Chapter 20 Lists, Stacks, Queues, and Priority Queues

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy (§20.2).
- To use the common methods defined in the Collection interface for operating collections (§20.2).
- To use the Iterator interface to traverse the elements in a collection (§20.3).
- To use a for-each loop to traverse the elements in a collection (§20.3).
- To explore how and when to use ArrayList or LinkedList to store elements (§20.4).
- To compare elements using the Comparable interface and the Comparator interface (§20.5).
- To use the static utility methods in the Collections class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections (§20.6).
- To develop a multiple bouncing balls application using ArrayList (§20.7).
- To distinguish between Vector and ArrayList and to use the Stack class for creating stacks (§20.8).
- To explore the relationships among Collection, Queue, LinkedList, and PriorityQueue and to create priority queues using the PriorityQueue class (§20.9).
- To use stacks to write a program to evaluate expressions (§20.10).

What is Data Structure?

A data structure is a collection of data organized in some fashion. The structure not only stores data, but also supports operations for accessing and manipulating the data.
Java Collections Framework

A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *lists*, *sets*, and *maps*.

Java Collections Framework

- Lists – Stores elements in sequential order
  - Ordered Collection
- Sets – lists allow duplicates, sets do not
  - Unordered Collection
- Maps – data structure based on {key, value} pair
  - Holds two objects per entry
  - May contain duplicate values
  - Keys are always unique

Java Collections Framework

Set and List are subinterfaces of Collection.
The Collection Interface

The Collection interface defines a set of methods that can be used to manipulate collections. Collections are used to store and manipulate a group of related objects. The methods defined in the Collection interface include:

- add(e: E): boolean - Adds an element to the collection.
- addAll(c: Collection<? extends E>): boolean - Adds all elements of a collection to the current collection.
- clear(): void - Removes all elements from the collection.
- contains(o: Object): boolean - Returns true if the collection contains the specified element.
- containsAll(c: Collection<?>): boolean - Returns true if this collection contains all the elements of the specified collection.
- isEmpty(): boolean - Returns true if the collection contains no elements.
- remove(o: Object): boolean - Removes the specified element from the collection.
- removeAll(c: Collection<?>): boolean - Removes all the elements of the specified collection from this collection.
- retainAll(c: Collection<?>): boolean - Retains only the elements that are also contained in the specified collection.
- size(): int - Returns the number of elements in the collection.
- toArray(): Object[] - Returns an array containing all elements of the collection.
- stream(): Stream - Returns a stream of elements contained in this collection.
- parallelStream(): Stream - Returns a parallel stream of elements contained in this collection.

The Iterable Interface

The Iterable interface is a marker interface that provides a way to iterate over the elements of a collection. It has a single method:

- iterator(): Iterator<E> - Returns an iterator for the elements in this collection.

The List Interface

A list is a data structure that stores elements in a sequential order, and allows the user to specify where the element is stored. The user can access the elements by index.

The List Interface, cont.

The List Interface adds methods that allow for the manipulation of elements at specific indices:

- add(index: int, element: Object): boolean - Inserts the specified element at the specified position in the list.
- addFirst(element: Object): void - Adds the specified element to the front of the list.
- addLast(element: Object): void - Adds the specified element to the end of the list.
- contains(element: Object): boolean - Returns true if the list contains the specified element.
- equals(other: Object): boolean - Returns true if the specified object is equal to this list.
- get(index: int): Object - Returns the element at the specified index.
- indexOf(element: Object): int - Returns the index of the first occurrence of the specified element in the list, or -1 if the element is not contained by the list.
- lastIndexOf(element: Object): int - Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not contained by the list.
- remove(index: int): Object - Removes and returns the element at the specified index.
- removeFirst(): Object - Removes and returns the first element of the list.
- removeLast(): Object - Removes and returns the last element of the list.
- set(index: int, element: Object): Object - Replaces the element at the specified index with the specified value.
- size(): int - Returns the number of elements in the list.
The List Iterator

```java
interface java.util.List<E>

interface java.util.ListIterator<E>

+add(E e): void
+hasPrevious(): boolean
+nextIndex(): int
+previous(E): int
+set(E e): void
```

Add the specified object to the list. Returns true if this list iterator has more elements when traversing backward. Returns the index of the next element. Returns the previous element in this list iterator. Returns the index of the previous element. Replaces the last element returned by the previous or next method with the specified element.

Array vs ArrayList vs LinkedList

- ArrayList class and the LinkedList class
  - Concrete implementations of the List interface.
  - Usage depends on your specific needs.
- Efficiency
  - ArrayList – Random access through an index
  - LinkedList - Insertion or deletion of elements at any location
  - Array - If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

**java.util.ArrayList**

```java
interface java.util.List<E>

interface java.util.ListIterator<E>

create

+add(E e): void
+hasPrevious(): boolean
+nextIndex(): int
+previous(E): int
+set(E e): void
```

Creates an empty list with the default initial capacity. Creates an array list from an existing collection. Creates an empty list with the specified initial capacity. Replaces the last element returned by the previous or next method with the specified element.

```java
interface java.util.Collection<E>

java.util.ArrayList<E>

java.util.LinkedList<E>
```

```java
interface java.util.List<E>

interface java.util.ListIterator<E>

java.util.ArrayList<E>

java.util.LinkedList<E>
```

```java
interface java.util.List<E>

interface java.util.ListIterator<E>

java.util.ArrayList<E>

java.util.LinkedList<E>
```

```java
interface java.util.List<E>

interface java.util.ListIterator<E>

java.util.ArrayList<E>

java.util.LinkedList<E>
```
Example: Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.
Comparable vs Comparator

- **Comparable**
  - Implemented with compareTo
  - Defines the natural order for the object

- **Comparator**
  - Implemented with compare()
  - Defines a different order for some purpose

The Comparator Interface

Sometimes you want to compare the elements of different types. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, define a class that implements the java.util.Comparator interface. The Comparator interface has the compare method for comparing two objects.

```
public int compare(Object element1, Object element2)
```

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.
The Collections Class

The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

The Collections Class UML Diagram

The Vector and Stack Classes

The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.
The Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, “The Collections Class.”

The Vector Class, cont.

The Stack Class

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.

```java
java.util.Stack<E>

+Stack() 
+empty(): boolean 
+peek(): E 
+pop(): E 
+push(o: E) : E 
+search(o: Object) : int

+Stack(java.util.Collection<? extends E> c) 
+clear(): void 
+clear(): void 
+isEmpty(): boolean 
+isEmpty(): int 
+size(): int
```

Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.
Queues and Priority Queues

A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

The Queue Interface

```
public interface Queue {
    boolean offer(Object element);
    Object poll();
    Object remove();
    Object element();
}
```

Using LinkedList for Queue

```
public class LinkedListQueue extends LinkedList implements Queue {
    public boolean offer(Object element) {
        // implementation
    }
    public Object poll() {
        // implementation
    }
    public Object remove() {
        // implementation
    }
    public Object element() {
        // implementation
    }
}
```
The PriorityQueue Class

```java
interface PriorityQueue<T>

PriorityQueue() creates a default priority queue with initial capacity 11.
PriorityQueue(Collection<? extends E>) creates a priority queue with the specified collection.
PriorityQueue(int initialCapacity, comparator: Comparator<? super E>) creates a priority queue with the specified initial capacity and the comparator.
```

Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.

Some examples

- 2 + 3
  - When we see + we haven’t seen operand 3 yet. Use an operandStack to push operands, and an operatorStack to push operators:
  - push (2, operandStack)
  - push (+, operatorStack)
  - push (3, operandStack)
  - End of expression: apply operator to operands
  - Why wait until we see the end or rest of expression?

- 2 + 3 * 4
\[ 2 + 3 - 4 \text{ is } (2 + 3) - 4, \text{ and NOT } 2 + (3 - 4) \]

- push (2, operandStack)
- push (+, operatorStack)
- push (3, operandStack)

Seeing -: apply operator on stack to operands
- push (-, operatorStack)
- push (4, operandStack)

End: apply operator(s) to operands

\[ 2 + 3 \times 4 - 5 \]

- push (2, operandStack)
- push (+, operatorStack)
- push (3, operandStack)
- push (*, operatorStack)
- push (4, operandStack)

*: has precedence over +, so
- push (+, operatorStack)
- push (5, operandStack)

End: apply operator(s) to operands

\[ 2 \times (3 + 4) / 5 \]

- push (2, operandStack)
- push (*, operatorStack)
- push (‘(’, operatorStack)
- push (3, operandStack)
- push (+, operatorStack)
- push (4, operandStack)

(: make a substack at top of operatorStack:
- push (‘(’, operatorStack)
- push (3, operandStack)
- push (+, operatorStack)
- push (4, operandStack)

): apply operators to operands until ‘(’, pop (‘(’
- push (‘)’, operatorStack)
- push (5, operandStack)

End: apply operator(s) to operands

<table>
<thead>
<tr>
<th>2 + 3</th>
<th>2 + 3</th>
<th>2 + 3...</th>
<th>2 + 3...</th>
<th>2 + 3...</th>
<th>2 + 3...</th>
<th>2 + 3...</th>
<th>2 + 3...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Algorithm

Phase 1: Scanning the expression
The program scans the expression from left to right to extract operands, operators, and the parentheses.

1.1. If the extracted item is an operand, push it to operandStack.
1.2. If the extracted item is a + or - operator, process all the operators at the top of operatorStack and push the extracted operator to operatorStack.
1.3. If the extracted item is a * or / operator, process the * or / operators at the top of operatorStack and push the extracted operator to operatorStack.
1.4. If the extracted item is a ( symbol, push it to operatorStack.
1.5. If the extracted item is a ) symbol, repeatedly process the operators from the top of operatorStack until seeing the ( symbol on the stack.

Phase 2: Clearing the stack
Repeatedly process the operators from the top of operatorStack until operatorStack is empty.

Example

<table>
<thead>
<tr>
<th>Expression</th>
<th>Scan</th>
<th>Action</th>
<th>operandStack</th>
<th>operatorStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>(</td>
<td>Phase 1.4</td>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>1</td>
<td>Phase 1.1</td>
<td>[1, 1]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>+</td>
<td>Phase 1.2</td>
<td>[1, 1, +]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>2</td>
<td>Phase 1.3</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>)</td>
<td>Phase 1.5</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>*</td>
<td>Phase 1.3</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>4</td>
<td>Phase 1.1</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>-</td>
<td>Phase 1.2</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>3</td>
<td>Phase 1.3</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>)</td>
<td>Phase 1.5</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
<tr>
<td>(1 + 2/4 - 3)</td>
<td>)</td>
<td>Phase 2</td>
<td>[1, 1, +, 2]</td>
<td></td>
</tr>
</tbody>
</table>