Inner Classes and Iterators

Readings: Chapters 15 and 16

Our list classes

• We have implemented the following two list collection classes:
  - ArrayList

<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>

  - LinkedList

  ![Linked List Diagram]

  - Problems:
    • Some methods are implemented the same way (redundancy).
    • Linked list carries around a clunky extra node class.
    • It is inefficient to get or remove each element of a linked list.
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?

An abstract list class

```java
// Superclass with common code for a list of integers.
public abstract class AbstractList implements List {
    public void add(int value) {
        add(size(), value);
    }

    public boolean contains(int value) {
        return indexOf(value) >= 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }
}

public class ArrayList extends AbstractList {...

public class LinkedList extends AbstractList {...
```
Our list classes

• We have implemented the following two list collection classes:
  - ArrayList
    - index 0 1 2
    - value 42 -3 17
  - LinkedList
    - Problems:
      • Some of their methods are implemented the same way (redundancy).
      • Linked list carries around a clunky extra node class.
      • It is inefficient to get or remove each element of a linked list.

Inner classes

• inner class: A class defined inside of another class.
  - can be created as static or non-static

• 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

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

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

- 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.

public class BeerList implements List {

    private class ListNode {

    }

    private class BeerListIterator implements Iterator {

    }
}
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

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

public class BeerList implements List {

    private static class ListNode {
    }
    private class BeerListIterator implements Iterator {
    }
}

What Happens...

• When we compile code containing inner classes?
  – Class files are made for each inner class, but the naming convention is different
  – BeerList$ListNode.class
  – BeerList$BeerListIterator.class
Anonymous Inner Classes

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

Our list classes

- We have implemented the following two list collection classes:
  - ArrayIntList
    | index | 0 | 1 | 2 |
    | value | 42 | -3 | 17 |
  - LinkedIntList
    | data  | next |
    | 42    | -3   |
    | -3    | 17   |

- Problems:
  - Some of their methods are implemented the same way (redundancy).
  - Linked list carries around a clunky extra node class.
  - It is inefficient to get or remove each element of a linked list.
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.

Runtime Efficiency (13.2)

• **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

• Assume the following:
  - Any single Java statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.
Efficiency examples

\[
\begin{align*}
\text{statement1;} & \quad \text{statement2;} & \quad \text{statement3;} & \quad 3 \\
\text{for (int } i = 1; \ i <= N; \ i++) { & \quad \text{statement4;} & \quad N \\
\} & \quad \text{for (int } i = 1; \ i <= N; \ i++) { & \quad \text{statement5;} & \quad \text{statement6;} & \quad \text{statement7;} & \quad 3N \\
\} & \quad 4N + 3
\end{align*}
\]

Efficiency examples 2

\[
\begin{align*}
\text{for (int } i = 1; \ i <= N; \ i++) { & \quad \text{for (int } j = 1; \ j <= N; \ j++) { & \quad \text{N}\textsuperscript{2} & \quad \text{statement1;} & \quad N\textsuperscript{2} + 4N \\
\} & \quad \text{for (int } i = 1; \ i <= N; \ i++) { & \quad \text{statement2;} & \quad \text{statement3;} & \quad \text{statement4;} & \quad \text{statement5;} & \quad 4N \\
\} & \quad N\textsuperscript{2} + 4N
\end{align*}
\]

• How many statements will execute if \( N = 10 \)? If \( N = 1000 \)?
Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs \(0.4N^3 + 25N^2 + 8N + 17\) statements.
  - Consider the runtime when N is extremely large.
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (\(N^3\)) dominates the overall runtime.
  - We say that this algorithm runs "on the order of" \(N^3\).
  - or \(O(N^3)\) for short ("Big-Oh of N cubed")

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(\log_2 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 \log_2 N))</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>(O(N^2))</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>(O(N^3))</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>exponential</td>
<td>(O(2^N))</td>
<td>multiplies drastically</td>
<td>(5 \times 10^{61}) years</td>
</tr>
</tbody>
</table>
Collection efficiency

- Efficiency of various operations on different collections:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(value)</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>indexOf</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1) or O(N) (Depends on how size() is implemented)</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?
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
  - Benefit: A common way to examine any collection's elements.

<table>
<thead>
<tr>
<th>Index</th>
<th>Data</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

-3

iterator

-3

current element: -3

current index: 1

Iterator methods

<table>
<thead>
<tr>
<th>Method</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 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--;
        }
    }
}
```

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();
  }
  ```
Final List interface (15.3, 16.5)

// Represents a list of values.
public interface List<E> implements 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();
}