JAVA - METHODS

A Java method 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, and apply method abstraction in the program design.

Creating Method:

Considering the following example to explain the syntax of a method:

```
public static int methodName(int a, int b) {
  // body
}
```

Here,

- **public static** : modifier.
- **int**: return type
- **methodName**: name of the method
- **a, b**: formal parameters
- **int a, int b**: list of parameters

Method definition consists of a method header and a method body. The same is shown below:

```
modifier returnType nameOfMethod (Parameter List) {
 // method body
}
```

The syntax shown above includes:

- **modifier:** It defines the access type of the method and it is optional to use.
- **returnType:** Method may return a value.
- **nameOfMethod:** This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List:** The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body:** The method body defines what the method does with statements.

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:

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;

   return min;
}
```

Method Calling:

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing *noreturnvalue*.

The process of method calling is simple. When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when:

- return statement is executed.
- reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Lets consider an example:

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

Example:

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

```
public class ExampleMinNumber{

   public static void main(String[] args) {
      int a = 11;
      int b = 6;
      int c = minFunction(a, b);
      System.out.println("Minimum Value = " + c);
   }

   /** returns the minimum of two numbers */
   public static int minFunction(int n1, int n2) {
      int min;
      if (n1 > n2)
         min = n2;
      else
         min = n1;

      return min;
   }
}
```

This would produce the following result:

```
inimum value = 6
```

The void Keyword:

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints*255.7;. It is a Java statement which ends with a semicolon as shown below.

Example:

```
public class ExampleVoid {

   public static void main(String[] args) {
      methodRankPoints(255.7);
```

```
    }

    public static void methodRankPoints(double points) {
       if (points >= 202.5) {
          System.out.println("Rank:A1");
       }
       else if (points >= 122.4) {
          System.out.println("Rank:A2");
       }
       else {
          System.out.println("Rank:A3");
       }
    }
}
```

This would produce the following result:

```
Rank:A1
```

Passing Parameters by Value:

While working under calling process, arguments is to be passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this the argument value is passed to the parameter.

Example:

The following program shows an example of passing parameter by value. The values of the arguments remains the same even after the method invocation.

```
public class swappingExample {

    public static void main(String[] args) {
       int a = 30;
       int b = 45;

       System.out.println("Before swapping, a = " +
                           a + " and b = " + b);

       // Invoke the swap method
       swapFunction(a, b);
       System.out.println("\n**Now, Before and After swapping values will be same
here**:");
       System.out.println("After swapping, a = " +
                          a + " and b is " + b);
    }

    public static void swapFunction(int a, int b) {

       System.out.println("Before swapping(Inside), a = " + a
                             + " b = " + b);
       // Swap n1 with n2
       int c = a;
       a = b;
       b = c;

       System.out.println("After swapping(Inside), a = " + a
                             + " b = " + b);
    }
}
```

This would produce the following result:

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

Method Overloading:

When a class has two or more methods by same name but different parameters, it is known as method overloading. It is different from overriding. In overriding a method has same method name, type, number of parameters etc.

Lets consider the example shown before for finding minimum numbers of integer type. If, lets say we want to find minimum number of double type. Then the concept of Overloading will be introduced to create two or more methods with the same name but different parameters.

The below example explains the same:

```
public class ExampleOverloading{

   public static void main(String[] args) {
      int a = 11;
      int b = 6;
      double c = 7.3;
      double d = 9.4;
      int result1 = minFunction(a, b);
      // same function name with different parameters
      double result2 = minFunction(c, d);
      System.out.println("Minimum Value = " + result1);
      System.out.println("Minimum Value = " + result2);
   }

  // for integer
   public static int minFunction(int n1, int n2) {
      int min;
      if (n1 > n2)
         min = n2;
      else
         min = n1;

      return min;
   }
   // for double
   public static double minFunction(double n1, double n2) {
     double min;
      if (n1 > n2)
         min = n2;
      else
         min = n1;

      return min;
   }
}
```

This would produce the following result:

```
inimum Value = 6
inimum Value = 7.3
```

Overloading methods makes program readable. Here, two methods are given same name but with different parameters. The minimum number from integer and double types is the result.

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 without parameters:

```
// 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);
   }
}
```

parametarized constructor

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 with parameter:

```
// 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);
   }
}
```

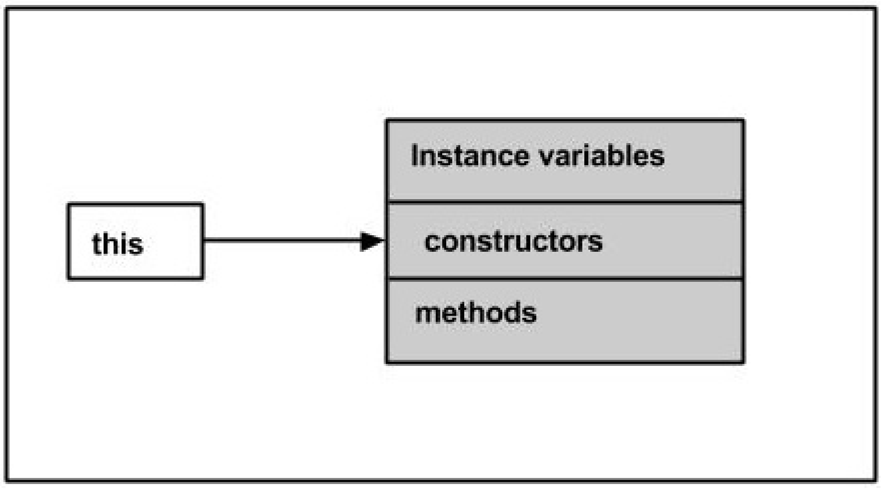
This would produce the following result:

```
10 20
```

The this keyword

this is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor. Using *this* you can refer the members of a class such as constructors, variables and methods.

NoteThe keyword *this* is used only within instance methods or constructors



In general the keyword *this* is used to :

- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student{

   int age;
   Student(int age){
   this.age=age;
   }

}
```

- Call one type of constructor*parametrizedconstructorordefault* from other in a class. It is known as explicit constructor invocation .

```
class Student{

   int age
   Student(){
   this(20);
   }

   Student(int age){
   this.age=age;
   }

}
```

Example

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the below given program in a file with name This_Example.java

```
public class This_Example {

   //Instance variable num
   int num=10;

   This_Example(){
      System.out.println("This is an example program on keyword this ");
   }

   This_Example(int num){
      //Invoking the default constructor
      this();

      //Assigning the local variable num to the instance variable num
      this.num=num;
   }

   public void greet(){
      System.out.println("Hi Welcome to Tutorialspoint");
   }

   public void print(){
      //Local variable num
      int num=20;

      //Printing the instance variable
      System.out.println("value of local variable num is : "+num);

      //Printing the local variable
      System.out.println("value of instance variable num is : "+this.num);
```

```
        //Invoking the greet method of a class
        this.greet();
    }

    public static void main(String[] args){
        //Instantiating the class
        This_Example obj1=new This_Example();

        //Invoking the print method
        obj1.print();

        //Passing a new value to the num variable through parametrized constructor
        This_Example obj2=new This_Example(30);

        //Invoking the print method again
        obj2.print();
    }

}
```

This would produce the following result:

```
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 10
Hi Welcome to Tutorialspoint
This is an example program on keyword this
value of local variable num is : 20
value of instance variable num is : 30
Hi Welcome to Tutorialspoint
```

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.

JAVA - PACKAGES

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Package can be defined as a grouping of related types *classes, interfaces, enumerationsandannotations* providing access protection and name space management.

Some of the existing packages in Java are::

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a package:

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements you have to do use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder

Example:

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes, interfaces.

Below given package example contains interface named *animals*:

```
/* File name : Animal.java */
package animals;
interface Animal {
   public void eat();
   public void travel();
}
```

Now, let us implement the above interface in the same package *animals*:

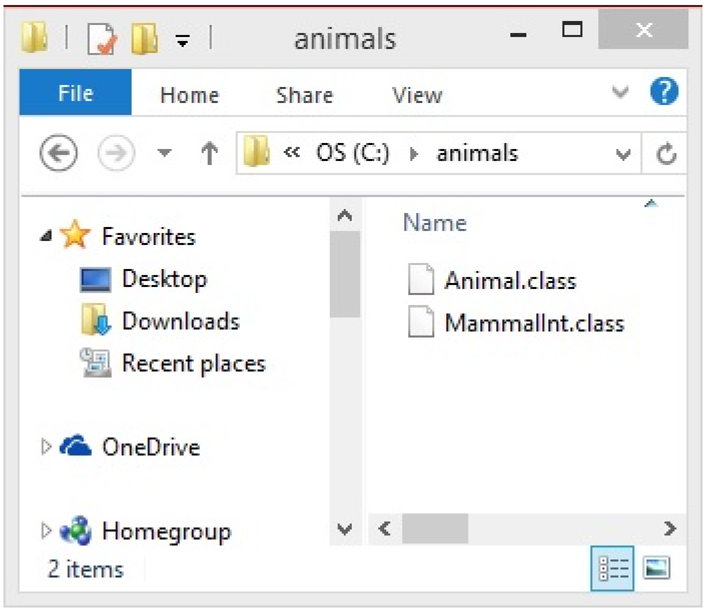
```
package animals;

/* File name : MammalInt.java */
```

```
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

Now compile the java files as shown below:

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown below.



Animal.class
MammalInt.class

You can execute the class file with in the package and get the result as shown below.

```
$ java animals.MammalInt
ammal eats
ammal travels
```

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;

public class Boss
{
   public void payEmployee(Employee e)
   {
      e.mailCheck();
   }
}
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card $*$. For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in Java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name :  Car.java

package vehicle;

public class Car {
   // Class implementation.
}
```

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:

```
....\vehicle\Car.java
```

Now, the qualified class name and pathname would be as below:

- Class name -> vehicle.Car
- Path name -> vehicle\Car.java $inwindows$

In general, a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**

For example:

```
// File Name: Dell.java

package com.apple.computers;
public class Dell{

}
class Ups{

}
```

Now, compile this file as follows using -d option:

```
$javac -d . Dell.java
```

This would put compiled files as follows:

```
.\com\apple\computers\Dell.class
.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in *\com\apple\computers\* as follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java

<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine *JVM* can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\compters.

A class path may include several paths. Multiple paths should be separated by a semicolon *Windows* or colon *Unix*. By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and UNIX

Bourneshell:

- In Windows -> C:\> set CLASSPATH
- In UNIX -> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use :

- In Windows -> C:\> set CLASSPATH=
- In UNIX -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH