

# Re: Java collections

---

*Source:* <http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2006-05/msg00281.html>

---

- *From:* Eric Sosman <[Eric.Sosman@xxxxxxx](mailto:Eric.Sosman@xxxxxxx)>
  - *Date:* Wed, 03 May 2006 16:44:59 -0400
- 

tim@xxxxxxxxxxxxxxx wrote On 05/03/06 15:44,:

So then, each row of the collection would automatically point to the location in memory for the Customer that was referenced by the Customer variable when the add was done. And the garbage collection would know not to destroy the old Customers that are being referenced by previous rows of the collection, even though they are no longer the location in memory referenced by the Customer variable.

It seems you're still a little bit confused, so let me try to hammer home a point that I think will clear things up for you. Chisel the following on a suitable slab of marble and hang it on the wall at your workplace:

## VARIABLES ARE NOT OBJECTS

Now, what on Earth could I possibly mean by this nonsense? Of course, variables are (or can be) objects! When I write `int i' it means "the variable i is an int," so when I write `Customer c' it must mean "the variable c is a Customer." I must be wacko to think otherwise, right?

Well, no: Analogy is not proof, and two syntactic forms that look similar don't necessarily mean the same thing. `Customer c' means that c is a *\*reference\** to a Customer, not that c *\*is\** a Customer. The Customer instance itself lives somewhere out in hyperspace, and c holds a reference to it. When you do

```
Customer c1 = new Customer("Tim");
Customer c2 = c1;
```

.... there is only one Customer instance in play, and both reference variables c1 and c2 point to it. If the above is followed by

## Re: Java collections

`c2 = null;`

this does not destroy the Customer instance, nor influence it in any way: the instance lives on, even if one of the many references to it has disappeared. (Write your phone number on a piece of paper, then burn the paper: Did your phone catch fire?)

Now: When you call `collection.add(c1)`, what manner of argument does the `add()` method receive? You're catching on: it receives a copy of the reference `c1`, so the argument is yet another reference to that lonely Customer instance. What do you suppose `add()` puts into the collection? Yes, it salts away a copy of its argument, that is, a reference to that very same Customer. After `add()` returns, there are two references to the Customer in your program: the variable `c1` and another one somewhere in the collection. (`c2`, you'll recall, has been set to null and no longer refers to the Customer.)

So:

- Making lots of copies of a reference doesn't make lots of copies of the Customer it refers to. (Photocopying your phone number doesn't clone your phone.)
- If you make a change to the Customer, that change becomes apparent via all the references. (Change the "I'm busy now" message on your phone, and all callers will hear the new message even if they got your number from an old piece of paper.)
- Any reference to a particular Customer is just as good as any other; they all work identically. (It doesn't matter which piece of paper somebody reads to find your phone number; your phone rings anyhow.)

The last point also applies to the garbage collector: as long as at least one reference to the Customer survives, Java knows that the Customer might still be used and therefore will not treat it as garbage. Since all references are equivalent, a reference inside a collection is every bit as effective as an ordinary reference variable: as long as the collection still refers to the Customer (and as long as the collection itself remains accessible), the Customer is accessible and Java won't discard it.

Concentrate on the distinction between references and instances, and I think much of your confusion will disappear.

—

Re: Java collections

Eric.Sosman@xxxxxxx