

# CS 228: Introduction to Data Structures

## Lecture 6

### Friday, September 2, 2016

**Example (Continued).** Recall Wednesday's two `Point` objects.

```
Point p = new Point(1, 2);  
Point q = new Point(3, 4);
```

If we now write:

```
p = q;
```

The object that `p` referenced before the statement (the `Point` with coordinates (1,2)) is now unreferenced — it becomes “garbage”. Unreferenced objects are eventually ***garbage collected***.

## Object Types versus Primitive Types

The next table compares object types and primitive types.

	Object types	Primitive types
<b>Contains a</b>	reference	built-in
<b>How defined?</b>	class definition	9, 42.5, 'h', false
<b>How created?</b>	"new"	default (usually zero)
<b>How used?</b>	method	operators: +, -, *, etc
<b>Testing equality</b>	<code>equals()</code> (override)	<code>==</code>

The last row deserves further explanation . . .

### Equality and the `equals()` Method

The operation "`x == y`" determines whether variables `x` and `y` have the same values. When `x` and `y` are of the same primitive type, this corresponds to the usual notion of equality testing. For instance, suppose `i` and `j` are `int` variables and we set `i = 2`. Then, if we set `j = 3`, `i == j` is false and if we set `j = 2`, `i == j` is true.

When `p` and `q` are of object types, however, “`p == q`” determines whether variables `p` and `q` **reference** the same object. It does not determine whether the objects are “the same”. For instance, suppose we execute the following statements.

```
Point p = new Point(1, 2);  
Point q = new Point(1, 2);
```

Intuitively, `p` and `q` are “the same”; however, `p == q` is false, because `p` and `q` reference different objects.

In Java, the notion of equality testing is captured by the `equals()` method. The default implementation of `equals()` in `java.lang.Object` just compares object references. Thus, if we used this default implementation, `p.equals(q)` would be false; i.e., no different than if we used `p == q`. If we want `p.equals(q)` to return `true` if and only if the `x` and `y` coordinates of `p` and `q` match (the usual notion of “sameness” of `Point` objects), we need to **override** the default implementation of `equals()`<sup>1</sup>.

---

<sup>1</sup>The Java `Point` class (`java.awt.Point`) implements `equals()` exactly as we want it: Points are equal only if they have the same coordinates.

The code below — which should be inserted within the body of the `Point` class — illustrates the standard way to override `equals()`.

```
@Override
public boolean equals(Object obj)
{
    if (obj == null ||
        obj.getClass() != this.getClass())
    {
        return false;
    }

    Point other = (Point) obj;
    return x == other.x && y == other.y;
}
```

Note that `obj`, the method argument, must be an `Object`, not a `Point`, because the signature of `equals()` in `java.lang.Object` is

```
boolean equals(Object obj)
```

and, to override a method, we must match its signature.

Our implementation of `equals()` first verifies that `obj` is not `null` and that it is also a `Point` object (if the other object is not a `Point`, we should not attempt to examine its coordinates). After verifying that `obj` is indeed a `Point`, we need to downcast `obj` to the `Point` type, to let the compiler know that we can access the instance variables `x` and `y`.<sup>2</sup>

**Note.** Some textbook authors and developers use the `instanceof` operator in the `equals()` method to test if objects are of the same class:

```
if (!(obj instanceof Point))  
    return false;
```

We will not do that here; instead, we will always test for class equality using `getClass()`. The reason is that `instanceof` does not work correctly when inheritance is involved; in fact it will be incorrect for subtypes of `Point`. To fully understand the issues, you need to understand the formal definition of `equals()`. We provide this formal definition next, for the benefit of the mathematically inclined — we will not cover it in class.

---

<sup>2</sup> Note that we can access the instance variables `x` and `y`, because `equals()` is defined within the body of the `Point` class.

**Formal Definition of `equals()`.** The Java documentation<sup>3</sup> specifies that `equals()` implements an *equivalence relation* on non-null object references. That is, it satisfies the following properties.

- **Reflexivity:** for any non-null reference value `x`, `x.equals(x)` should return `true`.
- **Symmetry:** for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- **Transitivity:** for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- **Consistency:** for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- **Nothing equals null except null:** For any non-null reference value `x`, `x.equals(null)` should return `false`.

---

<sup>3</sup> [http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals\(java.lang.Object\)](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals(java.lang.Object))

**Exercise:** Which of these properties imply that we should use `getClass()` instead of `instanceof()` in `equals()`?

## `equals()` and the `String` class

Suppose we do the following.

```
String s = "hurley";  
String t = "HURLEY".toLowerCase();
```

Now strings `s` and `t` contain the same characters, but they ***are not*** the same object.

```
System.out.println(s == t);           // false
```

As you probably saw in ComS 227, the proper way to test if two strings are the same (i.e., contain the same characters), is to use the `equals()` method.

```
System.out.println(s.equals(t));      // true
```

This behaves as you would expect, because the implementors of Java have done some work for you: the `String` class overrides `equals()` to check whether the characters are the same.

## Another Example

The same pattern we used for the `Point` class to override `equals()` for other classes. Here is an `equals()` method for the `Bee` class:

```
@Override
public boolean equals(Object o)
{
    if (o == null ||
        o.getClass() != getClass())
    {
        return false;
    }

    // typecast o to Bee so that we can
    // compare data members
    Bee b = (Bee) o;

    // Compare the data members and return
    // accordingly
    return b.size == size
}
```



```
        && (b.color == color ||  
        b.color != null &&  
        b.color.equals(color))  
        && (b.swarm == swarm ||  
        b.swarm != null &&  
        b.swarm.equals(swarm));  
    }
```

Note how `equals()` calls itself. Of course, this (recursive) call is to the `equals()` method for the `String` class (which we assume is implemented).

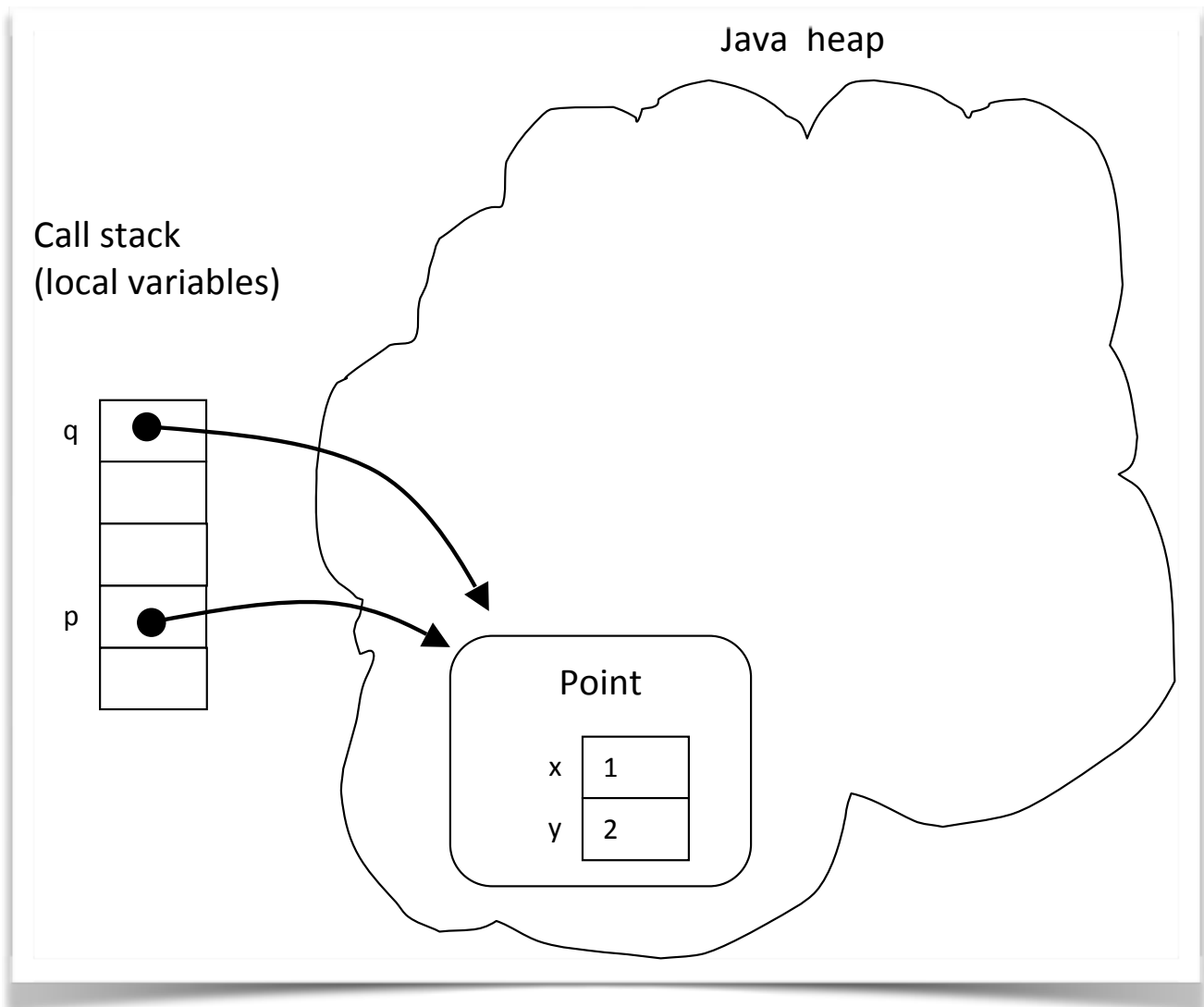
By inheritance, the same implementation can be used by any subclass of `Bee`, unless it is overridden by the subclass.

## Copying and Cloning

If we execute these two statements

```
Point p = new Point(1, 2);  
Point q = p;
```

the result is this:



The second assignment does not make a copy of the `Point` object, it just assigns the reference. That is, `q` is now an ***alias*** for `p`. This is potentially dangerous, since any changes we make through `q` affect `p` as well, which might not be what we want.

In order to make an actual copy of an object, we have to implement a special method to do so. Two common options are to

- write a ***copy constructor*** or
- write a cloning method.

We will illustrate these approaches using the `Point` class.

## Copy Constructors

A copy constructor initializes the object under construction (“this” object) using the values from an existing one:

```
public Point(Point existing)
{
    this.x = existing.x;
    this.y = existing.y;
}
```

Usage:

```
Point p = new Point(1, 2);
q = new Point(p);
```

## Cloning

A cloning method returns a ***new*** object from the values in this object. Here is an ad-hoc cloning method for the `Point` class.

```
public Point makeClone()  
{  
    Point copy = new Point();  
    copy.x = this.x;  
    copy.y = this.y;  
    return copy;  
}
```

Usage:

```
Point p = new Point(1, 2);  
q = p.makeClone();
```

Another option is to override Java's `Object.clone()` method. For this, you either have to explicitly declare that your class implements the `Cloneable` interface or some superclass of your class must implement `Cloneable`. Thus, the declaration for `Point` would be

```
public class Point implements Cloneable{...}
```

We will see how to override `clone()` next time.