Chapter VII

Creating Class Methods

Chapter VII Topics

7.1   Introduction
7.2   User-Declared Class Methods
7.3   User-Declared Parameter Methods
7.4   Void Methods and Return Methods
7.5   Making a Utility Library Class
7.6   The Payroll Case Study
7.7   Program Input with GUI Windows
7.8   Summary
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 and 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 data types like Math, Random and DecimalFormat. Each one 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 the 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 methods and 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 program 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.

<table>
<thead>
<tr>
<th>Additional Math Methods (not shown in earlier chapter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math.exp(p) returns the antilog of the p</td>
</tr>
<tr>
<td>Math.log(p) returns the log (base e) of p</td>
</tr>
<tr>
<td>Math.sin(p) returns the trigonometric sine of p</td>
</tr>
<tr>
<td>Math.cos(p) returns the trigonometric cosine of p</td>
</tr>
<tr>
<td>Math.tan(p) returns the trigonometric tangent of p</td>
</tr>
<tr>
<td>Math.toDegrees(p) returns the degrees of the radian value of p</td>
</tr>
<tr>
<td>Math.toRadians(p) returns the radians of the degrees value of p</td>
</tr>
<tr>
<td>Math.random() returns a value x, such that 0 &lt;= x &lt; 1</td>
</tr>
</tbody>
</table>

Figure 7.1
This program reviews using class methods and demonstrates most of the available <Math> class methods and data fields.

```java
public class Java0701 {
    public static void main (String args[]) {
        System.out.println("JAVA0701.JAVA");
        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 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));
    }
}
```

Java0701.java Output

JAVA0701.JAVA

The value of E is 2.718281828459045
The value of PI is 3.141592653589793
The absolute value of (-25) is 25
The square root of (1024) is 32.0
The ceiling of (5.00001) is 6.0
The floor of (5.99999) is 5.0
The round of (5.50001) is 6
The antilog of (4.605170185) is 99.9999999011909
The log of (100) is 4.605170185988092
The max of (1000,999) is 1000
The min of (1000,999) is 999
The power of (4,3) is 64.0
The random of () is 0.4785750816715798
The sine of (0) is 0.0
The cosine of (3.141592653) is -1.0
The tangent of (0.785398163) is 0.9999999992051033
The toDegrees of (3.141592653) is 179.99999996620733
The toRadians of (180) is 3.141592653589793

7.4 Exposure Java, 2005 Edition 08-06-05
7.2 User-Declared Class Methods

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, complete with 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? 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 step-by-step and this introduction is more comfortable on the neurons than the avalanche approach.

Program `Java0702.java`, in figure 7.2, displays a rather unimpressive mailing address. The three `println` statements could have been all placed in the main method, as they were in previous programs. 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, `Java0702`.

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 complexity of hundreds of program statements.

Figure 7.2
All your previous program examples always had a class. Java requires that programming is done with classes and an application program requires a **main** method. 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 **Java0703.java**, in figure 7.3, 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 be correct? Will that compile and execute?

**Figure 7.3**

```java
// Java0703.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, Java0703.

public class Java0703
{
    public static void main(String[] args)
    {
        System.out.println("JAVA0703.JAVA");
        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");
    }
}
```

**Java0703.java Output**

JAVA0703.JAVA
Kathy Smith
7003 Orleans Court
Kensington, Md. 20795

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 methods that are declared are all members of the **Java0703** 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 B116 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 J116. That makes sense. Now suppose that I have a letter on my desk for Tom Jones and I am in room J116. 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 J116. I am in room J116, the student delivering the letter is in room J116 and Tom Jones is in room J116. It does not seem necessary to add that piece of information to make the delivery possible.

Figure 7.4

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 *Java0704.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 *Java0704* class. This time *fullName*, *street* and *cityStateZip* are no longer members of the *Java0704* 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 should not be declared as **public**. Only the primary class with the same name as the file is public. *Java0704.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

```java
// Java0704.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 Java0704.

public class Java0704
{
    public static void main(String args[])
    {
        System.out.println("JAVA0704.JAVA");
        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");
    }
}

C:\Java\Progs07>javac Java0704.java
Java0704.java:12: cannot resolve symbol
symbol : method fullName ()
location: class Java0704
    fullName();
  ^
Java0704.java:13: cannot resolve symbol
symbol : method street ()
location: class Java0704
    street();
  ^
Java0704.java:14: cannot resolve symbol
symbol : method cityStateZip ()
location: class Java0704
    cityStateZip();
  ^
3 errors

Program Java0705.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

```java
// Java0705.java
// The problem of Java0704.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.

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");
    }
}

public class Java0705
{
    public static void main(String args[])
    {
        System.out.println("JAVA0705.JAVA");
        Address.fullName();
        Address.street();
        Address.cityStateZip();
        System.out.println();
    }
}
```

Java0705.java Output

JAVA0705.JAVA

Kathy Smith
7003 Orleans Court
Kensington, Md. 20795

7.3 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. Math.random() is one of the few methods, which did not use a parameter. As you looked at the program examples with user declared methods, you noticed that all these methods were void of parameters.

### Methods Calls With and Without Parameters

Parameter method example where **100** is the parameter or argument of the sqrt method:

Result = Math.sqrt(100);

No-Parameter method example:

RndNumber = Math.random();

You will find that most 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 Java0706.java, in figure 7.7, has a Demo class with four methods. You will note that the method declarations are identical to any other user-created methods you have seen previously with one addition. Inside the parentheses is a variable declaration. The variable declared in the method heading receives a copy of the values that are passed by the parameters of the method call. Look at the program example first and then we need to talk some more to make sure that you understand the difference between the parameters in the method call and the parameters in the method heading.

---

**Figure 7.7**

```java
// Java0706.java
// This program demonstrates how to write methods with parameters.

public class Java0706 {
```
public static void main(String args[]) {
    System.out.println("JAVA0706.JAVA\n");
    Demo.method1(); // no parameter method call
    Demo.method2(1000); // one int parameter method call
    Demo.method3(1000,2000); // two int parameters method call
    Demo.method4("Hans",30,3.575); // three different type parameters method call
    System.out.println();
}
}

class Demo {
    public static void method1() {
        System.out.println("Method1 has no parameters\n");
    }
    public static void method2(int intNum) {
        System.out.println("Method2 has one parameter, \n + intNum\n");
    }
    public static void method3(int intNum1, int intNum2) {
        System.out.println("Method3 has two parameters, \n + intNum1 + \n and \n + intNum2\n");
    }
    public static void method4(String studentName, int studentAge, double studentGPA) {
        System.out.println("Method4 has 3 parameters with three different types\n");
        System.out.println("Name: \n + studentName\n");
        System.out.println("Age: \n + studentAge\n");
        System.out.println("GPA: \n + studentGPA\n");
    }
}

Java0706.java Output

JAVA0706.JAVA
Method1 has no parameters
Method2 has one parameter, 1000
Method3 has two parameters, 1000 and 2000
Method4 has 3 parameters with three different types
Name: Hans
Age: 30
GPA: 3.575

We will start by examining the method calls. A method is called when the method identifier is used in a program statement. All method calls require the use of a set of parentheses following the method identifier. The parentheses are still required when no parameter information is passed to the method. If information
is passed, it must be placed between the parentheses, and this information must match the declaration of the called method.

Consider the declaration of method1, shown by itself in figure 7.8, along with the method call. No information is passed to the method. Parentheses are still used, but both the parentheses in the method call and the method heading are empty.

**Figure 7.8**

```java
public static void method1()
{
    System.out.println("Method1 has no parameters");
}

Demo.method1(); // no parameter method call
```

method2, in figure 7.9, has one int parameter. Both the method heading and the method call show a single parameter. There is a very important distinction. In the method heading the type of the parameter is declared. There is no such declaration in the method call.

**Figure 7.9**

```java
public static void method2(int intNum)
{
    System.out.println("Method2 has one parameter, "+ intNum);
}

Demo.method2(1000); // one int parameter method call
```

method3, in figure 7.10, continues the pattern with two int parameters. Make a very important observation here since you may think that variable declarations inside a method are the same as they are in a method parameter list. Inside a method, you can make a declaration like int Num1, Num2, but that is not
possible inside a parameter list, which requires that the data type \texttt{int} is used with each parameter identifier.

\textbf{Figure 7.10}

```
public static void method3(int intNum1, int intNum2)
{
    System.out.println("Method3 has two parameters, " + intNum1 + " and " + intNum2);
}
```

Demo.method3(1000,2000); // two int parameters method call

\textbf{Method3}, in figure 7.11, introduces another important parameter concept. In this case there are three parameters and each parameter is a different data type. Java has no problem with these different data types. The key is to make sure that the parameters in the method call match the parameters in the method heading.

\textbf{Figure 7.11}

```
public static void method4(String studentName, int studentAge, double studentGPA)
{
    System.out.println("Method4 has 3 parameters with three different types");
    System.out.println("Name:  " + studentName);
    System.out.println("Age:   " + studentAge);
    System.out.println("GPA:   " + studentGPA);
}
```

Demo.method4("Hans",30,3.575); // three different type parameters method call

\textbf{Important Rules About Using Methods With Parameters}

The number of parameters in the method call must match the number of parameters in the method heading.
The corresponding parameters in the method call must be the same type as the parameters in the heading.

The sequence of the parameters in the method call must match the sequence of the parameters in the heading.

The parameter identifiers in the method call may be the same identifier as the parameters in the heading or different.

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 Race1 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.

<table>
<thead>
<tr>
<th>Race1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
<td>US</td>
</tr>
<tr>
<td>GB</td>
<td>GB</td>
</tr>
<tr>
<td>FR</td>
<td>FR</td>
</tr>
<tr>
<td>NL</td>
<td></td>
</tr>
</tbody>
</table>

Race2 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 country men.

<table>
<thead>
<tr>
<th>Race2</th>
</tr>
</thead>
<tbody>
<tr>
<td>US</td>
</tr>
<tr>
<td>GB</td>
</tr>
<tr>
<td>FR</td>
</tr>
<tr>
<td>NL</td>
</tr>
</tbody>
</table>

Race3 is not a problem situation. This race demonstrates an analogy to help explain the naming 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.

<table>
<thead>
<tr>
<th>Race3</th>
</tr>
</thead>
<tbody>
<tr>
<td>US (John)</td>
</tr>
<tr>
<td>GB (George)</td>
</tr>
<tr>
<td>FR (Gerard)</td>
</tr>
<tr>
<td>NL (Hans)</td>
</tr>
</tbody>
</table>

The point of this analogy is that the names do not matter. What matters is that there are the same number of runners at the passing mark as there are 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.

### 7.4 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.changeChecking(1000.0);
```

The changeChecking 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. It is not possible to 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 methods in an actual class. In the last chapter all the classes and methods 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, to 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 Java0707.java, in figure 7.11, is sure to astound the most discriminating Computer Science student or even AP Computer Science student.

```java
// Java0707.java
// This program demonstrates how to use the <Calc> class with void methods.

public class Java0707
```
Program **Java0708.java**, in figure 7.12, 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, 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 7.12

```java
// Java0708.java
// This program demonstrates the difference between a
// void sum1 method and a return sum2 method.

public class Java0708
{
    public static void main(String args[])
    {
        System.out.println("JAVA0708.JAVA");
        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;
    }
}
```

Java0708.java Output

JAVA0708.JAVA
1000 + 100 = 1100
1000 + 100 = 1100

Program Java0709.java, in figure 7.13, demonstrates a variety of ways to call a return method. In this case getSum is called with an output display using println, with an assignment statement and with a boolean condition in a selection
statement. In each example you will note the value returned by the `getSum` method is used in the program statement.

Figure 7.13

```java
// Java0709.java
// This program demonstrates several ways to call a return method.

public class Java0709
{
    public static void main(String args[])
    {
        System.out.println("nJAVA0709AVA");
        System.out.println("Sum: "+ getSum(200,300));
        int sum = getSum(200,300);
        System.out.println("Sum: "+ sum);
        if (getSum(100,300) > 0)
            System.out.println("Sum > 0");
        else
            System.out.println("Sum <= 0");
        System.out.println();
    }

    public static int getSum(int n1, int n2)
    {
        int sum = n1 + n2;
        return sum;
    }
}
```

Java0709ava Output

JAVA0709.JAVA
Sum: 500
Sum: 500
Sum > 0

This section concludes by returning to the exciting `Calc` class. This time the methods are altered to return methods. For a calculation class return methods are
more commonly used. Program **Java0710.java**, in figure 7.14, shows the altered, and improved, version of the **Calc** class.

**Figure 7.14**

```java
// Java0710.java
// This program demonstrates the use of return methods by changing the previous void methods of the <Calc> class to return methods.

public class Java0710
{
    public static void main(String args[])
    {
        System.out.println("JAVA0710n");
        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));
    }
}

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;
    }
}
```

**Figure 7.14 Continued**

Java0710.java Output
7.5 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 demonstrates a Util, short for Utility, class. Do not get too excited, the Util is not all that exciting, but it does demonstrate the purpose of class methods very nicely. Program Java0711.java, in figure 7.15, 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 to help compute how to center text and right-justify text.

Figure 7.15
// Java0711.java
// This program demonstrates a user-declared <Util> with a variety
public class Java0711
{
    public static void main(String args[])
    {
        System.out.println("JAVA0711");
        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);
    }
}

JAVA0711java Output
JAVA0711
This message is left justified

This message is centered

This message is right justified

Can we do better than the previous Util class? Certainly, 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 **Java0712.java**, in figure 7.16, demonstrates the improvement.

**Figure 7.16**

```java
// Java0712.java
// This program adds the <heading> method to the <Util> class.

public class Java0712
{
    public static void main(String args[])
    {
        Util.heading("Leon Schram","Java0712.java","10-24-02","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();
        System.out.println(" ");
        System.out.println(" ");
        System.out.println(" ");
        System.out.println(" ");
        System.out.println(" ");
    }
}
```
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 \texttt{import} and there is an approach called \textit{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 \texttt{Java0713.java}, in figure 7.17, you will see that it is now a small program with one method. It uses the \texttt{Util} class, which is in a separate file, shown in figure 7.18. Both files are in the \texttt{Progs07} folder and everything will compile just nicely without a problem.

\textbf{Figure 7.17}

\begin{verbatim}
// Java0713.java
// This program is identical to Java0713.java with the <Util> class
// not included in this file.

class Java0713
{
    public static void main(String args[])
    {
        Util.heading("Leon Schram","Java0713.Java","10-24-02","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);
    }
}
\end{verbatim}

\textbf{Figure 7.18}

\begin{verbatim}
// 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);
        System.out.println();
    }

    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 void main(String args[])
    {
        Util.skip(2);
        Util.center("This message is centered");
        Util.skip(2);
        Util.rightJustify("This message is right justified");
        Util.skip(1);
    }
}
\end{verbatim}
7.6 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 present completed 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 add 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 mistake 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 `Java0714.java`, in figure 7.19. 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 7.19**

```java
// Java0714.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 Java0714 { 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("PAYROLL 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 = 0.100; break; default:
    f = 0.075; } i = g * f; h = g - i; System.out.println("Name: " + a);
    System.out.println("Hourly rate: " + output.format(c)); System.out.println("Hours worked: " + b);
    System.out.println("Dependants: " + 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("in/n");
} }
```

**Java0714.java Output**

PAYROLL CASE STUDY #1

Enter Name ===>>  Tom Jones
Enter Hours Worked ===>>  37
Enter Hourly Rate ===>>  8.75
Enter Dependents ===>>  2

Name:  Tom Jones
Hourly rate:  $8.75
Hours worked:  37.0
Dependants:  2
Tax rate:  $0.19
Regular pay:  $323.75
Overtime pay:  $0.00
Gross pay:  $323.75
Deductions:  $60.54
Net pay:  $263.21

**Payroll Case Study, Stage #2**

Stage #2 makes a modest, but very significant 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 that follows is identical to the first stage. Future stages will not include anymore program outputs. Each stage along the way features some improvement, but it does not alter the fundamental logic of the program.

Figure 7.20

```java
// Java0715.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 Java0715
{
    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("PAYROLL 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;
        }
    }
}
```
```java
    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("Dependants:   "+ 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));
    }
    }
```

---

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 `Java0716.java`, in figure 7.21, 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 7.21**

```java
// Java0716.java
// Payroll Case Study #3
// Stage 3 improves program readability by using meaningful identifiers.
import java.util.Scanner;
import java.text.*;
public class Java0716
{
    public static void main (String args[])
    {
        Scanner keyboard = new Scanner(System.in);
        String employeeName;
        double hoursWorked;
        double hourlyRate;
        int numDependants;
        double overtimeHours;
        double regularPay;
        double overtimePay;
        double taxRate;
        double grossPay;
        double taxDeductions;
        double netPay;
        DecimalFormat output = new DecimalFormat("$0.00");
        System.out.println("PAYROLL CASE STUDY #3");
        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    ===>>  ");
        numDependants = 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 (numDependants)
        {
            case 0 : taxRate = 0.295; break;
            case 1 : taxRate = 0.249; break;
            case 2 : taxRate = 0.187; break;
            // Other cases...
        }
    }
}
```
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
");
System.out.println("Name:       "+employeeName);
System.out.println("Hourly rate: "+output.format(hourlyRate));
System.out.println("Hours worked: "+hoursWorked);
System.out.println("Dependants: "+numDependants);
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));

Payroll Case Study, Stage #4

Program Java0717, in figure 7.22, 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 7.22

// Java0717.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 Java0717
{
   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 numDependants;  // 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 dependants, used for deduction computation
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");
employeeName = keyboard.nextLine();
hoursWorked = keyboard.nextDouble();
hourlyRate = keyboard.nextDouble();
umDependants = 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 (numDependants)
// compute proper tax rate based on declared dependants
// everybody gets 0.075 tax rate if dependants 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("Dependants:   "+ numDependants);
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

The first four stages showed improvements after the first stage, but there was no evidence of any type of object oriented programming. Some attempt was made with organization by grouping similar program statements together in commented segments. However, the program stuffed every program statement in the main method. It simply is not possible to create a large program by placing every statement in the main method. Furthermore, it is not very efficient.

Some program segments are needed very frequently. As a matter of fact, Java has tried to anticipate many of the programmer's needs and has created a large number of libraries with many methods that are ready to use. For instance, any time that it is necessary to round off a number you can use `Math.round(x)`. This is possible because some `round` method exists that can be used whenever necessary.

In stage #5 separate methods are used for each important program task. The only statements left in the main method are the variable declarations and the calls to each one of the methods. The comments that identified each one of the program segment have been removed to demonstrate that a program segment placed in a method with a well-chosen method identifier once again becomes an example of
self-commenting. The method calls have intentionally been shaded. This improvement sounds great, but there is a major problem, because the program does not compile.

Program **Java0718.java**, in figure 7.23, will indicate many compile errors. The errors are very similar and indicate that there are problems with the variables. Java does not seem to know where the variables are located. This problem actually is logical. The earlier program examples stuffed all the program statements - including the variable declarations - in the main method. Any program statement, which used some variable was happy. The variable was known and properly declared in the same block as the statements using the variables. Now the program has one large block, which is the **Java0718** class and lots of smaller blocks, each of which is a method.

You need to realize something if this was never clear earlier. Any variable declared within a block structure is only accessible inside the boundaries of that block. A block is indicated by starting and ending braces. You see these braces for a class, for a method and for multiple statements used by control structures. This basically means that all the variables are declared in the **main** method and therefore become local variables to the **main** method.

---

**Figure 7.23**

```java
// Java0718.java
// Payroll Case Study #5
// In all the previous stages the program statements were located in the <main> method.
// Stage #5 divides the program up in a series of <class> methods for improved program readability.
// The comments are intentionally removed to show that with well-chosen variables names and with
// user-defined methods, a program becomes very organized and readable.
// There are some problems, because the program does not compile, unable to find many variables.

import java.util.Scanner;
import java.text.*;

public class Java0718
{
    public static void main (String args[])
    {
        System.out.println("nPAYROLL CASE STUDY #5n");
        String employeeName;
        double hoursWorked;
        double hourlyRate;
        int numDependants;
        double overtimeHours;
        double regularPay;
        double overtimePay;
        double taxRate;
        double grossPay;
        double taxDeductions;
        double netPay;
        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    ===>>  ");
    numDependants = 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 (numDependants)
    {
    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("Name:       "+employeeName);
    System.out.println("Hourly rate:  "+output.format(hourlyRate));
    System.out.println("Hours worked: "+hoursWorked);
    System.out.println("Dependants:  "+numDependants);
    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));
Payroll Case Study, Stage #6

Stage #6 with Java0719.java, in figure 7.24, tries hard to cure the variable problem. If variables are only declared in the main method then these variables are unknown elsewhere in the program. So let us try another solution. Stage #6 declares any variable that is necessary in the method where the variable is used. Are we correct now? The compile errors change, but the program still has difficulties even though every method is provided with the necessary variables.

Figure 7.24

// Java0719.java
// Payroll Case Study #6
// Stage #6 tries to correct the variable problems of Stage #5. Variables are declared
// in every location where they are used to allow access to such variables.
// The idea seems logical, but the program now has different compile problems.

import java.util.Scanner;
import java.text.*;

public class Java0719
{
    public static void main (String args[])
    {
        System.out.println("PAYROLL CASE STUDY #6\n");
        enterData();
        computeGrosspay();
        computeDeductions();
        computeNetpay();
        printCheck();
    }

    public static void enterData()
    {
        String employeeName;
        double hoursWorked;
        double hourlyRate;
        int numDependants;
        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 Dependants    ===>>  ");
        numDependants = keyboard.nextInt();
        HourlyWages wage = new HourlyWages();
        System.out.println("Employee Name:  "+employeeName);
        System.out.println("Hours Worked:  "+hoursWorked);
        System.out.println("Hourly Rate:  "+hourlyRate);
        System.out.println("Dependants:  "+numDependants);
        wage.setEmployeeName(employeeName);
        wage.setHoursWorked(hoursWorked);
        wage.setHourlyRate(hourlyRate);
        wage.setNumDependants(numDependants);
        grossPay = wage.computeGrossPay();
        deductions = wage.computeDeductions();
        netPay = grossPay - deductions;
        printCheck();
    }

    public static void printCheck()
    {
        System.out.println("Check Details:");
        System.out.println("Gross Pay:  "+ output.format(grossPay));
        System.out.println("Deductions:   "+ output.format(deductions));
        System.out.println("Net pay:      "+ output.format(netPay));
        System.out.println("\n\n");
    }
}
public static void computeGrosspay()
{
    double hoursWorked;
    double hourlyRate;
    double overtimeHours;
    double regularPay;
    double overtimePay;
    double grossPay;

    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()
{
    int numDependants;
    double taxRate;
    double grossPay;
    double taxDeductions;

    switch (numDependants)
    {
        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()
{
    double grossPay;
    double taxDeductions;
    double netPay;
    netPay = grossPay - taxDeductions;
}

public static void printCheck()
{
    DecimalFormat output = new DecimalFormat("$0.00");
    String employeeName;
    double hoursWorked;
    double hourlyRate;
    int numDependants;
    double regularPay;
    double overtimePay;
    double taxRate;
    double grossPay;
    double taxDeductions;

    numDependants = keyboard.nextInt();
}
double netPay;

System.out.println("\n\n");
System.out.println("Name: \n\t\t" + employeeName);
System.out.println("Hourly rate: \n\t\t" + output.format(hourlyRate));
System.out.println("Hours worked: \n\t\t" + hoursWorked);
System.out.println("Dependants: \n\t\t" + numDependants);
System.out.println("Tax rate: \n\t\t" + taxRate);
System.out.println("Regular pay: \n\t\t" + output.format(regularPay));
System.out.println("Overtime pay: \n\t\t" + output.format(overtimePay));
System.out.println("Gross pay: \n\t\t" + output.format(grossPay));
System.out.println("Deductions: \n\t\t" + output.format(taxDeductions));
System.out.println("Net pay: \n\t\t" + output.format(netPay));
System.out.println("\n\n");

Payroll Case Study, Stage #7

The problem with the previous program is a lack of understanding what it means to use local variables. A variable declared inside any block - remember class, method or control structure block - is only local to that block. If you create a program and start with a variable in one method, followed by the exact same variable identifier in another method, you have two different variables. Sure the identifiers have the same name, but that does not matter.

Imagine the following scenario. A teacher explains parameter passing to Tom during second period. In fifth period there is another Tom and Tom is confused about parameter passing. The teacher informs 5th period Tom that she has already explained the topic to Tom. Our 5th period Tom is not too happy with such an explanation. Having the same name does not make you the same person. In other words, ten methods with ten locally declared boolean variables are ten totally different variables. The identical names mean nothing in this case. We need to look for another solution.

Ironically, the solution has been directly under our nose. Several times it was mentioned that a declared variable is accessible within the limits of its own block. The last couple of programs have declared variables inside a method and that is where the variable stayed. Program Java0720.java, in figure 7.25, declares any variable that is used in multiple methods in the class block. The shaded portion shows the group of variables that is within the boundaries of the Java0720 class, but not inside any method. Since all the methods are also part of the same Java0720 block, every method has access to these class variables. Now the program compiles and executes once again.
import java.util.Scanner;
import java.text.*;

public class Java0720 {
    static String employeeName;
    static double hoursWorked;
    static double hourlyRate;
    static int numDependants;
    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("PAYROLL CASE STUDY #7\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 Dependants    ===>>  ");
        numDependants = 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 (numDependants)
    {
        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("Dependants:   " + numDependants);
    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 #8

Stage #7 may seem nice, because it solved some problems of earlier stages, 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 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 the program sequence.
Program `Java0721.java` in figure 7.26, now has a separate `Payroll` class. The `Java0721` class is only concerned with the management of the program sequence. Notice also that all the variables have moved to the block limits of the `Payroll` class. This is a considerable improvement.

**Figure 7.26**

```java
// Java0721.java
// Payroll Case Study #8
// In Stage 8 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 Java0721
{
    public static void main (String args[])
    {
        System.out.println("PAYROLL CASE STUDY #8\n");
        Payroll.enterData();
        Payroll.computeGrosspay();
        Payroll.computeDeductions();
        Payroll.computeNetpay();
        Payroll.printCheck();
    }
}

class Payroll
{
    static String employeeName;
    static double hoursWorked;
    static double hourlyRate;
    static int numDependants;
    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 Dependants    ===>>  ");
        numDependants = 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 (numDependants)
{
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("Dependants:  " + numDependants);
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");
}

**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.

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.

7.7 Program Input with GUI Windows

The last section in this chapter does not fit the chapter title. It has nothing to do with creating class methods. In this section you will learn how to create some graphics windows for program input. You have already learned several graphics features, but all those features were output. You have also learned to do input in a text window, but that text style input does not work with graphics.

It is nice if you can do some GUI input stuff, but much of what you use will not be explained. Basically, consider the following: There is a JOptionPane class, which has a method for input called showInputDialog and another method for output called showMessageDialog. Both GUI methods are shown by Java0722.java, in figure 7.27. The program output shows three boxes that appear in sequence. They will not all three be visible at the same time.

Figure 7.27

// Java0722.java
// This program introduces program input using Swing dialog boxes.

import javax.swing.JOptionPane;
The last program demonstrated GUI input entering a first name string of characters and a last name string of characters. You learned that with the text input style it was possible to enter string, integer or double data types. This same process is not available with GUI input. Program `Java0723.java`, in figure 7.28, enters two pieces of information like the last program. The dialog boxes prompt for the entry of two numbers. These two inputs are `String` variables and require conversion with `Integer.parseInt`.

**Figure 7.28**

// Java0723.java
// This program shows that numerical program input with dialog boxes requires
// a conversion process in the same manner as text window input.
import javax.swing.JOptionPane;

public class Java0723
{
    public static void main (String args[])
    {
        String strNbr1 = JOptionPane.showInputDialog("Enter Number 1");
        String strNbr2 = JOptionPane.showInputDialog("Enter Number 2");
        int intNbr1 = Integer.parseInt(strNbr1);
        int intNbr2 = Integer.parseInt(strNbr2);
        int sum = intNbr1 + intNbr2;
        JOptionPane.showMessageDialog(null,intNbr1 + " + " + intNbr2 + " = " + sum);
        System.exit(0);
    }
}

The last two program examples provided GUI input and output dialog boxes for an application program. Program Java0724.java, in figure 7.29, repeats the last program as an applet program. You may be used to all applet programs including a paint method. This is true, but applets have several methods that are called automatically. The dialog windows require the init method, which is called before the paint method. The paint method is necessary to display graphics objects. The init method is used with dialog boxes.

This program creates the same dialog GUI boxes shown by the previous program. As an applet the small windows will be called applet windows, but otherwise they are identical in appearance.

Figure 7.29
This program is the same process as Java0723.java. This time the program is written as an Applet. Note that the program code is written in the <init> method. The <exit> method is necessary since the applet has no <paint> method.

```java
import javax.swing.*;

public class Java0724 extends JApplet {
    public void init() {
        String strNbr1 = JOptionPane.showInputDialog("Enter Number 1");
        String strNbr2 = JOptionPane.showInputDialog("Enter Number 2");
        int intNbr1 = Integer.parseInt(strNbr1);
        int intNbr2 = Integer.parseInt(strNbr2);
        int sum = intNbr1 + intNbr2;
        JOptionPane.showMessageDialog(null, intNbr1 + " + " + intNbr2 + " = " + sum);
        System.exit(0);
    }
}
```

This chapter will conclude by combining the dialog boxes in the init method with graphics output in the paint method. Program Java0725.java is an applet, in figure 7.30, uses the init method with a dialog input boxes to gather information about the size and the count of the random squares that will be displayed by the paint method.

**Figure 7.30**

```java
import javax.swing.*;
import java.awt.);
import java.util.*;
public class Java0725 extends JApplet {
    int size; // size of squares to be displayed
    int count; // the quantity of squares to be displayed

    public void init() {
        String str1 = JOptionPane.showInputDialog("Enter Square Size");
        String str2 = JOptionPane.showInputDialog("Enter Square Count");
        size = Integer.parseInt(str1);
        count = Integer.parseInt(str2);
    }

    public void paint(Graphics g) {
        Random rd = new Random(12345);
        for (int k = 1; k <= count; k++) {
            int x = rd.nextInt(800-size);
            int y = rd.nextInt(600-size);
            int red = rd.nextInt(256);
            int green = rd.nextInt(256);
            int blue = rd.nextInt(256);
            g.setColor(new Color(red, green, blue));
        }
    }
}
```
7.8 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 will be cleared up when creating object methods is demonstrated in the next chapter.

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.

---

**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:

- Program should use self-commenting identifiers
- Control structures and block structure 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.

The chapter concluded by demonstrating how to use GUI dialog boxes for program input. GUI input, like text input, is limited to strings of characters. If there is need for mathematical numbers, such information needs to be converted from a String variable to an `int` variable using `Integer.parseInt(String var.)`