Lecture 20

We continue the discussion of primitive types versus reference types. We show how this difference effects equality testing and parameter passing.

We also discuss garbage collection.

Annotate your copy of these notes with lots of pictures during the lecture

Equality testing using == compares contents

```
int x = 10;
int y = x;
int z = 10;
if (x == y)
{  System.out.println("Yes, x == y");
}
if (x == z)
{  System.out.println("Yes, x == z");
}

Location loc1 = new Location(8, 51);
Location loc2 = loc1;
Location loc3 = new Location(8, 51);
if (loc1 == loc2)
{  System.out.println("Yes, loc1 == loc2");
}
if (loc1 == loc3)
{  System.out.println("Yes, loc1 == loc3");
}
```


Using a method instead of ==

- For reference types, == is probably no good. So, include a method in every class definition for testing for equality in a more meaningful way, e.g.:

```
public boolean isEqualTo(Location aLoc)
{  return
      this.longitude == aLoc.longitude &&
      this.latitude == aLoc.latitude;
}
```

- Use this method when comparing:

```
if (loc1.isEqualTo(loc2))
{  System.out.println("Yes, equal");
}
if (loc1.isEqualTo(loc3))
{  System.out.println("Yes, equal");
}
```


Equality comparison for `String`s

- You often want to test whether two `String`s are equal

- It is tempting to write

```
if (s1 == s2) ...
```

  But this only returns `true` if `s1` and `s2` contain pointers to the same `String` object

- To compare the contents of two `String`s, the `String` class definition contains a method called `equals`:

```
if (s1.equals(s2)) ...
```


Parameter passing: it uses assignment

- Java uses *call-by-value*: the values of actual parameters are *assigned* into the formal parameters

- If the parameters are of primitive types,
  - the formal parameter merely contains a duplicate of the value of the actual parameter
  - so, any updates to the formal parameter within the method *do not* affect the actual parameter

- If the parameters are of reference types,
  - the formal parameter contains a duplicate of what was in the actual parameter, i.e. both reference ('point to') the same object
  - so, updates to the object referenced by the formal parameter *do* affect the original object

Parameter passing: primitive types

```
public class PassByValue
{
   public static void main(String[] args)
   {  int a = 8;
      int b = 51;
      System.out.println(
         "Before: " + a + ", " + b);
      method1(a, b);
      System.out.println(
         "After: " + a + ", " + b);
   }

   public static void method1(int x, int y)
   {  x = 10;
      y = 52;
      System.out.println(
         "In method1: " + x + ", " + y);
   }
}
```

Parameter passing: reference types

```
public class PassRef
{
   public static void main(String args[])
   {  Location loc = new Location(8, 51);
      System.out.println(
       "Before: " + loc);
      method2(loc);
      System.out.println(
        "After: " + loc);
   }

   public static void method2(Location aLoc)
   {  aLoc.move(10, 52);
      System.out.println(
         "In method2: " +  aLoc);
   }
}
```


Parameter passing

- Most methods that we write do not update their parameters anyway, so the above is not an issue. But in methods that do update their parameters, if those parameters have reference types, you need to take care

- You may even need to pass in a copy of the object so that the original is not updated:

  ```
  method2(new Location(loc.getLongitude(),
     loc.getLatitude()));
  ```

  or include a copy method in `Location` that returns a new object with the same contents:

  ```
  method2(loc.clone());
  ```

- Don't get hung up on this! It's rarely an issue

<table>
<tr><th>Primitive types</th><th>Reference types</th></tr>
<tr><th colspan="2">Equality testing</th></tr>
<tr><td>== tests for equal values</td><td>• == tests for equal references (pointing to same object)
• define your own equality testing method if you wish to compare, e.g., the values of instance variables instead of the pointers</td></tr>
<tr><th colspan="2">Passing as a parameter to a method</th></tr>
<tr><td>Copies the value, so changes made in the method do not affect the original</td><td>Copies the reference (hence two pointers to same object), so changes to the object made in the method do affect the original</td></tr>
</table>

Garbage collection

- In Java and C++, objects are created using `new`

- In C++ (but not in Java), objects are deleted and the memory space they occupy is reclaimed using `delete`

- Instead, in Java, a program, called the *garbage collector*, is run at certain times. It automatically detects unused objects, deletes them and frees up their memory space

- The C++ approach is more error-prone ('dangling pointers'); the Java approach is not suitable for time-critical applications

The Java garbage collector

- Which objects are eligible for deletion?
  - Objects that have no references (i.e. there are no longer any 'pointers' to them)
  - References are lost when variables reach the end of their lifetimes and when you set variables to `null`
- Even though an object is eligible for deletion, it may not be deleted immediately
- When is the garbage collector run?
  - When the amount of free memory has fallen below a threshold
  - At program termination
  - (If you invoke it explicitly e.g.: `System.gc();`)

When do objects become eligible for garbage collection?

```
public static void main(String[] args)
{  Location loc1 = new Location(8, 51);
   Location loc2 = new Location(8, 52);
   Location loc3 = new Location(8, 53);
   Location loc4 = new Location(8, 54);
   Location loc5 = loc1;
   loc = loc4;
   if (...)
   {  Location loc6 = new Location(8, 55);
      System.out.println(loc6);
   }
   loc2 = null;
   loc5 = null;
   loc3 = new Location(8, 56);
   System.out.println(loc3);
}

private static Location loc;
```