterData()  {  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter Name ===>> ");  employeeName = keyboard.nextLine();  System.out.print("Enter Hours Worked ===>> ");  hoursWorked = keyboard.nextDouble();  System.out.print("Enter Hourly Rate ===>> ");  hourlyRate = keyboard.nextDouble();  System.out.print("Enter Dependents ===>> ");  numDependents = keyboard.nextInt();  }      public static void computeGrosspay()  {  if (hoursWorked > 40)  {  overtimeHours = hoursWorked - 40;  regularPay = 40 \* hourlyRate;  overtimePay = overtimeHours \* hourlyRate \* 1.5;  }  else  {  regularPay = hoursWorked \* hourlyRate;  overtimePay = 0;  }  grossPay = regularPay + overtimePay;  }      public static void computeDeductions()  {  switch (numDependents)  {  case 0 : taxRate = 0.295; break;  case 1 : taxRate = 0.249; break;  case 2 : taxRate = 0.187; break;  case 3 : taxRate = 0.155; break;  case 4 : taxRate = 0.126; break;  case 5 : taxRate = 0.100; break;  default: taxRate = 0.075;  }  taxDeductions = grossPay \* taxRate;  }      public static void computeNetpay()  {  netPay = grossPay - taxDeductions;  }      public static void printCheck()  {  DecimalFormat output = new DecimalFormat("$0.00");  System.out.println("\n\n");  System.out.println("Name: " + employeeName);  System.out.println("Hourly rate: " + output.format(hourlyRate));  System.out.println("Hours worked: " + hoursWorked);  System.out.println("Dependents: " + numDependents);  System.out.println("Tax rate: " + taxRate);  System.out.println("Regular pay: " + output.format(regularPay));  System.out.println("Overtime pay: " + output.format(overtimePay));  System.out.println("Gross pay: " + output.format(grossPay));  System.out.println("Deductions: " + output.format(taxDeductions));  System.out.println("Net pay: " + output.format(netPay));  System.out.println("\n\n");  }  } |

**Payroll Case Study, Stage #6**

Stage #5 may seem nice, but it is not very good programming design. In object oriented program we want to think about the needs of our program. What is the mission? What are we trying to accomplish. This case study features some type of payroll program. This means that we need a **Payroll** class. For larger programs there will be multiple classes. The essence of program design is the creation of practical classes and the proper interaction of these classes. The previous program example turned the driving class, which is the class with the **main** method into the class handling all the work. That just is not very good design. We want to leave the **main** method to the job of controlling or testing the program sequence.

Program **Java0728.java**, in figure 7.30, now has a separate **Payroll** class. The **Java0728** class is now only concerned with the management, or testing, of the **Payroll** class. Notice also that all the variables have moved to the block limits of the **Payroll** class. This is a considerable improvement.

**Figure 7.30**

|  |
| --- |
| // Java0728.java  // Payroll Case Study #6  // In Stage #6 the <main> method is part of the "driving" class, which is the class  // responsible for the program execution sequence. The <main> method now contains  // method calls to objects of the <Payroll> class.  import java.util.Scanner;  import java.text.\*;  public class Java028  {  public static void main (String args[])  {  System.out.println("\nPAYROLL CASE STUDY #6\n");  Payroll.enterData();  Payroll.computeGrosspay();  Payroll.computeDeductions();  Payroll.computeNetpay();  Payroll.printCheck();  }  }      class Payroll  {  static String employeeName;  static double hoursWorked;  static double hourlyRate;  static int numDependents;  static double overtimeHours;  static double regularPay;  static double overtimePay;  static double taxRate;  static double grossPay;  static double taxDeductions;  static double netPay;    public static void enterData()  {  Scanner keyboard = new Scanner(System.in);  System.out.print("Enter Name ===>> ");  employeeName = keyboard.nextLine();  System.out.print("Enter Hours Worked ===>> ");  hoursWorked = keyboard.nextDouble();  System.out.print("Enter Hourly Rate ===>> ");  hourlyRate = keyboard.nextDouble();  System.out.print("Enter Dependents ===>> ");  numDependents = keyboard.nextInt();  }    public static void computeGrosspay()  {  if (hoursWorked > 40)  {  overtimeHours = hoursWorked - 40;  regularPay = 40 \* hourlyRate;  overtimePay = overtimeHours \* hourlyRate \* 1.5;  }  else  {  regularPay = hoursWorked \* hourlyRate;  overtimePay = 0;  }  grossPay = regularPay + overtimePay;  }    public static void computeDeductions()  {  switch (numDependents)  {  case 0 : taxRate = 0.295; break;  case 1 : taxRate = 0.249; break;  case 2 : taxRate = 0.187; break;  case 3 : taxRate = 0.155; break;  case 4 : taxRate = 0.126; break;  case 5 : taxRate = 0.100; break;  default: taxRate = 0.075;  }  taxDeductions = grossPay \* taxRate;  }    public static void computeNetpay()  {  netPay = grossPay - taxDeductions;  }    public static void printCheck()  {  DecimalFormat output = new DecimalFormat("$0.00");  System.out.println("\n\n");  System.out.println("Name: " + employeeName);  System.out.println("Hourly rate: " + output.format(hourlyRate));  System.out.println("Hours worked: " + hoursWorked);  System.out.println("Dependents: " + numDependents);  System.out.println("Tax rate: " + taxRate);  System.out.println("Regular pay: " + output.format(regularPay));  System.out.println("Overtime pay: " + output.format(overtimePay));  System.out.println("Gross pay: " + output.format(grossPay));  System.out.println("Deductions: " + output.format(taxDeductions));  System.out.println("Net pay: " + output.format(netPay));  System.out.println("\n\n");  }  } |

After this introduction to program design you should realize that many program examples that follow will actually not follow the very principles explained in this section. The aim of Exposure Java is to present new concepts with the smallest, clearest programs so that you learn using one small bite after another. Many program examples will in fact place the program statements in the **main** method. These programs are small and serve their purpose nicely. Creating separate classes for different purposes with all program examples requires a lot of space and may obscure the new topic that is introduced.

|  |
| --- |
| **Local Variables and Class Variables** |
| Variables that are declared inside a method or block are called *local variables*. Local variables are only accessible inside the method or block that they are defined in.  Variables that are declared inside a class, but outside any method, are *class variables*. Class variables are accessible by any method of the class.  Class variables are also called *attributes*.  If a variable is only used by one method, it should be declared inside that method as a *local variable*.  If a variable is used by two or more methods of a class, it should be declared as a *class variable*. |

|  |
| --- |
| **Program Design Notes** |
| This was the first introduction to program design. Additional design features will be introduced as you learn more Object-Oriented Programming. At this stage you can already consider the following:  • Programs should use self-commenting identifiers.  • Control structures and block structures need to use a  consistent indentation style.  • Specific tasks should be placed in modules called methods.  • Similar methods accessing the same data should be placed  in a class.  • The **main** method should be used for program sequence,  not large numbers of program statements. |

**7.8 Creating a GridWorld Library**

In several previous chapters you have created GridWorld projects. Each of the project folders always contained the **gridworld.jar** file. This file is required for any GridWorld project and therefore it was always placed in any folder used for GridWorld programs.

It is possible to locate the GridWorld required files in one library at a permanent location and indicate to JCreator – or some other IDE – where this library is located. Once this special library file is created, it is then possible to attach the library to any GridWorld project, regardless of its location. In the lab experiment that follows you will once again go step-by-step through the project creation stage, but this time you will create a special library.

|  |
| --- |
| **Lab Experiment Java0729**  Experience has proven that you the speedy students, who like to charge ahead, have trouble with this experiment. You are doing a GridWorld project differently than you have done previously. If you ignore the instructions and quickly create a project, as you have done previously, then your program will not compile. Have patience and follow the steps carefully. |

|  |
| --- |
| **01. Investigate Folder Java0729**  Navigate on your computer to the **Java0729** folder, similar to the one shown below. There is only one file present. The familiar **gridworld.jar** isabsent. The **gridworld.jar** file is still necessary, but in this experiment you will learn how to create a special library that includes the required **gridworld.jar** file and then attach it to your project.  All future GridWorld projects, experiments and lab assignments will assume that you are using the ***attached library*** approach. You will no longer see the **gridworld.jar** file located in any of your future GridWorld folders. |

|  |
| --- |
| **02. Start JCreator**  Use the window format shown below. This means that you display an *Edit* window, a *File View* windowand an *Output Window*. |

|  |
| --- |
| **03a. Create a New Project**  **Click File**  **Click New**  **Click Project** |

|  |
| --- |
| **03b. Create a New Project**  When the window below appears, **click Empty Project**  **Click Next** |

|  |
| --- |
| **03c. Create a New Project**  **Click the** **Browse Button** on the right side of the **Location** window. |

|  |
| --- |
| **03d. Create a New Project**  Navigate and find the **Java0729** folder, which should be in the **Programs07** folder.  Note that the location of the **Java0729** folder may vary.  **Click the Java0729 folder** and then **click OK**. |

|  |
| --- |
| **03e. Create a New Project**  JCreator has now automatically entered the correct information for all four windows.  STOP ! ! ! ! It now will be different ! ! ! !  **Click Next.** |
| **04a. Create the Required GridWorld Library**  When this window pops up **click Required Libraries** and click **New**. |

|  |
| --- |
| **04b. Create the Required GridWorld Library**  **Type GWCS Library in the Name window. (The name of the library does not matter)** |

|  |
| --- |
| **04c. Create the Required GridWorld Library**  **Click Add and then click Add Archive.** |

|  |
| --- |
| **04d. Create the Required GridWorld Library**  **Navigate to find the GridWorld-No Touch Files folder.**  **Click on the gridworld.jar file and then click open**.  **Click OK**. |
| **04e. Create the Required GridWorld Library**  **Check the box in front of GWCS Library**.  **Click Finish twice**. |

|  |
| --- |
| **05a. Compile and Execute the Project**  The project is now completed and the proper files are loaded by the project manager. The left **File View** window shows the files that are part of the project. The first file is automatically loaded into the edit window. In this case there is only one file.  **Click the Build or Compile Project icon.** |

|  |
| --- |
| **05b. Compile and Execute the Project**  If the correct location and the project is properly created, you will see the window below. You need to see **Process completed** without any error messages in the output window. |

|  |
| --- |
| **05c. Compile and Execute the Project**  If the compile stage completed without error messages, you can **Run** or **Execute** the project.  **Click the Run or Execute Project icon.**  After a short delay you will see a grid six objects. The six objects are placed on the grid at random locations. You should have a 10 X 10 grid and six objects, but the locations of the objects are randomly selected. |

**7.9 Observing the Bug act Method**

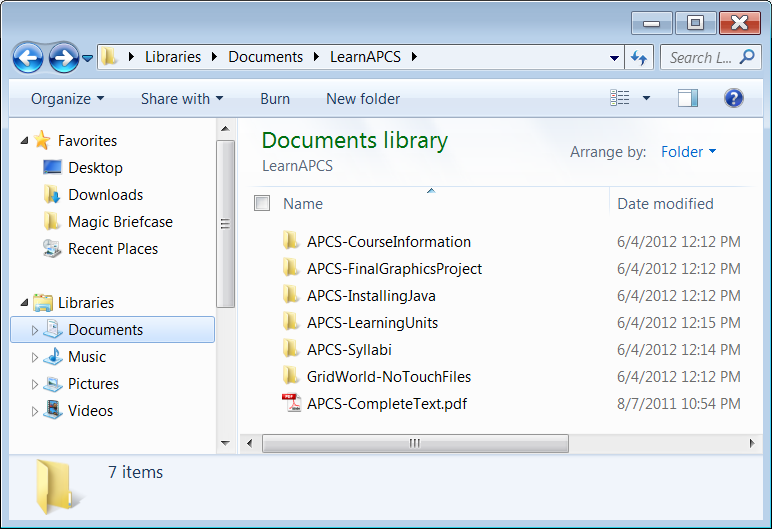
This is a chapter on creating methods. It does not matter how many tools Java has provided. There are always specialty needs that require that you create your very own tools or methods to make a program simpler to write. The same is true for the GridWorld Case Study. The GWCS is a very large program and it has many classes and many, many methods. Yet, it is always possible to create additional classes and additional methods. The creation of new classes will have to wait until the next chapter, but right now you are ready to try and see how you can make some small additions to a GridWorld program.

The GWCS lab experiments that follow later will concentrate on making some changes to the **Bug** class. For the convenience of the lab experiment that will follow, the **Bug.java** file has been placed in the lab experiment folder. This begs the question where these files are located? The answer is in the **GridWorld-NoTouchFiles** folder. Right now we are going to look inside that folder before we start actually making any changes to the **Bug** class.

**The GridWorld Case Study Files**

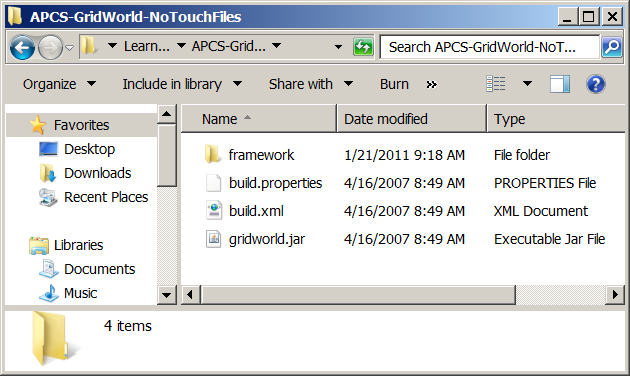
Your search for the GridWorld files starts by finding the G**ridWorld-NoTouchFiles** folder. It contains all the necessary GWCS files and it can be at many locations. Imagine now that it is located in the very cleverly named **LearnAPCS** folder shown in figure 7.31. Your folder may be in a different location than what is shown here. Your first step is to find the location of the **GridWorld-NoTouchFiles** folder.

**Figure 7.31**



The College Board does not use weird names like *NoTouchFiles*. They are much more proper than that. I changed the folder to that name, because it is a folder of important files that should not be moved, altered or touched in any way. This folder contains strictly *read-only* stuff. The immediate contents of this folder are shown in Figure 7.32. In particular, note the **gridworld.jar** file.

**Figure 7.32**



The **gridworld.jar** file is the file you used for your library and it contains all the GridWorld files compressed into one file. The **framework** folder contains other folders and everything that is compressed in **gridworld.jar** can be found and is visible as you navigate down the **framework** folder.

You will now see a quick succession of folders and files that start with the **framework** folder. It is not the intention right now that you comprehend the purposes of all these different files. This is more a brief tour of folders and files that you will learn more about in the future.

Inside the **framework** folder figure 7.33 shows the **info** folder.

And inside the **info** folder figure 7.34 shows the **GridWorld** folder.

Inside the **GridWorld** folder figure 7.35 shows four folders, which are the

**actor**, **grid**, **gui** and **world** folders.

Figure 7.36 finally shows some files, which can be found in the **actor** folder, like the **Bug.java** file, which we will use for the next lab experiment.

|  |  |
| --- | --- |
| **Figure 7.33** | **Figure 7.34** |
| **Figure 7.35** | **Figure 7.36** |

Figure 7.37 shows some files, which can be found in the **grid** folder.

Figure 7.38 shows some files, which can be found in the **gui** folder. You can right now realize that none of the **gui** files will ever be a concern this year. You will be able to complete every GWCS lab assignment and never again look at the **gui** folder and its files.

Figure 7.39 shows one file, which can be found in the **world** folder.

Figure 7.40 repeats the **gridworld.jar** file, which is the file you used first GWCS lab experiment and for the creation of your GridWorld library.

|  |  |
| --- | --- |
| **Figure 7.37** | **Figure 7.38** |
| **Figure 7.39** | **Figure 7.40** |

**Observe Bug Method act and turn**

Every single object that you will ever see placed on a GridWorld acts when the **Step** button is clicked. The objects act repeatedly when **Run** is clicked. You have observed the behavior of the **Actor**, **Rock**, **Bug** and **Flower** objects.

Now you are going get closer and start to look at some of the code that is used in a GridWorld program. We will start gentle and only look at the **Bug** class. There are many GridWorld features that cannot be explained until a later time. This happens, because GridWorld uses computer science concepts that you will learn in the future. For instance, the **Bug** class has some constructors. Constructors will be explained in Chapter VIII, which means that right now we will ignore any **Bug** constructor stuff.

We can look at the **act** method of the **Bug** class and a few other methods to help us prepare for the lab experiment where we will alter the **Bug** object’s behavior. Look at the code of the **act** method, shown in figure 7.41.

**Figure 7.41**

|  |
| --- |
| **public void act()**  **{**  **if (canMove())**  **move();**  **else**  **turn();**  **}** |

This very brief method is an excellent example of the clarity that is provided by using *self-documenting* identifiers. The **act** method has a selection control structure and various method calls. The names of the methods do a good job explaining the behavior.

How does a **Bug** object act? Read the code in figure 7.41. First, check if you can move. You should have observed that **Bug** objects only move over **Flower** objects and go around all other objects or turn before grid boundaries.

If it is possible to move, go ahead. What does **move** mean? You have not seen the actual **move** method, but you did see that a **Bug** object moves one cell in the direction that it is facing and drops a flower in the last cell visited.

In the event that it is not possible to move, the **act** method calls the **turn** method, which means a 45-degree clockwise turn.

For the purpose of this chapter and the upcoming lab experiment we will only be concerned with understanding the **act** method and the **turn** method. The **turn** method is shown in figure 7.42. You now see a frustrating problem with the process of learning about GridWorld in the early stages. You keep on running into methods that you do not know yet. The **turn** method *sets* a new direction for the **Bug** object by adding 45 degrees (**HALF\_RIGHT**) to the current direction. You are now ready to start making some new methods for the **Bug** class.

**Figure 7.42**

|  |
| --- |
| **public void turn()**  **{**  **setDirection(getDirection() + Location.HALF\_RIGHT);**  **}** |

**7.10 Creating New Bug Methods**

Creating a new method requires a method container and the program statements inside the method container that will be executed when the method is called. Remember that calling a method occurs when the method identifier is used in a program statement.

Creating the method container is easy. Figure 7.43 shows an empty method container, called **displaySum**. There is a heading indicating that this is a *public void* method and it will process two **int** parametervalues. Now we continue and place one or more statements inside the method body or container. This is shown in figure 7.44. You are now ready to start Lab Experiment Java0730.

**Figure 7.43**

|  |
| --- |
| **public void displaySum(int n1, int n2)**  **{**  **}** |

**Figure 7.44**

|  |
| --- |
| **public void displaySum(int n1, int n2)**  **{**  **System.out.println(n1 + n2);**  **}** |

|  |
| --- |
| **Lab Experiment 0730**  The mission of this lab experiment is to alter the behavior of the **Bug** object, such that it will make 90-degree clockwise turns when it cannot move rather than the current 45-degree clockwise turns. |

|  |
| --- |
| **01. Create a Project for Lab Experiment Java0730**  Remember that you need to attach the library that you created in **Java0729**.  It is not necessary to create the library again. That is now done permanently.  However, you do need to place a check in front of the library name for each new project. |
| **02. Compile and Execute Java0730**  The GridWorld execution shown below will be the same for everybody and for each time that you execute the program.  The **Bug** class is not yet altered to change the behavior of the **Bug** object.  **Click the Run button and observe the behavior of the objects.**  You will find that every object acts in the manner that you have observed before.  The **Rock** does nothing.  The **Actor** flips around doing 180-degree turns.  The **Bug** moves one cell and drops a flower, if possible, otherwise turns 45-degrees.  The **Flower** starts out red and slowly turns darker. |

|  |
| --- |
| **03. Change the Bug Behavior**  Our goal is to make the **Bug** object perform 90-degree right turns in place of 45-degrees.  There are several ways to accomplish this goal.  The first approach is to alter the **act** method and add a second **turn** method call.  **Change the act method as shown below in the Bug.java file.**  **public void act()**  **{**  **if (canMove())**  **move();**  **else**  **{**  **turn();**  **turn();**  **}**  **}**  **Re-compile the project.**  **Execute again and click Run.**  **Is the behavior as expected?** |

|  |
| --- |
| **04. Challenge**  You now need to make several changes to create a new **Bug** behavior.  This time the **Bug** objects needs to make 90-degree left turns.  There are several approaches that will work correctly.  Your solution must include a new method, called **leftTurn** that turns  90-degrees left. |

**7.11 Summary**

This chapter focused on writing **class methods**. Methods can be members in classes and they can also be members inside an object. The distinction between creating class methods and object methods was displayed with GridWorld examples. A class is a category and an object is one example of the category.

Class methods are called by using the class identifier followed by a period and the method identifier. This is called **dot.method** notation. This type of syntax is necessary because the class contains multiple members. It requires two identifiers to first identify the class and then identify the method within the class.

A popular provided class is the **Math** class. This class provides a variety of useful math functions, such as square root, absolute value, truncate, logarithmic and trigonometric functions.

Java allows programmers to declare their own classes and methods. It is possible to add methods to the existing main class of the program. It is also possible to declare a second class or multiple classes outside the main class.

If a statement calls a member method of the same class, the class identifier is optional. Anytime that a method is called from a class declared outside the calling statement’s class, the class identifier must be provided.

Methods can be declared with or without parameters. In either case a set of parentheses follows the method identifier. In the method declaration parameters need to be declared in the same manner as any other variable declaration, except that each parameter must have its own data type.

The parameters in the method call must match the parameters in the method declaration in quantity, type and sequence. Parameters in a method call can be literal constants, variables or expressions.

This chapter introduced the important concept of program design. Program design was demonstrated by using a case study that started with a very poorly written program and improved with small incremental stages.

In this chapter a special GridWorld library was created that can be used for all the future GridWorld projects. The GridWorld-NoTouchFiles folder cannot be moved after this library is created.

The GridWorld Case Study, like any Java program, allows classes to create new methods. Altering existing methods or adding new methods will alter the behavior of GridWorld objects.