Pass-by-Value Please (Cup Size continued)

If you haven't read the Cup Size story, this one won't make sense. Or it will make sense, but you'll think its really stupid. Or you won't think it's stupid but you'll find yourself... never mind, just go read it now and then come back.

I really care about my cups.

I don't want just anybody putting something in my cups. If I have something in a cup, I just want it to stay that way until I decide to change it! So back off!

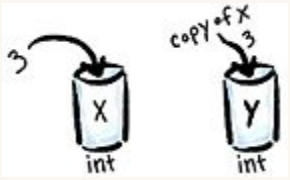
So if a Java variable is a cup, with a value in it, **what does it mean to "pass" a variable** to a method? And are primitives and references treated the same way?

We'll get there, but first let's start with simple assignment.

What does it mean to say:

1) `int x = 3;`

2) `int y = x;`



In line 1, a cup called x, of size int, is created and given the value 3.

In line 2, a cup called y, of size int, is created and given the value... 3.

The x variable is not affected!

Java COPIES the value of x (which is 3) and puts that COPY into y.

This is PASS-BY-VALUE. Which you can think of as PASS-BY-COPY. The value is copied, and that's what gets shoved into the new cup. You don't stuff one cup into another one.

Saying int y = x does NOT mean "put the x cup into y". It means "copy the value inside x and put that copy into y".

If I later change y:

```
y = 34;
```

Is x affected? Of course not. The x cup is still sitting there, all happy.

If I later change x:

```
x = 90;
```

Is y affected? Nope. They are disconnected from one another once the assignment was made (the COPY was made).

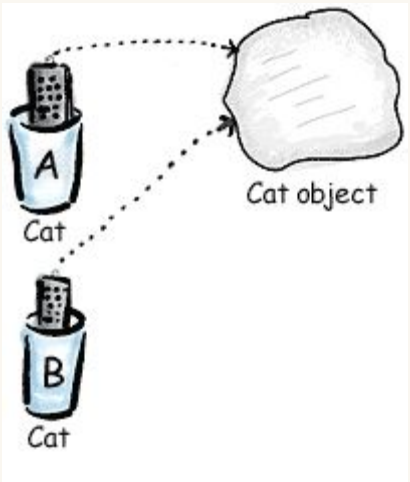
SO... what about Reference Variables (remote controls)? How does THAT work?

Not so tricky, in fact the rule is the same.

References do the same thing. You get a copy of the reference.

So if I say:

```
Cat A = new Cat();

Cat B = A;
```



The remote control in A is copied. Not the object it refers to.

You've still got just one Cat object.

But now you have **two different references** (remote controls) controlling the **same** Cat object.

NOW let's look at passing values to methods

Java is pass-by-value.

Always.

That means **"copy the value, and pass the copy."**

For primitives, it's easy:

```
int x = 5;

doStuff(x); // pass a COPY of x (the value 5) to the doStuff method
```

The doStuff method looks like this:

```
void doStuff(int y) {

   // use y in some way

}
```

A copy of the value in x, which is 5, is passed into the doStuff() method.

The doStuff() method has its own new cup, called y, waiting.

The y cup is a new, different cup. With a copy of what was in x at the time it was passed. From this point on, y and x have no affect on each other. If you change y, you don't touch x.

```
void doStuff(int y) {

   y = 27; // this does NOT affect x

}
```

And vice-versa. If you change x, you don't change y.

The only part x had in this whole business was to simply copy its value and send that copy into the doStuff() method.

How does pass-by-value work with references?

Way too many people say "Java passes primitive by value and objects by reference". This is not the way it should be stated. Java passes everything by value. With primitives, you get a copy of the contents. With references you get a copy of the contents.

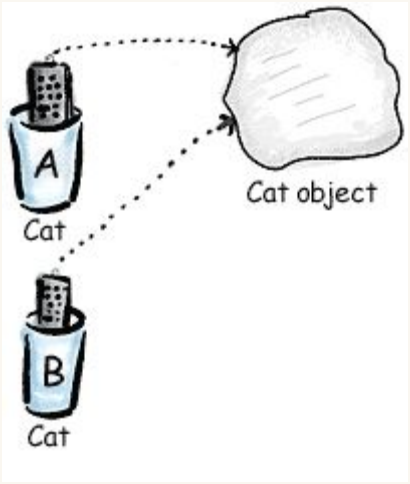
But what is the contents of a reference?

The remote control. The means to control / access the object.

When you pass an object reference into a method, you are passing a COPY of the REFERENCE. A clone of the remote control. The object is still sitting out there, waiting for someone to use a remote.The object doesn't care how many remotes are "programmed" to control it. Only the garbage collector cares (and you, the programmer).

So when you say:

```
Cat A = new Cat();

doStuff(A);

void doStuff(Cat B) {

   // use B in some way

}
```



There is still just ONE Cat object. But now TWO remote controls (references) can access that same Cat object.

So now, anything that B does to the Cat, will affect the Cat that A refers to, but it won't affect the A cup!

You can change the Cat, using your new B reference (copied directly from A), but you can't change A.

What the heck does that mean?

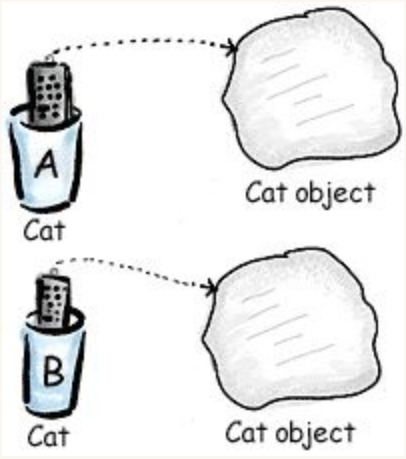
You can change the object A refers to, but you can't take the A reference variable and do something to it -- like redirect it to reference a different object, or null.

So if you change the B reference (not the Cat object B refers to, but the B reference itself) you don't change A. And the opposite is true.

So...

```
Cat A = new Cat();

doStuff(A);

void doStuff(Cat B) {

   B = new Cat(); //did NOT affect the A reference

}
```

Doing this simply "points" B to control a different object. A is still happy.



So repeat after me:

Java is pass-by-value.

(OK, once again... with feeling.)

Java is pass-by-value.

For primitives, you pass a copy of the actual value.

For references to objects, you pass a copy of the reference (the remote control).

You never pass the object. All objects are stored on the heap. Always.

Now go have an extra big cup of coffee and write some code.