Predefined Methods

Methods in Java are like automobile parts; they are building blocks. Methods are used to divide complicated programs into manageable pieces. There are both **predefined methods**, methods that are already written and provided by Java, and **user-defined methods,** methods that you create.

Using methods has several advantages:

• While working on one method, you can focus on just that part of the program and construct it, debug it, and perfect it.

• Different people can work on different methods simultaneously.

• If a method is needed in more than one place in a program, or in different programs, you can write it once and use it many times.

• Using methods greatly enhances the program's readability because it reduces the complexity of the method main.

Methods are often called modules. They are like miniature programs; you can put them together to form a larger program. When user-defined methods are discussed, you will see that this is the case. This ability is less apparent with predefined methods because their programming code is not available to us. However, because predefined methods are already written, you will learn these first so that you can use them when needed. To include a predefined method in your program(s), you only need to know how to use it.

In Java, predefined methods are organized as a collection of classes, called **class libraries**. For example, the class Math contains mathematical methods. Table 10-1 lists some of Java's predefined mathematical methods. The table gives the name of the method (in bold type), number of parameters, the data type of the parameters, and the method type. The method type is the data type of the value returned by the method. The table also shows how a method works. The class Math is contained in the package java.lang.

Table 1: Some Pre-Defined Mathematical Methods of Math Class (Malik, 2012)

Consider the following declarations:

```
double a = 3, b = 23, c = 34.45, d = 65.78;
int e = -57;
```

Method	Description	Example
`abs(x)`	Returns the absolute value of *x*	`Math.abs(d); //returns the value of 57`

`ceil(x)`	Returns a value of x of type double, which is the smallest integer value that is not less than *x*	`Math.ceil(c); //returns the value of 35.0`
		`Math.ceil(d); //returns the value of 66.0`

`exp(x)`	Returns Euler's number *e* (e^x) raised to the power of a double value, where *e* is approximately 2.7182818284590455	`Math.exp(3) //returns the value of 20.085536923187668`

`floor(x)`	Returns a value of *x* of type double, which is the largest integer value less than *x*	`Math.floor(c); //returns the value of 34.0`
		`Math.floor (d); //returns the value of 65.0`

`log(x)`	Returns a value of *x* of type double, which is the natural logarithm (base *e*) of *x*	`Math.log(a); //returns the value of 1.0986122886681098`
`log10(x)`	Returns a value of *x* of type double, which is the common logarithm (base 10) of *x*	`Math.log10(a); //returns the value of 0.47712125471966244`

`max(x, y)`	Returns the largest of *x* and *y*. If either of *x* and *y* is of type double, it returns a value of type double.	`Math.max(a, b); //returns the value of 23.0`
		`Math.max(12, a); //returns the value of 12.0`

`min(x, y)`	Returns the smallest of *x* and *y*.	`Math.min(a, b); //returns the value of 3.0`
		`Math.min(23, 12); //returns the value of 12`

`pow(x, y)`	Returns a value of type double, which is the value of the first argument raised to the power of the second argument (x^y)	`Math.pow(2, a); //returns the value of 8.0`

`round(x)`	Returns a value of long type that is the greater closest to *x*	`Math.round(c); //returns the value of 34`
		`Math.round(d); //returns the value of 66`

`sqrt(x)`	Returns a value of double type, which is the square root of *x*	`Math.sqrt(4); //returns the value of 2.0`
`cos(x)`	Returns the cosine of *x* measured in radians.	`Math.cos(34); //returns the value of -0.8485702747846052`
`sin(x)`	Returns the sine of *x* measured in radians	`Math.sin(34); //returns the value of 0.5290826861200238`
`tan(x)`	Returns the tangent of *x* measured in radians	`Math.tan(34); //returns the value of -0.1425465430742778`

When a method returns a value, the method can be used in an expression, for example, `int x = Math.abs(-57); // the return value is stored in variable x.`

When you use a method, you are said to invoke or call it. For example, your program invoked the `nextInt` method using objects of the `Scanner` class.

User-Defined Methods

User-Defined Methods Because Java does not provide every method that you will ever need, and designers cannot possibly know a user's specific needs, you must learn to write your own methods. User-defined methods in Java are classified into two categories:

• **Value-returning methods**—methods that have a return data type. These methods return a value of a specific data type using the return statement, which we will explain shortly.

• **Void methods**—methods that do not have a return data type. These methods do not use a return statement to return a value.

Value-Returning Methods

SYNTAX: VALUE-RETURNING METHOD

The syntax of a value-returning method is:

```
modifier(s) returnType methodName(formal parameter list)
{
    statements
}
```

In this syntax:

• **modifier(s)** indicates the visibility of the method, that is, where in a program the method can be used (called). Some of the modifiers are public, private, protected, static, abstract, and final. If you include more than one modifier, they must be separated with spaces. You can select one modifier among public, protected, and private. The modifier public specifies that the method can be called outside the class; the modifier private specifies that the method cannot be used outside the class. Similarly, you can choose one of the modifiers static or abstract.

• **returnType** is the type of value that the method returns. This type is also called the type of the value-returning method.

• **methodName** is a Java identifier, giving a name to the method.

• Statements enclosed between braces form the body of the method.

In Java, public, protected, private, static, and abstract are reserved words.

For example, for the method **abs** (*absolute*), the heading might look like:

```
public static int abs(int number)
```

Similarly, the method **abs** might have the following definition:

```
public static int abs(int number)
{
    if (number < 0)
        number = -number;

    return number;
}
```

The variable declared in the heading, within the parentheses, of the method **abs** is called the **formal parameter** of the method **abs**. Thus, the formal parameter of **abs** is **number**.

Another Example:

```
1  import java.util.*;
2  public class LargerNumber
3  {
4  static Scanner console = new Scanner(System.in);
5  public static void main(String[] args)
6  {
7  double num1;
8  double num2;
9
10 System.out.print(" Enter two "+ "numbers: ");
11 num1 = console.nextDouble();
12 num2 = console.nextDouble();
13 System.out.println();
14 System.out.println("The larger of "+ num1 + " and " + num2 + " is "+ larger(num1, num2));
15 }
16 public static double larger(double x, double y)
17 {
18 double max;
19 if (x >= y)
20 max = x;
21 else
22 max = y;
23 return max;
24 }
25 }
```

OUTPUT:

Parameter Passing

Java passed arguments, such as `int`, into functions by value. This means that any changes to the values of the parameters exist only within the scope of the function. When a function returns, the parameters are gone, and any changes to them are lost. The program below shows an example of passing a parameter by value. The values of the arguments remain the same, even after the method invocation:

```
public class PassByValue {
        public static void main (String[] args) {
                int x = 3;
                passMethod(x); //invoke passMethod with x as argument
                System.out.println("After invoking passMethod, x = " + x);
        }
        //change parameter in passMethod()
        public static void passMethod (int x) {
                x = 10;
                System.out.println("The value of x inside the passMethod is " + x);
        }
}
```

Sample Output:

```
The value of x inside the passMethod() is 10
After invoking passMethod, x = 3
```