

```
5  myMethod( obj );
6  String name = obj.getName(); // --- The name attribute was changed
   to 'Susan' inside the method
7  }
8  void myMethod( MyObjOne obj )
9  {
10     obj.setName("Susan");
11     ...
12     obj = new MyObjOne();
13     obj.setName("Sonya");
14     ...
15 }
```

On line 2, we created an object, on line 3 we set its name property to 'Christin'. On line 5 we called the myMethod(obj). Inside the method, we changed the name to 'Susan' through the passed in object reference. So that change will stay. Note however that after we reassigned the obj reference to a new object, that is no effect whatsoever on the passed in object.

Return Parameter

A method may or may not return a value. If the method does not return a value we use the **void** java keyword. Same as the parameter passing, the method can return a primitive type or an object reference. So a method can return only one value. What if you want to return more than one value from a method. You can always pass in an object reference to the method, and let the method modify the object properties. The modified values can be considered as an output value from the method. However better option, and cleaner if you create an Object array inside the method, assign the return values and return the array to the caller. You could have a problem however, if you want to mix primitive data types and object references as the output values from the method. There is a better approach. Defines special return object with the needed return values. Create that object inside the method, assign the values and return the reference to this object. This special object is "bound" to this method and used only for returning values, so do not use a public class. The best way is to use a nested class, see example below:

```
public class MyObject
{
    ...

    /** Nested object is for return values from 'getPersonInfoById'
method */
    public static class ReturnObj
    {
        private int    age;
        private String name;

        public void setAge( int val )
        {
            this.age = val;
        }

        public int getAge()
        {
            return age;
        }
    }
}
```

```
    }

    public void setName( String val )
    {
        name = val;
    }
    public String getName()
    {
        return name;
    }
} // --- End of nested class defination ---

/** Method using the nested class to return values */
public ReturnObj getPersonInfoById( int ID )
{
    int    age;
    String name;

    ...
    // --- Get the name and age based on the ID from the database ---
    ...
    ReturnObj ret = new ReturnObj();
    ret.setAge( age );
    ret.setName( name );

    return ret;
}
}
```

In the above example the 'getPersonInfoById' method returns an object reference that contains both values the name and age. See below how you may use that object:

```
{
    ...
    MyObject obj = new MyObject();
    MyObject.ReturnObj person = obj.getPersonInfoById( 102 );

    System.out.println( "Person Name=" + person.getName() );
    System.out.println( "Person Age =" + person.getAge() );
    ...
}
```

Special method, the Constructor

There is a special method for each class that will be executed each time an object is created from that class. That is the **Constructor**. Constructor does not have a return value and its name is the same as the class name. The Constructor can be overloaded; you can define more than one constructor with different parameters. For example:

```
public class MyClass
{
    private String memberVar;
    /**
     * MyClass Constructor, there is no input parameter
     */
    public MyClass()
    {
        ...
    }
    /**
     * MyClass Constructor, there is one input parameter
     */
    public MyClass( String param1 )
    {
        memberVar = param1;
        ...
    }
}
```

In the above example we defined two constructors, one with no input parameter, and one with one input parameter. You may ask which constructor will be called. It depends how the object is created with the **new** keyword. See below:

```
{
    ...
    MyClass obj1 = new MyClass();           // The constructor with
no input parameter will be called
    MyClass obj2 = new MyClass("Init Value"); // The constructor with
one input param. will be called
    ...
}
```

In the above example we created two objects from the same class, or we can also say that obj1 and obj2 both have the same type. The difference between the two is that in the first one the memberVar field is not initialized, in the second one that is initialized to 'Init Value'. obj1, and obj2 contains the reference to the object. Each class must have a constructor. If we do not define one, the compiler will create a default so called **empty constructor** automatically.

```
public class MyClass
{
    /**
     * MyClass Empty Constructor
     */
    public MyClass()
```