

JAVA METHODS

METHODS

A Java method is similar to function in C/C++. It is a collection of statements that are grouped together to perform an operation. When you call the `System.out.println` method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

CREATING A METHOD

In general, a method has the following syntax:

```
modifier returnType methodName(list of parameters) {  
    // Method body;  
}
```

A method definition consists of a **method header** and a **method body**. Here are all the parts of a method:

- **Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.
- **Return Type:** A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`.
- **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.
- **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
- **Method Body:** The method body contains a collection of statements that define what the method does.

EXAMPLE

Here is the source code of the above defined method called `max()`. This method takes two parameters `num1` and `num2` and returns the maximum between the two:

```
/** Return the max between two numbers */  
public static int max(int num1, int num2) {
```

```

int result;
if (num1 > num2)
    result = num1;
else
    result = num2;

return result;
}

```

CALLING A METHOD

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30, 40);
```

If the method returns *void*, a call to the method must be a statement. For example, the method *println* returns *void*. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

EXAMPLE

Following is the example to demonstrate how to define a method and how to call it:

```

public class TestMax {
    /** Main method */
    public static void main(String[] args) {
        int i = 5;
        int j = 2;
        int k = max(i, j);
        System.out.println("The maximum between " + i +
            " and " + j + " is " + k);
    }

    /** Return the max between two numbers */
    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
            result = num1;
        else
            result = num2;

        return result;
    }
}

```

```
}  
}
```

This would produce the following result:

```
The maximum between 5 and 2 is 5
```

This program contains the *main* method and the *max* method. The *main* method is just like any other method except that it is invoked by the operating system (better: by JVM).

The main method's header is always the same, like the one in this example, with the modifiers `public` and `static`, return value type `void`, method name `main`, and a parameter of the `String[]` type. `String[]` indicates that the parameter is an array of `String`.

THE VOID KEYWORD

This section shows how to declare and invoke a void method. Following example gives a program that declares a method named `printGrade` and invokes it to print the grade for a given score.

EXAMPLE

```
public class TestVoidMethod {  
  
    public static void main(String[] args) {  
        printGrade(78.5);  
    }  
  
    public static void printGrade(double score) {  
        if (score >= 90.0) {  
            System.out.println('A');  
        }  
        else if (score >= 80.0) {  
            System.out.println('B');  
        }  
        else if (score >= 70.0) {  
            System.out.println('C');  
        }  
        else if (score >= 60.0) {  
            System.out.println('D');  
        }  
        else {  
            System.out.println('F');  
        }  
    }  
}
```

This would produce the following result:

```
C
```

Here the `printGrade` method is a void method. It does not return any value. A call to a void method must be a statement. So, it is invoked as a statement in line 3 in the main method. This statement is like any Java statement terminated with a semicolon.

PASSING PARAMETERS BY VALUES

When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as **parameter order association**.

For example, the following method prints a message `n` times:

```
public static void nPrintln(String message, int n) {
    for (int i = 0; i < n; i++)
        System.out.println(message);
}
```

Here, you can use `nPrintln("Hello", 3)` to print "Hello" three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter, "Hello", to the parameter, `message`; passes 3 to `n`; and prints "Hello" three times.

However, the statement `nPrintln(3, "Hello")` would be wrong.

When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

In fact, when passing arguments by value, a local copy of arguments are created. If values of parameters are changed inside the method, it means that only local copies of arguments are changed; original arguments stay unchanged.

For simplicity, Java programmers often say passing an argument `x` to a parameter `y`, which actually means passing the value of `x` to `y`.

EXAMPLE

Following is a program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The `swap` method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

```
public class TestPassByValue {

    public static void main(String[] args) {
        int num1 = 1;
        int num2 = 2;

        System.out.println("Before swap method, num1 is " +
            num1 + " and num2 is " + num2);
    }
}
```

```

        // Invoke the swap method
        swap(num1, num2);
        System.out.println("After swap method, num1 is " +
                            num1 + " and num2 is " + num2);
    }
    /** Method to swap two variables */
    public static void swap(int n1, int n2) {
        System.out.println("\t\tInside the swap method");
        System.out.println("\t\t\tBefore swapping n1 is " + n1
                            + " n2 is " + n2);

        // Swap n1 with n2
        int temp = n1;
        n1 = n2;
        n2 = temp;

        System.out.println("\t\t\tAfter swapping n1 is " + n1
                            + " n2 is " + n2);
    }
}

```

This would produce the following result:

```

Before swap method, num1 is 1 and num2 is 2
    Inside the swap method
        Before swapping n1 is 1 n2 is 2
        After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2

```

OVERLOADING METHODS

The *max* method that was used earlier works only with the *int* data type. But what if you need to find which of two floating-point numbers has the maximum value? Yes, we could create another method with different name, but it is quite uncomfortable and confusing. We would use different names for all types!

The solution is to create another method with the same name but different parameters, as shown in the following code:

```

public static double max(double num1, double num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}

```

If you call *max* with *int* parameters, the *max* method that expects *int* parameters will be invoked; if you call *max* with *double* parameters, the *max* method that expects *double* parameters will be invoked. This is referred to as method overloading; that is, two methods have the same name but different parameter lists within one class.

The Java compiler determines which method is used based on the method signature. Overloading methods can make programs clearer and more readable. Methods that perform closely related tasks should be given the same name.

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the compiler cannot determine the most specific match. This is referred to as ambiguous invocation.

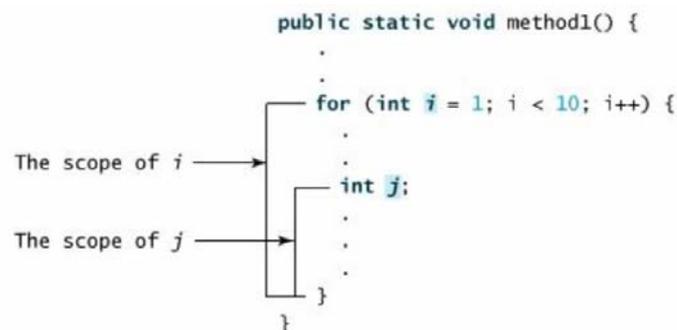
THE SCOPE OF VARIABLES

The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a local variable.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method.

A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable as shown below:



You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

USING COMMAND-LINE ARGUMENTS

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to `main()`.

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. They are stored as strings in the String array passed to `main()`.

EXAMPLE

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine {  
  
    public static void main(String args[]){  
        for(int i=0; i<args.length; i++){  
            System.out.println("args[" + i + "]: " +  
                               args[i]);  
        }  
    }  
}
```

Try executing this program as shown here:

```
java CommandLine this is a command line 200 -100
```

This would produce the following result:

```
args[0]: this  
args[1]: is  
args[2]: a  
args[3]: command  
args[4]: line  
args[5]: 200  
args[6]: -100
```

THE CONSTRUCTORS

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

EXAMPLE

Here is a simple example that uses a constructor:

```
// A simple constructor.  
class MyClass {  
    int x;  
  
    // Following is the constructor
```

```

    MyClass() {
        x = 10;
    }
}

```

You would call constructor to initialize objects as follows:

```

public class ConsDemo {

    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        System.out.println(t1.x + " " + t2.x);
    }
}

```

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

EXAMPLE

Here is a simple example that uses a constructor:

```

// A simple constructor.
class MyClass {
    int x;

    // Following is the constructor
    MyClass(int i ) {
        x = i;
    }
}

```

You would call constructor to initialize objects as follows:

```

public class ConsDemo {

    public static void main(String args[]) {
        MyClass t1 = new MyClass( 10 );
        MyClass t2 = new MyClass( 20 );
        System.out.println(t1.x + " " + t2.x);
    }
}

```

This would produce the following result:

```

10 20

```

VARIABLE ARGUMENTS(VAR-ARGS)

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

EXAMPLE

```
public class VarargsDemo {

    public static void main(String args[]) {
        // Call method with variable args
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }

    public static void printMax( double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");
            return;
        }

        double result = numbers[0];

        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result)
                result = numbers[i];
        System.out.println("The max value is " + result);
    }
}
```

This would produce the following result:

```
The max value is 56.5
The max value is 3.0
```

THE FINALIZE() METHOD

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called `finalize()`, and it can be used to ensure that an object terminates cleanly.

For example, you might use `finalize()` to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the `finalize()` method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the `finalize()` method, you will specify those actions that must be performed before an object is destroyed.

The *finalize()* method has this general form:

```
protected void finalize( )
{
    // finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to *finalize()* by code defined outside its class.

This means that you cannot know when or even if *finalize()* will be executed. For example, if your program ends before garbage collection occurs, *finalize()* will not execute.