Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Functions and Parameters in Java/5b1aa72413ce7007ced9f1bc

TUTORIAL

Functions and Parameters in Java

Topics

Function is a piece of code that performs a specific task. Even though it is possible to write each and every line of code in the main function, but writing different functions for different tasks is easy way to programming. Functions provide many benefits while writing programs like reusability, easy understanding, easy debugging etc. Most languages allows to write our own functions. In the first program of Java, we described the a class and inside it a function called main().

Similarly we can define more functions in java as shown below:

```
class Shape
{
    int length, width;
    public void getData(int l, int w)
    {
        length = l;
        width = w;
    }
    public void putData()
    {
        System.out.println("Length is = " + length + " and Width
is = " + width);
    }
}
```

To create an object of this class and call these methods we can use a main class as shown below: -

```java
class Main
{
  public static void main(String[] args)
  {
    Shape s1 = new Shape();             // Creation of Objects
    Shape s2 = new Shape();
    s1.getData(5,10);
    s2.getData(12,6);                   // Calling functions on objects
    System.out.println("Shape1 information: ");
    s1.putData();
    System.out.println("Shape2 information: ");
    s2.putData();
  }
}
```

All methods have a visibility scope outside the class in which it is defined. If a method is defined as public anyone can call it on objects of that class. But we can define private methods also.

Private methods

When implementing a class, we make all data fields private because public data are dangerous. But what about the methods? While most methods are public, private methods are used in certain circumstances. Sometimes, you may wish to break up the code for a computation into separate helper methods. Typically, these helper methods should not become part of the public interface—they may be too close to the current implementation or require a special protocol or calling order. Such methods are best implemented as private.

To implement a private method in Java, simply change the public keyword to private. By making a method private, you are under no obligation to keep it available if you change to another implementation. The method may well be harder to implement or unnecessary if the data representation changes: this is irrelevant. The point is that as long as the method is private, the designers of the class can be assured that it is never used outside the other class

operations and can simply drop it. If a method is public, you cannot simply drop it because other code might rely on it.

Also we can define static methods

Static methods

Static methods are methods that do not operate on objects. For example, the pow method of the Math class is a static method. The expression

```
Math.pow(5, 3)
```

computes the power (5 raised to power 3). It does not use any Math object to carry out its task. So if we define a method as static, it can be called without creating the object of that class i.e. using class name. For example,

Java

```
class Dummy
{
  static void staticMethod()    // Static method
  {
    System.out.println("Code Quotient - Get better at coding.");
  }
  void nonStaticMethod()        // non static method
  {
    System.out.println("Non Static : Code Quotient - Get better at coding.");
  }
}

class Main
{
  public static void main(String[] args)
  {
    Dummy.staticMethod();       // calling static method

```

```
19      Dummy m1 = new Dummy();             // Creation of
    Object to call non static method.
20      m1.nonStaticMethod();
21    }
22  }
```

Parameter Passing to Methods

The term call by value means that the method gets just the value that the caller provides. In contrast, call by reference means that the method gets the location of the variable that the caller provides. Thus, a method can modify the value stored in a variable that is passed by reference but not in one that is passed by value.

The Java programming language always uses call by value. That means that the method gets a copy of all parameter values. In particular, the method cannot modify the contents of any parameter variables that are passed to it. For example, consider the following call:

```
double percent = 10;
harry.raiseSalary(percent);
```

No matter how the method is implemented, we know that after the method call, the value of percent is still 10. Let us look a little more closely at this situation. Suppose a method tried to triple the value of a method parameter:

```
public static void tripleValue(double x)
{                       // doesn't work
        x = 3 * x;
}
```

When we call this method:

```
double percent = 10;
tripleValue(percent);
```

However, this does not work. After the method call, the value of percent is still 10. Here is what happens:

1. x is initialized with a copy of the value of percent (that is, 10).

2. x is tripled—it is now 30. But percent is still 10.

3. The method ends, and the parameter variable x is no longer in use.

There are, however, two kinds of method parameters:

```
·        Primitive types (numbers, boolean values)
·        Object references
```

You have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object parameters. You can easily implement a method that triples the salary of an employee:

```
public static void tripleSalary(Employee x)
{                // works
        x.raiseSalary(200);
}
```

When you call

```
harry = new Employee(. . .);
tripleSalary(harry);
```

then the following happens:

1. x is initialized with a copy of the value of harry, that is, an object reference.

2. The raiseSalary method is applied to that object reference. The Employee object to which both x and harry refer gets its salary raised by 200 percent.

3. The method ends, and the parameter variable x is no longer in use. Of course, the object variable harry continues to refer to the object whose salary was tripled.

As you have seen, it is easily possible—and in fact very common—to implement methods that change the state of an object parameter. The reason is simple. The method gets a copy of the object

reference, and both the original and the copy refer to the same object. Java does not have Call by reference technique. To understand it an example is shown below. Let's try to write a method that swaps two employee objects:

```
public static void swap(Employee x, Employee y)
{                        // doesn't work
        Employee temp = x;
        x = y;
        y = temp;
}
```

If the Java programming language used call by reference for objects, this method would work:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?
```

However, the method does not actually change the object references that are stored in the variables a and b. The x and y parameters of the swap method are initialized with copies of these references. The method then proceeds to swap these copies.

```
// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice
```

But ultimately, this is a wasted effort. When the method ends, the parameter variables x and y are abandoned. The original variables a and b still refer to the same objects as they did before the method call. This discussion demonstrates that the Java programming language does not use call by reference for objects. Instead, object references are passed by value.