An **argument in Java** is an actual value that is passed to a method when the method is called.

Whenever any particular method is called during the execution of the program, there are some values that are passed to call that particular method. These values are called arguments.

The passed argument values replace those parameters which have been used during method definition and the method is then executed with these values.

The type of argument values passed must be matched with the type specified for the corresponding parameter in the definition of method. Sometimes, an argument is also called **actual parameter**.

For example:

```
add(5,7);
sum(35, 47);
```

The values 5 and 7 are the arguments with which a method will be called.

Parameter in Java

In Java, a parameter is a variable name with type that is declared within the method signature. The list of parameters is enclosed in parenthesis.

Each parameter consists of two parts: type name, and variable name. A type name followed by a variable name defines the type of value that can be passed to a method when it is called. It is also often called **formal parameter**.

Parameters declared in method signature are always local variables that receive values when the method is called.

For example:

```
1.  public int add(int a, int b)
    {
        return (a+b);
     }
```

The method add() has two parameters, named a and b with data type integer. It adds the values passed in the parameters and returns the result to the method caller.

```
2.  void sum(int x, int y)
```

The sum() method has two parameters x and y. While passing the argument values to the parameters, the order of parameters and number of parameters is very important. These must be in the same order as their respective parameters declared in the method declaration.

```
3. public static void main(String[ ] args )
   {
      . . . . . .
   }
```

In the main() method, args is a String array parameter.

4. void sub()
Here, the sub() method has no parameter. If a method has no parameters, then only an empty pair of parentheses is used.

Parameter Types

Any data type such as primitive data types including int, float, double, char, String, and object reference variables for a parameter of a method and constructor can be used as parameters.
There is no standard limit to specify the number of parameters in the definition of a method but you should limit the number of parameters almost 6 to 7 and any more will have a negative effect on the readability of your code.

Example:

```
1. public class Sum
2. {
3. public static void main(String[] args)
4. {
5. Sum obj = new Sum();
6. int x = obj.sum(20, 10);
7. System.out.println(x);
8. }
9. int sum (int a, int b)
10. {
11. int s = a + b;
12. return s;
13. }
14. }
```

Explanation with Diagram:

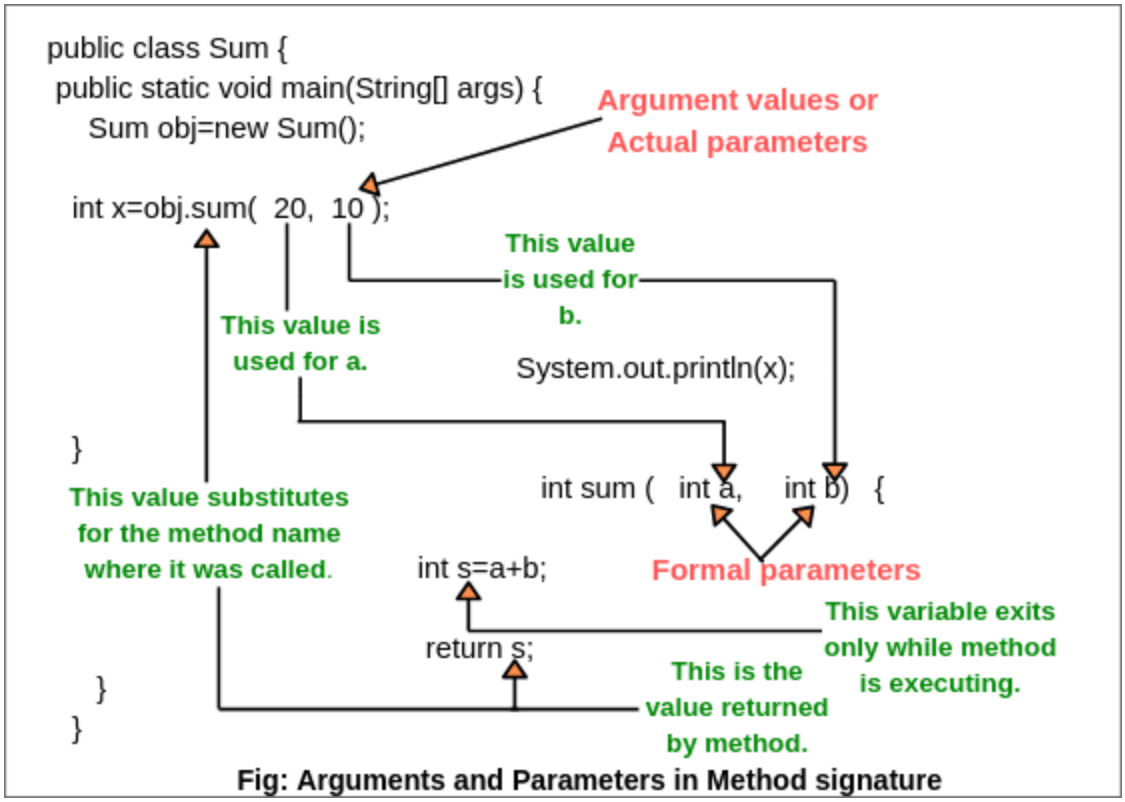

Fig: Arguments and Parameters in Method signature

In the preceding program and diagram, you can see that the method sum() has two Parameters, a and b, both of which are of type int.

So, both argument values must always be of type int. Since this method has not been defined as static, we can call it by creating an object of the class.

When we will call sum() method from another method called main(), the parameters defined within the method signature are replaced with the values of passing arguments and the method is then executed with these values.

The method sum() declares a variable s, which exists only within the body of the method. Each time this variable will be newly created when you will execute method and it will be destroyed when the execution of method ends.

All the variables that you declare within the body of a method, are considered as local variables. Variables declared within the body of a method are called **local variables**.

Difference between Argument and Parameter in Java

A parameter is a variable in the definition of a method whereas an argument is an actual value of this variable that is passed to the method's parameter.

A parameter is also called formal parameter whereas an argument is also called actual parameter.

During the time of call, each argument is always assigned to the parameter in the method definition whereas parameters are local variables that are assigned by value of the arguments when method is called.

Call by Value and Call by Reference in Java
In the previous tutorial, we looked at the difference between arguments and parameters in Java. Now, we need to be clear about how argument values are passed to a method in Java.

In Java, all argument values are passed to a method using only **call by value (pass by value)** mechanism. There is no call by reference (pass by reference) mechanism in Java. So, let's understand the mechanism "call-by-value" in detail.
Call by Value (Pass by Value) in Java

"Call by value" in Java means that argument's value is copied and is passed to the parameter list of a method. That is, when we call a method with passing argument values to the parameter list, these argument values are copied into the small portion of memory and a copy of each value is passed to the parameters of the called method.

When these values are used inside the method either for "read or write operations", we are actually using the copy of these values, not the original argument values which are unaffected by the operation inside the method.

That is, the values of the parameters can be modified only inside the scope of the method but such modification inside the method doesn't affect the original passing argument.

When the method returns, the parameters are gone and any changes to them are lost. This whole mechanism is called **call by value or pass by value**.
Let's understand the concept "pass by value" mechanism by example program and related diagram.
Example of Call by Value in Java
Let's create an example program where we will pass argument value to the parameter of the method using call by value.

```
package callbyValueExample;
1. public class Test
2. {
3. int x = 20;
4. void modify(int x)
5. {
6. x = x + 200;
7. System.out.println("Value of x after modification: " +x);
8. }
9. public static void main(String[] args)
10. {
11. Test t = new Test();
12. t.modify(t.x);
13. System.out.println("Original value of x: " +t.x);
14. }
15. }
```

Value of a after passing: 20

Value of x after modifying: 21

The below figure will illustrate how call by value works in the above program.

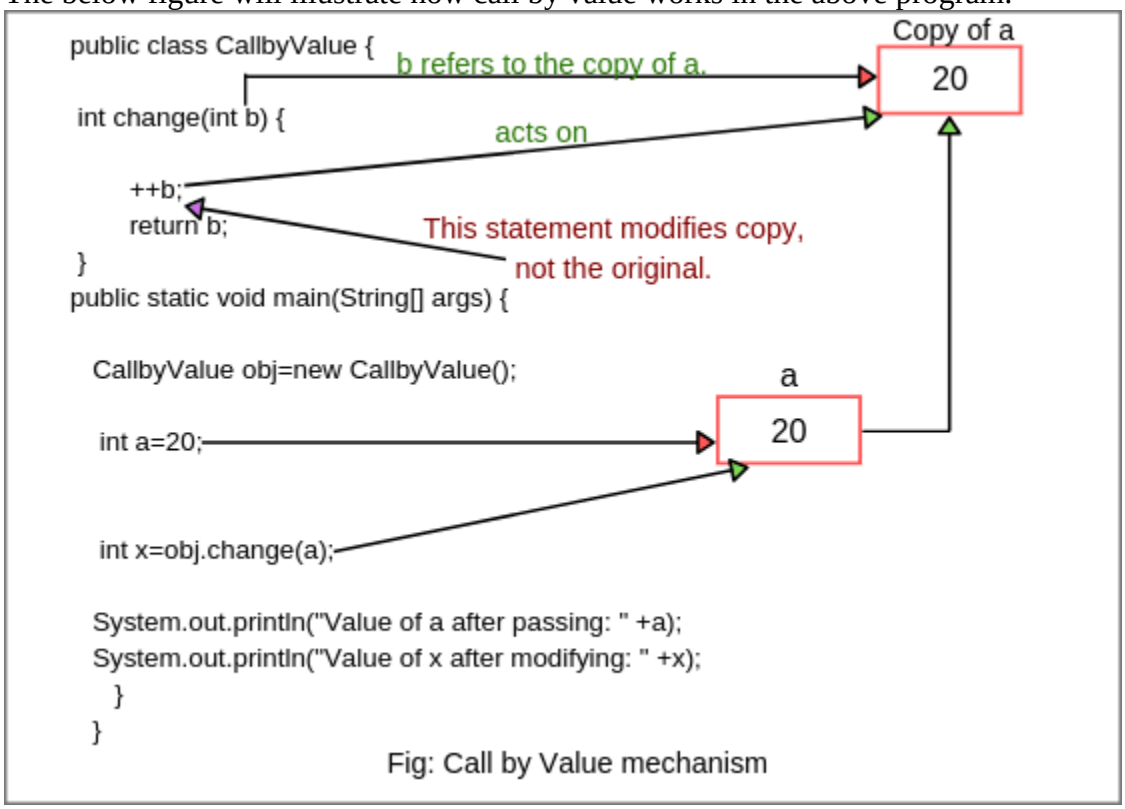

Fig: Call by Value mechanism

From the above figure, you can see that when we are passing argument value to a method, first, a copy of the value is automatically created into the small portion of memory and then a copy is passed to a method by referenced using parameter name “b”.

As shown in the above figure, the method change() will modify the copy of a, not the original value of “a”. That is, ++b will increment the copy of a. It will not increment the original value of a.

Thus, the value of b that will be returned, will be 21, not 20. Now, this value will be stored in the variable “x” when the return statement from the method will be executed.

However, the original value of a will remain at 20 because the modification does not affect the original argument.

Let’s take another example program based on Call by value mechanism.
Program source code 2:

```
package callbyValueExample;
1. public class Test
2. {
3. int x = 20;
4. void modify(int x)
5. {
6. x = x + 200;
```

```
7.  System.out.println("Value of x after modification: " +x);
8.  }
9.  public static void main(String[] args)
10. {
11. Test t = new Test();
12. t.modify(t.x);
13. System.out.println("Original value of x: " +t.x);
14. }
15. }
}
```

Call by Reference (Pass by Reference) in Java

In Java "**Call by Reference**" means passing a reference (i.e. address) of the object by value to a method. We know that a variable of class type contains a reference (i.e. address) to an object, not object itself.

Thus, when we pass a variable of class type as an argument to a method, actually, a copy of a reference (address) to an object is passed by value to the method, not a copy of the object itself because Java does not copy the object into the memory.

Actually, it copies reference of the object into the memory and passes the copy to the parameters of the called method. If we change the reference of the object then the original reference does not get changed because this reference is not original. It's a copy.

For example:

void m1(Emp e); where e is the object reference variable and Emp is the name of a class.

Key points:

Java passes the arguments both primitives and the copy of object reference by value. Java never passes object itself.

Let's take an example program to understand call by reference concept in Java.
In the above program, we passed integer values to the parameters of a method but in a real-time company project, we do not pass primitive values such as int, float, or double values to a method. We pass the reference (address) of the object as a value to the parameters of a method.

Suppose in a real-time project, we are developing a college application in which there are five modules such as Student, Library, Admin, Employee, and College.

We will create a class for each module and will pass variables of class type to call the method m1() and m2() in the school class. Let's see the following source code.

```
package callbyValueExample;
public class Student
{
```

```
 }
public class Library
{

 }
public class Admin
{

 }
public class Employee
{

 }
public class College
{
// Declare the instance method with two objects of Student and Library classes as parameters.
     void m1(Student s, Library l)
     {
      // s and l are object reference variables.
         System.out.println("m1 is calling.");
      }
// Similarly,
    public static void m2(Admin a, Employee ep)
    {
      System.out.println("m2 is calling");
    }
public static void main(String[] args)
{
// First, create an object of the class college.
    College c = new College();
// Now, create the object of the classes Student and Library.
     Student s = new Student(); // (1)
     Library l = new Library(); // (2)

// Pass object reference variables "s" and "l" as argument value to the method m1 for calling m1().
      c.m1(s, l); // (3)

// Above three lines of code 1, 2, and 3 can be replaced by a single line of code. Both are the same as working.
      c.m1(new Student(), new Library());
      Admin a = new Admin();
      Employee ep1 = new Employee();
      College.m2(a, ep1); // We can pass different Employee type reference variable. Reference Variable name is not important but Employee type is important. So, don't confuse in ep and ep1.

  // OR
      College.m2(new Admin(), new Employee());
   }
```

}
Output:

```
m1 is calling.
m1 is calling.
m2 is calling
m2 is calling
```

As you can see in the above program, we have passed the reference of objects of different classes as arguments by value to call m1() and m2() methods.