Chapter 15

Collection Considerations

15.1 ADTs, Collection Classes, and Data Structures

A collection is an object to store, retrieve, and manipulate a collection of objects in some meaningful way. Collection classes have the following characteristics:

- The main responsibility of a collection class is to store a collection of values.
- Elements may be added and removed from the collection.
- A collection class allows clients to access the individual elements.
- A collection class may have search-and-sort operations for locating a particular element.
- Some collections allow duplicate elements while other collections do not.
- Some collections are naturally ordered while other collections are not.

Some collections are designed to store large amounts of information where any element may need to be found quickly—to find a phone listing, for example. Other collections are designed to have elements that are added, removed, and changed frequently—an inventory system, for example. Some collections store elements in a first in first out basis—such as a queue of incoming packets on an Internet router. Other collections are used to help build computer systems—a stack that manages method calls, for example

Collection classes support many operations. The following is a short list of operations that are common to many collection classes:

- **add** Place an object into a collection (insertion point varies).
- **find** Get a reference to an object in a collection so you can send messages to it.
- **remove** Take the object out of the collection (extraction point varies).

The Java class is a convenient way to encapsulate algorithms and store data in one module. In addition to writing the class and method headings, decisions have to be made about what data structures to use.

Data Structures

A data structure is a way of storing data on a computer so it can be used efficiently. There are several structures available to programmers for storing data. Some are more appropriate than others, depending on how you need to manage your information. Although not a complete list, here are some of the storage structures you have or will see in this book:

1. arrays
2. linked structure
3. tables

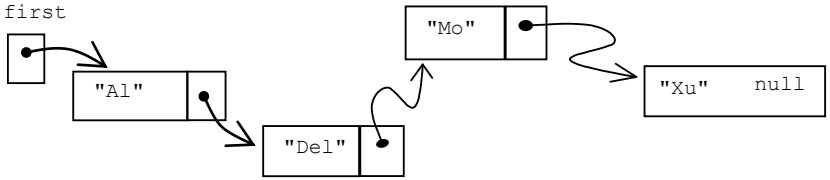
Data can be stored in contiguous memory with arrays or in non-contiguous memory with linked structures. Arrays allow you to reserve memory where each element can be physically located next to its predecessor and successor. Any element can be directly changed and accessed through an index.

```
String[] data = new String[5];
data[0] = "Al";
data[1] = "Di";
data[2] = "Mo";
data[3] = "Xu";
```

`data` (where data.length == 5):

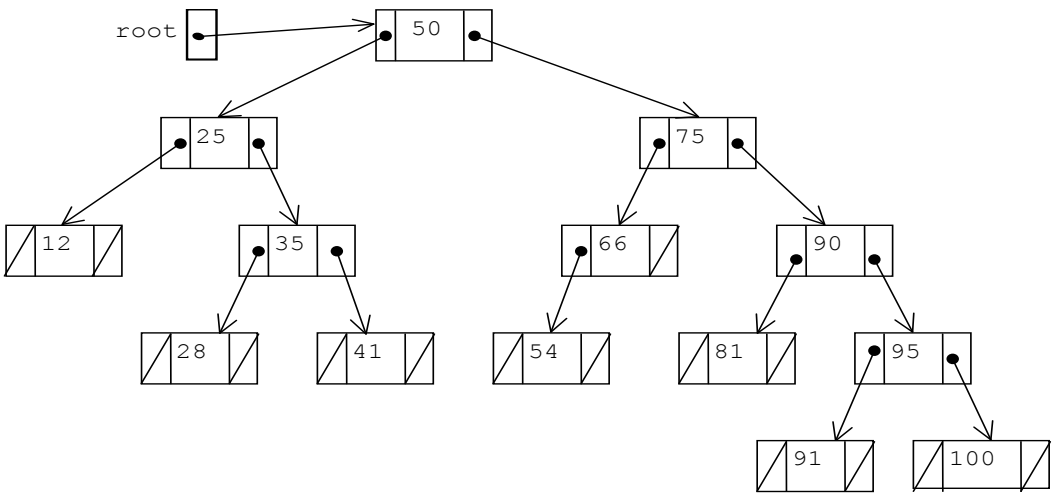
[0]	[1]	[2]	[3]	[4]
"Al"	"Di"	"Mo"	"Xu"	null

A linked structure contains nodes that store a reference to an element and a reference to another node. The reference to another node may be `null` to indicate there are no more elements stored.



You will see that both of these storage mechanisms — arrays and linked structures — implement the same List ADT in subsequent chapters.

You will also see how binary trees store data that can be added, removed, and found more quickly. The following picture indicates a linked structure where each node contains a reference to data (integers are used here to save space) and references to two other nodes, or to nothing (shown as diagonal lines).



You will also see another storage mechanism that uses a key—such as a student ID, employee number, or Social Security Number—to find the value very quickly.

Array Index	Key	Value mapped to the key (null or five instance variables shown)
[0]	"1023"	"Devon" 40.0 10.50 1 'S'
[1]	null	null
[2]	"5462"	"Al" 42.5 12.00 2 'M'
[3]	null	null
[4]	"3343"	"Ali" 20.0 9.50 0 'S'
…		
[753]	"0930"	"Chris" 0.0 13.50 1 'S'

One focal point of this textbook will see collections using different data structures to implement abstract data types.

Abstract Data Types

An **abstract data type** (ADT) describes a set of data values and associated operations that are precisely specified independent of any particular implementation. An abstract data type can be specified using axiomatic semantics. For example, here is the Bag ADT as described by the National Institute of Standards and Technology (NIST)[1].

[1] http://www.nist.gov/dads/HTML/bag.html

Bag

Definition: An unordered collection of values that may have duplicates.

Formal Definition: A bag has a single query function, occurencesOf(v, B), which tells how many copies of an element are in the bag, and two modifier functions, add(v, B) and remove(v, B). These may be defined with axiomatic semantics as follows.

1. new() returns a bag
2. occurencesOf(v, new()) = 0
3. occurencesOf(v, add(v, B)) = 1 + occurencesOf(v, B) [2]
4. occurencesOf(v, add(u, B)) = occurencesOf(v, B) if $v \neq u$
5. remove(v, new()) = new()
6. remove(v, add(v, B)) = B
7. remove(v, add(u, B)) = add(u, remove(v, B)) if $v \neq u$

where B is a bag and u and v are elements.

The predicate isEmpty(B) may be defined with the following additional axioms:

8. isEmpty(new()) = true
9. isEmpty(add(v, B)) = false

Also known as multi-set.
Note: A bag, or multi-set, is a set where values may be repeated. Inserting 2, 1, 2 into an empty set gives the set {1, 2}. Inserting those values into an empty bag gives {1, 2, 2}.

---

[2] This definition shows a bag B is passed as an argument. In an object-oriented language you send a message to an object of type B as in `aBag.add("New Element);` rather than `add("New Element", aBag);`

Although an abstract data type describes how each operation affects data, an ADT

- does not specify how data will be stored
- does not specify the algorithms needed to accomplish the listed operations
- is not dependent on a particular programming language or computer system.

An ADT can also be described as a Java interface that specifies the operations and JUnit assertions to specify behavior.

ADTs Described with Java Interfaces and JUnit Assertions

The Java `interface` introduced in the preceding chapter can also be used to specify an abstract data type. For example, the following Java interface specifies the operations for a Bag ADT as method headings. The `new` operation from NIST's Bag ADT is not included here simply because the Java interface does not allow constructor to be specified in an interface. We will use `new` later when there is some class that implements the interface. The syntactic element <E> will require the class to specify the type of element to be stored.

```
/**
 * This interface specifies the methods for a Bag ADT. It is designed to be
 * generic so any type of element can be stored.
 */
public interface Bag<E> {

  /**
   * Add element to this Bag.
   *
   * @param element: The element to insert.
   */
  public void add(E element);

  /**
   * Return true if no elements have been added to this bag.
   *
   * @return False if there is one or more elements in this bag.
   */
  public boolean isEmpty();
```

```
  /**
   * Return how many elements match element according to the equals method of
   * the type specified at construction.
   *
   * @param element The element to count in this Bag.
   */
  public int occurencesOf(E element);

  /**
   * Remove the first occurrence of element and return true if found. Otherwise
   * leave this Bag unchanged.
   *
   * @param element: The element to remove
   * @return true if element was removed, false if element was not equal to any
   *         in this bag.
   */
  public boolean remove(E element);
}
```

Like an ADT specified with axiomatic expressions, an `interface` does not specify the data structure and the algorithms to add, remove, or find elements. Comments and well-named identifiers imply the behavior of the operations; however the behavior can be made more explicit with assertions. For example, the assertions like those shown in the following test method help describe the behavior of `add` and `occurencesOf`. This code assumes that a class named `ArrayBag` implements `interface Bag<E>`.

```
  @Test
  public void testOccurencesOf() {
    Bag<String> names = new ArrayBag<String>();

    // A new bag has no occurrences of any string
    assertEquals(0, names.occurencesOf("NOT here"));
    names.add("Sam");
    names.add("Devon");
    names.add("Sam");
    names.add("Sam");
    assertEquals(0, names.occurencesOf("NOT Here"));
    assertEquals(1, names.occurencesOf("Devon"));
    assertEquals(3, names.occurencesOf("Sam"));
  }
```

One Class, Many Types

The Bag ADT from NIST does not specify what type of elements can be stored. The Bag interface does. Any class that implements the Bag interface can store any type of element. This is specified with the parameter E in `add`, `occurencesOf`, and `remove`. The compiler replaces the type specified in a construction wherever E is found. So, when a Bag is constructed to store BankAccount objects, you can add BankAccount objects. At compiletime, the parameter E in `public void add(E element)` is considered to be BankAccount.

```
Bag<BankAccount> accounts = new ArrayBag<BankAccount>();
accounts.add(new BankAccount("Chris", 100.00));
accounts.add(new BankAccount("Skyler", 2802.67));
accounts.add(new BankAccount("Sam", 31.13));
```

When a `Bag` is constructed to store `Point` objects, you can add `Point` objects. At compile time, the parameter `E in public void add(E element)` is considered to be `Point`.

```
Bag<Point> points = new ArrayBag<Point>();
points.add(new Point(2, 3));
points.add(new Point(0, 0));
points.add(new Point(-2, -1));
```

Since Java 5.0, you can add primitive values. This is possible because integer literals such as `100` are autoboxed as new `Integer(100)`. At compiletime, the parameter `E` in `public void add(E element)` is considered to be `Integer` for the Bag `ints`.

```
Bag<Integer> ints = new ArrayBag<Integer>();
ints.add(100);
ints.add(95);
ints.add(new Integer(95));
```

One of the advantages of using generics is that you cannot accidentally add the wrong type
of element. Each of the following attempts to add the element that is not of type E for that instance results in a compiletime error.

```
ints.add(4.5);
ints.add("Not a double");
points.add("A String is not a Point");
accounts.add("A string is not a BankAccount");
```

Implementing the Bag interface

A JUnit test describes the behavior of methods in a very real sense by sending messages and making assertions about the state of a `Bag` object. The assertions shown describe operations of the Bag ADT in a concrete fashion. We'll write a test and then the code to make it compile. We'll also need a data structure to store the elements. We'll use an array in this collection class and choose the name `ArrayBag`.

The first assertion shows that a bag is empty immediately after `new` and no longer empty after one `add` (see axioms 8 and 9 above).

```
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

// This unit test shows another view of the bag ADT, this time as a unit test
public class ArrayBagTest {

  @Test
  public void testIsEmptyWithOneAdd() {
    Bag<String> names = new ArrayBag<String>();
    assertTrue(names.isEmpty());
    names.add("Kim");
    assertFalse(names.isEmpty());
  }


    // More to come . . .


}
```

If any assertion fails, either that assertion is stated incorrectly or a method such as `isEmpty`, `add` or the constructor (`new`) may be wrong. In any case, you will know something is wrong with the concrete implementation or testing of the new type (a Java class). Here is the start of a generic class that implements an interface.

```java
// An ArrayBag object can store elements as a multi-set where elements can
// "equals" each other.  There is no specific order to the elements.
//
public class ArrayBag<E> implements Bag<E> {

  // --Instance variables
  private Object[] data;
  private int n;

  // Construct an empty bag that can store any type of element.
  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }

  // Return true if there are 0 elements in this bag.
   public boolean isEmpty() {
    return n == 0;
  }

  // Add element to this bag
  public void add(E element) {
    data[n] = element;
    n++;
  }

  public int occurencesOf(E element) {
    // TODO Change this method stub to a working method
    return 0;
  }

  public boolean remove(E element) {
    // TODO Change this method stub to a working method
    return false;
  }
}
```

There are three new things shown in this new class:

1. `ArrayBag` uses an array of `Object` references to store the elements. Since we want one class to store many different types, it cannot be one particular type. The array could also be declared to be type `E`. `Object` was chosen because at runtime that is exactly the type it will be: an array of `Object`. (It also avoids an ugly cast and a warning.)
2. Since the `ArrayBag<E>` class implements the `Bag<E>` interface, all methods from the interface must exist for class to compile. Two methods are still stubs, just enough to make it compile, but not yet working as specified: `occurencesOf` always returns 0 and `remove` does not do anything other than return false. (`occurencesOf` will be discussed next, but it will be your job to write the appropriate code for the `remove` method.)
3. The big-O runtimes are added as comments to describe the runtime behavior for each.

occurencesOf

The next test method provides another way to indicate that after adding an element with the value "Kim", the Bag has 1 more of that value than before (see axiomatic expression 3 above). It also ensures there are none of a particular value in a new Bag (see axiomatic expression 2 above).

```
@Test
public void testOccurencesOfWithOneElement() {
  // This code assumes ArrayBag<E> is a class that implements Bag<E>
  Bag<String> names = new ArrayBag<String>();
  assertEquals(0, names.occurencesOf("Kim"));
  names.add("Kim");
  assertEquals(1, names.occurencesOf("Kim"));
}
```

Another test method verifies that duplicate elements are allowed.

```
@Test
public void testOccurencesOf() {
  Bag<String> names = new ArrayBag<String>();
  names.add("Sam");
  names.add("Devon");
  names.add("Sam");
  names.add("Sam");
  assertEquals(1, names.occurencesOf("Devon"));
```

```
    assertEquals(3, names.occurencesOf("Sam"));
  }
```

Since there is no specified ordering to `Bag` objects, the element passed as an argument may be located at any index. Also, a value that equals the argument may occur more than once. Thus each element in indexes 0..n-1 must be compared. It makes the most sense to use the `equals` method, assuming `equals` has been overridden to compare the state of two objects rather than the reference values.

```
  // Return how many objects currently in the bag match an element, using
  // the equals method of whatever type of element is stored in this bag.
  public int occurencesOf(E element) {
    int result = 0;
    for (int i = 0; i < n; i++) {
      if (element.equals(data[i]))
        result++; // Found one that equals element
    }
    return result;
  }
```

Self-Check

15-1 Write a test method for `remove`.

15-2 Implement the `remove` method as if you were writing it inside the `ArrayBag` class.

Answers to Self-Check Questions

15-1 One possible test method for remove:

```
@Test
public void testRemove() {
  Bag<String> names = new ArrayBag<String>();
  names.add("Sam");
  names.add("Chris");
  names.add("Devon");
  names.add("Sandeep");

  assertFalse(names.remove("Not here"));

  assertTrue(names.remove("Sam"));
  assertEquals(0, names.occurencesOf("Sam"));
  // Attempt to remove after remove
  assertFalse(names.remove("Sam"));
  assertEquals(0, names.occurencesOf("Sam"));

  assertEquals(1, names.occurencesOf("Chris"));
  assertTrue(names.remove("Chris"));
  assertEquals(0, names.occurencesOf("Chris"));

  assertEquals(1, names.occurencesOf("Sandeep"));
  assertTrue(names.remove("Sandeep"));
  assertEquals(0, names.occurencesOf("SanDeep"));

  // Only 1 left
  assertEquals(1, names.occurencesOf("Devon"));
  assertTrue(names.remove("Devon"));
  assertEquals(0, names.occurencesOf("Devon"));
}
```

15-2 remove method in the ArrayBag class

```
// Remove element if it is in this bag, otherwise return false.
public boolean remove(E element) {

  for (int i = 0; i < n; i++) {
    if (element.equals(data[i])) {
      // Replace with the element at the send
      data[i] = data[n - 1];
      // Reduce the size
      n--;
      return true; // Found--end the search
    }
  }
  // Did not find element "equals" anything in this bag.
  return false;
}
```