CSE 143

Lecture 26: Advanced List Implementation

(ADTs; interfaces; abstract classes; inner classes; generics; iterators)
Implementing generics

// a parameterized (generic) class
public class name<Type> {
    ...
}

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as E or T.
  - You can require multiple type parameters separated by commas.

- The rest of your class's code can refer to that type by name.

- Exercise: Convert our list classes to use generics.
Generics and arrays (15.4)

```java
class Foo<T> {
    private T myField; // ok

    public void method1(T param) {
        myField = new T(); // error
        T[] a = new T[10]; // error

        myField = param; // ok
        T[] a2 = (T[]) (new Object[10]); // ok
    }
}
```

– You cannot create objects or arrays of a parameterized type.
– You can create variables of that type, accept them as parameters, return them, or create arrays by casting from `Object[]`. 
Comparing generic objects

```java
public class ArrayList<E> {
    ...
    public int indexOf(E value) {
        for (int i = 0; i < size; i++) {
            // if (elementData[i] == value) {
                if (elementData[i].equals(value)) {
                    return i;
                }
            }
        }
        return -1;
    }
}
```

- When testing objects of type E for equality, must use equals
Generic interface (15.3, 16.5)

// Represents a list of values.
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}

public class ArrayList<E> implements List<E> { ...
public class LinkedList<E> implements List<E> { ...
Inner classes

// outer (enclosing) class
public class name {
    ...

    // inner (nested) class
    private class name {
        ...
    }
}

– Only this file can see the inner class or make objects of it.
– Each inner object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.

– Exercise: Convert the linked node into an inner class.
Common code

- Notice that some of the methods are implemented the same way in both the array and linked list classes.
  - `add(value)`
  - `contains`
  - `isEmpty`

- Should we change our interface to a class? Why / why not?
  - How can we capture this common behavior?
Abstract classes (9.6)

• **abstract class**: A hybrid between an interface and a class.
  – defines a superclass type that can contain method declarations (like an interface) and/or method bodies (like a class)
  – like interfaces, abstract classes that cannot be instantiated (cannot use `new` to create any objects of their type)

• What goes in an abstract class?
  – implementation of common state and behavior that will be inherited by subclasses (parent class role)
  – declare generic behaviors that subclasses must implement (interface role)
Abstract class syntax

// declaring an abstract class
public abstract class name {
    ...

    // declaring an abstract method
    // (any subclass must implement it)
    public abstract type name(parameters);
}

• A class can be abstract even if it has no abstract methods
• You can create variables (but not objects) of the abstract type

• Exercise: Introduce an abstract class into the list hierarchy.
**Linked list iterator**

- The following code is particularly slow on linked lists:

```java
List<Integer> list = new LinkedList<Integer>();
...
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

- Why?
- What can we do to improve the runtime?
Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of a collection, regardless of its implementation.
  - Remembers a position within a collection, and allows you to:
    - get the element at that position
    - advance to the next position
    - (possibly) remove or change the element at that position
  - A common way to examine *any* collection's elements.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-3</td>
<td>17</td>
</tr>
</tbody>
</table>

- current element: -3
- current index: 1
## Iterator methods

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>returns true if there are more elements to examine</td>
</tr>
<tr>
<td>next()</td>
<td>returns the next element from the collection (throws a</td>
</tr>
<tr>
<td></td>
<td>NoSuchElementException if there are none left to examine)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes from the collection the last value returned by next() (throws</td>
</tr>
<tr>
<td></td>
<td>IllegalStateException if you have not called next() yet)</td>
</tr>
</tbody>
</table>

- every provided collection has an iterator method

```java
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...
```

- Exercise: Write iterators for our linked list and array list.
  - You don't need to support the remove operation.
public class ArrayList<E> extends AbstractIntList<E> {  
   ...
   // not perfect; doesn't forbid multiple removes in a row
   private class ArrayIterator implements Iterator<E> {  
      private int index;  // current position in list
      public ArrayIterator() {  
         index = 0;
      }
      public boolean hasNext() {  
         return index < size();
      }
      public E next() {  
         index++;
         return get(index - 1);
      }
      public void remove() {  
         ArrayList.this.remove(index - 1);
         index--;
      }
   }
}
Linked list iterator

```java
public class LinkedList<E> extends AbstractIntList<E> {

    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current; // current position in list

        public LinkedIterator() {
            current = front;
        }

        public boolean hasNext() {
            return current != null;
        }

        public E next() {
            E result = current.data;
            current = current.next;
            return result;
        }

        public void remove() { // not implemented for now
            throw new UnsupportedOperationException();
        }
    }

    ...
}
```
for-each loop and Iterable

• Java's collections can be iterated using a "for-each" loop:

```java
List<String> list = new LinkedList<String>();
...
for (String s : list) {
    System.out.println(s);
}
```

– Our collections do not work in this way.

• To fix this, your list must implement the `Iterable` interface.

```java
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```
// Represents a list of values.
public interface List<E> extends Iterable<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public Iterator<E> iterator();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}