THE PROTOTYPE PATTERN

The Protoype pattern is used when creating an instance of a class is very time-consuming or complex in some way. Then, rather than creating more instances, you make copies of the original instance, modifying them as appropriate.

Prototypes can also be used whenever you need classes that differ only in the type of processing they offer, for example in parsing of strings representing numbers in different radixes. In this sense, the prototype is nearly the same as the Examplar pattern described by Coplien [1992].

Let’s consider the case of an extensive database where you need to make a number of queries to construct an answer. Once you have this answer as a table or ResultSet, you might like to manipulate it to produce other answers without issuing additional queries.

In a case like one we have been working on, we’ll consider a database of a large number of swimmers in a league or statewide organization. Each swimmer swims several strokes and distances throughout a season. The “best times” for swimmers are tabulated by age group, and many swimmers will have birthdays and fall into new age groups within a single season. Thus the query to determine which swimmers did the best in their age group that season is dependent on the date of each meet and on each swimmer’s birthday. The computational cost of assembling this table of times is therefore fairly high.

Once we have a class containing this table, sorted by sex, we could imagine wanting to examine this information sorted just by time, or by actual age rather than by age group. It would not be sensible to recompute these data, and we don’t want to destroy the original data order, so some sort of copy of the data object is desirable.

Cloning in Java

You can make a copy of any Java object using the **clone** method.

```
Jobj j1 = (Jobj)j0.clone();
```

The clone method always returns an object of type Object. You must cast it to the actual type of the object you are cloning. There are three other significant restrictions on the clone method:

1. It is a protected method and can only be called from within the same class or the module that contains that class.
2. You can only clone objects which are declared to implement the Cloneable interface.
3. Objects that cannot be cloned throw the CloneNotSupported Exception.

This suggests packaging the actual clone method inside the class where it can access the real clone method:

```
public class SwimData implements Cloneable
{
  public Object clone()
  {
     try{
      return super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
}
```

This also has the advantage of encapsulating the try-catch block inside the public clone method. Note that if you declare this public method to have the same name “clone,” it must be of type Object, since the internal protected method has that signature. You could, however, change the name and do the typecasting within the method instead of forcing it onto the user:

```
public SwimData cloneMe()
  {
     try{
      return (SwimData)super.clone();
     }
     catch(Exception e)
            {System.out.println(e.getMessage());
     return null;
     }
  }
```

You can also make special cloning procedures that change the data or processing methods in the cloned class, based on arguments you pass to the clone method. In this case, method names such as *make* are probably more descriptive and suitable.

Using the Prototype

Now let’s write a simple program that reads data from a database and then clones the resulting object. In our example program, SwimInfo, we just read these data from a file, but the original data were derived from a large database as we discussed above.

Then we create a class called Swimmer that holds one name, club name, sex and time

```
class Swimmer
{  String name;
   int age;
   String club;
   float time;
   boolean female;
```

and a class called SwimData that maintains a vector of the Swimmers we read in from the database.

```
public class SwimData implements Cloneable
{
  Vector swimmers;
  public SwimData(String filename)
  {
     String s = "";
     swimmers = new Vector();
     //open data file
     InputFile f = new InputFile(filename);
     s= f.readLine();   //read in and parse each line
     while(s != null)
     {
        swimmers.addElement(new Swimmer(s));
        s= f.readLine();
     }
     f.close();
  }
```

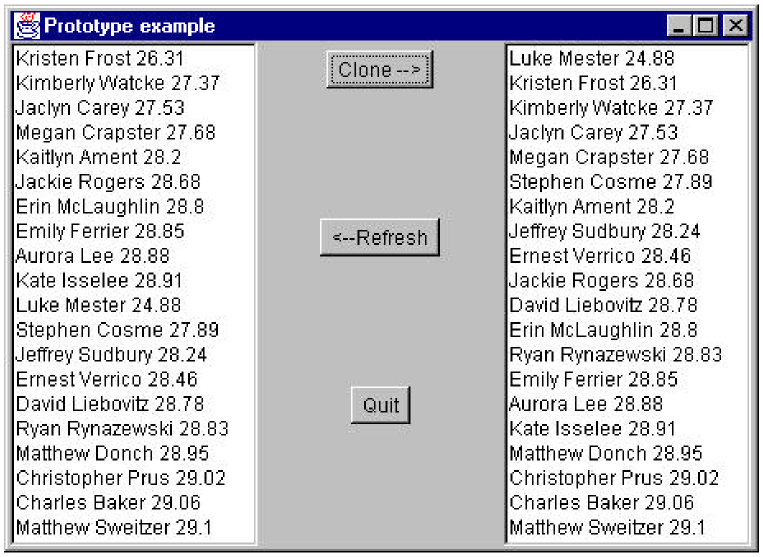
We also provide a *getSwimmer* method in SwimData and *getName*, *getAge* and *getTime* methods in the Swimmer class. Once we’ve read the data into SwimInfo, we can display it in a list box.

```
swList.removeAll();     //clear list
for (int i = 0; i < sdata.size(); i++)
{
 sw = sdata.getSwimmer(i);
 swList.addItem(sw.getName()+" "+sw.getTime());
}
```

Then, when the user clicks on the Clone button, we’ll clone this class and sort the data differently in the new class. Again, we clone the data because creating a new class instance would be much slower, and we want to keep the data in both forms.

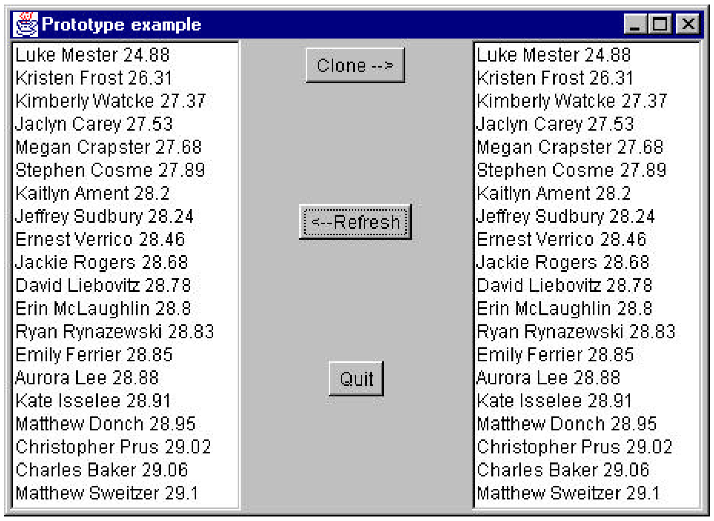
```
    sxdata = (SwimData)sdata.clone();
    sxdata.sortByTime();   //re-sort
    cloneList.removeAll(); //clear  list

    //now display sorted values from clone
    for(int i=0; i< sxdata.size(); i++)
      {
       sw = sxdata.getSwimmer(i);
       cloneList.addItem(sw.getName()+" "+sw.getTime());
     }
```

In the original class, the names are sorted by sex and then by time, while in the cloned class, they are sorted only by time. In the figure below, we see the simple user interface that allows us to display the original data on the left and the sorted data in the cloned class on the right:



```
Kristen Frost 26.31           Luke Mester 24.88
Kimberly Watcke 27.37         Kristen Frost 26.31
Jaclyn Carey 27.53            Kimberly Watcke 27.37
Megan Crapster 27.68          Jaclyn Carey 27.53
Kaitlyn Ament 28.2            Megan Crapster 27.68
Jackie Rogers 28.68           Stephen Cosme 27.89
Erin McLaughlin 28.8          Kaitlyn Ament 28.2
Emily Ferrier 28.85           Jeffrey Sudbury 28.24
Aurora Lee 28.88              Ernest Verrico 28.46
Kate Isselee 28.91            Jackie Rogers 28.68
Luke Mester 24.88             David Liebovitz 28.78
Stephen Cosme 27.89           Erin McLaughlin 28.8
Jeffrey Sudbury 28.24         Ryan Rynazewski 28.83
Ernest Verrico 28.46          Emily Ferrier 28.85
David Liebovitz 28.78         Aurora Lee 28.88
Ryan Rynazewski 28.83         Kate Isselee 28.91
Matthew Donch 28.95           Matthew Donch 28.95
Christopher Prus 29.02        Christopher Prus 29.02
Charles Baker 29.06           Charles Baker 29.06
Matthew Sweitzer 29.1         Matthew Sweitzer 29.1
```

The left-hand list box is loaded when the program starts and the right-hand list box is loaded when you click on the **Clone** button. Now, let’s click on the **Refresh** button to reload the left-hand list box from the original data.



Luke Mester 24.88	Luke Mester 24.88
Kristen Frost 26.31	Kristen Frost 26.31
Kimberly Watcke 27.37	Kimberly Watcke 27.37
Jaclyn Carey 27.53	Jaclyn Carey 27.53
Megan Crapster 27.68	Megan Crapster 27.68
Stephen Cosme 27.89	Stephen Cosme 27.89
Kaitlyn Ament 28.2	Kaitlyn Ament 28.2
Jeffrey Sudbury 28.24	Jeffrey Sudbury 28.24
Ernest Verrico 28.46	Ernest Verrico 28.46
Jackie Rogers 28.68	Jackie Rogers 28.68
David Liebovitz 28.78	David Liebovitz 28.78
Erin McLaughlin 28.8	Erin McLaughlin 28.8
Ryan Rynazewski 28.83	Ryan Rynazewski 28.83
Emily Ferrier 28.85	Emily Ferrier 28.85
Aurora Lee 28.88	Aurora Lee 28.88
Kate Isselee 28.91	Kate Isselee 28.91
Matthew Donch 28.95	Matthew Donch 28.95
Christopher Prus 29.02	Christopher Prus 29.02
Charles Baker 29.06	Charles Baker 29.06
Matthew Sweitzer 29.1	Matthew Sweitzer 29.1

Why have the names in the left-hand list box also been re-sorted?. This occurs in Java because the clone method is a *shallow copy* of the original class. In other words, the references to the data objects are copies, but they refer to the same underlying data. Thus, any operation we perform on the copied data will also occur on the original data in the Prototype class.

In some cases, this shallow copy may be acceptable, but if you want to make a deep copy of the data, there is a clever trick using the serializable interface. A class is said to be *serializable* if you can write it out as a stream of bytes and read those bytes back in to reconstruct the class. This is how Java remote method invocation (RMI) is implemented. However, if we declare both the Swimmer and SwimData classes as Serializable,

```
public class SwimData
   implements Cloneable, Serializable

class Swimmer implements Serializable
```

we can write the bytes to an output stream and reread them to create a complete data copy of that instance of a class:

```
public Object deepClone()
  {
    try{
     ByteArrayOutputStream b = new ByteArrayOutputStream();
     ObjectOutputStream out = new ObjectOutputStream(b);
     out.writeObject(this);
     ByteArrayInputStream bIn = new
              ByteArrayInputStream(b.toByteArray());
     ObjectInputStream oi = new ObjectInputStream(bIn);
     return (oi.readObject());
    }
    catch (Exception e)
    {  System.out.println("exception:"+e.getMessage());
       return null;
    }
  }
```

This deepClone method allows us to copy an instance of a class of any complexity and have data that is completely independent between the two copies. The program SwimInfo on the accompanying CD-ROM contains the complete code for this example, showing both cloning methods.

Consequences of the Prototype Pattern

Using the Prototype pattern, you can add and remove classes at run time by cloning them as needed. You can revise the internal data representation of a class at run time based on program conditions. You can also specify new objects at run time without creating a proliferation of classes and inheritance structures.

One difficulty in implementing the Prototype pattern in Java is that if the classes already exist, you may not be able to change them to add the required clone or deepClone methods. The deepClone method can be particularly difficult if all of the class objects contained in a class cannot be declared to implement Serializable. In addition, classes that have circular references to other classes cannot really be cloned.

Like the registry of Singletons discussed above, you can also create a registry of Prototype classes which can be cloned and ask the registry object for a list of possible prototypes. You may be able to clone an existing class rather than writing one from scratch.

Note that every class that you might use as a prototype must itself be instantiated (perhaps at some expense) in order for you to use a Prototype Registry. This can be a performance drawback.

Finally, the idea of having prototype classes to copy implies that you have sufficient access to the data or methods in these classes to change them after cloning. This may require adding data access methods to these prototype classes so that you can modify the data once you have cloned the class.