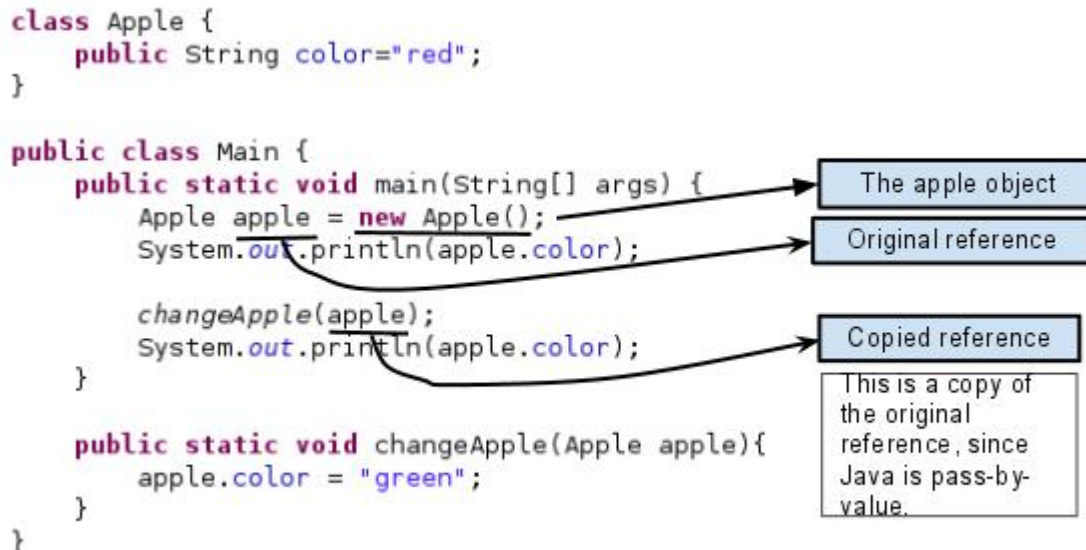


### 3. Passing Object Variable

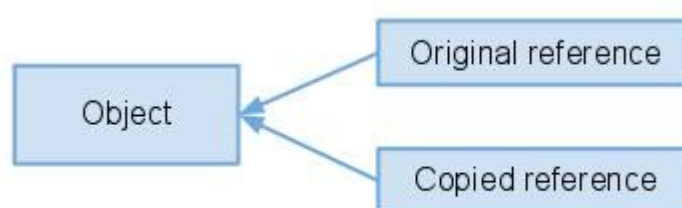
Java manipulates **objects by reference**, and all **object variables are references**.

However, **Java pass method arguments by value** and not by **Reference**

Question is: why the member value of the object can get changed?



Since the **original** and **copied** reference refer the **same object**, the **member value** gets **changed**.



<https://www.programcreek.com/2011/08/so-java-passes-object-by-reference-or-by-value/>

### Java Methods

What are Methods?

A method is a **set of code which is referred to by name** and can be **called (invoked)** at any point in a program simply by utilizing the method's name. Think of a method as a subprogram that acts on **data and often returns a value**.

Each method has its own name. When **that name is encountered in a program**, the execution of the program **branches to the body of that method**. When the method is **finished**, execution returns to the area of the **program code from which it was called**, and the program continues on to the **next line of code**.

How to invoke (call) a method (method invocation):

When a method is **invoked (called)**, a request is made to perform some action, such as setting a value, printing statements, returning an answer, etc. The **code to invoke the method contains the name of the method to be executed and any needed data that the receiving method requires**. The required **data for a method** are specified in the **method's parameter list**.

Consider this method that we have already been using from Breezy;

```
int number = Console.readInt("Enter a number"); //returns a value
```

The method `name is "readInt"` which is defined in the class `"Console"`. Since the method is defined in the class `Console`, the word `Console` becomes the calling object. This particular method returns an integer value which is assigned to an integer variable named `number`.

You invoke (call) a method by writing down the calling object followed by a dot, then the name of the method, and finally a set of parentheses that may (or may not) have information for the method.

<https://mathbits.com/MathBits/java/Methods/Lesson1.htm>

# Java Parameters

Java passes information to a method with parameters.

A copy of the *calling* parameter is assigned to the *receiving* parameter in the method.

Altering a simple data type parameter inside a method does not alter the value of the calling parameter.

Altering an object parameter inside a method alter the object value of the calling parameter, because objects pass the memory reference where object information is stored.

***It is true that strings are objects. However, strings are a special exception. Chapter 16 will deal with exactly what is going on with Strings. For now think of Strings as simple data types.***

<https://manifesto.co.uk/parameter-passing-in-java/>

[https://www.google.ch/search?](https://www.google.ch/search?q=calling+and+receiving+methods+in+java&dcr=0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiS0NbwqvPXAhWOH7QKHdjiCVAQ_AUICygC&biw=1280&bih=615#imgsrc=1eAOA5xaU_8VRM:)

[q=calling+and+receiving+methods+in+java&dcr=0&source=lnms&tbm=isch](https://www.google.ch/search?q=calling+and+receiving+methods+in+java&dcr=0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiS0NbwqvPXAhWOH7QKHdjiCVAQ_AUICygC&biw=1280&bih=615#imgsrc=1eAOA5xaU_8VRM:)


[&sa=X&ved=0ahUKEwiS0NbwqvPXAhWOH7QKHdjiCVAQ\\_AUICygC&biw=1280&](https://www.google.ch/search?q=calling+and+receiving+methods+in+java&dcr=0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiS0NbwqvPXAhWOH7QKHdjiCVAQ_AUICygC&biw=1280&bih=615#imgsrc=1eAOA5xaU_8VRM:)

[bih=615#imgsrc=1eAOA5xaU\\_8VRM:](https://www.google.ch/search?q=calling+and+receiving+methods+in+java&dcr=0&source=lnms&tbm=isch&sa=X&ved=0ahUKEwiS0NbwqvPXAhWOH7QKHdjiCVAQ_AUICygC&biw=1280&bih=615#imgsrc=1eAOA5xaU_8VRM:)

**Java tutorials**

<http://www.c4learn.com/java/java-declaring-object-in-class/>

# Parameter Passing in Java

by  Mario Martinez, Jul 14, 2015 | [Java development](#)

3 Min 0

One of the first things we learn about a programming language we just laid hands on is the way it manages the parameters when we call a method. The two most common ways that languages deal with this problem are called “*passing by value*” and “*passing by reference*”.

*Passing by value* means that, whenever a call to a method is made, the parameters are evaluated, and the result value is copied into a portion of memory. When the parameter is used inside the method, either for read or write, we are actually using the copy, not the original value which is unaffected by the operations inside the method.

On the other hand, when a programming language uses *passing by reference*, the changes over a parameter inside a method will affect the original value. This is because what the method is receiving is the reference, i.e the memory address, of the variable.

Some programming languages support *passing by value*, some support *passing by reference*, and some others support both. So the question is, what does Java support?

## Does Java pass parameters by value or by reference?

The answer to this question can be a little controversial, as there is some misunderstanding around how Java works this out. A lot of developers have the wrong idea that Java treats primitives and objects differently, so you often hear things like “*Java passes primitives by value and object by reference*”. Although,

this is not entirely true. The reality is that **Java always passes parameters by value.**

At this point, we could wonder where the confusion comes from. The explanation is that Java does things in its own special way.

If we look at how Java treats primitive parameters, it is clearly passing them by value, as there is nothing like a reference to a primitive in Java. But what happens for objects? Think about this snippet:

```
public static void main (String[] args) {  
    Person aPerson = new Person("Alice");  
    myMethod(aPerson);  
    System.out.println(aPerson.getName());  
}  
  
public static void myMethod(Person person) {  
    person.setName("Bob");  
}
```

It will print **"Bob"**, hence the **original variable has been modified**. If we go back to the definition of *"passing by reference"*, we can deduce that what is happening is that Java is passing a **reference to the variable aPerson**, therefore any modification **inside the method affects the original variable**.

Let's make a small change on our method code:

```
public static void myMethod(Person person) {  
    person = new Person("Bob");  
}
```

If the assumption we just made is right, and any modification on the parameter inside the method affects the original variable, this should print the same result as before (“Bob”). Although, if we execute this, the result is “Alice”.

## Java passes the reference of the object *by value*

So what is happening here? What Java really does with objects is pass the reference of the object *by value*. This is completely different from the proposition “Java passes object by reference”.

When we define something like `Person aPerson`, we are not defining an object `Person` itself, but a `pointer to` an object `Person`. When calling a method with `aPerson` as a parameter, Java doesn’t copy the `object`, copies the `reference` to the `object`. If we look back to the definition of passing by value, it matches this behaviour; we are copying the original value and passing the copy to the method. But remember, the original value is not the object itself but a reference to the object.

When `inside the method` we do `person.setName(“Bob”)`, we are `accessing the referenced object and changing it`. But when we do `person = new Person(“Bob”)`, what we are `changing is the reference to the object`, and remember that this reference is not the original, it is a copy, hence the original reference doesn’t get affected.

So, in summary, Java always passes parameter *by value* for both, primitives and object. When dealing with object, it passes the reference of the object *by value*, and not the object itself.

I hope this article helps you to have a better understanding on how Java works and to avoid confusion around the passing of parameters.