Chapter 17 Mutators, Replicators & Clones

17.0 Introduction

This chapter is about the relationship between Mutators & Replicators.

Mutators change an object.	— (Examples first appear in Chapter 4.)
Replicators copy an object.	— (Examples in section 17.3.)
The method *clone* is a Java specific replicator.	— (Examples in section 17.4.)

The issue that brought this chapter into being is illustrated by the following:

```
   Ogre ann, bob;
   ann = new Ogre();
   ann.setSize(5);
   ann.setColor(Color.green);
   ann.setSmiling(true);
→  bob = ann;
   bob.setSize(6);
```

One strongly suspects that the intent of the programmer is to create (and later display) two similar ogres; ogres which are the same except for size. This does not happen because only one ogre is created. The marked assignment just gives *bob* as an alternate reference for the ogre referenced by *ann*. At the end of the code above, there is just one, size 6, ogre. Displaying either *ann* or *bob* will show a size 6 image. For this to work as intended, the marked line should be something like:

```
bob = ann.duplicate();
```

The method *duplicate* should create a new ogre, transfer all of the attributes of the previous ogre to the new one, and then return the new ogre.

17.1 Immutable Objects

An immutable object is an object that cannot not be changed after it is constructed, an object with no mutators. The likelihood of programmer errors of the type discussed above is so high that objects are sometimes made without any mutators. Instead various kinds of 'make a modified copy methods' or modifying replicators are supplied. A simple example is the method *darker* in the *Color* class.

```
Color darkGreen = Color.green.darker().darker();
```

The line above creates a twice darkened version of *Color.green* without changing the original. The method *darker* makes a new, darker, *Color*. The example makes two new *Colors*, each darker than the last. The first of the two new *Colors* is discarded.

The java classes *Color*, *Font*, and *String* are examples of immutable objects. A programmer knows that giving a reference to one of these to another programmer's code will not result in a change to the original. *Color* and *String* have modifying replicators (*Color*: *brighter*, *darker*; *String*: *substring*, *toUpperCase*,…).

17.2 Distinguishing Mutators & Replicators

Mutators and Replicators can generally be distinguished by their return types.

Mutators: very often have a *void* return type. Occasionally a *boolean* value is returned indicating whether or not the change was made. Even less commonly a more complex data item is returned indicating the nature of the change made. Only in very special situations would a mutator return the same type as the object being mutated.

Replicators: must return the new object. As one would expect, the declared return type of a replicator is usually the type of object being replicated. However, the very important replicator *clone* declares its return type to be *Object*. The *clone* method is discussed in section 17.4.

17.3 Replicators

Replicators make a new object which is a copy of the object addressed. This copy is based on the addressed object. Sometimes an exact copy is returned. In other cases a modified copy is returned.

The queens on a chessboard example of section 15.2.2 contains an example of a replicator. The *makeFuller* method in the *Board* class makes a new chessboard containing an additional queen. It returns the new board as its result. If the modification fails, *null* is returned.

```
class Board
    {
    public static final int SIZE = 4;
    private int howMany;
    private int board[];

    public Board makeExactCopy()
        {
        Board b = new Board();
        b.howMany = howMany;
        for (int c=0; c<board.length; c++)
            b.board[c] = board[c];
        return b;
        }
```

The instance variables of the class *Board* from section 15.2.2 and a method *makeExactCopy* are shown at the left. The copy is made by creating a new board *b = new Board()* and then each instance variable is copied.

The elements of the array are copied one by one. It is tempting to write *b.board = board* with the expectation that this will copy the array. However, in Java, the = operator only copies scalar variables (*int*, *char*, *double*, *boolean*…). The expression *b.board = board* would not copy the elements of the array. The use of this expression would result in both objects referring to the same array — if an array element in one object were changed, the corresponding element in the other object would change with it.

Since whole objects (this includes arrays) are not copied by '=', it is necessary to explicitly copy their elements when making a copy. If an object contains objects which refer to other objects which refer to other objects… such copying is tedious at best. To help alleviate this problem, there is a standard way to copy objects.

The standard form for copying objects is the subject of the next section. Extensive special language features were not created for copying objects. As a result, the standard form for copying objects is rather complex.

17.4 Clones

A replicator that creates an unmodified copy is called a cloning method, the returned object being the clone. In Java, there is a standard signature for a cloning method:

- A clone method should have the signature: `public Object clone()`

Object is a special class in Java. All classes extend *Object*, directly or indirectly. When you create a class without declaring that you are extending some other class, the Java compiler creates your class as an extension of *Object*. Thus a return type of *Object* means that any kind of object can be returned.

The important positive consequence of all clone methods having the same signature is that the clone method of a subclass is called when appropriate. A less desirable consequence of this consistent signature is that the result of a clone method must be *cast* to the appropriate type. An object reference is *cast* by placing the type that the object should be considered to have in parentheses:

```
private Ogre ornery, grouch;
private AButton fewerButton, moreButton;
...
grouch = (Ogre)ornery.clone();               // ornery.clone() is cast as an Ogre
moreButton = (AButton)fewerButton.clone();   // fewerButton.clone() is cast as an AButton
```

Cloning an Object containing Scalar Variables and References to Immutable Objects

A *clone* method includes a call to *super.clone* — which calls the *clone* method in the class *Object*. The *clone* method in *Object* copies all scalar variables and object references (but not any objects the references refer to).

The standard way to write a *clone* method for an object with only scalar variables and immutable objects is shown in this example:

```
class Ogre implements Cloneable                  // A class containing a clone method
    {                                            // is declared to implement Cloneable.
    private int left, top;
    private boolean smiling;
    private String name;
    private Color skinColor;
    ...
        public Object clone()
        {
        try
            {
            return super.clone();                // super.clone makes a new Ogre and
            }                                    // copies all of the scalar variables
        catch (CloneNotSupportedException e)     // and object references.
            {
            return null;
            }
        }
    }
```

The *clone* method above is used as long as none of the instance variables refers to a mutable object. It is common practice to have multiple references to an immutable object. This is ok, an immutable object will always look the same as any copy, since neither can be changed.

Cloning an Object containing References to Mutable Objects

The *clone* method in *Object* copies all scalar variables and object references (but not any objects the references refer to). When there are instance variables that refer to mutable objects, those objects need to be explicitly cloned.

```
class OgreCouple implements Cloneable
   {
   private Ogre him, her;
   private boolean married;
   ...
   public Object clone()
      {
      try
         {
         OgreCouple c = (OgreCouple)super.clone();
         if (him != null)                            // If him or her is not null, the referenced
            c.him = (Ogre)him.clone();               // object is explicitly cloned.
         if (her != null)
            c.her = (Ogre)her.clone();
         return c;
         }
      catch (CloneNotSupportedException e)
         {
         return null;
         }
      }
   }
```

Cloning an Object containing References to One-dimensional Arrays

Arrays are mutable objects — their elements can be changed. Arrays must be explicitly cloned and then if the elements of the array are object references, the objects the elements reference must also be cloned.

```
class OgrePicnic implements Cloneable
   {
   private Ogre crowd[];
   private int tableSizes[];
   ...
   public Object clone()
      {
      try
         {
         OgrePicnic p = (OgrePicnic)super.clone();   // super.clone makes a new OgrePicnic,
         p.crowd = (Ogre[])crowd.clone();             //    The array crowd must be cloned.
         for (int n=0; n<crowd.length; n++)           // The mutable objects referenced by
            if (crowd[n] != null)                     //    crowd are explicitly cloned.
               p.crowd[n] = (Ogre)crowd[n].clone();
         p.tableSizes = (int[])tableSizes.clone();   // The array tableSizes must be cloned.
         return p;
         }
      catch (CloneNotSupportedException e)
         {
         return null;
         }
      }
   }
```

Cloning an Object containing References to Multi-dimensional Arrays

If an object contains a two-dimensional array, then the whole array and each non-null sub-array in the array must be cloned. If the entries are object references, then referenced non-null objects must also be cloned. If the crowd of *Ogres* above were two-dimensional, it would be cloned as follows:

```
p.crowd = (Ogre[][])crowd.clone();
for (int n=0; n<crowd.length; n++)
   if (crowd[n] != null)
      {
      p.crowd[n] = (Ogre[])crowd[n].clone();
      for (int m=0; m<crowd[n].length; m++)
         if (crowd[n][m] != null)
                    p.crowd[n][m] = (Ogre)crowd[n][m].clone();
      }
```

If an object contains a three-dimensional array, it is equally true that the whole array and each non-null sub-array must be cloned. If the entries are object references, then referenced non-null objects must also be cloned. If the crowd of *Ogres* above were three-dimensional, it would be cloned as follows:

```
p.crowd = (Ogre[][][])crowd.clone();
for (int n=0; n<crowd.length; n++)
   if (crowd[n] != null)
      {
      p.crowd[n] = (Ogre[][])crowd[n].clone();
      for (int m=0; m<crowd[n].length; m++)
         if (crowd[n][m] != null)
            {
            p.crowd[n][m] = (Ogre[])crowd[n][m].clone();
            for (int q=0; q<crowd[n][m].length; q++)
               if (crowd[n][m][q] != null)
                  p.crowd[n][m][q] = (Ogre)crowd[n][m][q].clone();
            }
      }
```

Cloning an Object containing References to Standard Java Objects

Many standard Java classes have no *clone* method. Generally, this is for good reason:

1. Immutable objects do not need to be cloned.
2. Some objects contain components that are not written in Java. Providing *clone* methods for these objects would be excessively difficult. Examples include *Buttons* & *TextFields*.