Chapter 4: Using Interfaces and Abstract Classes

Using Interfaces and Abstract Classes

Okay! We're finally through with generalities. Let's put all this into practice. We're going to create an interface, an abstract class, and a concrete class to do simple lists. That way we can see how they work together. After we create them, we're going to update our team roster project to use them so you can test them out.

So where do we start? Let's start with the interface, where we'll define what we want a list to do. And what do we normally do with lists? We add things to them, we count how big our list is, we scratch things off the list, and we look at specific items in the list. We may even be making a list for someone else (we parents do that, don't we?) and want to make a copy of it for others to see.

We'll put those operations into our lists. We'll even use the same names for them that the Java lists do: add, size, remove, get, and toString. We will define all these in our interface so anyone who wants to create a class from it will have to implement those methods.

Just to keep things straight, we will name our interface and classes differently so we don't confuse them with the Java lists. Let's name our interface MyList, our abstract class MyAbstractList, and our concrete class MyConcreteList. Here's the MyList interface:

```
public interface MyList<E>
{
/**
 *  The MyList interface defines the list methods we'll use.
 */


  /**
   *  The add method will add an item to the end of the list.
   */
  public void add(E element);


  /**
   *  The remove method will remove the item at the specified
   *  location in the list, moving the following items one position to
   *  the left.
   *
   *  @param index  the location at which to remove the element
   */
public void remove(int index);


/**
 *  The get method will return the item at the specified
 *  location in the list.
 *
 *  @param index  the location of the element to return
 */
public E get(int index);


/**
  /**
   *  The size method will return the size of the list
   */
  public int size();


    /**
     *  The toString method will return a string containing the contents
     *  of the list.
     */
    public String toString();
}
```

I included the comments so anyone using it can see what each of the methods does. We did not implement any of the methods; we just defined how to call them. The semicolon after each method signature ends it without an implementation.

You might be thinking that the one strange-looking part of this is the use of <E> and E in the interface. What's that all about?

Putting <E> after the interface name tells Java that this interface is going to be a *generic* one. In Java, the word *generic* has a very specific meaning. It means we're going to define a structure to hold data, but we don't yet know what type of data we're going to hold there! E (the specific letter doesn't matter, as long as we use the same one throughout) is just a placeholder. The actual type will be filled in when we pick a concrete class.

Lists can hold all kinds of things. I mentioned three earlier: shopping items, tasks to do, and guests. In Java, we can have lists of integers, Strings, floating point numbers, Players—anything you can imagine. We don't want to have to build different classes for each one, so Java has these generic classes that let us put any type of data in them. We used one for our player lists in earlier lessons, the ArrayList. Remember how we declared it? We said, `ArrayList<Player>`. We were telling Java that we wanted to use a generic structure to hold Player objects.

In our interface definition, every time we use the placeholder E, Java will replace the E with the actual type once we tell it what it is. Our abstract and concrete classes will work the same way.

That's all there is to an interface! On to the abstract class!

The abstract class implements the methods that don't need to change, no matter how the list is implemented. The add(), remove(), and get() methods depend on the final list implementation, so they need to be defined as abstract, but not implemented, in the abstract class. The size() and toString() methods can be implemented in the abstract class, though, since we can write them independently, like this:

```
public abstract class MyAbstractList<E> implements MyList<E>

{

    protected int listSize = 0;


    public abstract void add(E element);


  public abstract void remove(int index);


  public abstract E get(int index);


  public int size()

  {

    return listSize;

  }


  public String toString()

  {

    String toReturn = "";


    for (int i = 0; i < listSize; i++)

    {

        toReturn += this.get(i).toString();

    }


        return toReturn;

    }

}
```

The abstract class includes an instance variable, *listSize*, that will contain the size of the list. The add() and remove() methods in the concrete class will update this field. But size() can go ahead and use it to return the size of the list in this class, and toString() can use it to control the loop it uses to create the output String.

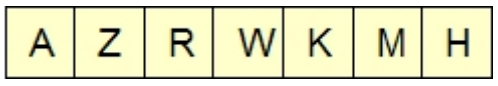
Notice that even though the get() method is abstract, toString can call it here as if it were already defined. That's because Java will ensure that it gets defined before it is ever actually called.

Now we're ready to define a concrete class we can use. It needs to define the three remaining abstract methods—add(), remove(), and get()—and we'll do that like this:

```
public class MyConcreteList<E>
    extends MyAbstractList<E>
    implements MyList<E>
{
    private Object[] myList;

  public MyConcreteList()
  {
    myList = new Object[100];
  }

  public MyConcreteList(int size)
  {
    myList = new Object[size];
  }

public void add(E element)
{
  myList[listSize] = element;
  listSize++;
}

public void remove(int index)
{
  listSize--;
  for (int i = index; i < listSize; i++)
  {
      myList[i] = myList[i+1];
  }
  }

    public E get(int index)
    {
        return (E) myList[index];
    }
}
```

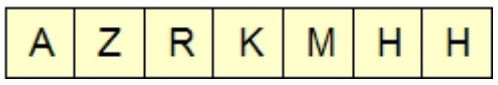
This concrete class uses an array to store the list. The default constructor creates a list with a maximum size of 100. The other constructor takes an argument that allows a user to define a different size list. The add() method puts a new element in the list and increments listSize. The remove() method deletes the list element at the given location by moving the next element into its place, then moving the element after that up one spot, and so on to the end of the list. When it's done, all the elements that followed the deleted one have been moved one location, writing over the deleted one.

For example, if we have this list of characters:



A list of characters

And we call remove with an index of 3, the array will then look like this:



The list after removing 'W'

We deleted W by copying the K, then the M replaced the K and the H replaced the M. Since we also reduced the list's size by one, to six, the final H will never be seen, so it won't matter if it's still there. The next character we add to the list will replace it.

Finally, the get() method lets a user get one of the elements out of the array if she or he needs to work with it.

Just one more thing about this implementation. We used an array in the class, but it was an array of Objects, not of type E. That's because Java doesn't allow arrays of generic types. Since every type we can create inherits from Object, we can use an array of Objects to hold anything. (Remember the "is a" relationship in our class hierarchy?) Since the array can hold anything, we have to do one more thing in the get() method that may have caught your eye already. We have to tell Java what type of item we are getting out of the array, since it could hold any type at all. That is what the (E) does in the return statement of the get() method. It tells Java that the item we are getting out of the array is of type E. Java calls this *casting*.

You've heard of typecasting in the entertainment industry, right? Well, this is how typecasting works in the programming world. It tells Java to take whatever type it may have and convert it into the type we say it is. If Java can't make that conversion, it will let us know.

Now, the next step is to actually use the list class for something. I am going to make that your assignment for this lesson, so we won't do it here. I'll explain what you need to do in the assignment.

I have one more item to show you, then we'll be through with today's lesson. We're going to look at a different implementation of the concrete list that works just the same from a user's standpoint, but it's different internally. And it's based on the same interface and abstract class. We're going to implement the list using an ArrayList instead of an array. Here is the second concrete class that inherits from MyList and from MyAbstractList:

```
import java.util.ArrayList;


public class MyConcreteArrayList<E>
    extends MyAbstractList<E>
    implements MyList<E>
{
    private ArrayList<E> myList;


  public MyConcreteArrayList()
  {
    myList = new ArrayList<E>(100);
  }


  public MyConcreteArrayList(int size)
  {
    myList = new ArrayList<E>(size);
  }


  public void add(E element)
  {
    myList.add(element);
    listSize++;
  }


    public void remove(int index)
    {
      listSize--;
      myList.remove(index);
    }


      public E get(int index)
      {
          return myList.get(index);
      }
}
```

Take a look at this class. It implements exactly the same methods as the first one and should work in someone else's program in just the same way. Other than the name, a programmer using these two classes could not tell the difference. But this implementation looks quite a bit different from the first one if you look at the internals.

You might be wondering why anyone would go to all that trouble to write two different classes that work the same way externally. The reason is internal performance. Remember how I mentioned a concrete Java class named LinkedList that behaves very much like ArrayList? For small applications, it would not make much difference which one you used. But for large applications, using the wrong one could mean a very slow process, and using the right one could mean a much faster, more successful process. These complex structures have important uses.