Contenido

[1. is-java-pass-by-reference-or-pass-by-value 1](#_Toc100153934)

# is-java-pass-by-reference-or-pass-by-value

Java is always **pass-by-value**. Unfortunately, when we deal with objects we are really dealing with object-handles called *references* which are passed-by-value as well. This terminology and semantics easily confuse many beginners.

It goes like this:

public static void main(String[] args) {

Dog aDog = new Dog("Max");

Dog oldDog = aDog;

// we pass the object to foo

foo(aDog);

// aDog variable is still pointing to the "Max" dog when foo(...) returns

aDog.getName().equals("Max"); // true

aDog.getName().equals("Fifi"); // false

aDog == oldDog; // true

}

public static void foo(Dog d) {

d.getName().equals("Max"); // true

// change d inside of foo() to point to a new Dog instance "Fifi"

d = new Dog("Fifi");

d.getName().equals("Fifi"); // true

}

In the example above aDog.getName() will still return "Max". The value aDog within main is not changed in the function foo with the Dog "Fifi" as the object reference is passed by value. If it were passed by reference, then the aDog.getName() in main would return "Fifi" after the call to foo.

Likewise:

public static void main(String[] args) {

Dog aDog = new Dog("Max");

Dog oldDog = aDog;

foo(aDog);

// when foo(...) returns, the name of the dog has been changed to "Fifi"

aDog.getName().equals("Fifi"); // true

// but it is still the same dog:

aDog == oldDog; // true

}

public static void foo(Dog d) {

d.getName().equals("Max"); // true

// this changes the name of d to be "Fifi"

d.setName("Fifi");

}

In the above example, Fifi is the dog's name after call to foo(aDog) because the object's name was set inside of foo(...). Any operations that foo performs on d are such that, for all practical purposes, they are performed on aDog, but it is **not** possible to change the value of the variable aDog itself.

For more information on pass by reference and pass by value, consult the following SO answer: <https://stackoverflow.com/a/430958/6005228>. This explains more thoroughly the semantics and history behind the two and also explains why Java and many other modern languages appear to do both in certain cases.