Cloning

Just like the name implies, cloning is making a copy of something. To be true to the nature of cloning, it should be an **exact** copy. While this can be very useful, it is not always necessary. For example, you have already been writing lots of programs and probably have never used cloning. So depending on your application, you may or may not need it. There are also manual processes that you can use that do the same thing as cloning. Regardless of whether or not you use cloning, you definitely need to understand how references to objects are handled or you can end up with programs that do unexpected and undesirable things.

Here is a sample class to kick things off:

```
import java.util.ArrayList;

public class Candy
{
     // Would need more accessors/mutators
     private int calories;
     private ArrayList<String> ingredients;

     public Candy()
     {
          ingredients = new ArrayList<String>();
          ingredients.add("sugar");
     }
     public void addIngredient(String ingred)
     {
          ingredients.add(ingred);
     }
     public void setCalories(int cal)
     {
          calories = cal;
     }
     public String toString()
     {
          String s;
          s = calories + " calories. Ingredients: ";
          for (String i : ingredients)
          {
               s = s + i + " ";
          }
          return s;
     }
}
```

```
public class Test
{
     public static void main(String[] args)
     {
          Candy twix = new Candy();
          twix.addIngredient("caramel");
          twix.addIngredient("chocolate");
          twix.setCalories(546);
          System.out.println(twix);
               // calls overriden toString()
     }
}
```

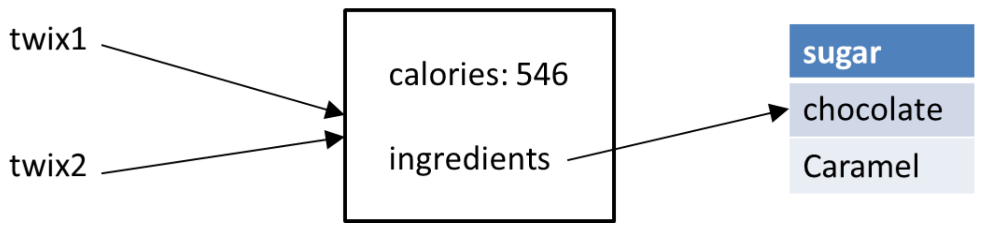
There shouldn't be any surprises so far. We are supposed to be talking about cloning, but first, we should point out that cloning is not the same as copying a reference:

```
          Candy twix1 = new Candy();
          twix1.addIngredient("caramel");
          twix1.addIngredient("chocolate");
          twix1.setCalories(546);

          Candy twix2 = twix1;

          System.out.println(twix2);
```

The output is the same as before. We have just copied a reference to the same place in memory:



We can see this in action if we change the calories, such as:

```
          Candy twix2 = twix1;
          twix1.setCalories(10);
          System.out.println(twix2);  // outputs 10
```

Note that there are some special exceptions to this general behavior. The most notable is the String class. A String in Java is immutable (not changeable) so when we change strings we are actually creating a new string, not editing the old string (the StringBuffer class edits the old string in memory).

e.g.:

```
String s1 = "hello"
String s2 = s1;
s1 = "bye";          // new string in memory
System.out.println(s2);    // Outputs hello
```

So going back to the Candy class, we saw that using = just makes a reference to the same class in memory. But what if we actually want twix2 to be set to a new copy of twix1? This is where clone will come in. We can make two kinds of copies of twix1. One type is called a **shallow copy**. In a shallow copy we just make a copy of all the member variables. This may or may not be sufficient or may result in errors. The other kind of copy is called a **deep copy** and it makes an exact copy of all structures in the class.

We can demonstrate first with a shallow copy. Here is a method on the Candy class that makes a shallow copy of the object and returns it.

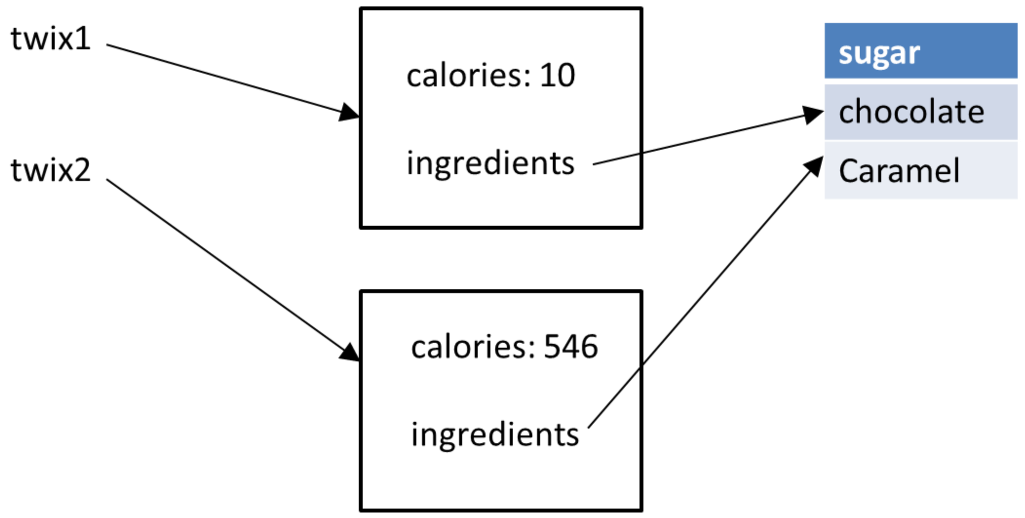
```
          public Candy shallowCopy()
          {
               Candy copy = new Candy();
               copy.calories = calories;
               copy.ingredients = ingredients;
               return copy;
          }

     In main:
               Candy twix2 = twix1.shallowCopy();
               twix1.setCalories(10);
               System.out.println(twix2);
```

This new version will output 546! It has created a new Candy object so twix1 and twix2 both reference different Candy objects, each with different calories. But what happens if we add a new ingredient to twix1?

```
               Candy twix2 = twix1.shallowCopy();
               twix1.setCalories(10);
               twix1.addIngredient("cocoa butter");
               System.out.println(twix2);
```

This outputs "cocoa butter" so we haven't really made totally separate versions of twix1. This is because the shallow copy just made a reference to the same ArrayList of ingredients:



It might be that this arrangement is fine – for example, if you are not going to be changing the ingredient list then you might not care and this would be easy code to write. On the other hand, if you are changing the ingredients but didn't realize that references were made to the same ingredient list, then you could get strange errors when you run your program.

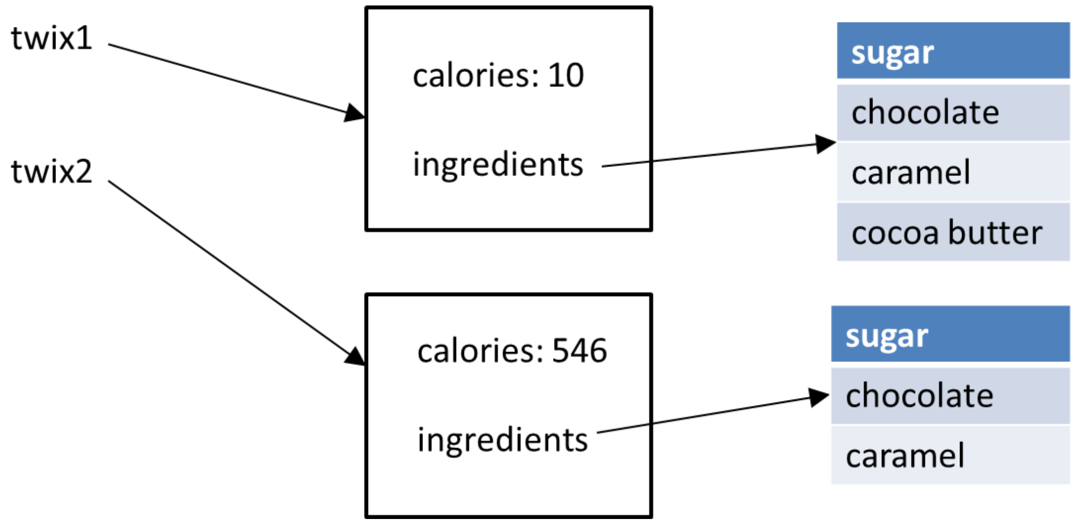
If we want to avoid this referencing problem then the solution is to make a deep copy. In a deep copy we make a duplicate copy of all structures referenced by the class. In this case we have to make a copy of the arraylist:

```
public Candy deepCopy()
{
     Candy copy = new Candy();
     copy.calories = calories;
     for (String i : ingredients)
          copy.addIngredient(i);
     return copy;
}
```

And in main:

```
Candy twix2 = twix1.deepCopy();
twix1.setCalories(10);
twix1.addIngredient("cocoa butter");
System.out.println(twix2);
```

Since we create a new ArrayList in the deepCopy method we now have no cross-interference between the two classes:



In this example we were able to make a deep copy pretty easily, but if we had more dynamic data structures (like linked lists) in the class then we might have a lot more complex copying to do.

Also note that we could avoid the deepCopy method and do the same thing in our calling code (main in this case) although it would generally be a lot messier, as you would have to get the calories, get the ingredients, then create a new Candy object and send in the calories and ingredients.

The Cloneable interface

Cloning is sufficiently important that Java comes with a standard interface for cloning. The interface is named **Cloneable** and has no methods but your classes inherit the clone() method from Object.

```
      protected Object clone()
```

If you try to clone a class that doesn't implement Cloneable then there is a CloneNotSupported exception. The clone method you write is supposed to:

1. Ensure that the copy != the calling object
2. Ensure that the classes of the copy and the calling object are the same
3. Ensure that copy.equals(calling object) is true

You get to decide if you want to do a deep or shallow copy in the clone method (if a shallow copy could break the equals method then you should make a deep copy), but you are supposed to invoke **super.clone()** to get an initial copy of your object to clone, then you can tack on specific things to copy from your class. This is important because if your class is a child class of a bunch of other classes, then we need to do whatever clone code is necessary to properly copy the parent class information. If the parent class is Object, then the implementation of clone() in the Object class checks if the actual class implements Cloneable, and creates an instance of that actual class.

```
public class Candy implements Cloneable
{
     …

     // Note can use sub-type of Object and still
     // override the method public Object clone()
     public Candy clone()
     {
          // First get copy via parent clone.
          // This must be in a try/catch block
          try
          {
               Candy copy = (Candy) super.clone();

               // Add on to copy specifics of Candy object
               // The line below is not needed because
               // super.clone() will at least do a shallow
               // copy for us
               // copy.calories = calories;
               copy.ingredients =
                    (ArrayList<String>)
                         ingredients.clone();
               // Need typecast since method returns Object
               return copy;
          }
          catch (CloneNotSupportedException e)
          {
               return null;
          }
     }
```

In main:

```
          Candy twix2 = twix1.clone();
          twix1.setCalories(10);
          twix1.addIngredient("cocoa butter");
          System.out.println(twix2);
```

The end result is the same as our home-brewed deepCopy but now we are using the standard clone interface supported by Java.

As you can see this interface is a little different than the other interfaces we have seen so far. Your class can only use the clone() method inherited from Object if it implements the Cloneable interface. This is called a **marker interface**.

If you think about a chain of classes in an inheritance hierarchy, each with a clone() method, then invoking super.clone() will call all the parent clone() methods until it gets to Object. At that point the Object class makes a new shallow copy of the child class and it is up to the overridden clone methods to fill in the rest of the details as needed.

Another subtle point worth mentioning is that the Object class itself does not implement the Cloneable interface (it wouldn't know how to do it for a generic object). It contains the clone() method for other classes that want to use it, but it is not possible to explicitly call the clone() method on an instance of the Object class although you can implicitly call it with super.clone() below the Object class.

Class exercise: Here is a class, help write the code to properly make a deep copy via clone:

```
public class Foo
{
      // This is an inner class, accessible
      // only in Foo. We will use it more
      // later.
      private class Node
      {
            public String str;
            public Node next;
            public Node(String str, Node next)
            {
                  this.str = str;
                  this.next = next;
            }
      }

      public String id;
      public Node name;

      public Foo(String id, String name1, String name2)
      {
            this.id = id;
            Node n2 = new Node(name2,null);
            Node n1 = new Node(name1, n2);
            name = n1;
      }
```

```
	public String toString()
	{
		return id + " " + name.str + " " + name.next.str;
	}
}

public class Test
{
	public static void main(String[] args)
	{
		Foo f1 = new Foo("39123","John","Doe");
		Foo f2 = f1.clone();
		f1.id = "99999";
		System.out.println(f2);  // Should be 39123 John Doe
	}
}
```