Is Java "pass-by-reference" or "pass-by-value"?

Asked 16 years, 3 months ago Modified 14 days ago

Viewed 2.8m times

7748

This question's answers are a community effort. Edit existing answers to improve this post. It is not currently accepting new answers or interactions.

I always thought Java uses **pass-by-reference**. However, I read a blog post which claims that Java uses **pass-by-value**. I don't think I understand the distinction the author is making.

What is the explanation?

java methods parameter-passing pass-by-reference pass-by-value

Share Follow

edited Feb 14, 2023 at 20:31

community wiki
26 revs, 22 users 12%
user4315

Comments disabled on deleted / locked posts / reviews

91 Answers

Sorted by: Highest score (default)

1 2 3 4 Next

7106

+150

The terms "pass-by-value" and "pass-by-reference" have special, precisely defined meanings in computer science. These meanings differ from the intuition many people have when first hearing the terms. Much of the confusion in this discussion seems to come from this fact.

The terms "pass-by-value" and "pass-by-reference" are talking about *variables.* Pass-by-value means that the *value* of a variable is passed to a function/method. Pass-by-reference means that a *reference* to that variable is passed to the function. The latter gives the function a way to change the contents of the variable.

By those definitions, Java is always **pass-by-value**. Unfortunately, when we deal with variables holding 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" do
        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
    with name member variable set to "Fifi"
        d = new Dog("Fifi");
        d.getName().equals("Fifi"); // true
    }
```

In this example, `aDog.getName()` will still return `"Max"`. The value `aDog` within `main` is not changed in the function `foo` by creating new `Dog` with name member variable set to `"Fifi"` because the object reference is passed by value. If the object reference was 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
        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 this example, `Fifi` is 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 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.

Share Improve this answer Follow

edited Sep 5 at 22:55

community wiki
38 revs, 31 users 29%
erlando

12 so what happens to "Fifi" in the 1st example? Does it cease to exist, was it never created, or does it exist in the heap but without a reference variable in the stack? – dbrewster Sep 29, 2020 at 18:03

116 To me, saying that an object's reference is passed by value is the same as saying that the object is passed by reference. I'm a Java novice, but I presume that (in

contrast) *primitive data* is pass by value. – user36800 Oct 27, 2020 at 18:46

27 @user36800: You're wrong. Did you work through the example with Fifi and look carefully through the results? Check that indeed `foo(aDog);` did **not** change `aDog` despite `foo` overwriting the value of `d`, showing that indeed all inputs to a function are passed by value. – user21820 Dec 23, 2020 at 15:03

23 @user36800: Well, both statements are wrong. To pass an object by reference would mean that if the function modifies the variable then it modifies the object itself. That is not what happens in Java; objects cannot be passed by reference, but instead one can only pass references as inputs to a function, and when a function performs `d = new Dog("Fifi");` it overwrites the input variable `d`, which **stores** a reference but is not 'the object passed by reference'. Contrast with `&d` in the function signature in C, which would be pass-by-reference. [cont] – user21820 Dec 28, 2020 at 5:49

74 @dbrewster i'm sorry but ... "Fifi" is not among us anymore – ghilesZ Feb 2, 2021 at 19:51

3614

I just noticed you referenced my article.

The Java Spec says that everything in Java is pass-by-value. There is no such thing as "pass-by-reference" in Java.

The key to understanding this is that something like

```
Dog myDog;
```

is *not* a Dog; it's actually a *pointer* to a Dog. The use of the term "reference" in Java is very misleading and is what causes most of the confusion here. What they call "references" act/feel more like what we'd call "pointers" in most other languages.

What that means, is when you have

```
Dog myDog = new Dog("Rover");
foo(myDog);
```

you're essentially passing the *address* of the created `Dog` object to the `foo` method.

(I say essentially because Java pointers/references aren't direct addresses, but it's easiest to think of them that way.)

Suppose the `Dog` object resides at memory address 42. This means we pass 42 to the method.

if the Method were defined as

```
public void foo(Dog someDog) {
    someDog.setName("Max");     // AAA
    someDog = new Dog("Fifi");  // BBB
    someDog.setName("Rowlf");   // CCC
}
```

let's look at what's happening.

- the parameter `someDog` is set to the value 42

- at line "AAA"
    - `someDog` is followed to the `Dog` it points to (the `Dog` object at address 42)
    - that `Dog` (the one at address 42) is asked to change his name to Max
- at line "BBB"
    - a new `Dog` is created. Let's say he's at address 74
    - we assign the parameter `someDog` to 74
- at line "CCC"
    - someDog is followed to the `Dog` it points to (the `Dog` object at address 74)
    - that `Dog` (the one at address 74) is asked to change his name to Rowlf
- then, we return

Now let's think about what happens outside the method:

Did `myDog` change?

There's the key.

Keeping in mind that `myDog` is a *pointer*, and not an actual `Dog`, the answer is NO. `myDog` still has the value 42; it's still pointing to the original `Dog` (but note that because of line "AAA", its name is now "Max" - still the same Dog; `myDog`'s value has not changed.)

It's perfectly valid to *follow* an address and change what's at the end of it; that does not change the variable, however.

Java works exactly like C. You can assign a pointer, pass the pointer to a method, follow the pointer in the method and change the data that was pointed to. However, the caller will not see any changes you make to where that pointer points. (In a language with pass-by-reference semantics, the method function *can* change the pointer and the caller will see that change.)

In C++, Ada, Pascal and other languages that support pass-by-reference, you can actually change the variable that was passed.

If Java had pass-by-reference semantics, the `foo` method we defined above would have changed where `myDog` was pointing when it assigned `someDog` on line BBB.

Think of reference parameters as being aliases for the variable passed in. When that alias is assigned, so is the variable that was passed in.

Update

A discussion in the comments warrants some clarification...

In C, you can write

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}

int x = 1;
int y = 2;
swap(&x, &y);
```

This is not a special case in C. Both languages use pass-by-value semantics. Here the call site is creating additional data structure to assist the function to access and manipulate data.

The function is being passed pointers to data, and follows those pointers to access and modify that data.

A similar approach in Java, where the caller sets up assisting structure, might be:

```
void swap(int[] x, int[] y) {
    int temp = x[0];
    x[0] = y[0];
    y[0] = temp;
}

int[] x = {1};
int[] y = {2};
swap(x, y);
```

(or if you wanted both examples to demonstrate features the other language doesn't have, create a mutable IntWrapper class to use in place of the arrays)

In these cases, both C and Java are *simulating* pass-by-reference. They're still both passing values (pointers to ints or arrays), and following those pointers inside the called function to manipulate the data.

Pass-by-reference is all about the function *declaration/definition*, and how it handles its parameters. Reference semantics apply to *every* call to that function, and the call site only needs to pass variables, no additional data structure.

These simulations require the call site and the function to cooperate. No doubt it's useful, but it's still pass-by-value.

Share Improve this answer Follow

edited Aug 12, 2022 at 17:42

community wiki
17 revs, 8 users 71%
Scott Stanchfield

3 @ebresie javarevisited.blogspot.com/2015/09/….
– Ravikumar Rajendran Mar 15, 2021 at 17:08

3 Java does not act exactly like C. If you pass a pointer to a function in C and modify where that pointer points to, the effect of reassigning that pointer is seen at the call sight, not just within the scope of the call. Seeking this behavior out of the languages is the purpose of the const keyword. Please stop saying java is just like C, because it's in many many fundamental ways entirely NOT c (or c++) and all you're doing is confusing people that do know C (or C++) and are trying to get a working overview of java. See:

courses.washington.edu/css342/zander/css332/passby.html
– Jonathan Jun 24, 2021 at 17:20

15 @Jonathan That link is C++, not C. C does not work that way. C is strictly pass by value, just like Java. If you pass a pointer to something, the *pointer* is the value that you can follow. You cannot change the pointer but can follow it and change the value it points to. If you re-point it, the caller does not see the change. In C++, you can pass a *reference* to something (seen in that page you reference as int&), which is similar to an alias; if you change it in a function/method it does actually change the object/primitive/pointer passed as an argument.
– Scott Stanchfield Jun 26, 2021 at 20:14

6 @Jonathan No. C only has pass-by-value (in your example you're passing the value of a pointer - see stackoverflow.com/questions/2229498/passing-by-reference-in-c for a good discussion, esp the answer by Ely). Pass-by-reference has a *very* specific meaning in compiler parlance. FYI - I'm a compiler guy and was on the ANSI C++ 98 committee... C++ has reference semantics; C does not. The difference is whether the actual argument can be modified. When you pass in &i the actual argument value is the address of i, not a reference to i. – Scott Stanchfield Jul 31, 2021 at 1:37

3 @Jonathan You're confusing 'reference' as a type with 'pass-by-reference'. There's a huge difference between "pass by a value by reference" (reference semantics) and "pass a pointer/reference by value" (value semantics). It's all about how you define the FORMAL parameters of a function/method, not the ACTUAL parameters. If the language gives you syntax to automatically de-reference values in the FORMAL parameters (the function/method signature), it has reference semantics. Creating a pointer to something in the ACTUAL arguments (the call site) is still only passing a value. – Scott Stanchfield Aug 9, 2021 at 8:39

2129

Java always passes arguments *by value*, NOT by reference.

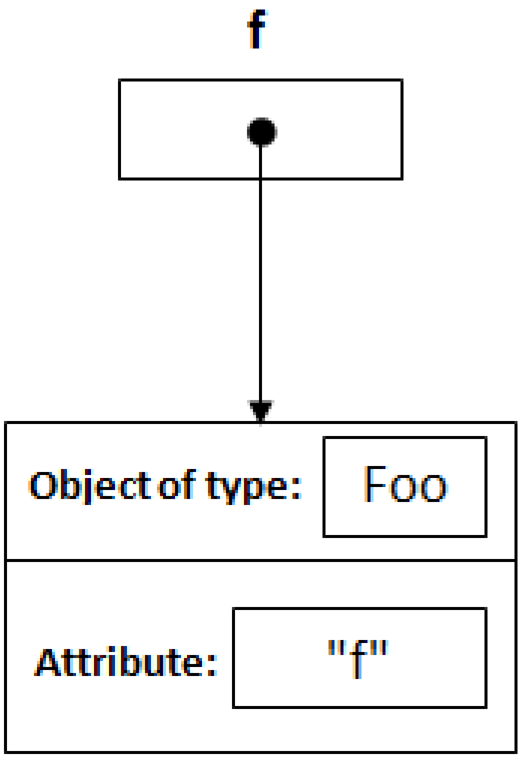
Let me explain this through an example:

```
public class Main {

     public static void main(String[] args) {
          Foo f = new Foo("f");
          changeReference(f); // It won't change the r
          modifyReference(f); // It will modify the ob
variable "f" refers to!
     }

     public static void changeReference(Foo a) {
          Foo b = new Foo("b");
          a = b;
     }

     public static void modifyReference(Foo c) {
          c.setAttribute("c");
     }
}
```

I will explain this in steps:

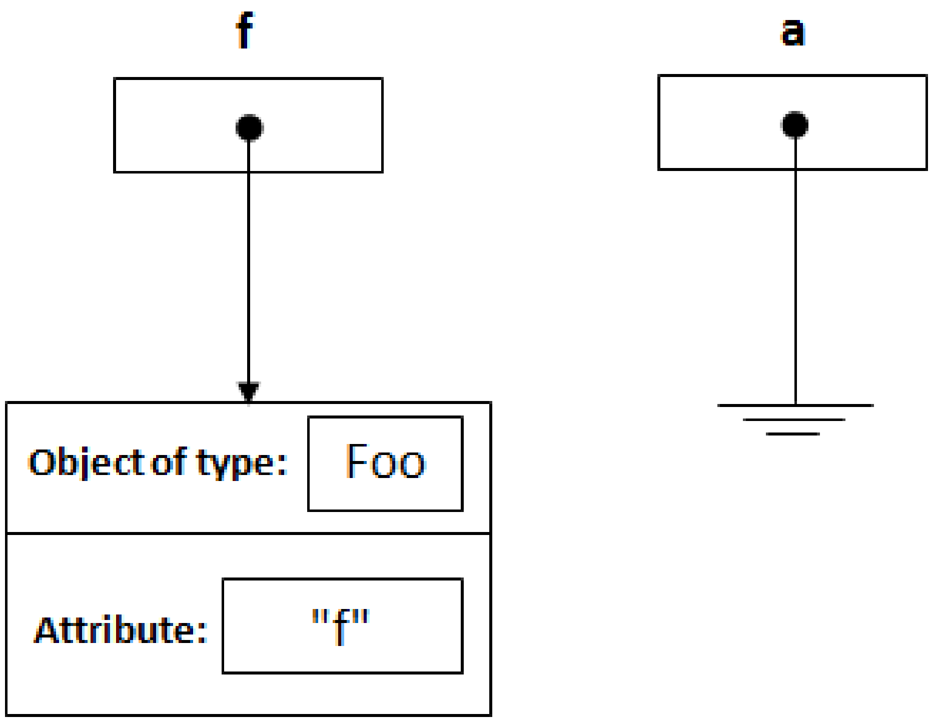
1. Declaring a reference named `f` of type `Foo` and assign it a new object of type `Foo` with an attribute `"f"`.

   ```
   Foo f = new Foo("f");
   ```



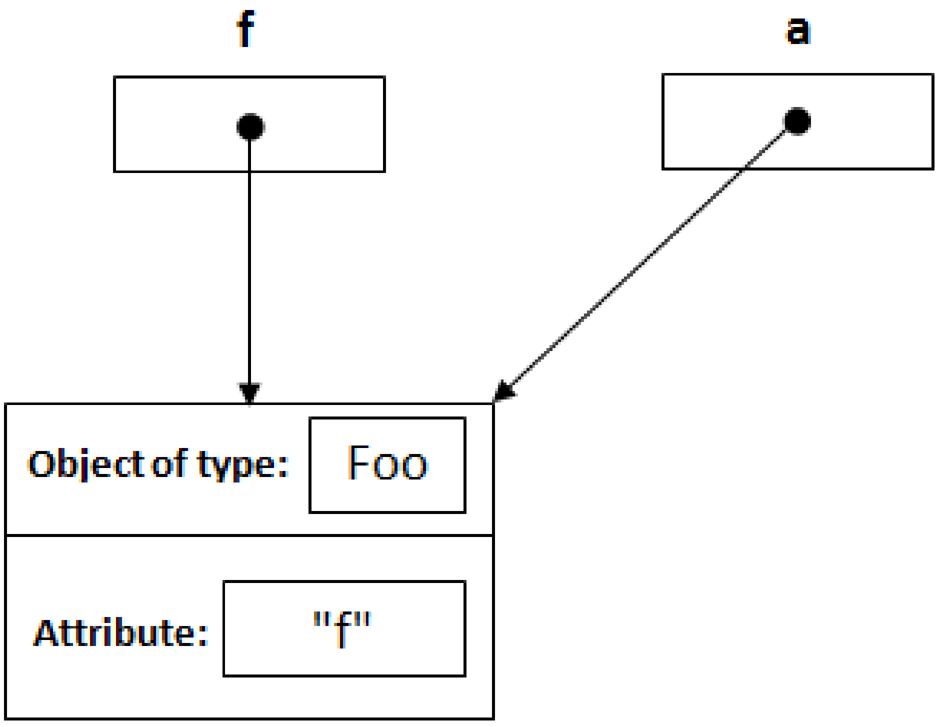
2. From the method side, a reference of type `Foo` with a name `a` is declared and it's initially assigned `null`.

   ```
   public static void changeReference(Foo a)
   ```



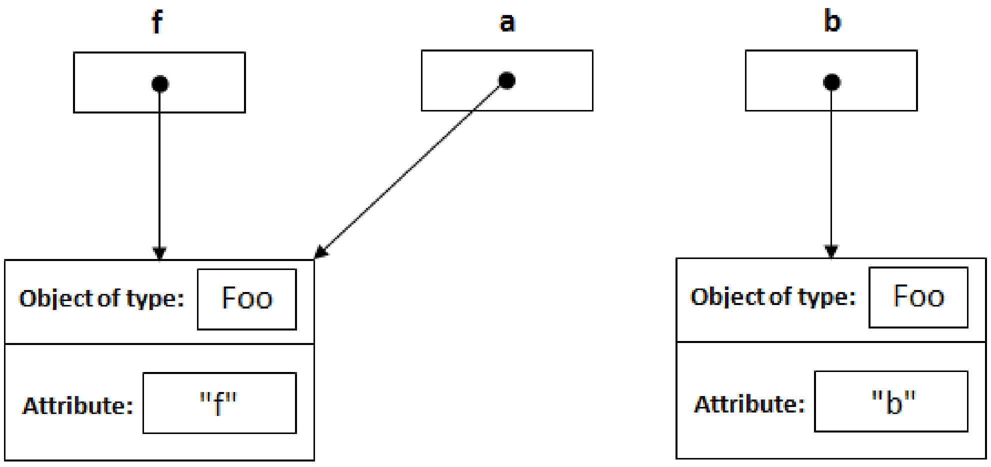
3. As you call the method `changeReference`, the reference `a` will be assigned the object which is passed as an argument.

   ```
   changeReference(f);
   ```

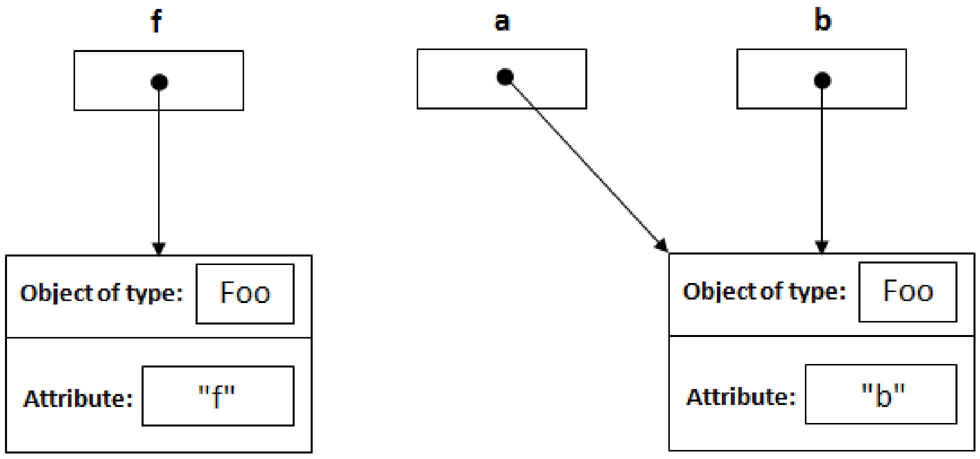


4. Declaring a reference named `b` of type `Foo` and assign it a new object of type `Foo` with an attribute `"b"`.

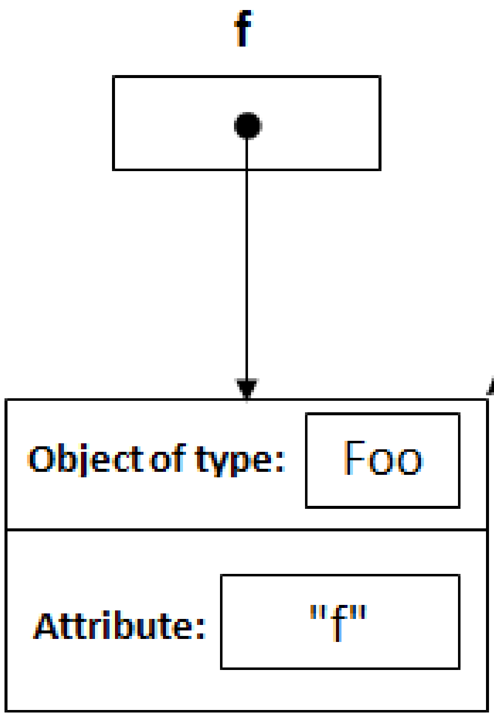
   ```
   Foo b = new Foo("b");
   ```



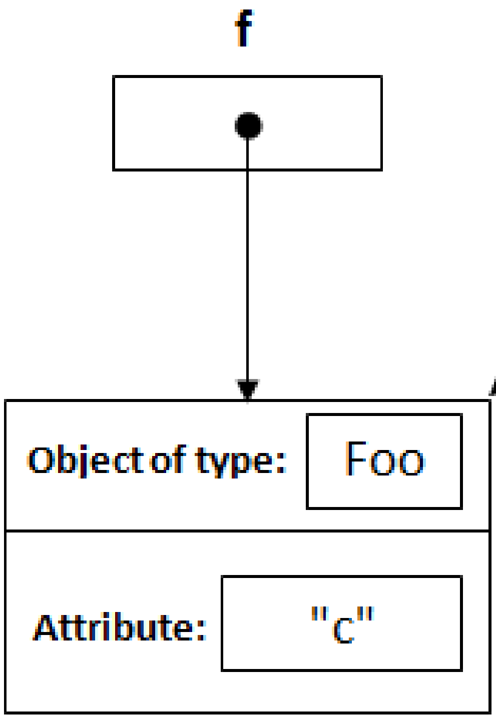
5. `a = b` makes a new assignment to the reference `a`, **not** `f`, of the object whose attribute is `"b"`.



6. As you call `modifyReference(Foo c)` method, a reference `c` is created and assigned the object with attribute `"f"`.



7. `c.setAttribute("c");` will change the attribute of the object that reference `c` points to it, and it's the same object that reference `f` points to it.



Share Improve this answer Follow

edited Jul 27, 2023 at 21:56

community wiki
9 revs, 9 users 73%
Eng.Fouad

82 Java always passes arguments by value, but what you are passing by value is a reference to an object, not a copy of the object. Simple eh? – dan carter Dec 7, 2020 at 8:48

1 "Object not by Reference", really? – Sam Ginrich Dec 4, 2021 at 12:47

1 This is the answer that made me clarify the question thanks to visually following the diagrams. – evaldeslacasa Dec 10 at 11:59

863

Java is always pass by value, with no exceptions, **ever**.

So how is it that anyone can be at all confused by this, and believe that Java is pass by reference, or think they have an example of Java acting as pass by reference? The key point is that Java **never** provides direct access to the values of *objects themselves*, in *any* circumstances. The only access to objects is through a *reference* to that object. Because Java objects are *always* accessed through a reference, rather than directly, it is common to talk about fields and variables *and method arguments* as being *objects*, when pedantically they are only *references to objects*. **The confusion stems from this (strictly speaking, incorrect) change in nomenclature.**

So, when calling a method

- For primitive arguments (`int`, `long`, etc.), the pass by value is *the actual value* of the primitive (for example, 3).
- For objects, the pass by value is the value of *the reference to the object*.

So if you have `doSomething(foo)` and `public void doSomething(Foo foo) { .. }` the two Foos have copied *references* that point to the same objects.

Naturally, passing by value a reference to an object looks very much like (and is indistinguishable in practice from) passing an object by reference.

Share Improve this answer Follow

edited Jan 22, 2016 at 3:37

community wiki
4 revs, 4 users 40%
SCdF

---

2 JVMS 2.2 makes this pretty clear: There are ... two kinds of values that can be stored in variables, passed as arguments, returned by methods, and operated upon: *primitive values* and *reference values*." Object references are values. Everything is passed by value. – Brian Goetz Nov 1, 2020 at 15:52

geeksforgeeks.org/g-fact-31-java-is-strictly-pass-by-value – georgiana_e Mar 2, 2021 at 7:28

The operative implication: `f(x)` (passing a variable) will never assign to `x` itself. There is no such thing as a variable address (*alias*) passed. *A solid language design decision.* – Joop Eggen Sep 20, 2021 at 9:37

So basically we're passing the address and we reference that address in our method for example in c `int test(int *a) { int b = *(a); return b;)` ? – bwass31 Feb 7, 2022 at 17:31

So, when I want to pass an object to some method, I'm doomed, because an object is "not a value" :(– Sam Ginrich Feb 16, 2022 at 15:34

803

This will give you some insights of how Java really works to the point that in your next discussion about Java passing by reference or passing by value you'll just smile :-)

Step one please erase from your mind that word that starts with 'p' "_ _ _ _ _ _ _", especially if you come from other programming languages. Java and 'p' cannot be written in the same book, forum, or even txt.

Step two remember that when you pass an Object into a method you're passing the Object reference and not the Object itself.

- *Student*: Master, does this mean that Java is pass-by-reference?
- *Master*: Grasshopper, No.

Now think of what an Object's reference/variable does/is:

1. **A variable holds the bits that tell the JVM how to get to the referenced Object in memory (Heap).**
2. When passing arguments to a method **you ARE NOT passing the reference variable, but a copy of the bits in the reference variable**. Something like this: 3bad086a. 3bad086a represents a way to get to the passed object.
3. So you're just passing 3bad086a that it's the value of the reference.
4. You're passing the value of the reference and not the reference itself (and not the object).
5. ***This value is actually COPIED and given to the method***.

In the following (please don't try to compile/execute this...):

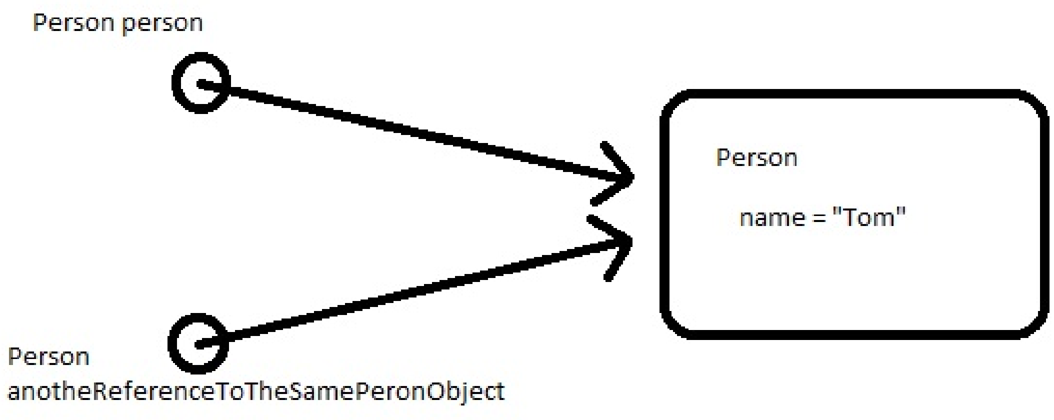
```
1. Person person;
2. person = new Person("Tom");
3. changeName(person);
4.
5. //I didn't use Person person below as an argument t
6. static void changeName(Person anotherReferenceToThe
7.     anotherReferenceToTheSamePersonObject.setName("
8. }
```

What happens?

- The variable *person* is created in line #1 and it's null at the beginning.

- A new Person Object is created in line #2, stored in memory, and the variable *person* is given the reference to the Person object. That is, its address. Let's say 3bad086a.
- The variable *person* holding the address of the Object is passed to the function in line #3.
- In line #4 you can listen to the sound of silence
- Check the comment on line #5
- A method local variable - *anotherReferenceToTheSamePersonObject*- is created and then comes the magic in line #6:
    - The variable/reference *person* is copied bit-by-bit and passed to *anotherReferenceToTheSamePersonObject* inside the function.
    - No new instances of Person are created.
    - Both "*person*" and "*anotherReferenceToTheSamePersonObject*" hold the same value of 3bad086a.
    - Don't try this but person==anotherReferenceToTheSamePersonObject would be true.
    - Both variables have IDENTICAL COPIES of the reference and they both refer to the same Person Object, the SAME Object on the Heap and NOT A COPY.

A picture is worth a thousand words:



Note that the anotherReferenceToTheSamePersonObject arrows is directed towards the Object and not towards the variable person!

If you didn't get it then just trust me and remember that it's better to say that **Java is pass by value**. Well, **pass by reference value**. Oh well, even better is ***pass-by-copy-of-the-variable-value! ;)***

Now feel free to hate me but note that given this **there is no difference between passing primitive data types and Objects** when talking about method arguments.

You always pass a copy of the bits of the value of the reference!

- If it's a primitive data type these bits will contain the value of the primitive data type itself.
- If it's an Object the bits will contain the value of the address that tells the JVM how to get to the Object.

> Java is pass-by-value because inside a method you can modify the referenced Object as much as you want but no matter how hard you try you'll never be able to modify the passed variable that will keep referencing (not p _ _ _ _ _ _ _) the same Object no matter what!

> The changeName function above will never be able to modify the actual content (the bit values) of the passed reference. In other word changeName cannot make Person person refer to another Object.

Of course you can cut it short and just say that **Java is pass-by-value!**

Share Improve this answer Follow

edited Jan 4, 2018 at 10:17

community wiki
3 revs, 2 users 95%
Gevorg

I tried this: <br /> File file = new File("C:/"); changeFile(file); System.out.println(file.getAbsolutePath()); } public static void changeFile(File f) { f = new File("D:/"); }` – Excessstone Jun 29, 2021 at 9:44

393

Java passes references by value.

So you can't change the reference that gets passed in.

Share Improve this answer Follow

answered Sep 2, 2008 at 20:20

community wiki
ScArcher2

Raises the question, whether Java is an Object Oriented of Reference Oriented language, rather than ´a mechanism for passing arguments´. en.wikipedia.org/wiki/Java_(programming_language)#Principles – Sam Ginrich Mar 8, 2022 at 12:48

303

I feel like arguing about **pass-by-reference** vs **pass-by-value** is not really helpful.

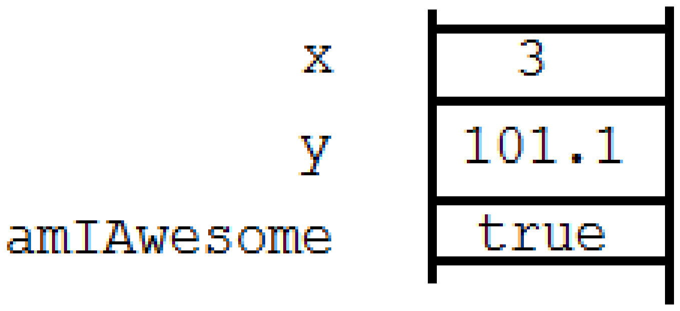
If you say that Java is **pass-by-whatever**, you are not providing a complete answer. Here is some additional information that will hopefully help you understand what actually happens in memory.

Crash course on stack/heap before we get to the Java implementation: Values go on and off the stack in a nice orderly fashion, like a stack of plates at a cafeteria. Memory in the heap (also known as dynamic memory) is haphazard and disorganized. The JVM just finds space wherever it can, and frees it up as the variables that use it are no longer needed.

Okay. First off, local primitives go on the stack. So this code:

```
int x = 3;
float y = 101.1f;
boolean amIAwesome = true;
```

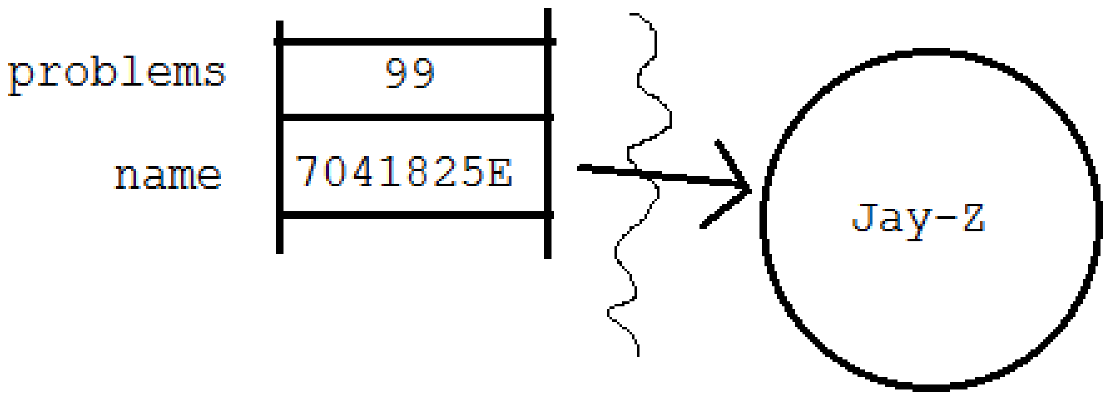
results in this:



When you declare and instantiate an object. The actual object goes on the heap. What goes on the stack? The address of the object on the heap. C++ programmers would call this a pointer, but some Java developers are against the word "pointer". Whatever. Just know that the address of the object goes on the stack.

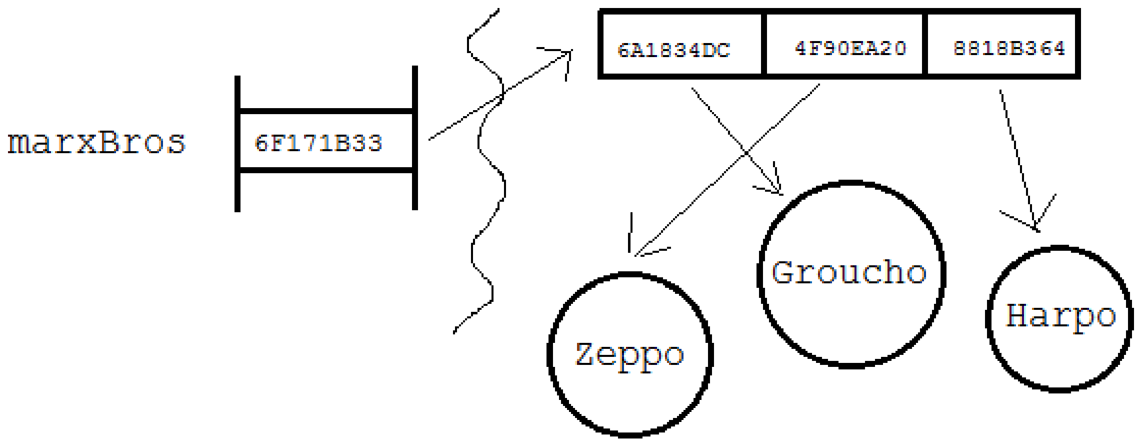
Like so:

```
int problems = 99;
String name = "Jay-Z";
```



An array is an object, so it goes on the heap as well. And what about the objects in the array? They get their own heap space, and the address of each object goes inside the array.

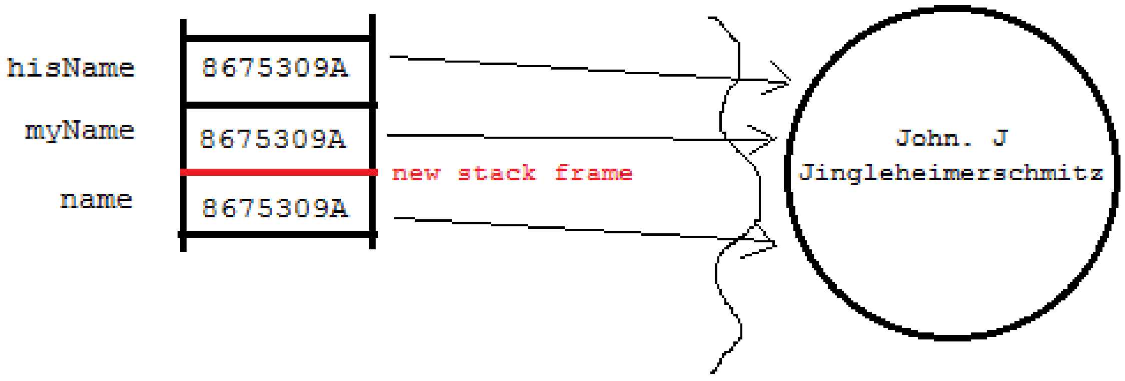
```
JButton[] marxBros = new JButton[3];
marxBros[0] = new JButton("Groucho");
marxBros[1] = new JButton("Zeppo");
marxBros[2] = new JButton("Harpo");
```



So, what gets passed in when you call a method? If you pass in an object, what you're actually passing in is the address of the object. Some might say the "value" of the address, and some say it's just a reference to the object. This is the genesis of the holy war between "reference" and "value" proponents. What you call it isn't as important as that you understand that what's getting passed in is the address to the object.

```
private static void shout(String name){
    System.out.println("There goes " + name + "!");
}

public static void main(String[] args){
    String hisName = "John J. Jingleheimerschmitz";
    String myName = hisName;
    shout(myName);
}
```

One String gets created and space for it is allocated in the heap, and the address to the string is stored on the stack and given the identifier `hisName`, since the address of the second String is the same as the first, no new String is created and no new heap space is allocated, but a new identifier is created on the stack. Then we call `shout()`: a new stack frame is created and a new identifier, `name` is created and assigned the address of the already-existing String.



So, value, reference? You say "potato".

Share Improve this answer edited Mar 12 at 23:27

Follow

community wiki
7 revs, 5 users 95%
cutmancometh

A reference is not a pointer, and it is not an address. It refers to the object via an intermediate data structure which contains *inter alia* the object's current address. This mechanism is required to make GC possible. – user207421
Mar 13 at 0:32

235

Basically, reassigning Object parameters doesn't affect the argument, e.g.,

```
private static void foo(Object bar) {
    bar = null;
}

public static void main(String[] args) {
    String baz = "Hah!";
    foo(baz);
    System.out.println(baz);
}
```

will print out `"Hah!"` instead of `null`. The reason this works is because `bar` is a copy of the value of `baz`, which is just a reference to `"Hah!"`. If it were the actual reference itself, then `foo` would have redefined `baz` to `null`.

Share Improve this answer edited Mar 7, 2022 at 7:47

Follow

community wiki
3 revs, 3 users 94%
Hank Gay

219

Just to show the contrast, compare the following C++ and Java snippets:

In C++: **Note: Bad code - memory leaks!** But it demonstrates the point.

```
void cppMethod(int val, int &ref, Dog obj, Dog &objRef
*&objPtrRef)
{
    val = 7; // Modifies the copy
    ref = 7; // Modifies the original variable
    obj.SetName("obj"); // Modifies the copy of Dog pa
    objRef.SetName("objRef"); // Modifies the original
    objPtr->SetName("objPtr"); // Modifies the origina
                               // by the copy of the p
    objPtr = new Dog("newObjPtr");  // Modifies the co
                                   // leaving the orig
    objPtrRef->SetName("objRefPtr"); // Modifies the o
                                    // by the original
    objPtrRef = new Dog("newObjPtrRef"); // Modifies t
passed
}

int main()
{
    int a = 0;
    int b = 0;
    Dog d0 = Dog("d0");
    Dog d1 = Dog("d1");
    Dog *d2 = new Dog("d2");
    Dog *d3 = new Dog("d3");
    cppMethod(a, b, d0, d1, d2, d3);
    // a is still set to 0
    // b is now set to 7
    // d0 still have name "d0"
    // d1 now has name "objRef"
    // d2 now has name "objPtr"
    // d3 now has name "newObjPtrRef"
}
```

In Java,

```
public static void javaMethod(int val, Dog objPtr)
{
   val = 7; // Modifies the copy
   objPtr.SetName("objPtr") // Modifies the original D
                            // by the copy of the poin
   objPtr = new Dog("newObjPtr");  // Modifies the cop
                                  // leaving the origi
}

public static void main()
{
    int a = 0;
    Dog d0 = new Dog("d0");
    javaMethod(a, d0);
    // a is still set to 0
    // d0 now has name "objPtr"
}
```

Java only has the two types of passing: by value for built-in types, and by value of the pointer for object types.

Share Improve this answer Follow

edited Sep 27, 2016 at 14:42

community wiki
4 revs, 4 users 94%
Eclipse

1 This shows that java is not pass by value as it doesn't copy the whole object onto the stack like C++ does, as shown in the example above - ..., Dog obj,... – Solubris Nov 25, 2020 at 23:03

3 No, Java passes references by value. That's why when you overwrite objPtr in the java example, the original Dog object doesn't change. But if modify the object being pointed to by objPtr, it does. – Eclipse Nov 26, 2020 at 22:03

212

Java passes references to objects by value.

Share Improve this answer Follow

answered Sep 2, 2008 at 20:23

community wiki
John Channing

What you mean is Java passes by copying the value to the reference. – skystar7 Jan 1, 2023 at 18:59

173

I can't believe that nobody mentioned Barbara Liskov yet. When she designed CLU in 1974, she ran into this same terminology problem, and she invented the term *call by sharing* (also known as *call by object-sharing* and *call by object*) for this specific case of "call by value where the value is a reference".

Share Improve this answer

Follow

answered Sep 7, 2010 at 22:07

community wiki
Jörg W Mittag

:) another term, feeding the confusion around the Java Island, just because it's politically incorrect to say "An Object is passed by reference, according to what we find on the stack". – Sam Ginrich Dec 6, 2021 at 18:11

141

The crux of the matter is that the word *reference* in the expression "pass by reference" means something completely different from the usual meaning of the word *reference* in Java.

Usually in Java *reference* means a a *reference to an object*. But the technical terms *pass by reference/value* from programming language theory is talking about a *reference to the memory cell holding the variable*, which is something completely different.

Share Improve this answer

Follow

edited Feb 6, 2018 at 10:18

community wiki
4 revs, 2 users 89%
JacquesB

Yes, a object reference is technically a handle, not yet the address, and so even a step further from "by value".

– Sam Ginrich Dec 6, 2021 at 18:14

111

There are already great answers that cover this. I wanted to make a small contribution by sharing a **very simple example** (which will compile) contrasting the behaviors between Pass-by-reference in c++ and Pass-by-value in Java.

A few points:

1. The term "reference" is a overloaded with two separate meanings. In Java it simply means a pointer, but in the context of "Pass-by-reference" it means a handle to the original variable which was passed in.

2. **Java is Pass-by-value**. Java is a descendent of C (among other languages). Before C, several (but not all) earlier languages like FORTRAN and COBOL supported PBR, but C did not. PBR allowed these other languages to make changes to the passed variables inside sub-routines. In order to accomplish the same thing (i.e. change the values of variables inside functions), C programmers passed pointers to variables into functions. Languages inspired by C, such as Java, borrowed this idea and continue to pass pointer to methods as C did, except that Java calls its pointers References. Again, this is a different use of the word "Reference" than in "Pass-By-Reference".

3. **C++ allows Pass-by-reference** by declaring a reference parameter using the "&" character (which happens to be the same character used to indicate "the address of a variable" in both C and C++). For example, if we pass in a pointer by reference, the parameter and the argument are not just pointing to the same object. Rather, they are the same variable. If one gets set to a different address or to null, so does the other.

4. In the C++ example below I'm passing a **pointer** to a null terminated string **by reference**. And in the Java example below I'm passing a Java reference to a String (again, the same as a pointer to a String) by value. Notice the output in the comments.

C++ pass by reference example:

```
using namespace std;
#include <iostream>

void change (char *&str){   // the '&' makes this a re
    str = NULL;
}

int main()
{
    char *str = "not Null";
    change(str);
    cout<<"str is " << str;      // ==>str is <null>
}
```

Java pass "a Java reference" by value example

```
public class ValueDemo{

    public void change (String str){
        str = null;
    }

     public static void main(String []args){
        ValueDemo vd = new ValueDemo();
        String str = "not null";
        vd.change(str);
        System.out.println("str is " + str);    // ==>
                                                // Not
                                                // pas
                                                // WOU
                                                // cal
     }
}
```

EDIT

Several people have written comments which seem to indicate that either they are not looking at my examples or they don't get the c++ example. Not sure where the disconnect is, but guessing the c++ example is not clear. I'm posting the same example in pascal because I think pass-by-reference looks cleaner in pascal, but I could be wrong. I might just be confusing people more; I hope not.

In pascal, parameters passed-by-reference are called "var parameters". In the procedure setToNil below, please note the keyword 'var' which precedes the parameter 'ptr'. When a pointer is passed to this procedure, it will be passed **by reference**. Note the behavior: when this procedure sets ptr to nil (that's pascal speak for NULL), it will set the argument to nil--you can't do that in Java.

```
program passByRefDemo;
type
   iptr = ^integer;
var
   ptr: iptr;

   procedure setToNil(var ptr : iptr);
   begin
       ptr := nil;
   end;

begin
   new(ptr);
   ptr^ := 10;
   setToNil(ptr);
   if (ptr = nil) then
       writeln('ptr seems to be nil');     { ptr shoul
will run. }
end.
```

EDIT 2

Some excerpts from **"THE Java Programming Language"** by Ken Arnold, **James Gosling (the guy who invented Java)**, and David Holmes, chapter 2, section 2.6.5

> **All parameters to methods are passed "by value"**. In other words, values of parameter variables in a method are copies of the invoker specified as arguments.

He goes on to make the same point regarding objects . . .

> You should note that when the parameter is an object reference, it is the object reference-not the object itself-that is **passed "by value"**.

And towards the end of the same section he makes a broader statement about java being only pass by value and never pass by reference.

> The Java programming language **does not pass objects by reference; it passes object references by value**. Because two copies of the same reference refer to the same actual object, changes made through one reference variable are visible through the other. There is exactly one parameter passing mode-**pass by value**-and that helps keep things simple.

This section of the book has a great explanation of parameter passing in Java and of the distinction between pass-by-reference and pass-by-value and it's by the creator of Java. I would encourage anyone to read it, especially if you're still not convinced.

I think the difference between the two models is very subtle and unless you've done programming where you actually used pass-by-reference, it's easy to miss where two models differ.

I hope this settles the debate, but probably won't.

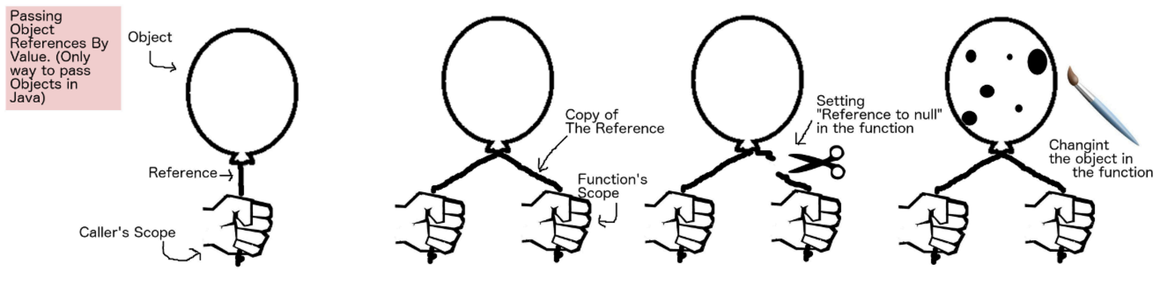
EDIT 3

I might be a little obsessed with this post. Probably because I feel that the makers of Java inadvertently spread misinformation. If instead of using the word "reference" for pointers they had used something else, say dingleberry, there would've been no problem. You could say, "Java passes dingleberries by value and not by reference", and nobody would be confused.

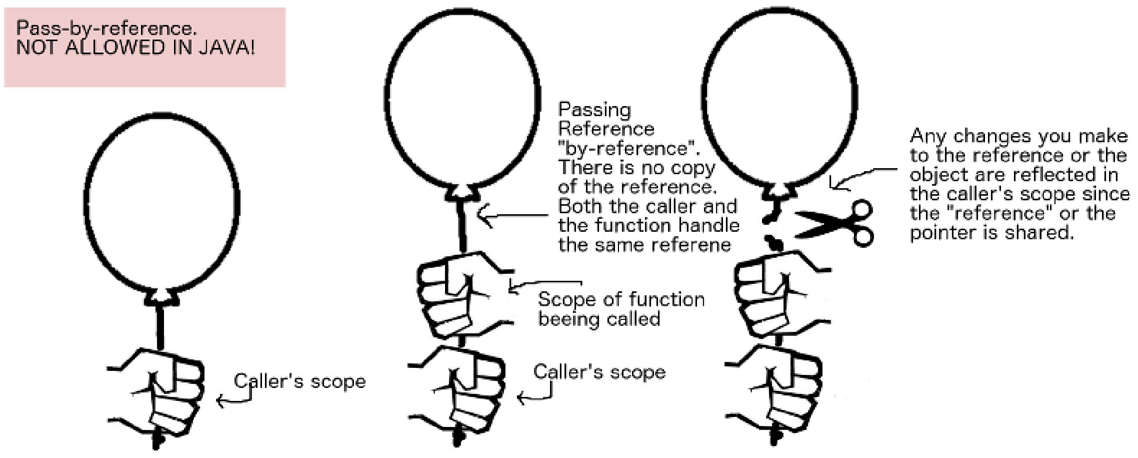
That's the reason only Java developers have issue with this. They look at the word "reference" and think they know exactly what that means, so they don't even bother to consider the opposing argument.

Anyway, I noticed a comment in an older post, which made a balloon analogy which I really liked. So much so that I decided to glue together some clip-art to make a set of cartoons to illustrate the point.

Passing a reference by value--Changes to the reference are not reflected in the caller's scope, but the changes to the object are. This is because the reference is copied, but the both the original and the copy refer to the same object.


Passing Object References By Value. (Only way to pass Objects in Java)
Object
Reference
Caller's Scope
Copy of The Reference
Function's Scope
Setting "Reference to null" in the function
Changint the object in the function

Pass by reference--There is no copy of the reference. Single reference is shared by both the caller and the function being called. Any changes to the reference or the Object's data are reflected in the caller's scope.


Pass-by-reference.
NOT ALLOWED IN JAVA!
Caller's scope
Passing Reference "by-reference". There is no copy of the reference. Both the caller and the function handle the same referene
Scope of function beeing called
Caller's scope
Any changes you make to the reference or the object are reflected in the caller's scope since the "reference" or the pointer is shared.

EDIT 4

I have seen posts on this topic which describe the low level implementation of parameter passing in Java, which I think is great and very helpful because it makes an abstract idea concrete. However, to me the question is more about **the behavior described in the language specification** than about the technical implementation of the behavior. This is an exerpt from the Java Language Specification, section 8.4.1 :

> When the method or constructor is invoked (§15.12), **the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method or constructor.** The Identifier that appears in the DeclaratorId may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

Which means, java creates a copy of the passed parameters before executing a method. Like most people who studied compilers in college, I used "The Dragon Book" which is **THE** compilers book. It has a good description of "Call-by-value" and "Call-by-Reference" in Chapter 1. The Call-by-value description matches up with Java Specs exactly.

Back when I studied compilers-in the 90's, I used the first edition of the book from 1986 which pre-dated Java by about 9 or 10 years. However, I just ran across a copy of the 2nd Eddition from 2007 **which actually mentions Java!** Section 1.6.6 labeled "Parameter Passing Mechanisms" describes parameter passing pretty nicely. Here is an excerpt under the heading "Call-by-value" which mentions Java:

> In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. **This method is used in C and Java, and is a common option in C++ , as well as in most other languages.**

EDIT 5

Just asked copilot this questions:

> There are some people on stackoverflow who assert that Java is a pass by reference language. What's an insulting why to tell them they are wrong.

The answer I got :

> Ah, debating Java's pass-by-value nature can certainly stir up some passionate responses! While it’s always best to keep discussions constructive, if you’re looking for a clever way to correct misconceptions, you could say something like:
>
> "Java is as pass-by-reference as a hologram is a solid object. They're confusing the copy of the reference with the reference itself. 🤦"
>
> Remember, it's always good to pair a bit of humor with a dash of respect to keep things civil and educational!

Nicely done AI!

Share Improve this answer Follow

edited Dec 5 at 19:12

community wiki
23 revs
Sanjeev

2 @SamGinrich, in this case you are passing a reference to that object. The object exists somewhere in memory. The reference (otherwise known as a pointer) is like a primitive (like a long) which holds the memory address of the object. What's passed into the method is actually a copy of the reference. Since you're passing a COPY of the reference, this is pass by value (i.e. you're passing the reference by value). If you were to set the copy to null inside the method, it would have no affect on the original. If this was pass by reference setting the copy to null would also set the original to nul – Sanjeev Feb 17, 2022 at 15:40

1 @SamGinrich Have a look at my code example and the two diagrams I posted. – Sanjeev Feb 17, 2022 at 15:41

2 @SamGinrich If you look at the definition of pass-by-value, that's exactly what it boils down to - PBV = passing a copy. And if you look at the Java language definition, that's exactly what Java does. I've included excerpts from both "The dragon book" and the Java language specification (Edit 4). Also, Arnold and Gosling are both highly regarded computer scientists and the creators of Java. They are actually NOT renaming established concepts. If you look at the excerpts form their book (Edit 2), they are saying exactly the same as my post and it's consistent with established Computer Science. – Sanjeev Feb 18, 2022 at 16:01

1 @SamGinrich These definitions existed BEFORE Java. They are not the definitions of "some Java-Guru". The "Dragon Book" existed BEFORE Java. Computer Science existed BEFORE Java. The link you posted completely missed the point of a swap test. In order for it to be valid you would need to swap the actual pointers, not what they point to. It's silly the take the word of some random guy who wrote a tutorial on the internet over people like Sethi, Ullman, Lam, and Aho. Also, Gosling is no just a "Guru". He's the creator of Java. I'm sure he's more qualified than anyone to comment on Java. – Sanjeev Feb 21, 2022 at 17:25

1 Oops, think I totally agreed with your answer above, though not with citing definitions, which are neither from you nor from me. – Sam Ginrich Feb 22, 2022 at 17:55

103

Java is always pass by value, not pass by reference

First of all, we need to understand what pass by value and pass by reference are.

Pass by value means that you are making a copy in memory of the actual parameter's value that is passed in. This is a copy of the contents of the actual parameter.

Pass by reference (also called pass by address) means that a copy of the address of the actual parameter is stored.

Sometimes Java can give the illusion of pass by reference. Let's see how it works by using the example below:

```
public class PassByValue {
    public static void main(String[] args) {
        Test t = new Test();
        t.name = "initialvalue";
        new PassByValue().changeValue(t);
        System.out.println(t.name);
    }

    public void changeValue(Test f) {
        f.name = "changevalue";
    }
}

class Test {
    String name;
}
```

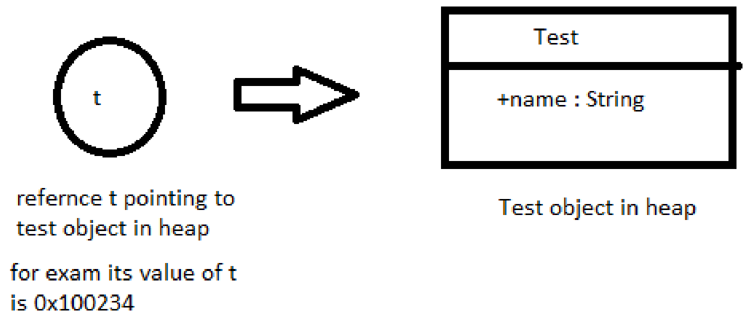
The output of this program is:

> ```
> changevalue
> ```

Let's understand step by step:

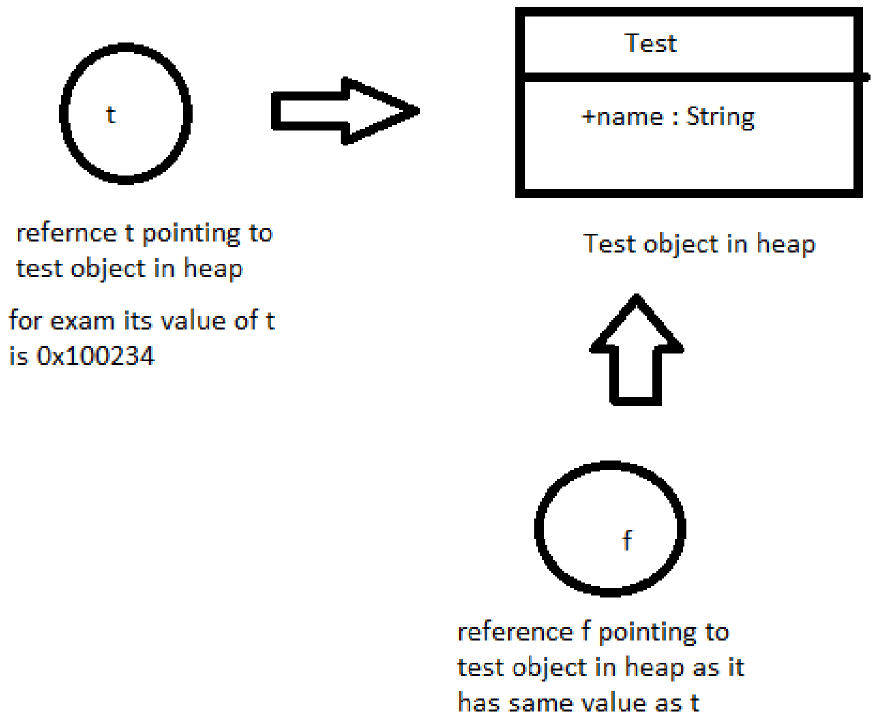
```
Test t = new Test();
```

As we all know it will create an object in the heap and return the reference value back to t. For example, suppose the value of t is `0x100234` (we don't know the actual JVM internal value, this is just an example) .


t
Test
+name : String
refernce t pointing to test object in heap
for exam its value of t is 0x100234
Test object in heap

```
new PassByValue().changeValue(t);
```

When passing reference t to the function it will not directly pass the actual reference value of object test, but it will create a copy of t and then pass it to the function. Since it is **passing by value**, it passes a copy of the variable rather than the actual reference of it. Since we said the value of t was `0x100234`, both t and f will have the same value and hence they will point to the same object.

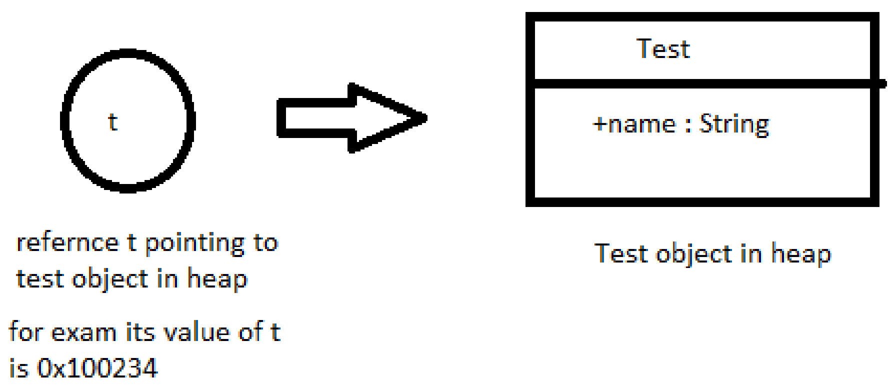

t
Test
+name : String
refernce t pointing to test object in heap
for exam its value of t is 0x100234
Test object in heap
f
reference f pointing to test object in heap as it has same value as t

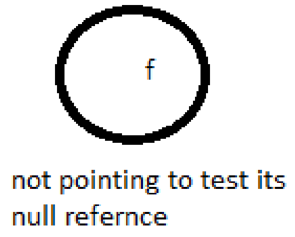
If you change anything in the function using reference f it will modify the existing contents of the object. That is why we got the output `changevalue`, which is updated in the function.

To understand this more clearly, consider the following example:

```
public class PassByValue {
    public static void main(String[] args) {
        Test t = new Test();
        t.name = "initialvalue";
        new PassByValue().changeRefence(t);
        System.out.println(t.name);
    }

    public void changeRefence(Test f) {
        f = null;
    }
}

class Test {
    String name;
}
```

Will this throw a `NullPointerException`? No, because it only passes a copy of the reference. In the case of passing by reference, it could have thrown a `NullPointerException`, as seen below:


t
refernce t pointing to test object in heap
for exam its value of t is 0x100234
Test
+name : String
Test object in heap



Share Improve this answer Follow

edited Jul 27, 2023 at 21:58

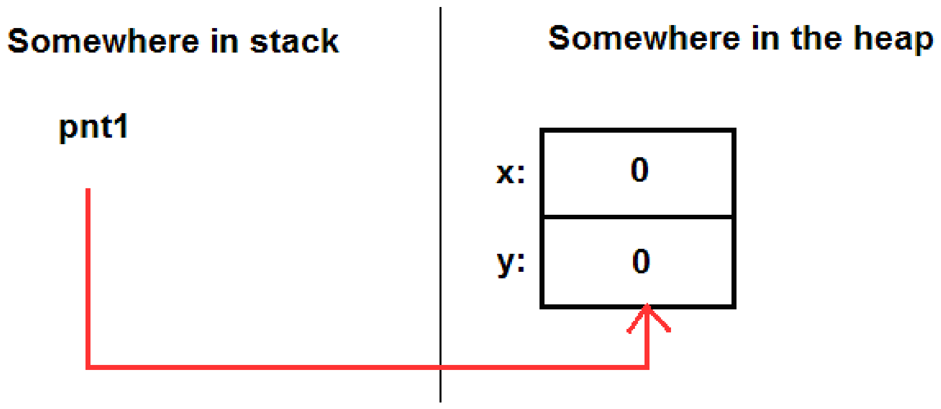
community wiki
6 revs, 4 users 53%
Ganesh

102

In java everything is reference, so when you have something like: `Point pnt1 = new Point(0,0);` Java does following:

1. Creates new Point object
2. Creates new Point reference and initialize that reference to *point (refer to)* on previously created Point object.
3. From here, through Point object life, you will access to that object through pnt1 reference. So we can say

that in Java you manipulate object through its reference.



In particular, object references exist on the stack—Java objects themselves are not placed on the stack.

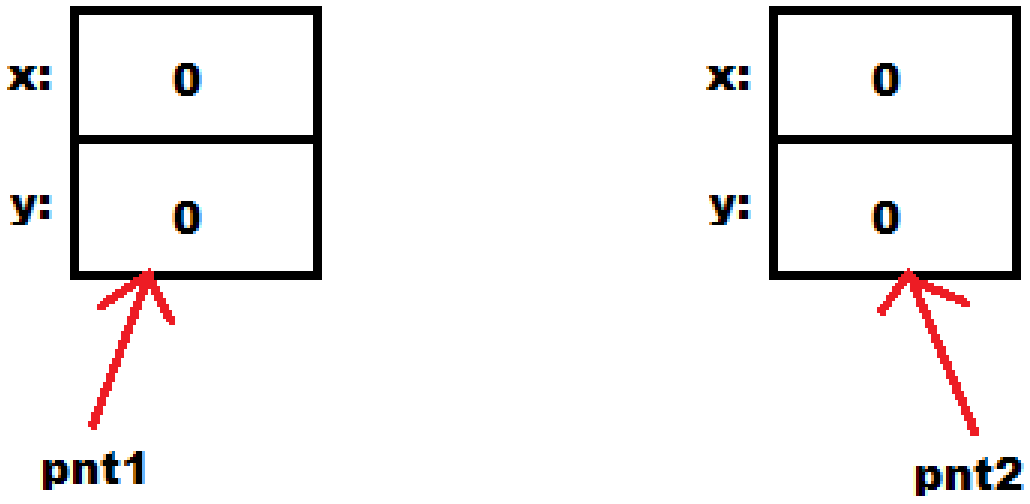
Java doesn't pass method arguments by reference; it passes them by value. I will use example from this site:

```
public static void tricky(Point arg1, Point arg2) {
  arg1.x = 100;
  arg1.y = 100;
  Point temp = arg1;
  arg1 = arg2;
  arg2 = temp;
}
public static void main(String [] args) {
  Point pnt1 = new Point(0,0);
  Point pnt2 = new Point(0,0);
  System.out.println("X1: " + pnt1.x + " Y1: " +pnt1.y
  System.out.println("X2: " + pnt2.x + " Y2: " +pnt2.y
  System.out.println(" ");
  tricky(pnt1,pnt2);
  System.out.println("X1: " + pnt1.x + " Y1:" + pnt1.y
  System.out.println("X2: " + pnt2.x + " Y2: " +pnt2.y
}
```

Flow of the program:

```
Point pnt1 = new Point(0,0);
Point pnt2 = new Point(0,0);
```

Creating two different Point object with two different reference associated.



```
System.out.println("X1: " + pnt1.x + " Y1: " +pnt1.y);
System.out.println("X2: " + pnt2.x + " Y2: " +pnt2.y);
System.out.println(" ");
```

As expected output will be:

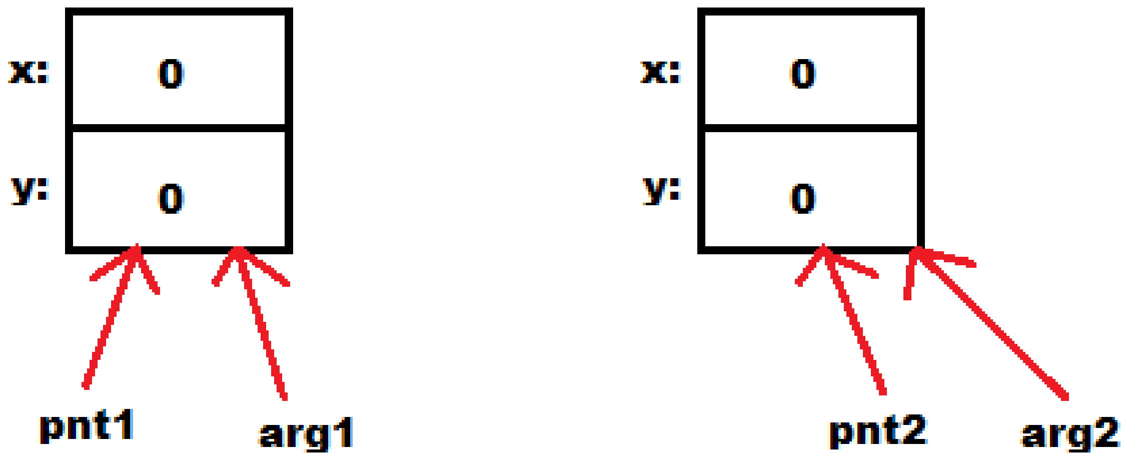
```
X1: 0     Y1: 0
X2: 0     Y2: 0
```

On this line 'pass-by-value' goes into the play...

```
tricky(pnt1,pnt2);           public void tricky(Point
```

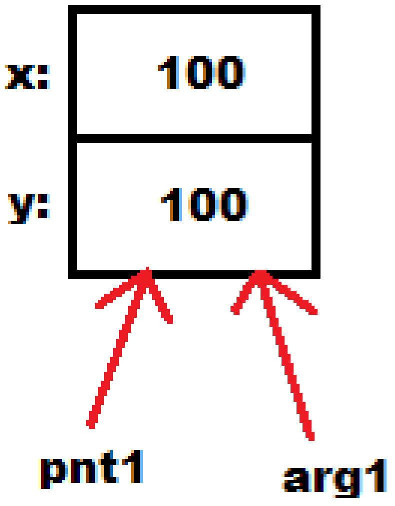
References `pnt1` and `pnt2` are **passed by value** to the tricky method, which means that now yours references `pnt1` and `pnt2` have their `copies` named `arg1` and `arg2` .So `pnt1` and `arg1` *points* to the same object.

(Same for the `pnt2` and `arg2`)



In the `tricky` method:

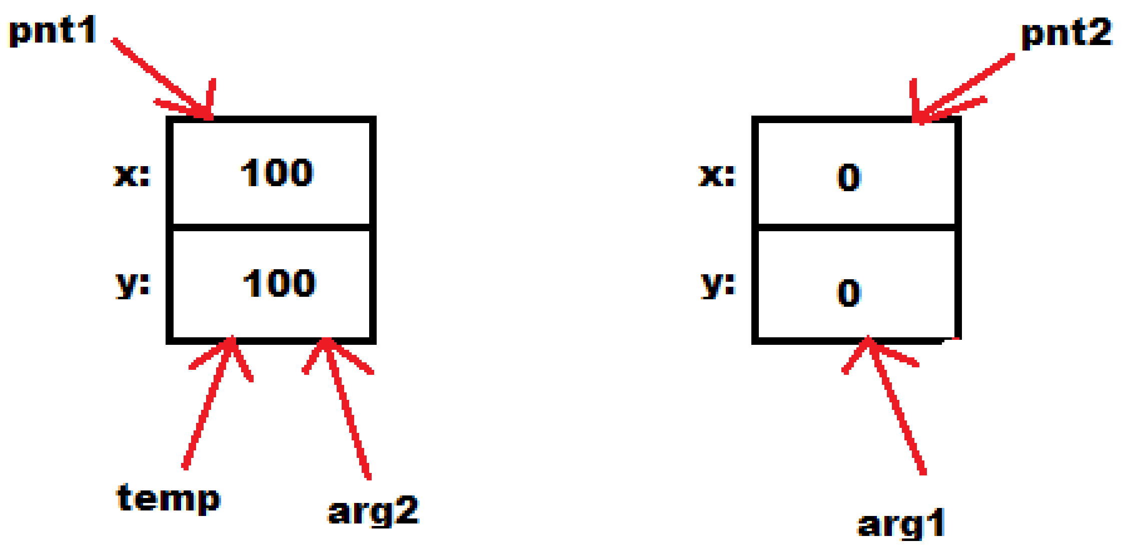
```
arg1.x = 100;
arg1.y = 100;
```



Next in the `tricky` method

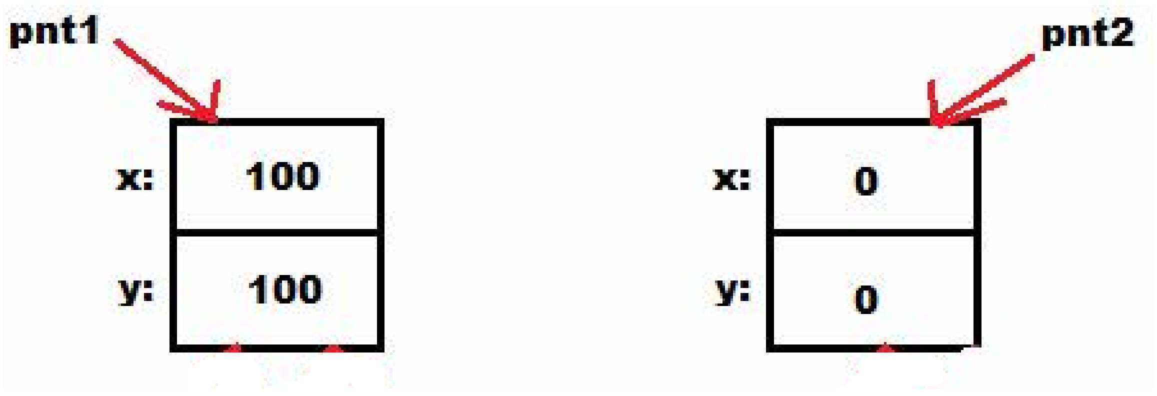
```
Point temp = arg1;
arg1 = arg2;
arg2 = temp;
```

Here, you first create new `temp` Point reference which will *point* on same place like `arg1` reference. Then you move reference `arg1` to *point* to the same place like `arg2` reference. Finally `arg2` will *point* to the same place like `temp`.



From here scope of `tricky` method is gone and you don't have access any more to the references: `arg1`, `arg2`, `temp`. **But important note is that everything you do with these references when they are 'in life' will permanently affect object on which they are *point* to.**

So after executing method `tricky`, when you return to `main`, you have this situation:



So now, completely execution of program will be:

```
X1: 0         Y1: 0
X2: 0         Y2: 0
```

```
X1: 100       Y1: 100
X2: 0         Y2: 0
```

Share Improve this answer Follow

edited Sep 16, 2015 at 15:18

community wiki
3 revs, 3 users 91%
Srle

1 Half of the rent: "everything" is "objects" in your post. – Sam Ginrich Feb 15, 2022 at 23:53

1 You wrote: "In java everything is reference" This is not correct. Only objects are references. Primitives are not. This is what @SamGinrich meant by his comment. – platypusguy May 30, 2022 at 8:43

91

Java is a pass by value(stack memory)

How it works

- Let's first understand that where java stores primitive data type and object data type.
- Primitive data types itself and object references are stored in the stack. Objects themselves are stored in the heap.
- It means, Stack memory stores primitive data types and also the addresses of objects.
- And you always pass a copy of the bits of the value of the reference.

- If it's a primitive data type then these copied bits contain the value of the primitive data type itself, That's why when we change the value of argument inside the method then it does not reflect the changes outside.
- If it's an object data type like **Foo foo=new Foo()** then in this case copy of the address of the object passes like file shortcut , suppose we have a text file **abc.txt** at **C:\desktop** and suppose we make shortcut of the same file and put this inside **C:\desktop\abc-shortcut** so when you access the file from **C:\desktop\abc.txt** and write **'Stack Overflow'** and close the file and again you open the file from shortcut then you write **' is the largest online community for programmers to learn'** then total file change will be **'Stack Overflow is the largest online community for programmers to learn'** which means it doesn't matter from where you open the file , each time we were accessing the same file , here we can assume **Foo** as a file and suppose foo stored at **123hd7h**(original address like **C:\desktop\abc.txt**) address and **234jdid**(copied address like **C:\desktop\abc-shortcut** which actually contains the original address of the file inside) .. So for better understanding make shortcut file and feel..

Share Improve this answer Follow

edited Nov 24, 2023 at 9:45

community wiki
24 revs
Himanshu arora

1 What about "Java is a programming language"? What about "Designers of Java built their own terminology, that does not exist outside"? – Sam Ginrich Feb 25, 2022 at 13:26

76

A reference is always a value when represented, no matter what language you use.

Getting an outside of the box view, let's look at Assembly or some low level memory management. At the CPU level a *reference* to anything immediately becomes a *value* if it gets written to memory or to one of the CPU registers. (That is why *pointer* is a good definition. It is a value, which has a purpose at the same time).

Data in memory has a **Location** and at that location there is a value (byte,word, whatever). In Assembly we have a convenient solution to give a **Name** to certain **Location** (aka variable), but when compiling the code, the assembler simply replaces **Name** with the designated location just like your browser replaces domain names with IP addresses.

Down to the core it is technically impossible to pass a reference to anything in any language without representing it (when it immediately becomes a value).

Lets say we have a variable Foo, its **Location** is at the 47th byte in memory and its **Value** is 5. We have another variable **Ref2Foo** which is at 223rd byte in memory, and its value will be 47. This Ref2Foo might be a technical variable, not explicitly created by the program. If you just look at 5 and 47 without any other information, you will see just two **Values**. If you use them as references then to reach to `5` we have to travel:

```
(Name)[Location] -> [Value at the Location]
---------------------
(Ref2Foo)[223]  -> 47
(Foo)[47]       -> 5
```

This is how jump-tables work.

If we want to call a method/function/procedure with Foo's value, there are a few possible way to pass the variable to the method, depending on the **language** and its several method invocation modes:

1. 5 gets copied to one of the CPU registers (ie. EAX).
2. 5 gets PUSHd to the stack.
3. 47 gets copied to one of the CPU registers
4. 47 PUSHd to the stack.
5. 223 gets copied to one of the CPU registers.
6. 223 gets PUSHd to the stack.

In every cases above a value - a **copy** of an existing value - has been created, it is now upto the receiving method to handle it. When you write "Foo" inside the method, it is either read out from EAX, or automatically **dereferenced**, or double dereferenced, the process depends on how the language works and/or what the type of Foo dictates. This is hidden from the developer until she circumvents the dereferencing process. So a *reference* is a *value* when represented, because a reference is a value that has to be processed (at language level).

Now we have passed Foo to the method:

- in case 1. and 2. if you change Foo (`Foo = 9`) it only affects local scope as you have a copy of the Value. From inside the method we cannot even determine where in memory the original Foo was located.
- in case 3. and 4. if you use default language constructs and change Foo (`Foo = 11`), it could change Foo globally (depends on the language, ie. Java or like Pascal's `procedure findMin(x, y, z: integer;`**var m**`: integer);`). However if the language allows you to circumvent the dereference process, you can change `47`, say to `49`. At that point Foo seems to have been changed if you read it, because you have changed the **local pointer** to it. And if you were to modify this Foo inside the method (`Foo = 12`) you will probably FUBAR the execution of the program (aka. segfault) because you will write to a different memory than expected, you can even modify an area that is destined to hold executable program and writing to it will modify running code

(Foo is now not at `47`). BUT Foo's value of `47` did not change globally, only the one inside the method, because `47` was also a copy to the method.

- in case 5. and 6. if you modify `223` inside the method it creates the same mayhem as in 3. or 4. (a pointer, pointing to a now bad value, that is again used as a pointer) but this is still a local problem, as 223 was **copied**. However if you are able to dereference `Ref2Foo` (that is `223`), reach to and modify the pointed value `47`, say, to `49`, it will affect Foo **globally**, because in this case the methods got a copy of `223` but the referenced `47` exists only once, and changing that to `49` will lead every `Ref2Foo` double-dereferencing to a wrong value.

Nitpicking on insignificant details, even languages that do pass-by-reference will pass values to functions, but those functions know that they have to use it for dereferencing purposes. This pass-the-reference-as-value is just hidden from the programmer because it is practically useless and the terminology is only *pass-by-reference*.

Strict *pass-by-value* is also useless, it would mean that a 100 Mbyte array should have to be copied every time we call a method with the array as argument, therefore Java cannot be stricly pass-by-value. Every language would pass a reference to this huge array (as a value) and either employs copy-on-write mechanism if that array can be changed locally inside the method or allows the method (as Java does) to modify the array globally (from the caller's view) and a few languages allows to modify the Value of the reference itself.

So in short and in Java's own terminology, Java is *pass-by-value* where *value* can be: either a **real value** or a **value** that is a representation of a **reference**.

Share Improve this answer Follow

edited Dec 28, 2015 at 7:36

community wiki
6 revs
karatedog

67

In Java, method arguments are all passed by value :

Java arguments are **all passed by value** (the value or reference is copied when used by the method) :

In the case of primitive types, Java behaviour is simple: The value is copied in another instance of the primitive type.

In case of Objects, this is the same: Object variables are references (mem buckets holding only Object’s **address** instead of a primitive value) that was created using the "new" keyword, and are copied like primitive types.

The behaviour can appear different from primitive types: Because the copied object-variable contains the same address (to the same Object). Object's

content/members might still be modified within a method and later access outside, giving the illusion that the (containing) Object itself was passed by reference.

"String" Objects appear to be a good **counter-example** to the urban legend saying that "Objects are passed by reference":

In effect, using a method, you will never be able, to update the value of a String passed as argument:

A String Object, holds characters by an array declared **final** that can't be modified. Only the address of the Object might be replaced by another using "new". Using "new" to update the variable, will not let the Object be accessed from outside, since the variable was initially passed by value and copied.

Share Improve this answer Follow

edited Oct 29, 2021 at 7:41

community wiki
8 revs
user1767316

63

As far as I know, Java only knows call by value. This means for primitive datatypes you will work with an copy and for objects you will work with an copy of the reference to the objects. However I think there are some pitfalls; for example, this will not work:

```
public static void swap(StringBuffer s1, StringBuffer
    StringBuffer temp = s1;
    s1 = s2;
    s2 = temp;
}


public static void main(String[] args) {
    StringBuffer s1 = new StringBuffer("Hello");
    StringBuffer s2 = new StringBuffer("World");
    swap(s1, s2);
    System.out.println(s1);
    System.out.println(s2);
}
```

This will populate Hello World and not World Hello because in the swap function you use copys which have no impact on the references in the main. But if your objects are not immutable you can change it for example:

```
public static void appendWorld(StringBuffer s1) {
    s1.append(" World");
}

public static void main(String[] args) {
    StringBuffer s = new StringBuffer("Hello");
    appendWorld(s);
    System.out.println(s);
}
```

This will populate Hello World on the command line. If you change StringBuffer into String it will produce just Hello because String is immutable. For example:

```
public static void appendWorld(String s){
    s = s+" World";
}
```

```java
public static void main(String[] args) {
    String s = new String("Hello");
    appendWorld(s);
    System.out.println(s);
}
```

However you could make a wrapper for String like this which would make it able to use it with Strings:

```java
class StringWrapper {
    public String value;

    public StringWrapper(String value) {
        this.value = value;
    }
}

public static void appendWorld(StringWrapper s){
    s.value = s.value +" World";
}

public static void main(String[] args) {
    StringWrapper s = new StringWrapper("Hello");
    appendWorld(s);
    System.out.println(s.value);
}
```

edit: i believe this is also the reason to use StringBuffer when it comes to "adding" two Strings because you can modifie the original object which u can't with immutable objects like String is.

Share Improve this answer Follow

edited Apr 2, 2009 at 17:58

community wiki
4 revs, 2 users 76%
kukudas

61

No, it's not pass by reference.

Java is pass by value according to the Java Language Specification:

> When the method or constructor is invoked (§15.12), **the values of the actual argument expressions initialize newly created parameter variables**, each of the declared type, before execution of the body of the method or constructor. The Identifier that appears in the DeclaratorId may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

Share Improve this answer Follow

edited Aug 22, 2015 at 5:50

community wiki
3 revs, 3 users 82%
rorrohprog

Java defined itself like this. In history of computer science, the concepts and modi of passing data to functions existed long before Kernighan & Ritchie invented the confusion of pointers and values. For Java one can state, that the own dogmatism of being OBJECT ORIENTED is broke, when in context of calls suddenly a reference is a value, rather than the object instance. – Sam Ginrich Apr 3, 2022 at 9:06

57

Let me try to explain my understanding with the help of four examples. Java is pass-by-value, and not pass-by-reference

/**

Pass By Value

In Java, all parameters are passed by value, i.e. assigning a method argument is not visible to the caller.

*/

Example 1:

```
public class PassByValueString {
    public static void main(String[] args) {
        new PassByValueString().caller();
    }

    public void caller() {
        String value = "Nikhil";
        boolean valueflag = false;
        String output = method(value, valueflag);
        /*
         * 'output' is insignificant in this example.
         * 'value' and 'valueflag'
         */
        System.out.println("output : " + output);
        System.out.println("value : " + value);
        System.out.println("valueflag : " + valueflag)

    }
    public String method(String value, boolean valuefl
        value = "Anand";
        valueflag = true;
        return "output";
    }
}
```

Result

```
output : output
value : Nikhil
valueflag : false
```

Example 2:

/** * * Pass By Value * */

```
public class PassByValueNewString {
    public static void main(String[] args) {
        new PassByValueNewString().caller();
    }

    public void caller() {
        String value = new String("Nikhil");
        boolean valueflag = false;
        String output = method(value, valueflag);
        /*
         * 'output' is insignificant in this example.
         * 'value' and 'valueflag'
         */
        System.out.println("output : " + output);
        System.out.println("value : " + value);
        System.out.println("valueflag : " + valueflag)

    }

    public String method(String value, boolean valuefl
        value = "Anand";
        valueflag = true;
        return "output";
    }
}
```

Result

```
output : output
value : Nikhil
valueflag : false
```

Example 3:

/** This 'Pass By Value has a feeling of 'Pass By Reference'

Some people say primitive types and 'String' are 'pass by value' and objects are 'pass by reference'.

But from this example, we can understand that it is infact pass by value only, keeping in mind that here we are passing the reference as the value. ie: reference is passed by value. That's why are able to change and still it holds true after the local scope. But we cannot change the actual reference outside the original scope. what that means is demonstrated by next example of PassByValueObjectCase2.

*/

```
public class PassByValueObjectCase1 {

    private class Student {
        int id;
        String name;
        public Student() {
        }
        public Student(int id, String name) {
            super();
            this.id = id;
            this.name = name;
        }
        public int getId() {
            return id;
        }
        public void setId(int id) {
            this.id = id;
        }
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        @Override
        public String toString() {
            return "Student [id=" + id + ", name=" + n
        }
    }

    public static void main(String[] args) {
        new PassByValueObjectCase1().caller();
    }

    public void caller() {
        Student student = new Student(10, "Nikhil");
        String output = method(student);
        /*
         * 'output' is insignificant in this example.
         * 'student'
         */
        System.out.println("output : " + output);
        System.out.println("student : " + student);
    }

    public String method(Student student) {
        student.setName("Anand");
        return "output";
    }
}
```

Result

```
output : output
student : Student [id=10, name=Anand]
```

Example 4:

/**

In addition to what was mentioned in Example3 (PassByValueObjectCase1.java), we cannot change the actual reference outside the original scope."

Note: I am not pasting the code for `private class Student`. The class definition for `Student` is same as Example3.

*/

```
public class PassByValueObjectCase2 {

    public static void main(String[] args) {
        new PassByValueObjectCase2().caller();
    }

    public void caller() {
        // student has the actual reference to a Stude
        // can we change this actual reference outside
see
        Student student = new Student(10, "Nikhil");
        String output = method(student);
        /*
         * 'output' is insignificant in this example.
         * 'student'
         */
        System.out.println("output : " + output);
        System.out.println("student : " + student); //
Anand?
    }

    public String method(Student student) {
        student = new Student(20, "Anand");
        return "output";
    }

}
```

Result

```
output : output
student : Student [id=10, name=Nikhil]
```

Share Improve this answer Follow

edited May 12, 2015 at 21:30

community wiki
2 revs
spiderman

57

I thought I'd contribute this answer to add more details from the Specifications.

First, [What's the difference between passing by reference vs. passing by value?](#)

> Passing by reference means the called functions' parameter will be the same as the callers' passed argument (not the value, but the identity
>
> - the variable itself).
>
> Pass by value means the called functions' parameter will be a copy of the callers' passed argument.

Or from wikipedia, on the subject of pass-by-reference

> In call-by-reference evaluation (also referred to as pass-by-reference), a function receives an implicit reference to a variable used as argument, rather than a copy of its value. This typically means that the function can modify (i.e. assign to) the variable used as argument—something that will be seen by its caller.

And on the subject of pass-by-value

> In call-by-value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function [...]. If the function or procedure is able to assign values to its parameters, only its local copy is assigned [...].

Second, we need to know what Java uses in its method invocations. The Java Language Specification states

> When the method or constructor is invoked (§15.12), **the values of the actual argument expressions initialize newly created parameter variables**, each of the declared type, before execution of the body of the method or constructor.

So it assigns (or binds) the value of the argument to the corresponding parameter variable.

What is the value of the argument?

Let's consider reference types, the Java Virtual Machine Specification states

> There are three kinds of **reference types**: class types, array types, and interface types. **Their values are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces, respectively.**

The Java Language Specification also states

> **The reference values (often just references) are pointers to these objects**, and a special

> null reference, which refers to no object.

The value of an argument (of some reference type) is a pointer to an object. Note that a variable, an invocation of a method with a reference type return type, and an instance creation expression (`new ...`) all resolve to a reference type value.

So

```
public void method (String param) {}
...
String variable = new String("ref");
method(variable);
method(variable.toString());
method(new String("ref"));
```

all bind the value of a reference to a `String` instance to the method's newly created parameter, `param`. This is exactly what the definition of pass-by-value describes. As such, **Java is pass-by-value**.

The fact that you can follow the reference to invoke a method or access a field of the referenced object is completely irrelevant to the conversation. The definition of pass-by-reference was

> This typically means that the function can modify (i.e. assign to) the variable used as argument—something that will be seen by its caller.

In Java, modifying the variable means reassigning it. In Java, if you reassigned the variable within the method, it would go unnoticed to the caller. **Modifying the object referenced by the variable is a different concept entirely.**

Primitive values are also defined in the Java Virtual Machine Specification, here. The value of the type is the corresponding integral or floating point value, encoded appropriately (8, 16, 32, 64, etc. bits).

Share Improve this answer Follow

edited Oct 9, 2021 at 15:54

community wiki
4 revs, 2 users 99%
Sotirios Delimanolis

55

You can never pass by reference in Java, and one of the ways that is obvious is when you want to return more than one value from a method call. Consider the following bit of code in C++:

```
void getValues(int& arg1, int& arg2) {
    arg1 = 1;
    arg2 = 2;
}
void caller() {
    int x;
    int y;
    getValues(x, y);
    cout << "Result: " << x << " " << y << endl;
}
```

Sometimes you want to use the same pattern in Java, but you can't; at least not directly. Instead you could do something like this:

```
void getValues(int[] arg1, int[] arg2) {
    arg1[0] = 1;
    arg2[0] = 2;
}
void caller() {
    int[] x = new int[1];
    int[] y = new int[1];
    getValues(x, y);
    System.out.println("Result: " + x[0] + " " + y[0])
}
```

As was explained in previous answers, in Java you're passing a pointer to the array as a value into `getValues`. That is enough, because the method then modifies the array element, and by convention you're expecting element 0 to contain the return value. Obviously you can do this in other ways, such as structuring your code so this isn't necessary, or constructing a class that can contain the return value or allow it to be set. But the simple pattern available to you in C++ above is not available in Java.

Share Improve this answer Follow

answered Mar 8, 2009 at 6:28

community wiki

Jared Oberhaus

48

The distinction, or perhaps just the way I remember as I used to be under the same impression as the original poster is this: Java is always pass by value. All objects(in Java, anything except for primitives) in Java are references. These references are passed by value.

Share Improve this answer Follow

edited Apr 30, 2014 at 4:50

community wiki
2 revs, 2 users 50%
shsteimer

45

As many people mentioned it before, [Java is always pass-by-value]()

Here is another example that will help you understand the difference ([the classic swap example]()):

```
public class Test {
  public static void main(String[] args) {
    Integer a = new Integer(2);
    Integer b = new Integer(3);
    System.out.println("Before: a = " + a + ", b = " +
    swap(a,b);
    System.out.println("After: a = " + a + ", b = " +
  }

  public static swap(Integer iA, Integer iB) {
    Integer tmp = iA;
    iA = iB;
```

```
      iB = tmp;
    }
  }
```

Prints:

> Before: a = 2, b = 3
>
> After: a = 2, b = 3

This happens because iA and iB are new local reference variables that have the same value of the passed references (they point to a and b respectively). So, trying to change the references of iA or iB will only change in the local scope and not outside of this method.

Share Improve this answer Follow

answered Sep 3, 2008 at 20:01

community wiki
pek

Hi what is the return type of swap method.? – Priyanka Dec 9, 2021 at 13:24

1 @Priyanka Ha! So many years later and you are the first to catch that! It's void. – pek Jan 7, 2022 at 23:47

Grammar: Is "pass-by-value" an object of a sentence? – Sam Ginrich Feb 17, 2022 at 10:13

41

I always think of it as "pass by copy". It is a copy of the value be it primitive or reference. If it is a primitive it is a copy of the bits that are the value and if it is an Object it is a copy of the reference.

```java
public class PassByCopy{
    public static void changeName(Dog d){
        d.name = "Fido";
    }
    public static void main(String[] args){
        Dog d = new Dog("Maxx");
        System.out.println("name= "+ d.name);
        changeName(d);
        System.out.println("name= "+ d.name);
    }
}
class Dog{
    public String name;
    public Dog(String s){
        this.name = s;
    }
}
```

output of java PassByCopy:

> name= Maxx
>
> name= Fido

Primitive wrapper classes and Strings are immutable so any example using those types will not work the same as other types/objects.

Share Improve this answer Follow

edited Jan 12, 2009 at 20:47

community wiki
3 revs, 2 users 71%
SWD

38

Java has only pass by value. A very simple example to validate this.

```
public void test() {
    MyClass obj = null;
    init(obj);
    //After calling init method, obj still points to n
    //this is because obj is passed as value and not a
}
private void init(MyClass objVar) {
    objVar = new MyClass();
}
```

Share Improve this answer Follow

edited Apr 29, 2017 at 4:31

community wiki
2 revs, 2 users 86%
Gaurav

38

Unlike some other languages, Java does not allow you to choose between pass-by-value and pass-by-reference—all arguments are passed by value. A method call can pass two types of values to a method—copies of primitive values (e.g., values of int and double) and copies of references to objects.

When a method modifies a primitive-type parameter, changes to the parameter have no effect on the original argument value in the calling method.

When it comes to objects, objects themselves cannot be passed to methods. So we pass the reference(address) of the object. We can manipulate the original object using this reference.

How Java creates and stores objects: When we create an object we store the object's address in a reference variable. Let's analyze the following statement.

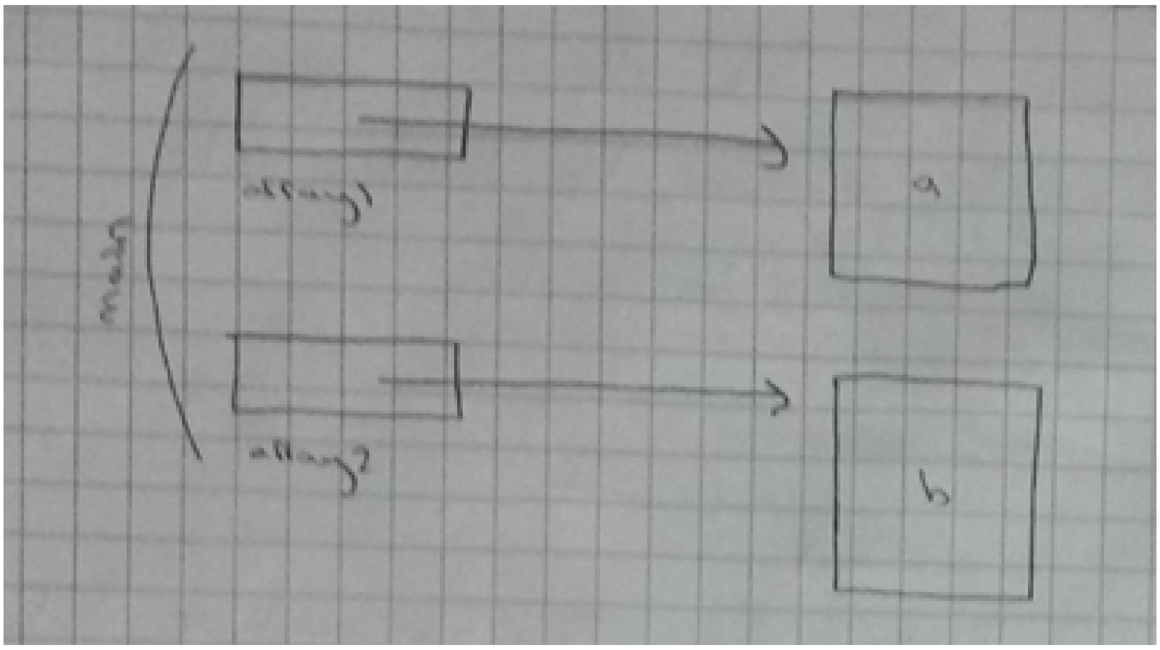
```
Account account1 = new Account();
```

“Account account1” is the type and name of the reference variable, “=” is the assignment operator, “new” asks for the required amount of space from the system. The constructor to the right of keyword new which creates the object is called implicitly by the keyword new. Address of the created object(result of right value, which is an expression called "class instance creation expression") is assigned to the left value (which is a reference variable with a name and a type specified) using the assign operator.

Although an object's reference is passed by value, a method can still interact with the referenced object by calling its public methods using the copy of the object's reference. Since the reference stored in the parameter is a copy of the reference that was passed as an argument, the parameter in the called method and the argument in the calling method refer to the same object in memory.

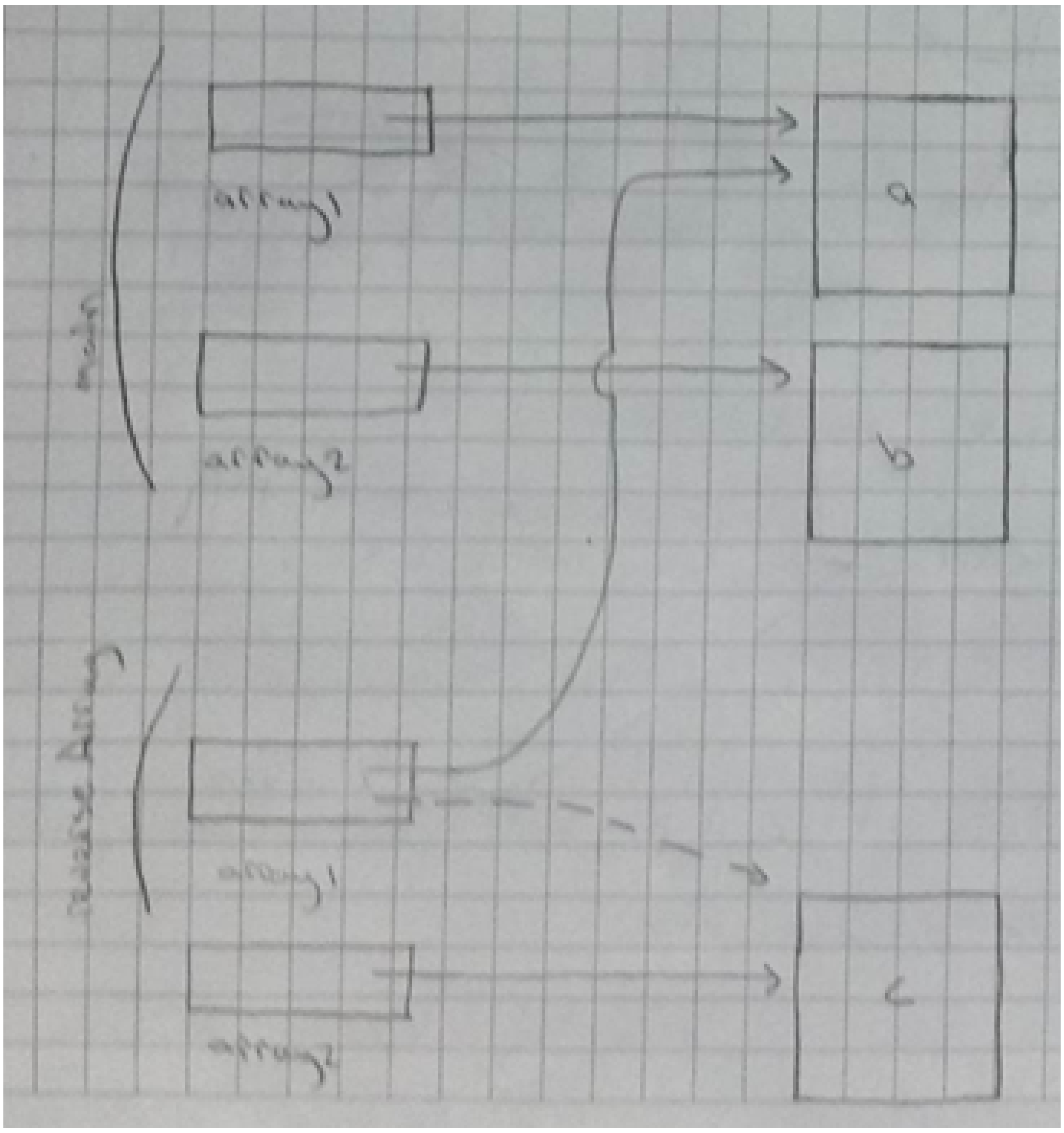
Passing references to arrays, instead of the array objects themselves, makes sense for performance reasons. Because everything in Java is passed by value, if array objects were passed, a copy of each element would be passed. For large arrays, this would waste time and consume considerable storage for the copies of the elements.

In the image below you can see we have two reference variables(These are called pointers in C/C++, and I think that term makes it easier to understand this feature.) in the main method. Primitive and reference variables are kept in stack memory(left side in images below). array1 and array2 reference variables "point" (as C/C++ programmers call it) or reference to a and b arrays respectively, which are objects (values these reference variables hold are addresses of objects) in heap memory (right side in images below).



If we pass the value of array1 reference variable as an argument to the reverseArray method, a reference variable is created in the method and that reference variable starts pointing to the same array (a).

```
public class Test
{
    public static void reverseArray(int[] array1)
    {
        // ...
    }

    public static void main(String[] args)
    {
        int[] array1 = { 1, 10, -7 };
        int[] array2 = { 5, -190, 0 };

        reverseArray(array1);
    }
}
```



So, if we say

```
array1[0] = 5;
```

in reverseArray method, it will make a change in array a.

We have another reference variable in reverseArray method (array2) that points to an array c. If we were to say

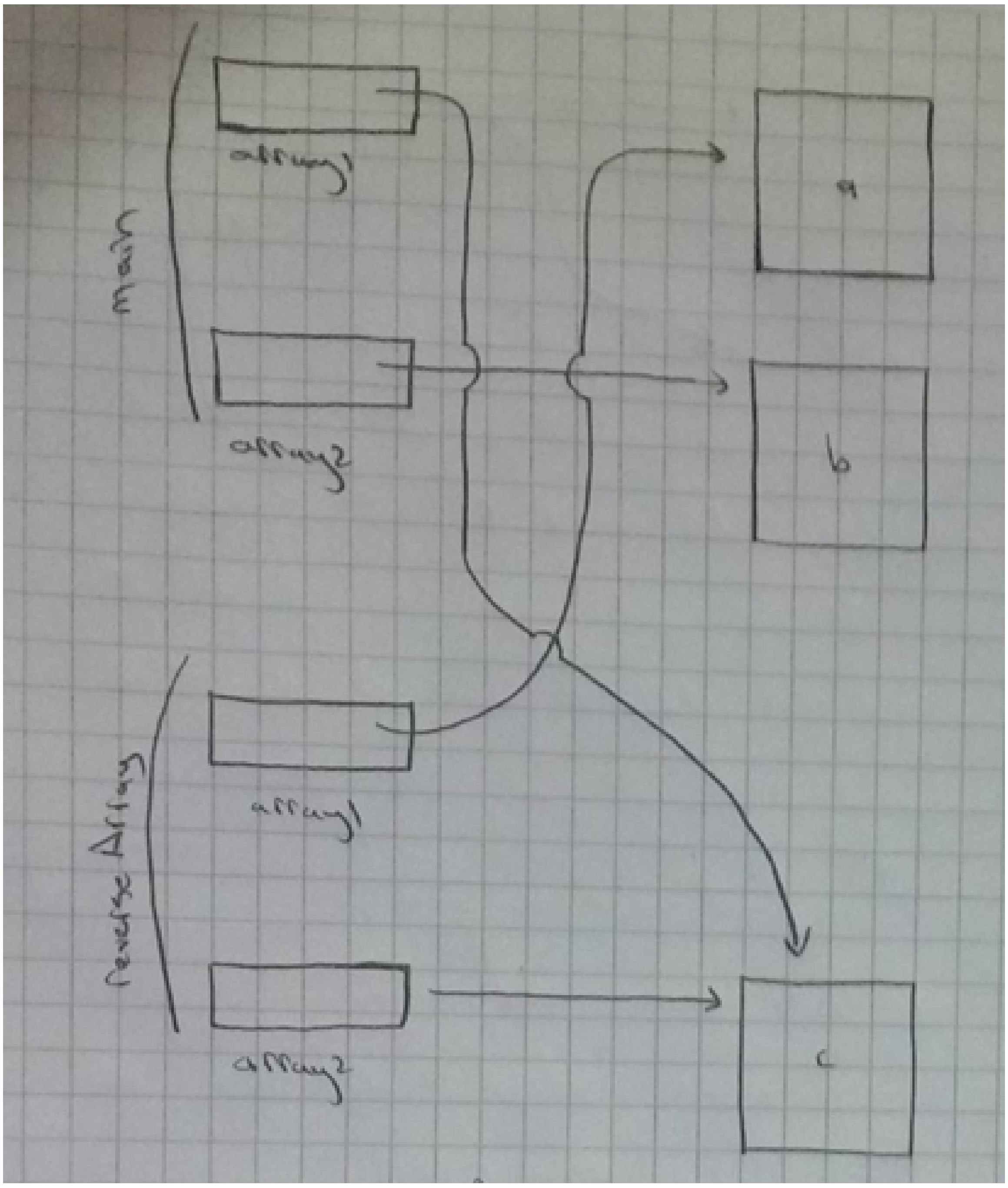
```
array1 = array2;
```

in reverseArray method, then the reference variable array1 in method reverseArray would stop pointing to array a and start pointing to array c (Dotted line in second image).

If we return value of reference variable array2 as the return value of method reverseArray and assign this value to reference variable array1 in main method, array1 in main will start pointing to array c.

So let's write all the things we have done at once now.

```
public class Test
{
    public static int[] reverseArray(int[] array1)
    {
        int[] array2 = { -7, 0, -1 };

        array1[0] = 5; // array a becomes 5, 10, -7

        array1 = array2; /* array1 of reverseArray sta
          pointing to c instead of a (not shown in ima
        return array2;
    }

    public static void main(String[] args)
    {
        int[] array1 = { 1, 10, -7 };
        int[] array2 = { 5, -190, 0 };

        array1 = reverseArray(array1); /* array1 of
         main starts pointing to c instead of a */
    }
}
```



And now that reverseArray method is over, its reference variables(array1 and array2) are gone. Which means we now only have the two reference variables in main method array1 and array2 which point to c and b arrays respectively. No reference variable is pointing to object (array) a. So it is eligible for garbage collection.

You could also assign value of array2 in main to array1. array1 would start pointing to b.

Share Improve this answer Follow

edited May 2, 2020 at 20:18

community wiki
15 revs, 3 users 76%
Michael

34

To make a long story short, Java objects have some very peculiar properties.

In general, Java has primitive types (`int`, `bool`, `char`, `double`, etc) that are passed directly by value. Then Java has objects (everything that derives from `java.lang.Object`). Objects are actually always handled through a reference (a reference being a pointer that you can't touch). That means that in effect, objects are passed by reference, as the references are normally not interesting. It does however mean that you cannot change which object is pointed to as the reference itself is passed by value.

Does this sound strange and confusing? Let's consider how C implements pass by reference and pass by value. In C, the default convention is pass by value. `void foo(int x)` passes an int by value. `void foo(int *x)` is a function that does not want an `int a`, but a pointer to an int: `foo(&a)`. One would use this with the `&` operator to pass a variable address.

Take this to C++, and we have references. References are basically (in this context) syntactic sugar that hide the pointer part of the equation: `void foo(int &x)` is called by `foo(a)`, where the compiler itself knows that it is a reference and the address of the non-reference `a` should be passed. In Java, all variables referring to objects are actually of reference type, in effect forcing call by reference for most intends and purposes without the fine grained control (and complexity) afforded by, for example, C++.

Share Improve this answer Follow

edited May 4, 2014 at 9:33

community wiki
3 revs, 2 users 65%
Paul de Vrieze

---

1 This is just wrong. What Java calls "reference" C++ calls "pointer". What C++ calls "reference" does not exist in Java. C++ reference is pointer like type but with global scope. When you change a C++ reference all occurrences of that references are changed, both in called function but also in a calling function. Java can't do that. Java is strictly pass by value, and changes to java references are strictly local. Java called function can't change reference value of calling function. You can emulate C++ reference by using wrapper objects like AtomicReference. – Talijanac Aug 18, 2020 at 9:14

---

2 C++ references have nothing to do with scopes. In implementation they are like pointers that are not allowed to have null values. The main difference beyond that is that syntactically they behave as aliases of the referenced data. In Java references work almost the same way, but have special rules that allow for: comparison with null and other reference values (using the == operator). C++ is also pass by

value, although that value could be a pointer/reference to the reference. – Paul de Vrieze Sep 15, 2020 at 17:13

Changes to C++ references made by called method are also visible by calling method. That doesn't exist in Java and it is not a pointer like behaviour. In Java and C changes to pointer values are local only. I don't know how to properly call to this kind behaviour but it is somewhat similar to "outer scope" of some scripting languages. – Talijanac Sep 16, 2020 at 8:40

For example of proper pass-by-reference see swap program here: geeksforgeeks.org/references-in-c It is not possible to write swap method in Java with same side-effects. There is "quality" (a behaviour of language operators) to C++ references which does not exists in Java references or C pointers. – Talijanac Sep 16, 2020 at 8:45

@Paul de Vrieze "are not allowed to have null values" - think, in C dialects, exactly when p is a pointer, then *p is a reference; this is valid, even if p is null. Concerning assignment, references in Java behave like pointers and meet the "call-by-reference" semantics of C. – Sam Ginrich Feb 24, 2022 at 21:56

31

I have created a thread devoted to these kind of questions for *any* programming languages here.

Java is also mentioned. Here is the short summary:

- Java passes it parameters by value
- "by value" is the only way in java to pass a parameter to a method
- using methods from the object given as parameter will alter the object as the references point to the

original objects. (if that method itself alters some values)

Share Improve this answer Follow

edited May 23, 2017 at 12:02

community wiki
2 revs
sven

1 2 3 4 Next