THE ITERATOR PATTERN

The Iterator is one of the simplest and most frequently used of the design patterns. The Iterator pattern allows you to move through a list or collection of data using a standard interface without having to know the details of the internal representations of that data. In addition you can also define special iterators that perform some special processing and return only specified elements of the data collection.

Motivation

The Iterator is useful because it provides a defined way to move through a set of data elements without exposing how it does it. Since the Iterator is an *interface*, you can implement it in any way that is convenient for the data you are returning. *Design Patterns* suggests that a suitable interface for an Iterator might be

```
public interface Iterator
{
public Object First();
public Object Next();
public boolean isDone();
public Object CurrentItem();
}
```

where you can move to the top of the list, move through the list, find out if there are more elements and find the current list item. This interface is easy to implement and it has certain advantages, but the Iterator of choice in Java is Java's built-in Enumeration type.

```
public interface Enumeration
{
public boolean hasMoreElements();
public Object nextElement();
}
```

While not having a method to move to the top of a list may seem restrictive a first, it is not a serious problem in Java, because it is customary to obtain a new instance of the Enumeration each time you want to move through a list. One disadvantage of the Java Enumeration over similar constructs in C++ and Smalltalk is the strong typing of the Java language. This prevents the *hasMoreElements()* method from returning an object of the actual type of the data in the collection without an annoying requirement to cast the returned Object type to the actual type. Thus, while the Iterator or

Enumeration interface is that is intended to be polymorphic, this is not directly possible in Java.

Enumerations in Java

The Enumeration type is built into the Vector and Hashtable classes. Rather than the Vector and Hashtable implementing the two methods of the Enumeration directly, both classes contain an *elements* method that returns an Enumeration of that class's data:

```
public Enumeration elements();
```

This *elements()* method is really a kind Factory method that produces instances of an Enumeration class.

Then, you move through the list with the following simple code:

```
Enumeration e = vector.elements();
while (e.hasMoreElements())
       {
       String name = (String)e.nextElement();
       System.out.println(name);
       }
```

In addition, the Hashtable also has the *keys* method, which returns an enumeration of the keys to each element in the table:

```
public Enumeration keys();
```

This is the preferred style for implementing Enumerations in Java and has the advantage that you can have any number of simultaneous active enumerations of the same data.

Filtered Iterators

While having a clearly defined method of moving through a collection is helpful, you can also define filtered Enumerations that perform some computation on the data before returning it. For example, you could return the data ordered in some particular way, or only those objects that match a particular criterion. Then, rather than have a lot of very similar interfaces for these filtered enumerations, you simply provide a method which returns each type of enumeration, with each one of these enumerations having the same methods.

Sample Code

Let's reuse the list of swimmers, clubs and times we described in the Interpreter chapter, and add some enumeration capabilities to the KidData class. This class is essentially a collection of Kids, each with a name, club and time, and these Kid objects are stored in a Vector.

```
public class KidData
{
   Vector kids;
//----------------------------------------
   public KidData(String filename)    {
     //read in the kids from the text file
      kids = new Vector();
      InputFile f = new InputFile(filename);
      String s = f.readLine();
      while(s != null)   {
         if(s.trim().length() > 0)  {
         Kid k = new Kid(s);
         kids.addElement(k);
         }
         s = f.readLine();
      }
   }
   //--------------------------------
   public Enumeration elements()    {
       //return an enumeration of the kids
      return kids.elements();
   }
```

To obtain an enumeration of all the Kids in the collection, we simply return the enumeration of the Vector itself.

The Filtered Enumeration

Suppose, however, that we wanted to enumerate only those kids who belonged to a certain club. This necessitates a special Enumeration class that has access to the data in the KidData class. This is very simple, because the *elements()* method we just defined gives us that access. Then we only need to write an Enumeration that only returns kids belonging to a specified club:

```
public class kidClub
   implements Enumeration
{
   String clubMask;     //name of club
   Kid kid;             //next kid to return
   Enumeration ke;      //gets all kids
   KidData kdata;       //class containing kids
//----------------------------------------
   public kidClub(KidData kd, String club)    {
```

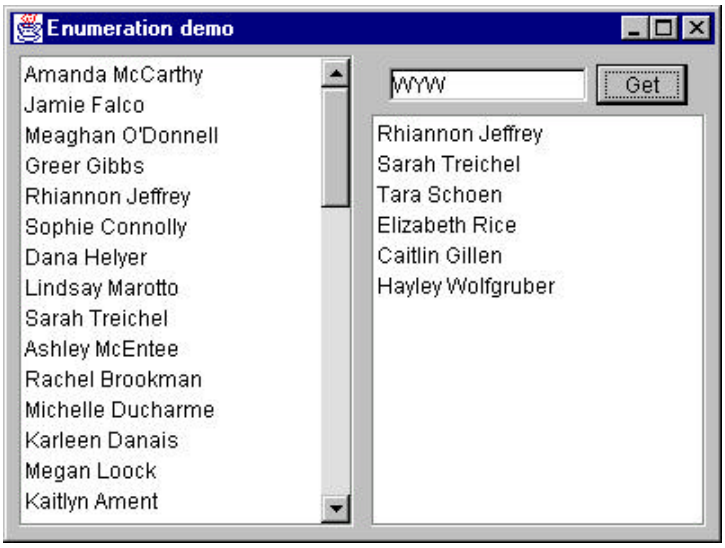
```
      clubMask = club;     //save the club
      kdata = kd;          //copy the class
      kid = null;          //default
      ke = kdata.elements();  //get Enumerator
   }
//---------------------------------------
   public boolean hasMoreElements()    {
   //return true if there are any more kids
   //belonging to the specified club
   boolean found = false;
   while(ke.hasMoreElements() && ! found)       {
      kid = (Kid)ke.nextElement();
      found = kid.getClub().equals(clubMask);
      }
   if(! found)
      kid = null;    //set to null if none left
   return found;
   }
//---------------------------------------
   public Object nextElement()    {
      if(kid != null)
         return kid;
      else
         //throw exception if access past end
         throw new NoSuchElementException();
   }
}
```

All of the work is done in the *hasMoreElements()* method, which scans through the collection for another kid belonging to the club specified in the constructor, and saves that kid in the *kid* variable, or sets it to *null*. Then, it returns either true or false. The *nextElement()* method either returns that next kid variable or throws an exception if there are no more kids. Note that under normal circumstances, this exception is never thrown, since the *hasMoreElements* boolean should have already told you not to ask for another element.

Finally, we need to add a method to KidData to return this new filtered Enumeration:

```
public Enumeration kidsInClub(String club)    {
      return new kidClub(this, club);
   }
```

This simple method passes the instance of KidClub to the Enumeration class kidClub along with the club initials. A simple program is shown below, that displays all of the kids on the left side and those belonging to a single club on the right.



Consequence of the Iterator Pattern

1. *Data modification.* The most significant question iterators may raise is the question of iterating through data while it is being changed. If your code is wide ranging and only occasionally moves to the next element, it is possible that an element might be added or deleted from the underlying collection while you are moving through it. It is also possible that another thread could change the collection. There are no simple answers to this problem. You can make an enumeration thread-safe by declaring the loop to be *synchronized*, but if you want to move through a loop using an Enumeration, and delete certain items, you must be careful of the consequences. Deleting or adding an element might mean that a particular element is skipped or accessed twice, depending on the storage mechanism you are using.

2. *Privileged access.* Enumeration classes may need to have some sort of privileged access to the underlying data structures of the original container class, so they can move through the data. If the data is stored in a Vector or Hashtable, this is pretty easy to accomplish, but if it is in some other collection structure contained in a class, you probably have to make that structure available through a *get* operation. Alternatively, you could make the Iterator a derived class of the containment class and access the data directly. The *friend* class solution available in C++ does not apply in Java. However, classes defined in the same module as the containing class do have access to the containing classes variables.

3. *External versus Internal Iterators.* The *Design Patterns* text describes two types of iterators: external and internal. Thus far, we have only described external iterators. Internal iterators are methods that move through the entire collection, performing some operation on each element directly, without any specific requests from the user. These are less common in Java, but you could imagine methods that normalized a collection of data values to lie between 0 and 1 or converted all of the strings to a particular case. In general, external iterators give you more control, because the calling program accesses each element directly and can decide whether to perform an operation on.

Composites and Iterators

Iterators, or in our case Enumerations, are also an excellent way to move through Composite structures. In the Composite of an employee hierarchy we developed in the previous chapter, each Employee contains a Vector whose *elements()* method allows you to continue to enumerate down that chain. If that Employee has no subordinates, the *hasMoreElements()* method correctly returns false.