

## Defining a Type Specific Wrapper Class

The `java.util.Stack` class is generic in that it uses the class `Object` as the argument and result type from methods. However, in many situations, you will be using a `Stack` that contains only one type of object, e.g., `String` objects. Rather than being forced to do an explicit type case in every location in your program where you call `Stack.pop()` or `Stack.peek()`, you can write a simple **wrapper class** that wraps the generic `Stack` class with a more concrete class that has methods that are declared to take `Strings` as arguments and return them as results, and do the explicit type casting for you. For example:

```
import java.util.*;

public class StringStack {
    Stack stk;

    StringStack() { stk = new Stack(); }

    public boolean empty() { return stk.empty(); }
    public void push(String s) { stk.push(s); }
    public String pop() { return (String) stk.pop(); }
    public String peek() { return (String) stk.peek(); }
}
```

**Q1: Why can't you just inherit from `Stack`, instead of having to wrap it in this way?**

**Q2: Do you think it would be beneficial if Java had parametric polymorphism (i.e., templates) as in C++, so that container types could be defined to take a type parameter as an argument?**

It turns out that Dr. Phil Wadler at AT&T Bell Labs, Guy Steele, and many other people (including Greg Lavender) think the answer to Q2 is “yes”. They have developed an extension to Java called “Generic Java” or GJ, that allows you to program with templates in Java, especially for implementing collection or container objects. If you are interested in learning more, you can download the GJ system from the following URL:

<http://cm.bell-labs.com/cm/cs/who/wadler/pizza/gj/Distribution/index.html>

## A Possible Generic Java Vector Interface

```
public interface Stack [T] {  
    boolean empty();  
    T pop() throws EmptyStackException;  
    T peek() throws EmptyStackException;  
    void push (T x);  
}
```

In C++, we use “templates” to provide this kind of generic, or parameterized type. This kind of polymorphism is called **parametric polymorphism**. Parameterized types are not implemented in Java today, but there are enough people working towards having the language include this feature, that you can expect it to appear before too much longer. Templates were not in C++ initially either, they were added later.

Note that in Java, a Vector is a collection of objects of type **Object**. So, you can put any kind of object in a vector. With a template like the Stack[T] template above, you are restricted to having a stack of objects that are of type T, or implement an interface of T. So, you can always be assured that the things in the stack have the interface of an object of type T.

In the case of a Vector in Java (or the Stack class that inherits from Vector), you have no way to enforce that only objects of type T are put into the Vector. You can restrict yourself to only putting objects of type T into the Vector, but the compiler doesn't enforce it. You also have to remember to downcast from Object to type T each time you access an element of the Vector, or write a wrapper class like the one shown on the previous page.

We will talk more about parametric polymorphism and collections of similarly typed objects when we cover C++. I just wanted you to be aware of the fact that this type of polymorphism has some advantages, it is currently missing from Java, and it is very likely to appear in a future version of the language.

I encourage you to download the Generic Java compiler and experiment with templates in Java to see how they work.

## The Enumeration Interface

The `java.util.Enumeration` interface defines the interface specification for how enumerators are used to enumerate the elements of a linear collection of objects.

For example, the `java.util.Vector` class has associated with it a “friendly” `VectorEnumerator` class, that implements the `Enumeration` interface by providing implementations of the `hasMoreElements` and the `nextElement` that are specific to the implementation of the `Vector` class.

```
public interface Enumeration {
    /**
     * Tests if this enumeration contains more elements.
     *
     * @return <code>true</code> if this enumeration contains more elements;
     *         <code>false</code> otherwise.
     * @since   JDK1.0
     */
    boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration.
     *
     * @return   the next element of this enumeration.
     * @exception NoSuchElementException if no more elements exist.
     * @since    JDK1.0
     */
    Object nextElement();
}
```

## Enumerating Elements in a Vector

The `java.util.Vector` class has a method called `elements()` which will allow you to obtain a handle to an Enumeration object, that will allow you to enumerate (or iterate) over all the elements contained in the Vector. Notice that the return type of the `elements()` method is of type Enumeration, but the type of object that is returned is of type `VectorEnumerator`. This is because the user does not care about the implementation, just the interface.

```
// method defined in class java.util.Vector
public final synchronized Enumeration elements() { return new VectorEnumerator(this); }
```

The `VectorEnumerator` class provides a `Vector` specific implementation of the Enumeration interface. Note that the `VectorEnumerator.nextElement()` method uses what is called a **synchronized block**, which locks the `Vector` object prior to accessing an element of the vector. This is because in a concurrent program, you need *exclusive access* to the vector.

```
final class VectorEnumerator implements Enumeration {
    Vector vector;
    int count;

    VectorEnumerator(Vector v) { vector = v; count = 0; }

    public boolean hasMoreElements() { return count < vector.elementCount; }

    public Object nextElement() {
        synchronized (vector) {
            if (count < vector.elementCount) {
                return vector.elementData[count++];
            }
        }
        throw new NoSuchElementException("VectorEnumerator");
    }
}
```

## An Example of Enumerating through a Vector

A simple example of enumerating through a vector is the `Vector.toString()` method. `public final synchronized String toString()` {

```
    int max = size() - 1;
    StringBuffer buf = new StringBuffer();
    Enumeration e = elements();
    buf.append("[");

    for (int i = 0 ; i <= max ; i++) {
        String s = e.nextElement().toString();
        buf.append(s);
        if (i < max) {
            buf.append(", ");
        }
    }
    buf.append("]");
    return buf.toString();
}
```

Notice that this method is defined as a **synchronized method**, which means that the method first locks the `Vector` object on entry and unlocks the `Vector` object on exit. Locking an object means that other threads that may be executing cannot also access the object while it is locked by another thread (i.e., the method that called the `toString` method).

## The Cloneable Interface

Any subclass of the class `Object` that wishes to override the `Object.clone()` method must implement the interface `Cloneable`, which is an empty interface! This seems a bit odd, but is a “feature” of the way object cloning works in Java.

```
public interface Cloneable {  
  
}
```

As an example, the `Vector` class, which is cloneable, and it is also “serializable”, which means it can be used as an argument to a remote method using Java Remote Method Invocation (RMI):

```
public class Vector implements Cloneable, java.io.Serializable {  
    ...  
  
    public synchronized Object clone() {  
        try {  
            Vector v = (Vector)super.clone();  
            v.elementData = new Object[elementCount];  
            System.arraycopy(elementData, 0, v.elementData, 0, elementCount);  
            return v;  
        } catch (CloneNotSupportedException e) {  
            // this shouldn't happen, since we are Cloneable  
            throw new InternalError();  
        }  
    }  
  
    ...  
}
```

If a class does not implement the `Cloneable` interface, and tries to call the `Object.clone()` method, then a `CloneNotSupportedException` is raised.