Enumeration

an important Java interface

What is an Enumeration?

- An abstract way of getting elements of some sequence in a specific order.
- The sequence can be
  - elements of a list
  - elements of an array
  - elements of some other kind of structure, or
  - generated on the fly

Example

```
// One way to print the elements of an OpenList
import java.util.*;
public static void main(String arg[])
  {
  OpenList L = OpenList.list("a", "b", "c", "d");

  Enumeration e = new OpenListEnumeration(L);

  while( e.hasMoreElements() )
    {
    System.out.println(e.nextElement());
    }
  }
}
```

An Enumeration is an interface

- It is defined in java.util.Enumeration.
- Two methods are required in an implementation:
  - `Object nextElement()`
  - `boolean hasMoreElements()`

The Advantage to Using Enumerations

- In order to provide a sequence of elements to some method, we could either:
  - pass it a structure containing the sequence
  - pass it an enumeration
- In the latter case, the method does not need to know where the sequence came from.
- The sequence could also be infinite, which precludes the structural version.

The Typical Way to Get an Enumeration

- Many structures provide a method:
  - `Enumeration elements()`
- Calling this method returns an enumeration of the structure.

Example in Java Libraries

- java.util.StringTokenizer *implements* java.util.Enumeration
  - **`StringTokenizer`** `(String str, String delim)`
    `Constructs a string tokenizer for the specified string.`

StringTokenizer Usage

```
public static void main(String arg[])
  {
  String S = "The quick brown fox just plays with my very l

  String delimiter = " ";

  StringTokenizer T = new StringTokenizer(S, delimiter);

  while( T.hasMoreElements() )
    {
    System.out.println(T.nextElement());
    }
  }
```

```
The
quick
brown
fox
just
plays
with
my
very
lazy
dog.
```

Example in Java Libraries

- java.util.Vector:
  - `Enumeration` **`elements`** `()`
    `Returns an enumeration of the components of this vector.`
  - Vector itself does not *implement* Enumeration. Some inner class not normally seen does.

Class java.util.Vector

- A very useful class.
- Provides a "growing array":
  - **`void addElement(Object)`**
    adds a new element to the end of the array
  - **`int size()`**
    returns the current number of elements
  - **`Object elementAt(int)`**
    gets the element at a specific index
  - **`elements()`**
    returns an enumeration of all of the elements

Example Vector code

```
public static void main(String arg[])
  {
  int n = 10;

  Vector V = new Vector();
  for( int i = 0; i < n; i++ )
    {
    V.addElement(new Integer(i));
    }

  for( int i = V.size()-1; i > 0 ; i-- )
    {
    System.out.println(V.elementAt(i));
    }

  for( Enumeration E = V.elements(); E.hasMoreElements(); )
    {
    System.out.println(E.nextElement());
    }
```

Enumeration Caution

- The nextElement() method has two effects:
  - main effect: returning the "current" element
  - side effect: advancing to the "next" element
- If you want to use an element more than once, you need to **assign** it to a local variable:
  - `Object current = E.nextElement();`

Referential Transparency

- "Referential Transparency" means that a given expression in a given context has the *same value* each time it is used.
- This property is usual for functional languages.
- It does NOT hold for

  `nextElement()`
- Using **`nextElement()`** twice in a loop body, for example, will usually give *different* elements.

Enumeration On-the-Fly
A Functional Sequence Generator

```
class SequenceGenerator implements Enumeration
{
Object current;          // the current state of the generator
Function1 output;        // the output function: current ==> result
Function1 update;        // the update function: current ==> current
Function1 stopper;       // the stopper function: current ==> Boolean

SequenceGenerator(Object initial, Function1 _output, Function1 _update, Function1 _stopper)
  {
  current = initial;
  output  = _output;
  update  = _update;
  stopper = _stopper;
  }

public Object nextElement()
  {
  Object result = output.apply(current);
  current = update.apply(current);
  return result;
  }

public boolean hasMoreElements()
  {
  return !((Boolean)stopper.apply(current)).booleanValue();
  }
```

Using the Sequence Generator

```
public static void main(String arg[])
  {
  Enumeration E =
    new SequenceGenerator(new Integer(0),
                          new Outputter(),
                          new Updater(),
                          new Stopper(100));

  while( E.hasMoreElements() )
    {
    System.out.println(E.nextElement());
    }
  }
```

Example Functional Components

```
class Outputter implements Function1
{
Public Object apply(Object arg)
   {
   long value = ((Number)arg).longValue();
   return new Long(value *  value);
   }
}

class Updater implements Function1
{
public Object apply(Object arg)
   {
   long value = ((Number)arg).longValue();
   return new Long(value + 2);
   }
}
```

```
0
4
16
36
64
100
144
196
256
324
400
484
576
676
784
900
1024
1156
```

Iterators

- **Iterator** is an interface similar to Enumeration. It offers:
  - `Object next()`
  - `boolean hasNext()`
  - `void remove()`
    (`remove` is "optional":
    throw `UnimplementedOperationException`
    if not implemented)
  - (Doing it this way was not a good design choice IMHO.)
- Iterator is the more common **generic** term now (used in software patterns, C++, etc.)

LinkedList class

- `LinkedList` provides a method:

  `iterator()`

  that returns an `Iterator`.

- `LinkedList` is an example of a "closed list".

Previous Vector Code, transcribed to LinkedList

```
public static void main(String arg[])
  {
  int n = 10;

  LinkedList L = new LinkedList();
  for( int i = 0; i < n; i++ )
    {
    L.add(new Integer(i));
    }

  for( int i = L.size()-1; i > 0 ; i-- )
    {
    System.out.println(L.get(i));
    }

  for( Iterator I = L.iterator(); I.hasNext(); )
    {
    System.out.println(I.next());
    }
```

List Interface

- List is implemented by:
  - LinkedList
  - Vector
  - ArrayList
- So if an argument type of a method is List, ***any*** of these can be passed to that method.

List Interface: Some Key Methods

- `void` **add** `(int index, Object element)`
  Inserts the specified element at the specified position in this list (optional).
- `boolean` **add** `(Object o)`
  Appends the specified element to the end of this list (optional).
- `boolean` **addAll** `(Collection c)`
  Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional).
- `Object` **get** `(int index)`
  Returns the element at the specified position in this list.
- `boolean` **isEmpty** `()`
  Returns true if this list contains no elements.
- `Iterator` **iterator** `()`
  Returns an iterator over the elements in this list in proper sequence.
- `Object` **remove** `(int index)`
  Removes the element at the specified position in this list (optional).
- `int` **size** `()` Returns the number of elements in this list.
- `Object []` **toArray** `()` Returns an array containing all of the elements in this list in proper sequence.

List Interface Example

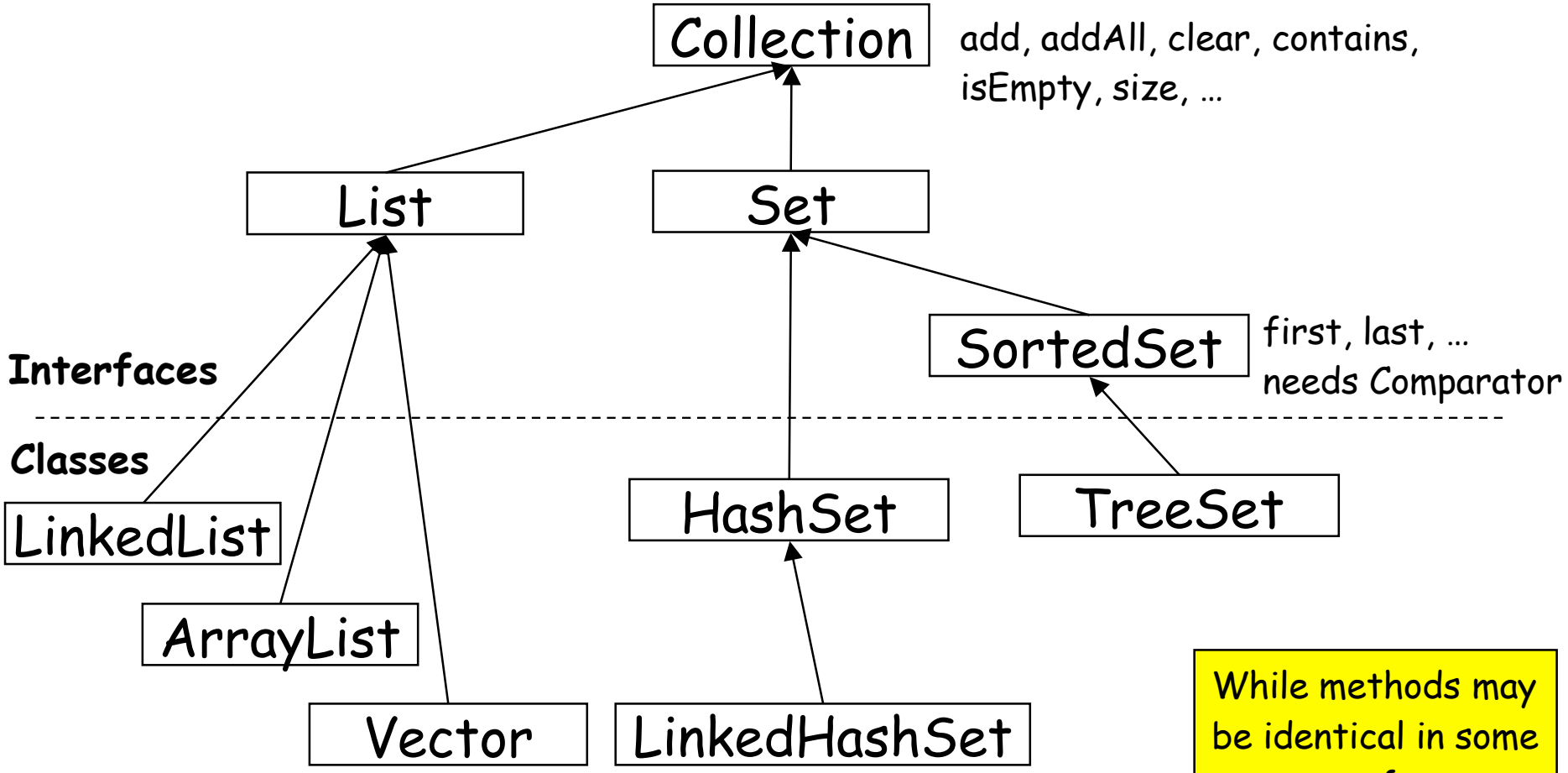
```
static void test(List L)
  {
  int n = 10;
  for( int i = 0; i < n; i++ )
    {
    L.add(new Integer(i));
    }

  for( int i = L.size()-1; i > 0 ; i-- )
    {
    System.out.println(L.get(i));
    }

  for( Iterator I = L.iterator(); I.hasNext(); )
    {
    System.out.println(I.next());
    }
  }
```

```
public static void main(String arg[])
  {
  Vector V     = new Vector();          test(V);
  LinkedList L = new LinkedList();      test(L);
  ArrayList  A = new ArrayList();       test(A);
  }
```

Java Standard Data Structures



While methods may be identical in some cases, performance can be different.