CS125 : Introduction to Computer Science

Lecture Notes #20
Copying and Mutability

Lecture 20 : Copying and Mutability

Copying an object in a constructor

Recall our code from the constructor lecture notes:

```
public class Clock
{
   private int hour;
   private int minutes;
   private boolean AM;

   // a constructor
   public Clock()
   {
      this.hour = 12;
      this.minutes = 0;
      this.AM = true;
   }

   // another constructor
   public Clock(int theHour, int theMinutes,
                boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }

   // a third constructor
   public Clock(Clock c)
   {
      this.hour = c.hour;
      this.minutes = c.minutes;
      this.AM = c.AM;
   }
   // a non-constructor instance method
   public void printClock()
   {
      System.out.print("Time is " + this.hour + ":");
      if (this.minutes < 10)
         System.out.print("0");
      System.out.print(this.minutes + " ");
      if (this.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }

   // another non-constructor instance method
   public void setTime(int theHour, int theMinutes, boolean theAM)
   {
      this.hour = theHour;
      this.minutes = theMinutes;
      this.AM = theAM;
   }
}  // end of class

public class ClockTest
{
   public static void main(String[] args)
   {
      // declare reference variables
      Clock home;
      Clock office;
      Clock car;

      // create objects; initialize references
      home = new Clock(2, 15, true);
      office = new Clock();
      car = new Clock(home);

      // change time on "home" clock
      home.setTime(8, 13, false);

      // call instance method to print
      //   instance variables
      home.printClock(); // Time is 8:13 PM.
      office.printClock(); // Time is 12:00 AM.
      car.printClock(); // Time is 2:15 AM.
   }
}
```

We pointed out at the time, that after the assignment of `car` was completed, the object that `car` pointed to, was a `copy` of the object that `home` pointed to. You did not have two references pointing to the same object; if we had wanted that, we would have assigned car as follows:

```
   car = home;
```

Instead, we used the following line:

```
   car = new Clock(home);
```

which called the following constructor:

```
   // a third constructor
   public Clock(Clock c)
   {
      this.hour = c.hour;
      this.minutes = c.minutes;
      this.AM = c.AM;
   }
```

and thus assigned the object that `this` pointed to (which in the end, is the same object that `car` points to), to hold the same values as the object that `c` points to (which is the same object that `home`, the constructor argument, points to. You have two different objects, which hold the same values.

If you are making a copy of an object like this, you need to be a bit more careful when reference variables come into play. For example, consider our `ExamScores` class from Lecture Notes #16 (though we've changed the access permission on the instance variables here):

```
public class ExamScores
{
   private int examNumber;
   private double average;
   private int[] scores;
}
```

If you write out a constructor to initialize a copy, like this:

```
public class ExamScores
{
   private int examNumber;
   private double average;
   private int[] scores;

   public ExamScores(ExamScores origVal)
   {
      this.examNumber = origVal.examNumber
      this.average = origVal.average;
      this.scores = origVal.scores;
   }
}
```

Then we have a small problem. First of all, we want to make it clear that access permissions are *not* the problem here – even though the instance variables are `private`, they can still be used directly by name *anywhere* in the `ExamScores` class. So, it's fine for the constructor to not only read and write the variables in the object that `this` points to, but also, to read and write the variables in the object that `origVal` points to. Instance methods of `ExamScores` can read and write any `ExamScores` object they have a reference to. It is when we are *outside* of the `ExamScores` class that the access permissions are relevant.

(We also want to make it clear that we *would* need other methods in this class, if we wanted it to be a useful class. Even if we only show the one or two methods we are talking about, we would want others as well.)

That said, the problem with the above constructor is that we are copying the *address* in `origVal.scores`, into `this.scores`. Copying the value from one variable to another is fine when you are copying a primitive type, such as with the other two assignments. Both `examNumber` variables hold their own copy of the same integer, and both `average` variables hold their own copy of the same floating-point value. But, similarly, both `scores` variables hold their own copy of the same address. But unlike with primitive-type variables, references lead to objects!! – if two reference variables hold the same address, then they point to the same object!

This can lead to trouble if it is possible to alter the array at all as a client. For example:

```
public class ExamScores
{
   private int examNumber;
   private double average;
   private int[] scores;

   public ExamScores(ExamScores origVal)
   {
      this.examNumber = origVal.examNumber
      this.average = origVal.average;
      this.scores = origVal.scores;
   }

   public void initialize()
   {
      for (int i = 0; i < this.scores.length; i++)
         this.scores[i] = 0;
   }
}
```

If you make a copy using the given constructor, then both `ExamScores` objects point to the same array. Now, if you call `initialize(...)` on one of the instances, the other instance gets initialized as well – writing to the array cells of one `ExamScores` object, means writing to the array cells of the other `ExamScores` object as well, since the two `ExamScores` objects point to the same array!!

This is known as a *soft copy* – we wanted to make a copy of an `ExamScores` object, but we made the copy in a way that allowed both objects to share a piece of dynamic memory. As a result, if we change that dynamic memory for one object, it's changed for the other as well – meaning the objects are not actually independent copies of each other. They are still tied together, since we can't change one without changing the other.

If we want a truly independent copy – a *hard copy* – then we need to copy the actual array object, not just its location:

```
public class ExamScores
{
   private int examNumber;
   private double average;
   private int[] scores;

   public ExamScores(ExamScores origVal)
   {
      this.examNumber = origVal.examNumber
      this.average = origVal.average;
      this.scores = new int[origVal.scores.length];
      for (int i = 0; i < this.scores.length; i++)
         this.scores[i] = origVal.scores[i];
   }

   public void initialize()
   {
      for (int i = 0; i < this.scores.length; i++)
         this.scores[i] = 0;
   }
}
```

Now the two `ExamScores` objects are *completely* independent. The array references even point to different arrays – though for now, those arrays hold the same primitive-type values. So, you can change one object in any way you want, and it will have no effect on the other object. That is what we prefer.

Note that you can sometimes get away with a "partial" soft copy. For example, what if we have a name for our exam, rather than a number:

```
public class ExamScores
{
   private String examName;
   private double average;
   private int[] scores;
}
```

In such a case, it would seem that the constructor for making a copy should be written like this:

```
public class ExamScores
{
   private String examName;
   private double average;
   private int[] scores;

   public ExamScores(ExamScores origVal)
   {
      this.examName = new String(origVal.examName);
      this.average = origVal.average;
      this.scores = new int[origVal.scores.length];
      for (int i = 0; i < this.scores.length; i++)
         this.scores[i] = origVal.scores[i];
   }
}
```

Such code takes advantage of the "initialize as a copy" constructor that exists in the `String` class. Actually, though, even though the above code is correct, we can also do this for that method:

```
   public ExamScores(ExamScores origVal)
   {
      this.examName = origVal.examName;
      this.average = origVal.average;
      this.scores = new int[origVal.scores.length];
      for (int i = 0; i < this.scores.length; i++)
         this.scores[i] = origVal.scores[i];
   }
```

Why is the soft copy okay for a `String` object? Because the `String` type is *immutable*. That is, once a `String` object is created, none of the variables are `public` *and* none of the instance methods allow any of the instance variables to be changed. Or in other words, it's impossible to change a `String` object once it is created. So, it's safe for two `ExamScores` objects to have references to the same `String` object – since you can't change the `String` object through either `ExamScores` object, you'd never have a situation where it was changed through one object and then as a result was changed for the other object as well. The worst you could do is have one of the `String` reference variables named `scores`, point to a different `String` object, but in that case, the other `ExamScores` object's `scores` variable could still point to the first `String`. Changing the `scores` reference variable of one `ExamScores` object won't change the other reference variable in the other object.

In contrast, objects that are *mutable* – objects that can be changed – would need to be properly copied. This is what we did with our array.