Lab 3: The Object Class

In this lab, you will create two classes that override methods from *Object*.

Objectives:

- To properly override methods from the Object class

Part 1: Getting Started

Steps:

_1. In Eclipse, create a new Java project named **Lab03**.

_2. To save you typing, we will provide you with two basic classes to which you will add overrides of *Object* methods. To import the classes into your Eclipse workspace and project, follow these steps:

   a. Open Windows Explorer or My Computer and browse to **{Lab Installation Directory}**/starters/lab03.

      If your computer has the standard lab setup on Windows Vista or later, the directory is:

      ```
      c:\Users\username\javaclass-p2\starters\lab03
      ```

   b. Right-click on the *logic* folder and choose *Copy* to copy the folder to the Windows clipboard.

   c. Back in Eclipse, in the Package Explorer, right-click on the *Lab03/src* folder and choose *Paste*.

_3. Examine the two imported classes and ensure that you understand them. In particular, note that each defines several fields and that neither overrides any methods from *Object*. That's your job, starting in the next part of lab.

Part 2: Overriding Methods from Object

In this part, you will modify both classes to override:

- toString
- equals
- hashCode
- clone

Steps:

_1. Edit both classes and override *toString*: both methods should return labeled values of the class's fields.

_2. Edit both classes and examine their fields so that you can override *equals*. Before you write any code, you need to decide what it means for objects of each class to be the same.

   Starting with *Policy*, note that it defines *policyName* and *amount* fields, the combination of which should be unique across all Policy objects.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

For *Customer*, the situation's a bit different. For Customers, the only way to determine uniqueness is to compare all of the fields. Please understand that this is probably not the best way to design a Customer class, but assume that due to the application domain, we cannot include any unique field in the Customer class.

Then override *equals* in both classes. You can use the sample code in the course notes to help you get started.

_3. Now you must override *hashCode* in both classes so that objects of the classes will work correctly in hashtable-based collections. Remember that the general contract for *hashCode* states that the value returned from *hashCode* should depend on the same fields as in the *equals* method.

Bloch presents a general recipe for calculating a suitable hashcode in the book *Effective Java Programming Guide*, published by Addison-Wesley Professional. The following is an extract from that book:

```
1. Store some constant nonzero value, say 17, in an int
   variable called result.

2. For each significant field f in your object (each field
   taken into account by the equals method, that is), do
   the following:

    a. Compute an int hash code c for the field:

      i. If the field is a boolean, compute (f ? 0 : 1).
     ii. If the field is a byte, char, short, or int, compute (int)f.
    iii. If the field is a long, compute (int)(f ^ (f >>> 32)).
     iv. If the field is a float compute Float.floatToIntBits(f).
      v. If the field is a double, compute Double.doubleToLongBits(f),
         and then hash the resulting long as in step 2.a.iii.
     vi. If the field is an object reference and this class's equals
         method compares the field by recursively invoking equals,
         recursively invoke hashCode on the field.
    b. Combine the hash code c computed in step a into result as
       follows:

        result = 37*result + c;

3. Return result.
4. When you are done writing the hashCode method, ask yourself whether
   equal instances have equal hash codes. If not, figure out why and
   fix the problem.
```

For the *Customer* class, since all of the fields are Strings, you can apply 2.a.vi for each of the field String objects, using code like:

```
int result = 17;

result = 37*result + firstName.hashCode();
result = 37*result + lastName.hashCode();
. . .
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

For the *Policy* class, you can do the same for the *policyName* String, but for the *amount* use step 2.a.v on the *double* to calculate an *int* that you can combine with the result calculated from the *policyNumber*. Your code might look something like:

```
long l = Double.doubleToLongBits(amount);
int i = (int)(l ^(l >>>32));
result = 37*result + i;
```

_4. Next, it's time to override *clone*. For each class, follow these steps:

   a. Ensure that the class implements the *Cloneable* interface.

   b. Override the *clone* method, performing a deep copy for all non-primitive, mutable fields.

      Note that java.time.LocalDate is an immutable class (unlike java.util.Date).

Part 3: Testing the Classes

In this part, you will write a program that tests the functionality of your Object-class overrides.

Steps:

_1. Create a new class named **Testing** in the *part3* package. This class should have a *main* method.

_2. First, test the *toString* override. In *main*, create a *Customer* and a *Policy*, then call System.out.println on each of the objects.

   Run the Testing class as a Java Application -- you should see the results of your toString methods. If you don't like the way they look, fix the methods before continuing.

_3. Now let's test the *equals* method. For each class, create three objects: two with identical values for the important fields, the third with at least one important field different. (An "important" field is one that is examined by *equals*.) Then write code to call *equals* to compare objects that should be equal, and objects that should not be.

   Run and test your program.

_4. Next, let's test the *hashCode* method. Follow these steps for the Customer class:

   a. Modify the Customer class's *hashCode*, *equals* and *toString* methods so that they print out an "in xxxx" message to the console, where "xxxx" is the method name.

   b. Then, back in *main*, create another couple of *Customer* objects, ensuring that they have different fields than any you've already created.

   c. Then create a *HashSet* and add all of the unique Customer object references, bracketing the add with console output. Your code should look something like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
HashSet<Customer> hs = new HashSet<Customer>();
Customer c4 = new Customer("Slim", "Dunlap", "K", "Jr.");
Customer c5 = new Customer("Bob", "Stinson", "Q", "");

System.out.println("Adding customers ---------");
hs.add(c1);
hs.add(c3);
hs.add(c4);
hs.add(c5);
System.out.println("Done adding customers ---------");
```

Run your program and note how the HashSet calls the *hashCode* method as the object references are added.

d. Create another Customer object with identical data as the last one you added to the collection. Then pass this new object's reference to the HashSet's *contains* method, which searches the collection for the specified object. Display the resulting boolean to the console. Your code should look something like:

```
Customer c6 = new Customer("Bob", "Stinson", "Q", "");

System.out.println("Finding a customer");
boolean b = hs.contains(c6);
System.out.println("Found customer in HashSet? " + b);
```

e. Run your program and note how the collection first calls *hashCode* on the brand new object. The collection then internally uses the hash code to quickly find one or more candidate objects in the collection. To confirm the choice, the collection then calls the slower equals method on the candidate object(s) -- that's required because it's possible for un-equal objects to have the same hash code.

f. Repeat the above steps to test the *Policy* class's hashCode implementation.

_5. Finally, let's test the *clone* method in each class. Follow these steps:

a. Create a new Customer with different fields than any other Customer. We will refer to this object as *c7*.

b. Create a new Policy with different fields -- pass the *c7* reference to the Policy constructor.

c. Create a clone of the new Policy.

d. Retrieve the Customer reference from the cloned Policy. We will refer to this reference as *c8*. If your Policy clone method did a proper deep copy, this should be a new, separate Customer.

e. Call the *setFirstName* method on *c8* to assign a unique first name -- if you have executed a deep copy, this should not affect *c7*.

f. Print the *c7* and *c8* references to the console and compare the first name fields. They should be different.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.