Inner Classes and Iterators

Readings: RS Chapter 15 & 16

Inner Classes

- Inner Class: A class defined inside of another class.
  - can be created as static or non-static
  - May also be called a nested class

Usefulness:
- Inner classes (and their public fields) are hidden from other classes (encapsulated)
- Inner objects can access/modify the fields of the outer object (if the inner class is not static)
Inner class syntax

```java
// outer (enclosing) class
public class Enclosing {
    ...
    // inner (nested) class
    private class Nested {
        ...
    }
}
```

- Only this outer class/object 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.
  - If necessary, can refer to outer object as `OuterClassName.this`

Adding Inner Classes

- Inner classes can be declared in a method or within an entire enclosing class
- You’re telling code inspectors that the class is only of interest to the enclosing body.
  - Composition relationship where the enclosing class controls the lifecycle of the contained

```java
public class LinkedList<E> extends AbstractSequentialList<E> {
    private class ListNode<T> {
    }
    private class ListIterator implements Iterator<E> {
    }
}  
```
Public Inner Classes

- Public inner classes are visible outside of the outer class
  - Access via: `<outer class>..<inner class>`

- If your inner class is public, then you should make your inner class' members private to preserve encapsulation

What Happens...

- When a method of your inner class has the same name as a method in your outer class, and you want to call the outer class' method from your inner class?
  - Java won't let you compile because it can't tell what method you want to call

- You need to refer to the method of the outer class explicitly
  `OuterClass.this.outerMethod();`
Static Inner Classes

• You can also make inner classes static
  – If the inner class DOES NOT access the outer object
  – Example: ListNode
• By making the inner class static, we minimize extra storage required for the connections between the inner and outer classes
• Static inner classes cannot use instance variables (fields) of the outer class

Revisiting Our Inner Classes

```java
public class LinkedList<E> extends AbstractSequentialList<E> {
    private static class ListNode<T> { }

    private class ListIterator<T> implements Iterator<T> { }
}
```
What Happens…

• When we compile code containing inner classes?
  – Class files are made for each inner class, but the naming
    convention is different

  LinkedList$ListNode.class
  LinkedList$ListIterator.class

Anonymous Inner Classes

• Anonymous objects don’t have names
  – Example:
    list.add(new Course(...));

• We can also have anonymous classes
  – Typically create anonymous classes when using
    Listeners in GUIs
Example Anonymous Inner Class

```java
button = new JButton("Click Me!");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Handle actions for button
    }
});
```

Generics and inner classes

```java
public class Foo<T> {
    private class Inner<T> {}  // incorrect

    private class Inner {}     // correct
}
```

- If an outer class declares a type parameter, inner classes can also use that type parameter.
- Inner class should NOT redeclare the type parameter. (If you do, it will create a second type parameter with the same name.)
Why Use ADTs?

- Why would we want more than one kind of list, queue, etc.?
- Answer: Each implementation is more efficient at certain tasks.
  - ArrayList is faster for adding/removing at the end;
    LinkedList is faster for adding/removing at the front/middle.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.
  - And you can change out the underlying type later if you find something more efficient!

Complexity Classes

- Complexity Class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double N, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>O(1)</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>O(log₂ N)</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>O(N)</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>O(N log₂ N)</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>O(N²)</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>O(N³)</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>O(2ⁿ)</td>
<td>multiplies drastically</td>
<td>5 * 10⁶¹ years</td>
</tr>
</tbody>
</table>
Linked list iterator

- The following *client* 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 is this client code slow?
  - What is the runtime efficiency?
  - What can we do to improve the runtime?

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();
  }
  ```
Iterators

- **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
  - Benefit: A common way to examine any collection’s elements.

```
index | 0 | 1 | 2
value | 42 | -3 | 17
```

```
front → 42 ← next

front → -3 ← next

front → 17 ← next
```

Current element: -3
Current index: 1

```
Iterator methods

<table>
<thead>
<tr>
<th>hasNext()</th>
<th>returns true if there are more elements to examine</th>
</tr>
</thead>
<tbody>
<tr>
<td>next()</td>
<td>returns the next element from the collection (throws a 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 IllegalStateException if you have not called next() yet)</td>
</tr>
</tbody>
</table>
```

• every provided collection has an iterator method

```java
LinkedList<String> list =
    new LinkedList<String>();
...
Iterator<String> itr = list.iterator();
...```
Array List Iterator

```java
public class ArrayList<E> extends AbstractList<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() {
            throw new UnsupportedOperationException();
        }
    }
}
```
AbstractSequentialList

- Part of Java API
- Used for sequential access – linked list
- All standard List methods are implemented for you
  - Use an Iterator as returned from the listIterator() method
  - Provide implementation for listIterator() and size()
  - Write a class that implements the Iterator interface
  - Leading and trailing node with null data; iterator between nodes

![Diagram of AbstractSequentialList](image)