PEARSON NEW INTERNATIONAL EDITION

Data Structures and Algorithms in Java

Peter Drake

First Edition

ALWAYS LEARNING™

PEARSON®

Pearson New International Edition

Data Structures and Algorithms in Java

Peter Drake

First Edition

PEARSON®

The class ArrayList is similar to the one we wrote in this chapter. It is also similar to the somewhat out-of-date class Vector. As Java has evolved over time, new classes and interfaces have been added to the built-in library. To provide ***backward compatibility***—that is, to allow old Java programs to run under new versions of Java—classes are rarely removed from the library. Thus, even though new programs should always use ArrayList, the Vector class is still present.

The framework does not include a Stack interface, but it does include a Stack class, which is similar to our ArrayStack class. The Stack class extends Vector. If an attempt is made to `peek()` at or `pop()` an empty Stack, a java.util.EmptyStackException is thrown.

Many object-oriented programmers feel that it is a bad idea for Stack to extend Vector. The reason is that Stack inherits some public methods like `get()`. These methods allow us to do unstackly things with a Stack, such as looking at the bottom element. Since the ability to do this depends on a particular implementation, it violates the encapsulation of the abstract data type. These issues aside, the java.util.Stack class is the one that comes with Java. We must either swallow our pride and use it or write our own.

Curiously, there is a Queue interface, but no array-based implementation. There is a non-array-based implementation, which we will see in Chapter 6.

Abstract Classes

List is implemented by the class AbstractList, which in turn is extended by ArrayList and Vector. AbstractList is an ***abstract class***, meaning that we can't create an instance of it. An abstract class exists only to be extended by other classes. Conceptually, an abstract class is something like "vehicle" or "animal." It wouldn't make sense to have an instance of a category like this, but it would make sense to have an instance of "bicycle" or "lion."

An abstract class is similar to an interface. While it is a class—and therefore is extended rather than implemented—an abstract class can contain ***abstract methods*** which must be provided by any (nonabstract) subclass. All of the methods in an interface are implicitly abstract. (We could explicitly declare them to be abstract, but there would be no point.)

The key difference between an abstract class and an interface is that an abstract class can specify *both responsibilities and (partial) implementation*, while an interface can specify *only responsibilities*. In more technical terms, an abstract class can contain fields, abstract methods, and non-abstract methods, while an interface can contain only (implicitly) abstract methods.

For a more concrete example of an abstract class, we could have defined Stack as an abstract class (Figure 5–50). Any class extending this class would have to provide `isEmpty()`, `pop()`, and `push()`, but would inherit `peek()`.

Exercises

5.28 Through experimentation, determine what happens if you try to extend an abstract class without providing one of the abstract methods specified.

5.29 Speculate on whether a class can extend more than one abstract class.

```
 1 /** A last-in, first-out stack of Objects. */
 2 public abstract class Stack {
 3
 4   /** Return true if the Stack is empty. */
 5   public abstract boolean isEmpty();
 6
 7   /**
 8    * Return the top Object on the Stack, but do not modify the
 9  Stack.
10    * @throws EmptyStructureException if the Stack is empty.
11    */
12   public Object peek() {
13     Object result = pop();
14     push(result);
15     return result;
16   }
17
18   /**
19    * Remove and return the top Object on the Stack.
20    * @throws EmptyStructureException if the Stack is empty.
21    */
22   public abstract Object pop();
23
24   /** Add an Object to the top of the Stack. */
25   public abstract void push(Object target);
26
27 }
```

Figure 5–50: Stack as an abstract class. Most of the methods are abstract (and therefore must be provided by subclasses), but an implementation of `peek()` is provided.

5.30 The method `peek()` in Figure 5–50 could begin with the code:

```
if (isEmpty()) {
  throw new EmptyStructureException();
}
```

Explain why this is not necessary.

Summary

The size of an array cannot be changed, but the array-based structures discussed in this chapter are able to grow and shrink. They do this by maintaining a separate field indicating how many of the array positions are currently in use. This field also provides the index of the next available

position. If all of the positions are in use, an array-based structure can be stretched by copying all of its elements into a new, larger array.

These techniques for stretching and shrinking are used in the ArrayStack class. The ArrayQueue class must do a little more work because, as elements are added to the back and removed from the front, the in-use portion of the array marches to the right. When it hits the right end of the array, it wraps around to the beginning. This is accomplished using the % operator.

The List interface describes a much more general data structure. It is implemented by the ArrayList class. In addition to providing basic methods for accessing particular elements and so on, a List can return an Iterator. An Iterator allows us to traverse the List, visiting each element.

Java's collections framework, in the java.util package, includes a List interface, an ArrayList class, a Stack class, and a Queue interface. We will see more interfaces and classes from this framework in later chapters.

Vocabulary

abstract class. Class that cannot be instantiated but may contain abstract methods. Unlike an interface, an abstract class can also contain fields and nonabstract methods.

abstract method. Method signature with no body, to be provided by another class. Found in interfaces and abstract classes.

backward compatibility. Of a compiler or other system, ability to work with old software.

iterator. Object allowing us to traverse a data structure. In Java, an instance of the java.util.Iterator class.

Java collections framework. Set of classes in the java.util package descending from Collection. These are standard implementations of many common data structures.

traverse. Go through a data structure, visiting each element.

visit. Do something with or to an element of a data structure.

Problems

5.31 In the Card class, replace the constant `ACE` with two constants `ACE_LOW=1` and `ACE_HIGH=14`. Make Deck abstract and provide two subclasses AceLowDeck (which has low aces) and AceHighDeck (which has high aces).

5.32 Rewrite the ArrayQueue class (Figure 5–23) so that, instead of a field `size` indicating the number of Objects in the queue, there is a field `back` indicating the next available position. How will you know when the queue is full?

5.33 Rewrite the ArrayQueue class (Figure 5–23) so that the `remove()` method shifts all of the data to the left, keeping the front of the queue at position 0. This allows the field

`front` to be removed. Which methods become simpler as a result of this change? Which become more complicated?

5.34 Define a class ReverseArrayIterator, which implements java.util.Iterator but visits an ArrayList's elements in *reverse* order. Add a method `reverseIterator()` to the ArrayList class. The method returns an instance of ReverseArrayIterator.

5.35 Write a Deck class whose only field is of type ArrayList<Card>. Use the static `shuffle()` method in the API for the Collections class (which should not be confused with the Collection interface).

Projects

5.36 Write an abstract class ArrayBasedStructure which is extended by ArrayStack, ArrayQueue, and ArrayList. How much of the implementation can be moved up into this abstract superclass? Discuss whether this makes the overall code more or less clear.

5.37 Write an array-based implementation of the Deque interface from Problem 4.18.

This page intentionally left blank

6

Linked Structures

This chapter introduces linked structures. The structures in this chapter are made from chains of nodes, connected by references. These nodes are introduced in Section 6.1. The Stack and Queue interfaces are implemented in Section 6.2 and the List interface in Section 6.3. Finally, in Section 6.4, we examine the linked structures in the Java collections framework.

6.1 List Nodes

The structures in this chapter are composed of ***list nodes***. A list node contains only one element, but it also contains (a reference to) another list node. A list node is represented by an instance of a class called, not surprisingly, ListNode (Figure 6–1).

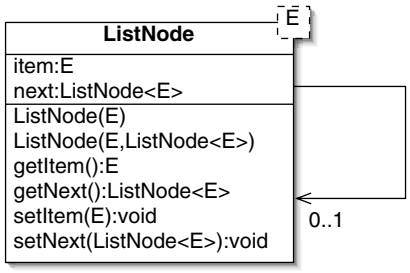

Figure 6–1: A ListNode contains another ListNode unless `next` is `null`.