More on objects and inheritance

How to define **equals**

We just discussed how every class we write is automatically a child class of the `Object` class, and that the `Object` class has a method called `equals`. You are supposed to override this class with a proper definition. Sometimes you might forget, because if you use built-in classes, the equals method does what you would expect.

For example consider the following code:

```
import java.util.Arrays;

public class Test
{
  public static void main(String[] args)
  {
        String[] a = {"Foo","Bar","Zot"};
        if (Arrays.asList(a).contains("Bar"))
                System.out.println("Contains Bar");
        else
                System.out.println("No Bar");
        if (Arrays.asList(a).contains("Bah"))
                System.out.println("Contains Bah");
        else
                System.out.println("No Bah");

  }
}
```

This outputs “Contains Bar” and “No Bah” as you would expect. This code uses the Arrays class to turn the array a into a list, which then has a contains() method to search the list for a target. We’ll talk more about lists a bit later in the class. How does the contains() method work? It scans through each element in the list and called equals to see if it returns true when compared to “Bar” and “Bah”. It works because someone wrote the method as part of the String class that correctly compares the characters in the strings to see if they match.

What if we try this with our own class?

```
class Myclass
{
      private int val;
      public Myclass()
      {
            val = 0;
      }
      public Myclass(int newVal)
      {
            val = newVal;
      }
}
```

```
public class Test
{
  public static void main(String[] args)
  {
        Myclass[] a = new Myclass[2];
        a[0] = new Myclass(1);
        a[1] = new Myclass(1);
        Myclass target = new Myclass(1);
        if (Arrays.asList(a).contains(target))
                System.out.println("Contains target");
        else
                System.out.println("No target");

  }
}
```

The contains method doesn't work on a new Myclass(1), even though the classes contain the same thing. This is because we didn't define our own equals method for our class, so Java just compares the addresses in memory where each object is stored to see if they match.

To fix this problem, we override the equals method.

First, what is the difference between these two methods?

```
public boolean equals(Myclass otherclass)
{
      return otherclass.val == this.val;
}

public boolean equals(Object otherclass)
{
      Myclass otherObj = (Myclass) otherclass;
      return otherObj.val == this.val;
}
```

The two equals methods have different parameter types, so the first one actually doesn't override the definition of equals defined at the Object level. We have merely overloaded the method equals. The class now has two methods named equals. If passed in a Myclass object, it will use the first definition.

With the second definition, we will get "true" returned for both tests for contains. This is because Java will now call the equals method for each element in the array and we get true when the values are the same. Note that we have to "downcast" from Object to Myclass. If we were given something that isn't really a Myclass object, then Java will likely crash (in general it is safe to "upcast").

To flesh out our implementation more, we should check to make sure that a Myclass object is passed in and that it is not null. If it's null we will get an error trying to access .val:

```
public boolean equals(Object otherclass)
{
      if (otherclass == null)
            return false;
      else if (getClass() != otherclass.getClass())
            return false;
      else
      {
            Myclass otherObj = (Myclass) otherclass;
            return otherObj.val == this.val;
      }
}
```

The getClass() method returns a representation of the class used to create the object. This method is marked final, which means it can't be overridden. The return value can be used with == and !=.

There is a similar method, instanceof, that tells you if an object is an instance of another class. For example:

```
      else if (!otherclass instanceof Myclass)
            return false;
```

The only problem with this is that if you have a class and subclass (like in our Person and Man/Woman example) and pass a Woman object in for a Person object (e.g. person.equals(woman)) then this will return false since woman is not an instance of a Person object.