CHAPTER 15

Collections and Iterators

15 Collections and Iterators

Science is built up with facts, as a house is with stones.
But, a collection of facts is no more science than a heap
of stones is a house.

Jules Henri Poincaré, Quoted by Bertrand Russell in the preface to *Science and Method*

Introduction

collection

A **collection** is a data structure for holding elements. For example, an array is a collection. If you read Chapter 14, the linked lists and trees discussed there are collections. Java has a repertoire of interfaces and classes that give a uniform treatment of collections.

All the interfaces and classes introduced in this chapter are in the `java.util` package.

Rather than start with an abstract discussion of collections, we will begin with one very useful concrete example of a collection class, namely the class `Vector`. If you do not want to cover the collection framework, you can cover only the material on vectors (Section 15.1). None of this chapter is required for the rest of this book.

Prerequisites

For Section 15.1 on vectors you can get by with only Chapters 1 through 7, but it would be preferable to have also covered Section 8.1 (polymorphism), Chapter 9 (exception handling), and Section 13.1 (interfaces). If you have not covered Chapter 9 on exception handling, then you should interpret any statements about "throwing an exception" as meaning that a run-time error message is given. The one subsection entitled "Vector Iterators" is the only subsection that requires the material on interfaces in Section 13.1 of Chapter 13, but you may omit this subsection without losing continuity. Although Chapter 8 is not required, it may help to understand some of the code in Section 15.1 if you have read Section 8.1 on polymorphism.

Sections 15.2 and 15.3 can be considered one single large topic. These two sections require Section 15.1, Chapters 1 through 9, and Section 13.1 of Chapter 13, which covers interfaces. The material on inner classes in Chapter 13 (sections 13.2 and 13.3) is not needed except for the last subsection, "Defining Your Own Iterator Classes," which requires Section 13.2 (but not 13.3).

None of the material in this chapter is needed for the material on Swing and GUIs. So, you may skip this chapter and go directly to Chapter 16 if you

prefer to cover Swing GUIs before considering the material of this chapter. Of course, this also means that you may cover only Section 15.1 on vectors and then go on to Chapter 16.

15.1 Vectors

> *"Well, I'll eat it," said Alice, "and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door; so either way I'll get into the garden. . . ."*
>
> Lewis Carroll, *Alice's Adventures in Wonderland*

`Vector` is a class. When we speak of a **vector** we simply mean an object of the class `Vector`. A vector can be thought of as being like an array that can grow (and shrink) in length while your program is running. In Java, you can read in the length of an array when the program is run, but once your program creates an array of that length, it cannot change the length of the array. Vectors serve the same purposes as arrays but have the added advantage of being able to grow and shrink. Vectors also have a number of useful methods that perform tasks on vectors that would require writing significant additional code if done with arrays instead of vectors. vector

It often seems that every silver lining has a cloud, and vectors are no exception to this rule. Vectors have lots of advantages over arrays, but they do have at least one conspicuous disadvantage: The elements in a vector must be objects; they cannot be values of a primitive type, such as `int`, `double`, or `char`.

VECTOR BASICS

Vectors are used in much the same way as arrays, but there are some important differences. First, any code that uses the class `Vector` must contain an import statement such as the following: import statement

```
import java.util.Vector;
```

A vector is created and named in the same way as an object of any other class. For example:

```
Vector v = new Vector(100);
```

This makes `v` the name of a vector that has an *initial* **capacity** of 100 items. When we say that a vector has a certain capacity, we mean that it has been allocated memory for that many items, but if it needs to hold more items, the system will automatically allocate more memory. By carefully choosing the initial capacity of a vector, you can often make your code more efficient, but this capacity has no effect on how many items the vector can hold. capacity

size

The capacity of a vector is the number of locations currently allocated for the vector. The **size** of a vector is the number of elements stored in the vector. The size is always less than or equal to the capacity and is typically strictly less than the capacity.

The elements of a vector can be any kind of objects, but for simplicity our examples will usually use objects of type `String`.

Creating a Vector

An object of the class `Vector` is created and named in the same way as any other object.

Examples:

```
Vector vectorObject = new Vector();
Vector anotherVector = new Vector(50);
```

When a number is given as an argument to the constructor, that number determines the initial capacity of the vector.

VECTOR OPERATIONS

no square brackets

Vectors can be used like arrays, but they do not have the array square bracket notation. If you would use the following for an array of strings `a`,

```
a[index] = "Hello";
```

set
get

then the analogous statement for a vector `v` would be

```
v.set(index, "Hello");
```

If you would use the following for an array of strings `a`,

```
String temp = a[index];
```

then the analogous statement for a vector `v` would be

```
String temp = (String)v.get(index);
```

The type cast `(String)` is needed because the base type of all vectors is `Object`. This point is discussed in more detail later in this chapter. The two methods `set` and `get` give vectors approximately the same functionality that the square brackets give to arrays. However, you need to be aware of one important point. The method invocation

```
v.set(index, "Hello");
```

is *not* always completely analogous to

```
a[index] = "Hello";
```

The method `set` can replace any existing element, but unlike an array, you cannot use `set` to put an element at just any index. The method `set` is used to change elements, not to set them for the first time. To set an element for the first time, you usually use the method `add`. The method `add` adds elements at index position `0`, position `1`, position `2`, and so forth in that order. This means that vectors must always be filled in this order. But your code can then go back and change any individual element, just as it can in an array.

`set` restrictions

`add`

Vectors check for indices out of bounds. The index used with `set` must be an integer greater than or equal to `0` and strictly less than the current size of the vector. If it is out of this range, an `ArrayIndexOutOfBoundsException` is thrown.

Although you can use the `set` method as if it were a `void` method, it actually returns a value of type `Object`. The method `set` returns the element its argument replaces.

Accessing at an Index

If `v` is a vector, its elements can be accessed as follows:

Examples:

```
v.set(index, "Hello"); //Sets the element
                       //at index to "Hello".
String temp = (String)v.set(index, "Hello"); //Sets element at
                 //index and returns the element formerly at index.
String temp2 = (String)v.get(index); //The expression
    //v.get(index) returns the element at position index.
```

The `index` must be greater than or equal to `0` and strictly less than the current size of the vector `v`. If it is not in this range, an `ArrayIndexOutOfBoundsException` is thrown.

The method `set` can be used as a `void` method, but it actually returns a value of type `Object`, namely the value replaced by its argument.

The method invocations `v.get(index)` and `v.set(index, "Hello")` always return their values as values of type `Object` and so typically need a type cast.

The method `add` is overloaded. The one-argument version adds an element to the end of the list of elements in the vector. For example:

`add`

```
v.add("I'm at the end of the vector.");
```

The two-element version of `add` allows you to add an element at any vector index from `0` through (and including) the size of the vector. For example,

```
v.add(i, "Hello");
```

ADD

The method add is overloaded. Using the one-argument version of add, elements are added to a vector at index 0, then 1, then 2, and so forth in that order.

EXAMPLES:

```
v.add("I'm at index zero");
v.add("I'm at index one");
v.add("I'm at index two");
```

The object v is a vector. The one-argument version of add always adds its argument at location `v.size()` and increases the size of the vector by one.

The two-argument version of add allows you to add an element at any location from 0 to the size of the array. All the elements at indices greater than the index receiving the new element are moved up one index. So, both the one-argument and two-argument versions of add increase the size of the vector by one.

EXAMPLES:

```
v.add(1, "new value at one");
v.add(v.size(), "I'm new at end.");
```

The object v is the same vector as in the previous examples. If the index used as the first argument is not in the range 0 to the size of the vector (inclusive), then an `ArrayIndexOutOfBoundsException` is thrown.

adds the element `"Hello"` at index `i`. All elements that were originally at locations `i` or higher have their indices increased by one. The size of the vector is thus increased by one.

size

You can find out how many elements are stored in the vector by using the method `size`. If `v` is a vector, `v.size()` returns the **size** of the vector, which is the number of elements stored in it. The indices of these elements go from `0` to 1 less than `v.size()`.

These basic vector operations are illustrated in Display 15.1.

With arrays, the square brackets and the instance variable `length` are the only tools automatically provided for you. If you want to use arrays for other things, you must write code to manipulate the arrays. Vectors, on the other hand, come with a number of powerful methods that can do many of the things you would need to write code to do with arrays. For example, the class `Vector` has a method to insert a new element between two elements in the vector. Most of these methods are described in Display 15.2.

Display 15.1 A Vector Demonstration *(Part 1 of 2)*

```
1  import java.util.Vector;

2  public class VectorDemo
3  {
4      public static void main(String[] args)
5      {
6          Vector poem = new Vector(10);


7          poem.add("A diller,");
8          poem.add("a dollar,");
9          poem.add("a ten o'clock vector scholar.");


10         System.out.println("The vector poem contains:");
11         System.out.println();
12         int index;
13         int vectorSize = poem.size();
14         for (index = 0; index < vectorSize; index++)
15             System.out.println(poem.get(index));
16         System.out.println();


17         String oldElement =
18                        (String)poem.set(1, "a dollar fifty,");
19         System.out.println("\"" + oldElement
20                                    + "\" is now replaced with");
21         System.out.println("\"" + poem.get(1) + "\"");
22         System.out.println();


23         System.out.println("The vector poem now contains:");
24         System.out.println();
25         vectorSize = poem.size();
26         for (index = 0; index < vectorSize; index++)
27             System.out.println(poem.get(index));
28     }

29 }
```

Notice the type cast.

Display 15.1 **A Vector Demonstration *(Part 2 of 2)***

SAMPLE DIALOGUE

```
The vector poem contains:

A diller,
a dollar,
a ten o'clock vector scholar.

"a dollar," is now replaced with
"a dollar fifty,"

The vector poem now contains:

A diller,
a dollar fifty,
a ten o'clock vector scholar.
```

THE METHOD size

The method `size` returns the number of elements in a vector.

EXAMPLE:

```
for (int i = 0; i < v.size(); i++)
    System.out.println(v.get(i));
```

`v` is a vector.

remove

Among other methods in this table are a number of methods for removing elements from a vector. For example, you can remove the `String` `"Bad String"` from the vector `v` as follows:

```
if (v.remove("Bad String"))
    System.out.println("String \"Bad String\" removed.");
else
    System.out.println("\"Bad String\" not in vector.");
```

Note that the method `remove` returns `true` if the argument is removed and `false` if the argument was not in the vector.

Display 15.2 Some Methods in the Class `Vector` *(Part 1 of 5)*

The `Vector` class and the `Iterator` interface are in the `java.util` package.

All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause. (If you have not yet studied exceptions, you can consider the exceptions to be run-time error messages.)

`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

```
public Vector(int initialCapacity, int capacityIncrement)
```

Constructs an empty vector with the specified initial capacity and capacity increment. When the vector needs to grow, it will add room for `capacityIncrement` more items.

Throws an `IllegalArgumentException` if `initialCapacity` is negative.

```
public Vector(int initialCapacity)
```

Creates an empty vector with the specified initial capacity. When the vector needs to increase its capacity, the capacity doubles.

Throws an `IllegalArgumentException` if `initialCapacity` is negative.

```
public Vector()
```

Creates an empty vector with an initial capacity of `10`. When the vector needs to increase its capacity, the capacity doubles.

```
public Vector(Vector v)
```

Creates a vector that contains all the elements of the vector `v` in the same order as they have in `v`. In other words, the elements have the same index in the vector created as they do in `v`. This is not quite a true copy constructor because it does not preserve capacity. The capacity of the created vector will be `v.size()`, not `v.capacity`.

The vector created is only a shallow copy of the vector argument. The vector created contains references to the elements in `v` (not references to clones of the elements in `v`).

Throws a `NullPointerException` if `v` is `null`.

(The parameter type is really `Collection` not `Vector`, but a `Vector` is also a `Collection`; so you can safely act as if the parameter type is `Vector`. We have not yet discussed `Collection`. It is discussed in Section 15.2.)

ARRAYLIKE METHODS

```
public Object set(int index, Object newElement)
```

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned. If you draw an analogy between the vector and an array `a`, this is analogous to setting `a[index]` to the value `newElement`. The `index` must be a value greater than or equal to `0` and strictly less than the current size of the vector.

Throws an `ArrayIndexOutOfBoundsException` if the `index` is not in this range.

Display 15.2 Some Methods in the Class Vector *(Part 2 of 5)*

```
public Object get(int index)
```

Returns the element at the specified index. This is analogous to returning `a[index]` for an array `a`. The `index` must be a value greater than or equal to `0` and less than the current size of the vector.

Throws an `ArrayIndexOutOfBoundsException` if the `index` is not in this range.

METHODS TO ADD ELEMENTS

```
public boolean add(Object newElement)
```

Adds `newElement` to the end of the calling vector and increases its size by 1. The capacity of the vector is increased if that is required. Returns `true` if the add was successful. This method is often used as if it were a `void` method.

```
public void add(int index, Object newElement)
```

Inserts `newElement` as an element in the calling vector at the specified index and increases the size of the calling vector by 1. Each element in the vector with an index greater than or equal to `index` is shifted upward to have an index that is 1 greater than the value it had previously. The `index` must be a value greater than or equal to `0` and less than *or equal to* the size of the vector (before this addition).

Throws an `ArrayIndexOutOfBoundsException` if the index is not in this range.

Note that you can use this method to add an element after the last current element. The capacity of the vector is increased if that is required.

METHODS TO REMOVE ELEMENTS

```
public Object remove(int index)
```

Deletes the element at the specified index and returns the element deleted. The size of the calling vector is decreased by 1. The capacity of the calling vector is not changed. Each element in the vector with an index greater than or equal to `index` is decreased to have an index that is 1 less than the value it had previously. The `index` must be a value greater than or equal to `0` and less than the size of the vector (before this removal).

Throws an `ArrayIndexOutOfBoundsException` if the `index` is not in this range.

```
public boolean remove(Object theElement)
```

Removes the first occurrence of `theElement` from the calling vector. If `theElement` is found in the vector, then each element in the vector with an index greater than or equal to `theElement`'s index is decreased to have an index that is 1 less than the value it had previously. Returns `true` if `theElement` was found (and removed). Returns `false` if `theElement` was not found in the calling vector. If the element was removed, the size is decreased by 1. The capacity is not changed.

```
public void clear()
```

Removes all elements from the calling vector and sets its size to zero.

SEARCH METHODS

```
public boolean isEmpty()
```

Returns `true` if the calling vector is empty (that is, has size `0`); otherwise returns `false`.

```
public boolean contains(Object target)
```

Returns `true` if `target` is an element of the calling vector; otherwise returns `false`. Uses the method `equals` of the object `target` to test for equality.

Display 15.2 Some Methods in the Class Vector *(Part 3 of 5)*

`public int indexOf(Object target)`

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.

`public int indexOf(Object target, int startIndex)`

Returns the index of the first element that is equal to `target`, but only considers indices that are greater than or equal to `startIndex`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.

Throws an `IndexOutOfBoundsException` if `startIndex` is greater than or equal to the size of the array.

`public int lastIndexOf(Object target)`

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.

`public Object firstElement()`

Returns the first element of the calling vector.

Throws a `NoSuchElementException` if the vector is empty.

`public Object lastElement()`

Returns the last element of the calling vector.

Throws a `NoSuchElementException` if the vector is empty.

ITERATORS

`public Iterator iterator()`

Returns an iterator for the calling vector.

CONVERTING TO AN ARRAY

`public Object[] toArray()`

Returns an array containing all of the elements in the calling vector. The elements of the array are indexed the same as in the calling vector.

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling vector. The elements of the array are indexed the same as in the calling vector.

The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:

The type of the returned array is that of `a`. If the collection fits in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.

If `a` has more elements than the calling vector, then the element in `a` immediately following the end of the elements copied from the calling vector is set to `null`.

Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the vector.

Throws a `NullPointerException` if `a` is `null`.

Display 15.2 Some Methods in the Class Vector *(Part 4 of 5)*

MEMORY MANAGEMENT

```
public int size()
```

Returns the number of elements in the calling vector.

```
public int capacity()
```

Returns the current capacity of the calling vector.

```
public void ensureCapacity(int newCapacity)
```

Increases the capacity of the calling vector to ensure that it can hold at least `newCapacity` elements. Using `ensureCapacity` can sometimes increase efficiency, but its use is not needed for any other reason.

```
public void trimToSize()
```

Trims the capacity of the calling vector to be the vector's current size. This is used to save storage.

```
public void setSize(int newSize)
```

Sets the size of the calling vector to `newSize`. If `newSize` is greater than the current size, the new elements receive the value `null`. If `newSize` is less than the current size, all elements at index `newSize` and greater are discarded.

Throws an `ArrayIndexOutOfBoundsException` if `newSize` is negative.

MAKE A COPY

```
public Object clone()
```

Returns a clone of the calling vector. The clone is an identical copy of the calling vector.

OLDER METHODS

These are methods that are not part of the newer collection framework, but are retained for backward compatibility. You should use the above newer methods instead. But, you may find these older methods used in older code.

```
public void setElementAt(Object newElement, int index)
```

Same as `set` with the arguments reversed but does not return the element replaced.

Throws an `IndexOutOfBoundsException` if `index` is out of range.

```
public Object elementAt(int index)
```

Same as `get`.

```
public void addElement(Object newElement)
```

Same as `add`.

```
public void insertElementAt(Object newElement, int index)
```

Same as `add`.

Display 15.2 Some Methods in the Class `Vector` *(Part 5 of 5)*

```
public void removeElementAt(int index)
```

Same as `remove` but does not return the element removed.

```
public boolean removeElement(Object theElement)
```

Same as `remove`.

```
public void removeAllElements()
```

Same as `clear`.

base type

The base type of an array can be any type whatsoever. On the other hand, all vectors have the base type `Object`, so to store an item in a vector, it must be of type `Object`. As you will recall, every class is a descendent class of the class `Object`. Thus, every object of every class type is also of type `Object`. So you can add elements of any class type to a vector. In fact, you can even add elements of different class types to the same vector, but this can be a dangerous thing to do. On the other hand, you cannot add elements of any primitive type, such as `int`, `double`, or `char`, to a vector.

primitive types

If you want the equivalent of a vector of elements of some primitive type, such as the type `int`, you must use the corresponding wrapper class, in this case `Integer`. You can have a vector of elements that are of type `Integer`. Wrapper classes are discussed in Chapter 5.

Pitfall

Vector Elements Are of Type `Object`

The fact that an element added to a vector must be an `Object` has more consequences than you might at first think. Consider the following:

```
Vector v = new Vector();
String stringVariable = "Hello";
v.add(stringVariable);
System.out.println(
             "Length is " + (v.get(0)).length());
```

Although this may look fine, it will produce an error message telling you that `v.get(0)` does not have a method named `length`.

You might protest that `v.get(0)` is of type `String`, and so it does have a method named `length`. You would be right, but Java acts as if it does not know that `v.get(0)` is of type `String`. It knows only that it is an element of a vector, and all it admits to knowing about elements of a vector is that they are of type `Object`. You need to tell Java that `v.get(0)` is of type `String` by using a type cast as follows:

```
(String)(v.get(0))
```

So the troublesome output statement needs to be rewritten to the following, which will work fine:

```
System.out.println("Length is " +
                    ((String)(v.get(0))).length());
```

There are certain special cases where a type cast is not required (although it would cause no harm). Note that `System.out.println` and certain other methods automatically add an invocation of `toString()` to their argument. Since the class `Object` does have a method named `toString()`, the following works fine:

```
System.out.println(v.get(0));
```

because it is equivalent to

```
System.out.println( (v.get(0)).toString() );
```

(and dynamic binding ensures that the correct version of `toString()` is invoked).

The Base Type of a Vector Is `Object`

All vectors have base type `Object`, but all classes are descendent classes of the class `Object`. This means that an element of a vector can be an object of any class, but you cannot have vector elements of a primitive type such as `int`, `double`, or `char`.

Since vector elements are normally returned as values of type `Object`, they usually require a type cast before you can do much with them.

Self-Test Exercises

1. Suppose `v` is a vector. How do you add the string `"Hello"` to the vector `v`?
2. Suppose `v` is a vector with the string `"Hello"` at index position `10`. How do you change the string at index position `10` to `"Good-bye"`?
3. Can you use the method `set` to place an element in a vector at any index you want?
4. Can you use the method `add` to place an element in a vector at any index you want? Can you use the method `add` to insert an element at any position (any index) for which you cannot use `set`?
5. If you create a vector with the following, can the vector contain more than 50 elements?

   ```
   Vector v = new Vector(50);
   ```

6. Give code that will output all the elements in a vector `v` to the screen. Assume that the elements all have a suitable `toString()` method.

7. Write a class for sorting strings into lexicographic order that follows the outline of the class `SelectionSort` in Display 6.9 of Chapter 6. Your definition, however, will use a vector of elements (all of which happen to be strings) rather than an array of elements of type `int`. Remember, you can compare two strings to see which is lexicographically first by using the `String` method `compareTo`. For strings `s1` and `s2`, `s1.compareTo(s2)` returns a negative number if `s1` is lexicographically before `s2`, returns `0` if `s1` equals `s2`, and returns a positive number if `s1` is lexicographically after `s2`. Call your class `StringSelectionSort`.

Tip

Comparing Vectors and Arrays

Vectors are used for the same sorts of applications as arrays. Each has its advantages and disadvantages. The advantage of vectors is that they have many built-in features. For example, a vector is automatically a partially filled vector. The method `size` keeps track of how much of the vector is filled with meaningful elements. This is illustrated in the sample program in Display 15.1. Vectors also have built-in methods to accomplish many of the common tasks that would require you to design your own code if you were using arrays. For example, with vectors, you have a method to insert an element at any specified point in the vector, a method to delete an element from any place in the vector, and a method to test whether or not an element is in the vector. (See Display 15.2.)

Perhaps the biggest advantage of vectors over arrays is that vectors automatically increase their capacity should your program need room for more elements. Your program can determine the size of an array when the program is run, but once the array is created, the size cannot be changed.

Some advantages of arrays are that they are more efficient, they have a very nice notation that uses the square brackets, and, perhaps most importantly, the base type of an array can be of any type. The base type of a vector is always the type `Object`. This is not a disadvantage if you want to store objects of some class, but if you want to store values of a primitive type in a vector, you need to use the wrapper class corresponding to the primitive type. With an array, you can simply make the primitive type the base type of the array.

VECTOR ITERATORS

An **iterator** is an object that allows your code to produce the elements in a vector or other container one after the other, producing each element exactly once. (If there are repeated elements in the vector, the iterator repeats them the same number of times.) Iterators will be discussed in detail later in this chapter, but we will give you a preview of iterators by describing how they work with vectors.

iterator

An iterator for a vector satisfies the `Iterator` interface, which has only the following three method headings that must be implemented:

`Iterator` interface

```
public Object next();
```
Returns the next object in the vector.

```
public boolean hasNext();
```
Returns `true` if the method `next` has not yet returned all the elements in the vector; returns `false` otherwise.

`public void remove()`
Removes the last element returned by `next()`. This method can be called at most once for each call of `next()`.

In the case of a vector, the iterator produces the elements in order going in order from the element at index `0` to the element at the last index used.

The vector should not have elements added or removed while an iteration through the elements is in progress, except, that is, by the iterator method `remove()`; otherwise the behavior of the iterator methods is no longer guaranteed to be as described.

Vectors have a method that produces an iterator for the vector. The method is called `iterator`, so the following produces an iterator named `i` for the vector `v`:

```
Iterator i = v.iterator();
```

You can then do something to all the elements in the vector as follows:

```
while (i.hasNext())
    Do something with i.next().
```

For example, if the vector contains `String` objects, one concrete example is

```
Iterator i = v.iterator();
int count = 0;
while (i.hasNext())
    count = count + ((String)i.next()).length();
```

This code sets the value of `count` equal to the total number of characters in all the elements of the vector `v`.

Use of the type name `Iterator` requires an import statement, such as the following:

```
import java.util.Iterator;
```

Another example of using an iterator for a vector is given in Display 15.3. If you compare Displays 15.1 and 15.3, you will quickly realize that you can always use an `int` variable for a vector index in place of an iterator for the vector. However, some might say that the use of an iterator is cleaner than the use of an `int` variable for an index. Our main reason for doing this example is to have a concrete example of an iterator to motivate our general discussion of iterators later in this chapter.

Tip

Use `trimToSize` to Save Memory

Vectors automatically increase their capacity when your program needs them to have additional capacity. However, the capacity may increase beyond what your program requires. Also, when your program needs less capacity in a vector, the vector does not automatically shrink. If your

Display 15.3 A Vector Iterator

```
1  import java.util.Vector;
2  import java.util.Iterator;

3  public class VectorIteratorDemo
4  {
5      public static void main(String[] args)
6      {
7          Vector poem = new Vector(10);

8          poem.add("A diller,");
9          poem.add("a dollar,");
10         poem.add("a ten o'clock vector scholar.");

11         System.out.println("The vector poem contains:");

12         Iterator i = poem.iterator();
13         while (i.hasNext())
14             System.out.println(i.next());

15         i.remove();

16         System.out.println();
17         System.out.println("The vector poem now contains:");

18         i = poem.iterator();
19         while (i.hasNext())
20             System.out.println(i.next());

21         System.out.println("End of program.");
22     }
23 }
```

SAMPLE DIALOGUE

```
The vector poem contains:
A diller,
a dollar,
a ten o'clock vector scholar.

The vector poem now contains:
A diller,
a dollar,
End of program.
```

vector has a large amount of excess capacity, you can save memory by using the methods `setSize` and `trimToSize` to shrink the capacity of a vector.

setSize

If v is a vector, an invocation of `v.setSize(n)` will set the size of v to n and discard any elements in positions n or higher; if n is greater than the current size of the vector, the new element positions will be set to `null`.

trimToSize

The invocation `v.trimToSize()` will shrink the capacity of the vector v down to the size of v, so that there is no unused capacity in v. Normally, you should use `trimToSize` only when you know the vector will not later need its extra capacity.

Self-Test Exercises

8. What is the base type of a vector?
9. Can you store a value of type `int` in a vector?
10. Suppose v is a vector. What is the difference between `v.capacity()` and `v.size()`?
11. Suppose v is a vector and `v.size()` returns `10`. Now suppose that your program has the following invocation:

    ```
    v.setSize(20);
    ```

    What will be the values of the new elements at indices `10` through `19`? (Garbage values? Some default value? What default value? Something else?)
12. Rewrite the following method so it uses an iterator in place of the `for` loop:

```
/**
 Returns the lexicographically first value among
 v.get(0), v.get(1),..., v.get(v.size() – 1)
 Precondition: v contains only Strings; v.size() > 0.
*/
private static String smallest(Vector v)
{
    String min = (String)v.get(0);
    int index;
    for (index = 1; index < v.size(); index++)
       if (((String)(v.get(index))).compareTo(min) < 0)
           min = (String)v.get(index);
    return min;
}
```

15.2 Collections

Put all your eggs in one basket and
—WATCH THAT BASKET.

Mark Twain, *Pudd'nhead Wilson*

A Java collection is a class that holds objects. This concept is made precise by the `Collection` interface. A Java **collection** is any class that implements the `Collection` interface. One example of a Java collection class is the `Vector` class. The `Collection` interface allows you to write code that applies to all Java collections so that you do not have to rewrite the code for each specific collection. There are other interfaces and abstract classes that are in some sense or another produced from the `Collection` interface. Some of these are shown in Display 15.4. In this section we give you an introduction to this Java collection framework. The topic is too large to treat exhaustively in this book, so this can only be an introductory treatment.

collection

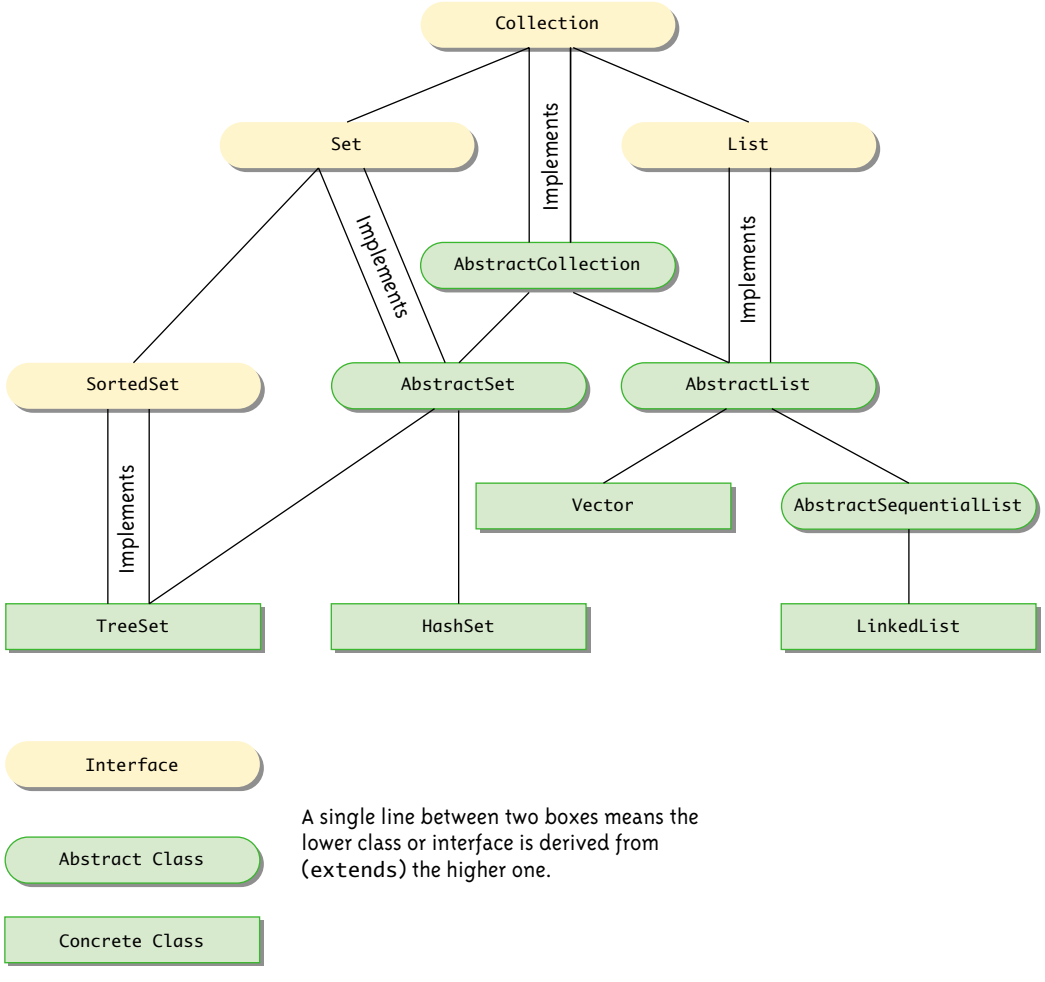
Collections are used along with *iterators*, which are discussed in Section 15.3. Separating collections and iterators into two sections turns out to be a handy way of organizing the material, but the two topics of collections and iterators are intimately intertwined and in practice you normally use them together.

THE COLLECTION FRAMEWORK

The `Collection` interface is the highest level of Java's framework for collection classes. The `Collection` interface describes the basic operations that all collection classes should implement. These operations (method headings) for the `Collection` interface are given in Display 15.5. Since an interface is a type, you can define methods with a parameter of type `Collection` and that parameter can be filled in with an argument that is an object of any class in the collection framework (that is, any class that implements the `Collection` interface). This turns out to be a very powerful tool. Let's explore the possibilities. So far, we have seen one class that implements the `Collection` interface, namely the class `Vector`. In addition to the methods given in Section 15.1 for the class `Vector`, the class `Vector` also implements all the methods given in Display 15.5. There are a number of different predefined classes that implement the `Collection` interface, and you can define your own classes that implement the `Collection` interface. If you write a method to manipulate a parameter of type `Collection`, it will work for all of these classes. Also, the methods in the `Collection` interface ensure that you can intermix the use of different collection classes. For example, consider the method

```
public boolean containsAll(Collection collectionOfTargets)
```

Display 15.4 The Collection Landscape



You can use this with two `Vectors` (one the calling object and one the argument) to see if one contains all the elements of the other, but you can also use it with a `Vector` object and an object of any other class that implements the `Collection` interface to compare the elements in these two different kinds of `Collections`.

Display 15.5 Method Headings in the `Collection` Interface *(Part 1 of 3)*

The `Collection` interface is in the `java.util` package.

All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Collection` interface should have at least two constructors: A no-argument constructor that creates an empty `Collection` object, and a constructor with one parameter of type `Collection` that creates a `Collection` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

```
public boolean contains(Object target)
```

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.

Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

```
public boolean containsAll(Collection collectionOfTargets)
```

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an `element` in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object.

Throws a `ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).

Throws a `NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).

Throws a `NullPointerException` if `collectionOfTargets` is `null`.

```
public boolean equals(Object other)
```

This is the `equals` of the collection, not the `equals` of the elements in the collection. Overrides the inherited method `equals`. Although there are no official constraints on `equals` for a collection, it should be defined as we have described in Chapter 7 and also to satisfy the intuitive notion of collections being equal.

```
public int size()
```

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Display 15.5 Method Headings in the `Collection` Interface *(Part 2 of 3)*

`Iterator iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The array returned should be a new array so that the calling object has no references to the returned array. (You might also want the elements in the array to be clones of the elements in the collection. However, this is apparently not required by the interface, since library classes, such as `Vector`, return arrays that contain references to the elements in the collection.)

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:

The type of the returned array is that of `a`. If the elements in the calling object fit in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.

If `a` has more elements than the calling object, the element in `a` immediately following the end of the copied elements is set to `null`.

If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order. (Iterators are discussed in Section 15.3.)

Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.

Throws a `NullPointerException` if `a` is `null`.

`public int hashCode()`

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is only here to make the definition of the `Collection` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have the method throw an `UnsupportedOperationException`.

OPTIONAL METHODS

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if for some reason you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

`public boolean add(Object element)` *(Optional)*

Ensures that the calling object contains the specified `element`. Returns `true` if the calling object changed as a result of the call. Returns `false` if the calling object does not permit duplicates and already contains `element`.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

Throws a `ClassCastException` if the class of `element` prevents it from being added to the calling object.

Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements.

Throws an `IllegalArgumentException` if some other aspect of `element` prevents it from being added to the calling object.

Display 15.5 Method Headings in the `Collection` Interface *(Part 3 of 3)*

`public boolean addAll(Collection collectionToAdd)` *(Optional)*

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

Throws a `ClassCastException` if the class of an element of `collectionToAdd` prevents it from being added to the calling object.

Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

Throws an `IllegalArgumentException` if some aspect of an element of `collectionToAdd` prevents it from being added to the calling object.

`public boolean remove(Object element)` *(Optional)*

Removes a single instance of the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`public boolean removeAll(Collection collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

Throws a `ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling collection (optional).

Throws a `NullPointerException` if `collectionToRemove` contains one or more `null` elements and the calling object does not support `null` elements (optional).

Throws a `NullPointerException` if `collectionToRemove` is `null`.

`public void clear()` *(Optional)*

Removes all the elements from the calling object.

Throws an `UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`public boolean retainAll(Collection saveElements)`

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws a `ClassCastException` if the types of one or more elements in `saveElements` are incompatible with the calling object (optional).

Throws a `NullPointerException` if `saveElements` contains one or more `null` elements and the calling object does not support `null` elements (optional).

Throws a `NullPointerException` if `saveElements` is `null`.

Collections Store Values of Type Object

All the collection classes and interfaces discussed in this chapter have base type `Object`, but all classes are descendent classes of the class `Object`. This means that an element of a collection can be an object of any class, but you cannot have a collection of elements of a primitive type such as `int`, `double`, or `char`.

Since collection elements are normally returned as values of type `Object`, they usually require a type cast before you can do much with them.

Set
List
interfaces

The relationships between some of the classes and interfaces that implement or extend the `Collection` interface are given in Display 15.4. There are two main interfaces that extend the `Collection` interface: the `Set` interface and the `List` interface. Classes that implement the `Set` interface do not allow an element in the class to occur more than once. Classes that implement the `List` interface have their elements ordered as on a list, so there is a zeroth element, a first element, a second element, and so forth. A class that implements the `List` interface allows elements to occur more than once. The `Vector` class implements the `List` interface. The methods in the `Set` and `List` interfaces are given in Displays 15.6 and 15.7, respectively. The `Set` interface has the same method headings as the `Collection` interface, but in some cases the semantics (intended meanings) are different and methods that are optional in the `Collection` interface are required in the `Set` interfaces. The `List` interface has more method headings than the `Collection` interface, and some of the methods inherited from the `Collection` interface receive somewhat different semantics.

Display 15.6 Methods in the `Set` Interface *(Part 1 of 4)*

The `Set` interface is in the `java.util` package.

The `Set` interface extends the `Collection` interface.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Set` interface should have at least two constructors: A no-argument constructor that creates an empty `Set` object, and a constructor with one parameter of type `Collection` that creates a `Set` object with the same elements as the constructor argument.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

Display 15.6 Methods in the Set Interface *(Part 2 of 4)*

`public boolean contains(Object target)`

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.

Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

`public boolean containsAll(Collection collectionOfTargets)`

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an `element` in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object. If `collectionOfTargets` is itself a `Set`, this is a test to see if `collectionOfTargets` is a subset of the calling object.

Throws a `ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).

Throws a `NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).

Throws a `NullPointerException` if `collectionOfTargets` is `null`.

`public boolean equals(Object other)`

If the argument is a `Set`, returns `true` if the calling object and the argument contain exactly the same elements; otherwise returns `false`. If the argument is not a `Set`, `false` is returned.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

`Iterator iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. A new array must be returned so that the calling object has no references to the returned array.

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are described in the table for the `Collection` interface (Display 15.5).

Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.

Throws a `NullPointerException` if `a` is `null`.

Display 15.6 Methods in the Set Interface *(Part 3 of 4)*

`public int hashCode()`

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is here only to make the definition of the `Set` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have it throw an `UnsupportedOperationException`.

ADDING AND REMOVING ELEMENTS

Unlike the `Collection` interface, the following methods are required for the `Set` interface.

`public boolean add(Object element)`

If `element` is not already in the calling object, `element` is added to the calling object and `true` is returned. If `element` is in the calling object, the calling object is unchanged and `false` is returned.

Throws a `ClassCastException` if the class of `element` prevents it from being added to the set.

Throws a `NullPointerException` if `element` is `null` and the set does not support `null` elements.

Throws an `IllegalArgumentException` if some other aspect of `element` prevents it from being added to this set.

`public boolean addAll(Collection collectionToAdd)`

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise. Thus, if `collectionToAdd` is a `Set`, then the calling object is changed to the union of itself with `collectionToAdd`.

Throws a `ClassCastException` if the class of some element of `collectionToAdd` prevents it from being added to the calling object.

Throws `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

Throws an `IllegalArgumentException` if some aspect of some element of `collectionToAdd` prevents it from being added to the calling object.

`public boolean remove(Object element)`

Removes the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`public boolean removeAll(Collection collectionToRemove)`

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws a `ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling object (optional).

Throws a `NullPointerException` if the calling object contains a `null` element and `collectionToRemove` does not support `null` elements (optional).

Throws a `NullPointerException` if `collectionToRemove` is `null`.

Display 15.6 Methods in the `Set` Interface *(Part 4 of 4)*

```
public void clear()
```

Removes all the elements from the calling object.

```
public boolean retainAll(Collection saveElements)
```

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`. If the argument is itself a `Set`, this changes the calling object to the intersection of itself with the argument.

Throws a `ClassCastException` if the types of one or more elements in the calling object are incompatible with `saveElements` (optional).

Throws a `NullPointerException` if `saveElements` contains a `null` element and the calling object does not support `null` elements (optional).

Throws a `NullPointerException` if `saveElements` is `null`.

Display 15.7 Methods in the `List` Interface *(Part 1 of 6)*

The `List` interface is in the `java.util` package.

The `List` interface extends the `Collection` interface.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `List` interface should have at least two constructors: A no-argument constructor that creates an empty `List` object, and a constructor with one parameter of type `Collection` that creates a `List` object with the same elements as the constructor argument. If the argument imposes an ordering on its elements, then the `List` created should preserve this ordering.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

```
public boolean contains(Object target)
```

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.

Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

Display 15.7 Methods in the `List` Interface *(Part 2 of 6)*

`public boolean containsAll(Collection collectionOfTargets)`

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For an `element` in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object. The elements need not be in the same order or have the same multiplicity in `collectionOfTargets` and in the calling object.

`public boolean equals(Object other)`

If the argument is a `List`, returns `true` if the calling object and the argument contain exactly the same elements in exactly the same order; otherwise returns `false`. If the argument is not a `List`, `false` is returned.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

`Iterator iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. A new array must be returned so that the calling object has no references to the returned array.

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are described in the table for the `Collection` interface (Display 15.5).

Throws an `ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.

Throws a `NullPointerException` if `a` is `null`.

`public int hashCode()`

Returns the hash code value for the calling object. Neither hash codes nor this method are discussed in this book. This entry is here only to make the definition of the `list` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have it throw an `UnsupportedOperationException`.

OPTIONAL METHODS

As with the `Collection` interface, the following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if for some reason you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

Display 15.7 Methods in the `List` Interface *(Part 3 of 6)*

`public boolean add(Object element)` *(Optional)*

Adds `element` to the end of the calling object's list. Normally returns `true`. Returns `false` if the operation failed, but if the operation failed, something is seriously wrong and you will probably get a run-time error anyway.

Throws an `UnsupportedOperationException` if the `add` method is not supported by the calling object.

Throws a `ClassCastException` if the class of `element` prevents it from being added to the calling object.

Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements.

Throws an `IllegalArgumentException` if some aspect of `element` prevents it from being added to the calling object.

`public boolean addAll(Collection collectionToAdd)` *(Optional)*

Adds all of the elements in `collectionToAdd` to the end of the calling object's list. The elements are added in the order they are produced by an iterator for `collectionToAdd`.

Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.

Throws a `ClassCastException` if the class of an element in `collectionToAdd` prevents it from being added to the calling object.

Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

Throws an `IllegalArgumentException` if some aspect of an element in `collectionToAdd` prevents it from being added to the calling object.

`public boolean remove(Object element)` *(Optional)*

Removes the first occurrence of `element` from the calling object's list, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws a `ClassCastException` if the type of `element` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.

`public boolean removeAll(Collection collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws an `UnsupportedOperationException` if the `removeAll` method is not supported by the calling object.

Throws a `ClassCastException` if the types of one or more elements in the calling object are incompatible with `collectionToRemove` (optional).

Throws a `NullPointerException` if the calling object contains one or more `null` elements and `collectionToRemove` does not support `null` elements (optional).

Throws a `NullPointerException` if `collectionToRemove` is `null`.

Display 15.7 Methods in the `List` Interface *(Part 4 of 6)*

`public void clear()` *(Optional)*

Removes all the elements from the calling object.

Throws an `UnsupportedOperationException` if the `clear` method is not supported by the calling object.

`public boolean retainAll(Collection saveElements)` *(Optional)*

Retains only the elements in the calling object that are also in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws an `UnsupportedOperationException` if the retainAll method is not supported by the calling object.

Throws a `ClassCastException` if the types of one or more elements in the calling object are incompatible with `saveElements` (optional).

Throws a `NullPointerException` if the calling object contains one or more `null` elements and `saveElements` does not support `null` elements (optional).

Throws a `NullPointerException` if the `saveElements` is `null`.

NEW METHOD HEADINGS

The following methods are in the `List` interface but were not in the `Collection` interface. Those that are optional are noted.

`public void add(int index, Object newElement)` *(Optional)*

Inserts `newElement` in the calling object's list at location `index`. The old elements at location `index` and higher are moved to higher indices.

Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if this add method is not supported by the calling object.

Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.

Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.

Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

`public boolean addAll(int index, Collection collectionToAdd)` *(Optional)*

Inserts all of the elements in `collectionToAdd` to the calling object's list starting at location `index`. The old elements at location `index` and higher are moved to higher indices. The elements are added in the order they are produced by an iterator for `collectionToAdd`.

Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index <= size()
```

Throws an `UnsupportedOperationException` if the `addAll` method is not supported by the calling object.

Throws a `ClassCastException` if the class of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

Display 15.7 Methods in the `List` Interface *(Part 5 of 6)*

Throws a `NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

Throws an `IllegalArgumentException` if some aspect of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

```
public Object get(int index)
```

Returns the object at position `index`.

Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

`0 <= index < size()`

```
public Object set(int index, Object newElement) (Optional)
```

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned.

Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

```
0 <= index < size()
```

Throws an `UnsupportedOperationException` if the `set` method is not supported by the calling object.

Throws a `ClassCastException` if the class of `newElement` prevents it from being added to the calling object.

Throws a `NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.

Throws an `IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

```
public Object remove(int index) (Optional)
```

Removes the element at position `index` in the calling object. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the calling object.

Throws an `UnsupportedOperationException` if the `remove` method is not supported by the calling object.

Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

`0 <= index < size()`

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns −1 if `target` is not found.

Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

Display 15.7 Methods in the `List` Interface *(Part 6 of 6)*

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `–1` if `target` is not found.

Throws a `ClassCastException` if the type of `target` is incompatible with the calling object (optional).

Throws a `NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

```
public List subList(int fromIndex, int toIndex)
```

Returns a *view* of the elements at locations `fromIndex` to `toIndex` of the calling object; the object at `fromIndex` is included; the object, if any, at `toIndex` is not included. The *view* uses references into the calling object; so, changing the view can change the calling object. The returned object will be of type `List` but need not be of the same type as the calling object. Returns an empty `List` if `fromIndex` equals `toIndex`.

Throws an `IndexOutOfBoundsException` if `fromIndex` and `toIndex` do not satisfy:

```
0 <= fromIndex <= toIndex <= size()
```

```
ListIterator listIterator()
```

Returns a list iterator for the calling object. (Iterators are discussed in Section 15.3.)

```
ListIterator listIterator(int index)
```

Returns a list iterator for the calling object starting at `index`. The first element to be returned by the iterator is the one at `index`. (Iterators are discussed in Section 15.3.)

Throws an `IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index <= size()
```

Collection Interfaces

The primary interfaces for collection classes are the `Collection`, `Set`, and `List` interfaces. Both the `Set` and the `List` interfaces are derived from the `Collection` interface. The `Set` interface is for collections that do not allow repetition of elements and do not impose an order on their elements. The `List` interface is for collections that do allow repetition of elements and do impose an order on their elements.

Pitfall

Optional Operations

What is the point of an optional method heading in an interface? The whole point of an interface is to specify what methods can be used with an object of the interface type so that you can write

code for an arbitrary object of the interface type. The reasoning behind these optional methods is that they normally would be implemented but in unusual situations a programmer may leave them "unsupported." (The alternative would be to have two interfaces, one with and one without the optional operations. Uncharacteristic of the Java designers, they opted for a smaller number of interfaces.) But, there is still more to the story.

The optional methods are not, strictly speaking, optional. Like the other methods in an interface, the optional methods must have a method body so that the optional method heading is converted to a complete method definition. So, what's optional? The "optional" refers to the semantics of the method. If the method is optional, then you may give it a trivial implementation and you will not be considered to have shirked your responsibly to follow the (unenforced) semantics for the interface.

To keep these optional methods from producing unexplained failures, the interface semantics say that if you do not give an optional method a "real" implementation, then you should have the method body throw an `UnsupportedOperationException`. For example, the `add` method of the `Collection` interface is optional and so can be implemented as follows (provided you have good reason for this):

```
public boolean add(Object element)
{
    throw new UnsupportedOperationException();
}
```

The `UnsupportedOperationException` class is a derived class of the `RunTimeException` class and so an `UnsupportedOperationException` need not be caught in a `catch` block or declared in a `throws` clause.

The intention is that the code for a class that implements an interface with optional methods would be written and used in such a way that this `UnsupportedOperationException` would only be thrown during debugging. These rules on optional methods are part of the semantics of the interface, and like all other parts of the semantics of an interface, they depend entirely on the good will and responsibility of the programmer defining the class that implements the interface.

Optional Methods

When an interface lists a method as "optional," you still need to implement it when defining a class that implements the interface. However, if you do not want to give it a "real" definition, you can simply have the method body throw an `UnsupportedOperationException`.

Tip

Dealing with All Those Exceptions

The tables of methods for the various collection interfaces and classes are liberally sprinkled with statements that certain exceptions are thrown. All these exception classes are of the kind that

need not be caught in a `catch` block and need not be declared in a `throws` clause. They are there primarily for debugging. If you are using an existing collection class, you can view them as run-time error messages. If you are defining a class as a derived class of some other collection class, then most or all of the exception throwing will be inherited, so you need not worry too much about it. If you are defining a collection class from scratch and want your class to implement one of the collection interfaces, then you do need to throw suitable exceptions as specified for the interface.

With one exception (no pun intended), all the exception classes mentioned in this chapter are in the package `java.lang` and so do not require any import statement. The one exception is the `NoSuchElementException`, which is used with vectors in Section 15.1 and with iterators in Section 15.3. The `NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class.

Self-Test Exercises

13. Give the definition of a `boolean` valued static method named `inSome`. The method `inSome` has two parameters of type `Collection` and one parameter of type `Object`. The method returns `true` if the `Object` is in either (or both) `Collections`; it returns `false` otherwise.
14. Give the definition of a static method named `getFirst` that has one parameter of type `List` and a return type of `Object`. The method returns the first element in the `List` or `null` if the `List` is empty.
15. Give the definition of a static `boolean` valued method named `noNull`. The method `noNull` has one parameter of type `Set` and removes `null` from the set if `null` is in the set; otherwise it leaves the set unchanged. The method returns `true` if the set is changed and `false` if it is not changed.

CONCRETE COLLECTION CLASSES

AbstractSet
AbstractList
classes

The abstract classes `AbstractSet` and `AbstractList` are there for convenience when implementing the `Set` and `List` interfaces, respectively. They have almost no methods beyond those in the interfaces they implement. A list of the methods in these two abstract classes is given in Appendix 4. Although these two abstract classes have only a few abstract methods, the other (nonabstract) methods have fairly useless implementations that must be overridden. When defining a derived class of either `AbstractSet` or `AbstractList`, you need to define not just the abstract methods but also all the methods you intend to use. It usually makes more sense to simply use (or define derived classes of) the `HashSet` or `Vector` classes, which are derived classes of `AbstractSet` and `AbstractList`, respectively, and are full implementations of the `Set` and `List` interfaces, respectively.

AbstractCollection

The abstract class AbstractCollection is a skeleton class for the Collection interface. Although it is perfectly legal, you seldom, if ever, need to define a derived class of the AbstractCollection class. Instead, you normally define a derived class of one of the descendent classes of the AbstractCollection class. A list of the methods in AbstractCollection is given in Appendix 4.

HashSet

If you want a class that implements the Set interface and do not need any methods beyond those in the Set interface, you can use the concrete class HashSet. So, after all is said and done, if all you need is a collection class that does not allow elements to occur more than once, then you can use the HashSet class and need not worry about all the other classes and interfaces in Display 15.4. The word "Hash" refers to the fact that the HashSet class is implemented using a *hash table*. We do not cover hash tables in this text, but since this is just part of the implementation of the class HashSet, you do not need to know anything about hash tables to use the class HashSet. The HashSet, of course, implements all the methods in the Set interface (Display 15.6) and it adds no other methods beyond constructors. A summary of the HashSet constructors and other methods is given in Display 15.8. If you want to define your own class that implements the Set interface, you would probably be better off using the HashSet class rather than the AbstractSet class as a base class.

Vector

Similarly, if you want a class that implements the List interface and do not need any methods beyond those in the List interface, you can use the Vector class. So, after all is said and done, if all you need is a collection class that does allow elements to occur more than once, or you need a collection that orders its elements as on a list (that is, as in an array), or you need a class that has both of these properties, then you can use the Vector class and need not worry about all the other classes and interfaces in Display 15.3. The Vector class implements all the methods in the List interface. A table of methods for the vector class is given in Display 15.2. A more complete list of the methods in the Vector class is given in Appendix 4. If you want to define your own class that implements the List interface, you would probably be better off using the Vector class rather than the AbstractList class as a base class.

AbstractSequentialList

LinkedList

The abstract class AbstractSequentialList is derived from the AbstractList class. Although it does override some methods inherited from the class AbstractList, it adds no completely new methods. The point of the AbstractSequentialList class is that it provides for efficient implementation of sequentially moving through the list at the expense of having inefficient implementation of random access to elements (that is, inefficient implementation of the get method). The LinkedList class is a concrete derived class of the abstract class AbstractSequentialList. The implementation of the LinkedList class is similar to that of the linked list classes we discussed in Chapter 14. If you need a List with efficient random access to elements (that is, efficient implementation of the get method), then use the Vector class or a class derived from the Vector class. If you do not need efficient random access but need to efficiently move sequentially through the list, then use the LinkedList class or a class derived from the LinkedList class.

Display 15.8 Methods in the HashSet Class

The HashSet class is in the `java.util` package.

The HashSet class extends the AbstractSet class and implements the Set interface.

The HashSet class implements all of the methods in the Set interface (Display 15.6). The only other methods in the HashSet class are the constructors. The two constructors that do not involve concepts beyond the scope of this book are given below.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public HashSet()
```

Creates a new, empty set.

```
public HashSet(Collection c)
```

Creates a new set that contains all the elements of c.

Throws a NullPointerException if c is null.

```
public HashSet(int initialCapacity)
```

Creates a new, empty set with the specified capacity.

Throws an IllegalArgumentException if initialCapacity is less than zero.

The methods are the same as those described for the Set interface (Display 15.6).

SortedSet
TreeSet

The interface SortedSet and the concrete class TreeSet are designed for implementations of the Set interface that provide for rapid retrieval of elements (efficient implementation of the contains and similar methods). The implementation of the class is similar to the binary tree class discussed in Chapter 14 but with more sophisticated ways to do inserting that keep the tree balanced. We will not discuss the SortedSet interface or the TreeSet class in this text, but you should be aware of their existence so you know what to look for in the Java documentation should you need them.

Self-Test Exercises

16. Can an object that instantiates the HashSet class contain multiple copies of some element?
17. Suppose you want to define a class that orders its elements like a List but does not allow multiple occurrences of an element like a Set. Would it be better to make it a derived class of the Vector class or a derived class of the HashSet class?

A PEEK AT THE MAP FRAMEWORK ✣

The Java **map** framework is similar in character to the collection framework, but it deals with collections of ordered pairs. Objects in the map framework can implement mathematical functions and relations and so can be used to construct database classes. Think of the pair as consisting of a key (to search for) and an associated value. For example, the key might be a social security number and the value might be the salary of the person with that social security number. We will not discuss the map framework in this text, but you should be aware of its existence so you know what to look for in the Java documentation should you need it. Look for the `Map` interface and the `AbstractMap` class and classes derived from the `AbstractMap` class.

map

15.3 Iterators

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said, very gravely, "And go on till you come to the end: then stop."

Lewis Carroll, *Alice in Wonderland*

An **iterator** is an object that is used with a collection to provide sequential access to the elements in the collection. In Section 15.1 we gave you a brief introduction to iterators used with vectors. In this section we present a more detailed and more general discussion of iterators.

iterator

THE ITERATOR CONCEPT

In the next subsection we will discuss the Java `Iterator` interface, but before that let's consider the intuitive idea of an iterator. An iterator is something that allows you to examine and possibly modify the elements in a collection in some sequential order. So, an iterator imposes an ordering on the elements of a collection even if the collection, such as the class `HashSet`, does not impose any order on the elements it contains.

Something that is not an object—and thus not, strictly speaking, a Java `Iterator`—but that satisfies the intuitive idea of an iterator is an `int` variable `i` used with an array `a`. This iterator `i` can be made to start out at the first array as follows:

```
i = 0;
```

The iterator can give you the current element; the current element is simply `a[i]`. The iterator can go to the next element and give you the next element as follows:

```
i++;
```
"Gives you `a[i]`"

The concept of an iterator is simple but powerful enough to be used frequently.

THE `Iterator` INTERFACE

`Iterator` interface

Java formalizes the concept of an iterator with the `Iterator` interface. Any object of any class that satisfies the `Iterator` interface is an `Iterator`. So, an array index is not a Java `Iterator`. However, the index could be an instance variable in an object of an `Iterator` class.

An `Iterator` does not stand on its own. It must be associated with some collection object. How is the association accomplished? In Java, any class that satisfies the `Collection` interface must have a method, named `iterator()`, that returns an `Iterator`. To make things concrete, let's say `c` is an instance of the `HashSet` collection class. You can obtain an iterator for `c` as follows:

```
Iterator iteratorForC = c.iterator();
```

You may not know what class the `iteratorForC` is an instance of, but you do know it satisfies the `Iterator` interface and so you know it has the methods in the `Iterator` interface. These methods are given in Display 15.9.

Display 15.10 contains a simple demonstration of using an iterator with a `HashSet` object. A `HashSet` object imposes no order on the elements in the `HashSet` object, but the iterator imposes an ordering on the elements, namely the order in which they are produced

Display 15.9 Methods in the `Iterator` Interface

The `Iterator` interface is in the `java.util` package.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public Object next()
```

Returns the next element of the collection that produced the iterator.

Throws a `NoSuchElementException` if there is no next element.

```
public boolean hasNext()
```

Returns `true` if `next()` has not yet returned all the elements in the collection; returns `false` otherwise.

```
public void remove() (Optional)
```

Removes from the collection the last element returned by `next`.

This method can be called only once per call to `next`.

Throws `IllegalStateException` if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.

Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

Display 15.10 An Iterator

```
1  import java.util.HashSet;
2  import java.util.Iterator;

3  public class HashSetIteratorDemo
4  {
5      public static void main(String[] args)
6      {
7          HashSet s = new HashSet();

8          s.add("health");
9          s.add("love");
10         s.add("money");

11         System.out.println("The set contains:");

12         Iterator i = s.iterator();
13         while (i.hasNext())
14             System.out.println(i.next());

15         i.remove();

16         System.out.println();
17         System.out.println("The set now contains:");

18         i = s.iterator();
19         while (i.hasNext())
20             System.out.println(i.next());

21         System.out.println("End of program.");
22     }
23 }
```

Sample Dialogue

```
The set contains:
money
love
health

The set now contains:
money
love
End of program.
```

The `HashSet` object does not order the elements it contains, but the iterator imposes an order on the elements. (It would not be illegal if the elements were listed in a different order on your computer, but it would be unexpected.)

by `next()`. There are no requirements on this ordering. It is likely that if you run the program in Display 15.10 twice, the order of the elements output will be the same each time. However, it would not be an error if they were output in different orders each time the program is run.

If the collection used with an `Iterator` imposes an ordering on its elements, such as a vector does, then the `Iterator` will output the elements in that order. See Display 15.3 for an example of this.

Iterators

An iterator is something that allows you to examine and possibly modify the elements in a collection in some sequential order. Java formalizes this concept with the two interfaces `Iterator` and `ListIterator`.

LIST ITERATORS

ListIterator

The collection framework has two iterator interfaces: the `Iterator` interface, which you have already seen and that works with any collection class that implements the `Collection` interface, and the `ListIterator` interface, which is designed to work with collections that satisfy the `List` interface. A `ListIterator` has all the methods that an `Iterator` has plus more methods that provide two new abilities: A `ListIterator` can move in either direction along the list of elements in the collection, and a `ListIterator` has methods, such as `set` and `add`, that can be used to change the elements in the collection. The methods for the `ListIterator` interface are given in Display 15.11.

cursor

The general idea of *next* and *previous* is clear, but we need to make it precise if you are to understand the `next()` and `previous()` methods of the `ListIterator` interface. Every `ListIterator` has a position marker in the list known as the **cursor**. If the list has *n* elements, they are numbered by indices 0 through *n*–1, but there are *n*+1 cursor positions, as indicated in Display 15.12. When `next()` is invoked, the element immediately following the cursor position is returned and the cursor is moved to the next cursor position. When `previous()` is invoked, the element immediately before the cursor position is returned and the cursor is moved back to the proceeding cursor position.

The ListIterator Interface

The `ListIterator` interface differs from the `Iterator` interface by adding the following abilities: A `ListIterator` can move in either direction along the list of elements in the collection, and a `ListIterator` has methods, such as `set` and `add`, that can be used to change the elements in the collection.

Display 15.11 Methods in the `ListIterator` Interface *(Part 1 of 2)*

The `ListIterator` interface is in the `java.util` package.

The *cursor position* is explained in the text and in Display 15.12.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

`NoSuchElementException` is in the `java.util` package, which requires an import statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any import statement.

```
public Object next()
```

Returns the next element of the list that produced the iterator. More specifically, returns the element immediately after the cursor position.

Throws a `NoSuchElementException` if there is no next element.

```
public Object previous()
```

Returns the previous element of the list that produced the iterator. More specifically, returns the element immediately before the cursor position

Throws a `NoSuchElementException` if there is no previous element.

```
public boolean hasNext()
```

Returns `true` if there is a suitable element for `next()` to return; returns `false` otherwise.

```
public boolean hasPrevious()
```

Returns `true` if there is a suitable element for `previous()` to return; returns `false` otherwise.

```
public int nextIndex()
```

Returns the index of the element that would be returned by a call to `next()`. Returns the list size if the cursor position is at the end of the list.

```
public int previousIndex()
```

Returns the index that would be returned by a call to `previous()`. Returns –1 if the cursor position is at the beginning of the list.

```
public void add(Object newElement) (Optional)
```

Inserts `newElement` at the location of the iterator cursor (that is, before the value, if any, that would be returned by `next()` and after the value, if any, that would be returned by `previous()`).

Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.

Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

Throws a `ClassCastException` if the class of `newElement` prevents it from being added.

Throws an `IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

Display 15.11 Methods in the `ListIterator` Interface *(Part 2 of 2)*

`public void remove()` *(Optional)*

Removes from the collection the last element returned by `next()` or `previous()`.

This method can be called only once per call to `next()` or `previous()`.

Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.

Throws an `UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

`public void set(Object newElement)` *(Optional)*

Replaces the last element returned by `next()` or `previous()` with `newElement`.

Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

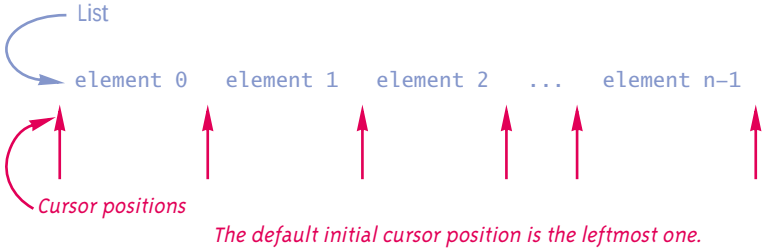
Throws an `UnsupportedOperationException` if the `set` operation is not supported by this `Iterator`.

Throws `IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has been called since the last call to `next()` or `previous()`.

Throws an `ClassCastException` if the class of `newElement` prevents it from being added.

Throws an `IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

Display 15.12 `ListIterator` Cursor Positions



The default initial cursor position is the leftmost one.

Pitfall

`next` Can Return a Reference

If `i` is an iterator, then `i.next()` returns an element of the collection that created `i`, but there are two senses of "return an element." (1) The invocation `i.next()` could return a copy of the element in the collection (for example, using a copy constructor or a `clone` method). (2) Alternatively,

`i.next()` could return a reference to the element in the collection. In case (1), modifying `i.next()` *will not* change the element in the collection. In case (2), modifying `i.next()` *will* change the element in the collection. The APIs for both the `Iterator` and `ListIterator` interfaces are vague on whether you should follow policy (1) or (2), but the iterators for the standard predefined collection classes, such as `Vector` and `HashSet`, return references. So, you can modify the elements in the collection by using mutator methods on `i.next()`. This is illustrated in Display 15.13. The comments we made about `i.next()` also apply to `i.previous()`.

The fact that `next` and `previous` return references to elements in the collection is not necessarily bad news. It means you must be careful, but it also means you can cycle through all the elements in the collection and perform some processing that might modify the elements. For example, if the elements in the collection are records of some sort, you can use mutator methods to update the records.

If you read the APIs for the `Iterator` and `ListIterator` interfaces, they say that a `ListIterator` can change the collection but, presumably, a plain old `Iterator` cannot. These API comments do not refer to whether or not a reference is returned by `i.next()`. They simply refer to the fact that the `ListIterator` interface has a `set` method while the `Iterator` interface does not have a `set` method. Do not confuse this with the point discussed in the previous paragraph.

Tip

Defining Your Own Iterator Classes

There really is little need to define your own `Iterator` or `ListIterator` classes. The most common and easiest way to define a collection class is to make it a derived class of one of the library collection classes, such as `Vector` or `HashSet`. When you do that, you automatically get the method `iterator()`, and if need be the method `listIterator()`, and that takes care of iterators. However, if you should need to define a collection class in some other way, then the best way to define your iterator class or classes is to define them as inner classes of your collection class.

Self-Test Exercises

18. Does a `HashSet` have a method to produce a `ListIterator`? Does `Vector` have a method to produce a `ListIterator`?

19. Suppose `i` is a `ListIterator` will an invocation of `i.next()` followed by `i.previous()` return the same element før each of the two invocations or might they return two different elements? What about `i.previous()` followed by `i.next()`?

Display 15.13 **An Iterator Returns a Reference *(Part 1 of 2)***

The class `Date` is defined in Display 4.11, but you can easily guess all you need to know about `Date` for this example.

```
1  import java.util.Vector;
2  import java.util.Iterator;

3  public class IteratorReferenceDemo
4  {
5      public static void main(String[] args)
6      {
7          Vector birthdays = new Vector();


8          birthdays.add(new Date(1, 1, 1990));
9          birthdays.add(new Date(2, 2, 1990));
10         birthdays.add(new Date(3, 3, 1990));


11         System.out.println("The vector contains:");


12         Iterator i = birthdays.iterator();
13         while (i.hasNext())
14             System.out.println(i.next());

15         i = birthdays.iterator();
16         Date d = null; //To keep the compiler happy.
17         System.out.println("Changing the references.");
18         while (i.hasNext())
19         {
20             d = (Date)i.next();
21             d.setDate(4, 1, 1990);
22         }


23         System.out.println("The vector now contains:");


24         i = birthdays.iterator();
25         while (i.hasNext())
26             System.out.println(i.next());


27         System.out.println("April fool!");
28     }
29 }
```

Display 15.13 **An Iterator Returns a Reference *(Part 2 of 2)***

Sample Dialogue

```
The vector contains:
Jan 1, 1990
Feb 2, 1990
Mar 3, 1990
Changing the references.
The vector now contains:
Apr 1, 1990
Apr 1, 1990
Apr 1, 1990
April fool!
```

Chapter Summary

- Vectors can be thought of as arrays that grow and shrink in length.
- The base type of a vector is always `Object`. Therefore, a vector may contain objects of any class but may not contain values of a primitive type.
- The main collection interfaces are `Collection`, `Set`, and `List`. The `Set` and `List` interfaces extend the `Collection` interface. The library classes that are standard to use and that implement these interfaces are `HashSet`, which implements the `Set` interface, and `Vector`, which implements the `List` interface.
- A `Set` does not allow repeated elements and does not order its elements. A `List` allows repeated elements and orders its elements.
- An iterator is something that allows you to examine and possibly modify the elements in a collection in some sequential order. Java formalizes this concept with the two interfaces `Iterator` and `ListIterator`.
- An `Iterator` (with only the required methods implemented) goes through the elements of the collection in only one direction, from the beginning to the end. A `ListIterator` can move through the collection list in both directions, forward and back. A `ListIterator` has a `set` method; the `Iterator` interface does not require a `set` method.

Answers to Self-Test Exercises

1. `v.add("Hello");`
2. `v.set(10, "Good-bye");`

3. No. The index for `set` must be greater than or equal to `0` and less than the size of the vector. Thus, you can replace any existing element, but you cannot place the element at any higher index. This is unlike an array. If an array is partially filled to index `10`, you can add an element at index `20`, as long as the array is that large. With a vector, you cannot add an element to a new unused position (except that methods other than `set` will let you add one in the first unused position).

4. The index for the two-argument version of `add` must be greater than or equal to `0` and less than *or equal to* the size of the vector. Thus, you can add an element at position `v.size()` of a vector `v` with the method `add` but you cannot add it at `v.size()` with the method `set`.

5. Yes. The vector can contain more than 50 elements. The number `50` used as an argument to the constructor merely gives the initial memory allocation for the vector. More memory is automatically allocated when it is needed.

6.
```
int index;
for (index = 0; index < v.size(); index++)
    System.out.println(v.get(index));
```

extra code on CD

7. The following code is in the file `StringSelectionSort.java` and a demonstration program is in the file `StringSelectionSortDemo.java` on the accompanying CD.

```
import java.util.Vector;

/**
 Class for sorting a vector of Strings lexicographically
*/
public class StringSelectionSort
{

  /**
   Sorts the vector a so that v.get(0), v.get(1),...
   v.get(a.size() – 1) are in lexicographic order.
   Assumes the vector contains only Strings.
  */
   public static void sort(Vector v)
   {
       int index, indexOfNextSmallest;
       for (index = 0; index < v.size() – 1; index++)
       {//Place the correct value in position index:
           indexOfNextSmallest =
                           indexOfSmallest(index, v);
           interchange(index,indexOfNextSmallest, v);
           //v.get(0), v.get(1),...,
            //v.get(index) are sorted. The rest of
           //the elements are in the remaining positions.
       }
   }
```

```
    /**
     Precondition: i and j are legal indices for the vector a.
     Postcondition: The values of v.get(i) and
      v.get(j) have been interchanged.
    */
    private static void interchange(
                            int i, int j, Vector v)
    {
        Object temp;
        temp = v.get(i);
        v.set(i, v.get(j));
        v.set(j, temp);
    }

    /**
     Returns the index of the lexicographically first value among
     v.get(startIndex), v.get(startIndex+1),...,
     v.get(v.size() – 1)
    */
    private static int indexOfSmallest(
                                  int startIndex, Vector v)
    {
        String min = (String)v.get(startIndex);
        int indexOfMin = startIndex;
        int index;
        for (index = startIndex + 1;
                            index < v.size(); index++)
            if (((String)(v.get(index))).compareTo(min) < 0)
            {
                min = (String)v.get(index);
                indexOfMin = index;
            }
        return indexOfMin;
    }
}
```

8. `Object`

9. No, you can only store objects in a vector. You cannot store values of any primitive type. (You could store the `int` value by embedding it an `Integer` object and storing the `Integer` object in the vector. So, the answer depends a little on how you interpret the question.)

10. The method invocation `v.size()` returns the number of elements in the vector `v`. The method invocation `v.capacity()` returns the number of elements for which the vector currently has memory allocated.

11. The new elements at indices `10` through `19` will have `null` for their values.

12. `/**`

```
     Returns the lexicographically first value among
     v.get(0), v.get(1),..., v.get(v.size() – 1)
     Precondition: v contains only Strings; v.size() > 0.
   */
   private static String smallest(Vector v)
   {
       Iterator vIterator = v.iterator();
       String min = (String)vIterator.next();
       String nextElement;
       while (vIterator.hasNext())
       {
           nextElement = (String)vIterator.next();
           if (nextElement.compareTo(min) < 0)
              min = nextElement;
       }

       return min;
   }
```

13. ```
    public static boolean inSome(Object target,
                                 Collection c1, Collection c2)
    {
        return (c1.contains(target) || c2.contains(target));
    }
    ```

14. ```
    public static Object getFirst(List aList)
    {
        if (aList.isEmpty())
            return null;
        else
           return aList.get(0);
    }
    ```

15. ```
    public static boolean noNull(Set s)
    {
        return (s.remove(null));
    }
    ```

16. No.

17. It would make more sense to make it a derived class of the `Vector` class. Then the elements are ordered. You can ensure against repeated elements by redefining all methods that add elements so that the methods check to see if the element is already in the class before entering it. A derived class of the `HashSet` class would automatically ensure that no element is repeated but it would seem to take a good deal of work to maintain the elements in order.

18. A `HashSet` does not. A `Vector` does.

19. The answer to both questions is the same: They will return the same element.
    ```

PROGRAMMING PROJECTS

1. Redo Programming Project 6 in Chapter 6, but this time do it for a vector of strings to be sorted into lexicographic order.

2. Define a class called `StringVector` which is to be a like the `Vector` class but it enforces the constraint that all elements in the collection are objects of the class `String`. Make your `StringVector` class a derived class of the `Vector` class. Also write a suitable test program.
3. Define a class called `IntegerSet` that is just like the class `HashSet` except that it stores values of type `int`. Objects of your class `IntegerSet` will have an instance variable of type `HashSet` that it uses to hold values of the wrapper class `Integer`. The instance variable of type `HashSet` is really the heart and soul of your `IntegerSet` class. All your methods do little more than invoke the methods of this `HashSet`, but they need to convert `int` values to their corresponding `Integer` object and vice versa. Also write a suitable test program.

For additional online Programming Projects, click the CodeMate icons below.

15.4 

15.5