Methods in Java

Difficulty Level : Easy Last Updated : 14 Dec, 2021

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class that is different from languages like C, C++, and Python.

> ***Note:*** *Methods are time savers and help us to reuse the code without retyping the c ode.*

Modifiers are divided into *two* groups:

1. Access Modifiers controls the access level
2. Non-Access Modifiers do not control access level but provide other functionality

There are *four* types of Java access modifiers:

1. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package, and outside the package.

There are some Java Non-Access Modifiers:

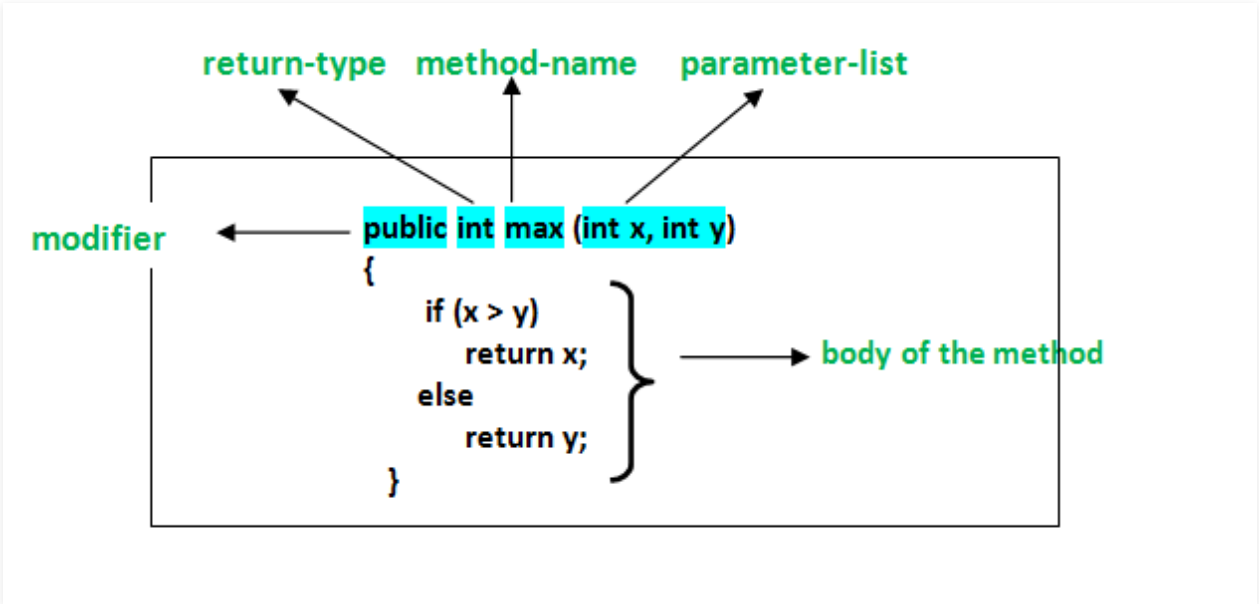
1. **static:** The member belongs to the class, not to objects of that class.
2. **final:** Variable values can’t be changed once assigned, methods can’t be overridden, classes can’t be inherited.
3. **abstract:** If applied to a method – has to be implemented in a subclass, if applied to a class – contains abstract methods
4. **synchronized:** Controls thread access to a block/method.

5. **volatile:** The variable value is always read from the main memory, not from a specific thread's memory.
6. **transient:** The member is skipped when serializing an object.

Method Declaration

In general, method declarations has six components :

- **Modifier**-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 types of access specifiers.
  - public: accessible in all classes in your application.
  - protected: accessible within the class in which it is defined and in its subclass/es
  - private: accessible only within the class in which it is defined.
  - default (declared/defined without using any modifier): accessible within the same class and package within which its class is defined.
- **The return type**: The data type of the value returned by the method or void if does not return a value.
- **Method Name**: the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list**: Comma-separated list of the input parameters is defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list**: The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body**: it is enclosed between braces. The code you need to be executed to perform your intended operations.



Method signature: It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.

Method Signature of above function:

```
max(int x, int y)
```

How to name a Method?: A method name is typically a single word that should be a **verb** in lowercase or multi-word, that begins with a **verb** in lowercase followed by an **adjective, noun…..** After the first word, the first letter of each word should be capitalized. For example, findSum,
computeMax, setX and getX

Generally, A method has a unique name within the class in which it is defined but sometimes a method might have the same name as other method names within the same class as method overloading is allowed in Java.

Method Calling

The method needs to be called for using its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

Example

```
// Java Program to Illustrate Methods

// Importing required classes
import java.io.*;

// Class 1
// Helper class
class Addition {

    // Initially taking sum as 0
    // as we have not started com[utation
```

```
    int sum = 0;

    // Method
    // To add two numbers
    public int addTwoInt(int a, int b)
    {

        // Adding two integer value
        sum = a + b;

        // Returning summation of two values
        return sum;
    }
}

// Class 2
// Helper class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of class 1 inside main() method
        Addition add = new Addition();

        // Calling method of above class
        // to add two integer
        // using instance created
        int s = add.addTwoInt(1, 2);

        // Printing the sum of two numbers
        System.out.println("Sum of two integer values :"
                           + s);
    }
}
```

Output

```
Sum of two integer values :3
```

Example 2

```
// Java Program to Illustrate Method Calling
// Via Different Ways of Calling a Method
```

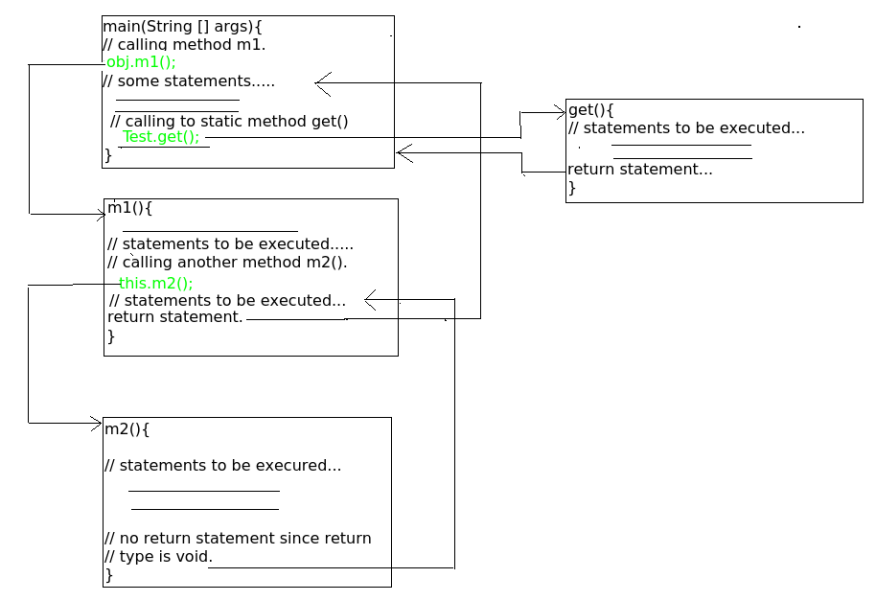
```
// Importing required classes
import java.io.*;

// Class 1
// Helper class
class Test {

    public static int i = 0;

    // Constructor of class
    Test()
    {

        // Counts the number of the objects of the class
        i++;
    }

    // Method 1
    // To access static members of the class and
    // and for getting total no of objects
    // of the same class created so far
    public static int get()
    {

        // statements to be executed....
        return i;
    }

    // Method 2
    // Instance method calling object directly
    // that is created inside another class 'GFG'.

    // Can also be called by object directly created in the
    // same class and from another method defined in the
    // same class and return integer value as return type is
    // int.
    public int m1()
    {

        // Display message only
        System.out.println(
            "Inside the method m1 by object of GFG class");

        // Calling m2() method within the same class.
        this.m2();

        // Statements to be executed if any
        return 1;
    }

    // Method 3
    // Returns nothing
    public void m2()
    {
```

```
            // Print statement
            System.out.println(
                "In method m2 came from method m1");
        }
    }

    // Class 2
    // Main class
    class GFG {

        // Main driver method
        public static void main(String[] args)
        {

            // Creating object of above class inside thi class
            Test obj = new Test();

            // Calling method 2 inside main() method
            int i = obj.m1();

            // Display message only
            System.out.println(
                "Control returned after method m1 :" + i);

            // Call m2() method
            // obj.m2();
            int no_of_objects = Test.get();

            // Print statement
            System.out.print(
                "No of instances created till now : ");

            System.out.println(no_of_objects);
        }
    }
```

Output

```
Inside the method m1 by object of GFG class
In method m2 came from method m1
Control returned after method m1 :1
No of instances created till now : 1
```

The control flow of the above program is as follows:



Memory allocation for methods calls

Methods calls are implemented through a stack. Whenever a method is called a stack frame is created within the stack area and after that, the arguments passed to and the local variables and value to be returned by this called method are stored in this stack frame and when execution of the called method is finished, the allocated stack frame would be deleted. There is a stack pointer register that tracks the top of the stack which is adjusted accordingly.

Related articles:

- Java is strictly passed by value
- Method overloading and Null error in Java
- Can we overload or override static methods in Java?
- Java Quizzes

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.