Chapter 19

Collection Classes

Collections are classes designed for holding groups of objects. Almost all nontrivial programs need to maintain one or more collections of objects. Although the Java library provides only a few different forms of collection, the features provided by these classes are very general, making them applicable to a wide variety of problems. In this chapter we will first describe some of the basic concepts common to the collection classes, then summarize the basic collections provided by Java, and finally describe a few container types that are not provided by the standard library, but which can be easily constructed by the programmer.

19.1 Elements Types and primitive value Wrappers

With the exception of the `array` data type, all the collections provided by the Java library maintain their values in variables of type `Object`. There are two important consequences of this feature:

- Since primitive types, such as integers, booleans, characters and floating point values, are not subclasses of `Object`, they cannot be directly stored in these collections.
- When values are removed from the collection, they must be cast back to their original type.

One way to circumvent the first restriction is through the use of *wrapper classes*. A wrapper class maintains a primitive data value, but is itself an object, and can thus be stored in a container. Methods are typically provided to both to construct an instance of the wrapper class from a primitive value, and to recover the original value from the wrapper. The following, for example, stores two integers into instances of class `Integer`, then recovers the original values so that an arithmetic operation can be performed:

	Integer **wrapper**
new Integer(int value)	build Integer from int
Integer(String value)	parse integer in String
intValue()	value of Integer as int
toString()	return decimal string representation of Integer
toBinaryString()	return binary representation
toOctalString()	return octal representation
toHexString()	return hex representation
	Character **wrapper**
new Character(char value)	convert char to Character
charValue()	return char value of Character
isLetter()	determine if character is letter
isDigit()	true if character is digit
	Boolean **wrapper**
new Boolean (boolean value)	convert boolean to Boolean
booleanValue ()	retrieve boolean value from Boolean
toString()	generate string representation of boolean
	Double **wrapper**
new Double(double)	convert double to Double
new Double(String)	construct Double from string
doubleValue()	return double value of Double

Figure 19.1: Wrapper Classes and Selected Behaviors

```
Integer a = new Integer(12);
Integer b = new Integer(3);
    // must recover the int values to do arithmetic
int c = a.intValue() * b.intValue();
```

In addition, many wrapper classes provide other useful functionality, such as the ability to parse string values. A common way to convert a string containing an integer value literal into an int, for example, is to use an Integer as a middle step:

```
String text = "123"; // example string value
Integer val = new Integer(text); // first convert to Integer
int ival = val.intValue(); // then convert Integer to int
```

Figure 19.1 summarizes the most common wrapper classes and a few of their more useful behaviors.

19.2 Enumerators

All collections can be envisioned as a linear sequence of elements, however the particular means used to access each element differs from one collection type to another. For example, a vector is indexed using an integer position key, while a hash table can use any type of object as a key. It is frequently desirable to abstract away these differences, and access the elements of the collection in sequence without regard to the technique used to obtain the underlying values. This facility is provided by the Enumeration interface, and its various implementations. The Enumeration interface specifies two methods:

hasMoreElements()	A boolean value that indicates whether or not there are any more elements to be enumerated
nextElement()	Retrieves the next element in the enumeration

These two operations are used together to form a loop. The following, for example, shows how all the elements of a hash table could be printed:

```
for (Enumeration e = htab.elements(); e.hasMoreElements(); ) {
    System.out.println (e.nextElement());
    }
```

The methods hasMoreElements() and nextElement should always be invoked in tandem. That is, nextElement should never be invoked unless hasMoreElements has first determined that there is another element, and nextElement should never be invoked twice without an intervening call on hasMoreElements. If it is necessary to refer more than once to the value returned by nextElement, the result of the nextElement call should be assigned to a local variable:

```
for (Enumeration e = htab.elements(); e.hasMoreElements(); ) {
    Object value = e.nextElement();
    if (value.equals(Test))
        System.out.println ("found object " + value);
    }
```

With the exception of the array, all the collections provided by the Java library provide a function that generates an enumeration. In addition, several other classes that are not necessarily collections also support the enumeration protocol. For example, a StringTokenizer is used to extract words from a string value. The individual words are then accessed using enumeration methods. In Section 19.6.1 we will describe how the programmer can create a new type of enumeration.

19.3 The Array

The most basic collection form in Java is the array. As we have noted in earlier chapters, the creation and manipulation of an array value is different in Java than in many other programming languages. The Java language makes a separation between (a) declaring a variable of array type, (b) defining the size of the array and allocating space, and (c) assigning values to the array. In many other languages the first and second tasks are merged together. Merging these concepts makes it difficult to, for example, write functions that will operate on arrays of any size. These problems are largely eliminated in Java's approach to arrays.

An array variable is declared by indicating the type of elements the array will contain, and a pair of square brackets that indicate an array is being formed. The following, for example, is from the solitaire game case study examined in Chapter 9.

```
static public CardPile allPiles [ ];
```

The brackets can be written either after the variable name, or after the type. The following, for example, is an equivalent declaration:

```
static public CardPile [ ] allPiles;
```

Note that the array is the only collection in the Java library that requires the programmer to specify the type of elements that will be held by the container. An important consequence of this is that the array is the *only* collection that can maintain non-object types, such as integers, booleans, or characters. Other containers hold their values as instances of class Object, and can therefore only hold primitive values if they are surrounded by wrapper classes (Integer, Double, and so on).

An array *value* is created, as are all values, using the new operator. It is only when the array value is created that the size of the array is specified.

```
allPiles = new CardPile [ 13 ]; // create array of 13 card piles
```

Arrays can also be created as an initialization expression in the original declaration. The initialization expression provides the values for the array, as well as indicating the number of elements.

```
int primes [ ] = {2, 3, 5, 7. 11, 13, 17, 19};
```

Elements of an array are accessed using the subscript operator. This is used both to retrieve the current value held at a given position, and to assign a position a new value:

```
primes[3] = primes[2] + 2;
```

Legal index values range from zero to one less than the number of elements held by the array. An attempt to access a value with an out of range index value results in an IndexOutOfBoundsException being thrown.

The integer field length describes the number of elements held by an array. The following loop shows this value being used to form a loop to print the elements in an array:

```
for (int i = 0; i < primes.length; i++)
    System.out.println("prime " + i + " is " + primes[i]);
```

Arrays are also *cloneable* (see Section 11.3.1). An array can be copied, and assigned to another array value. The clone operation creates a shallow copy.

```
int numbers [ ];
numbers = primes.clone(); // creates a new array of numbers
```

19.4 The Vector collection

The Vector data abstraction is similar to an array of objects, however it provides a number of high level operations not supported by the array abstraction. Most importantly, a vector is *expandable*, meaning it grows as necessary as new elements are added to the collection. Thus, the programmer need not know the eventual collection size at the time the vector is created. To use this collection the Vector class must be imported from java.util.Vector. A new vector is created using the new operator.

```
import java.util.Vector;
...
Vector numbers = new Vector ();
```

Because objects held by a Vector must be subclasses of Object, a vector can not be used to hold primitive values, such as integers or floats. However, wrapper classes can be used to convert such values into objects.

Table 19.1 summarizes the most useful operations provided by the Vector data abstraction. The class as been carefully designed so that it can be used in a variety of different ways. The following sections describe some of the more common uses.

19.4.1 Using a Vector as an array

Unlike an array, a vector is not created with a fixed size. A vector can be given a specific size either by adding the appropriate number of elements (using the addElement operation)

	Size Determination
size()	returns number of elements in collection
isEmpty()	returns true if collection is empty
capacity()	return current capacity of vector
setSize()	set size of vector, truncating or expanding as necessary
	Element Access
contains()	determines whether a value is in this vector
firstElement()	returns first element of collection
lastElement()	returns last element of collection
elementAt(index)	returns element stored at given position
	Insertion and Modification
addElement(value)	add new value to end of collection
setElementAt(value, index)	change value at position
insertElementAt(value, index)	insert value at given index
	Removal
removeElementAt(index)	remove element at index, reducing size of vector
removeElement(value)	remove all instances of value
removeAllElements()	delete all values from vector
	Search
indexOf(value)	return index of first occurrence
lastIndexOf(value)	return index of last occurrence
	Miscellaneous
clone()	return shallow copy of vector
toString()	return string representation of vector

Table 19.1: Operations provided by the Vector data type

or by using setSize. In the latter case a null value will be stored in any index locations that have not previously been assigned a value:

```
Vector aVec = new Vector (); // create a new vector
aVec.setSize (20); // allocate 20 locations, initially undefined
```

Once sized, the value stored at any position can be accessed using the method elementAt, while positions can be modified using setElementAt. Note that in the latter, the index of the position being modified is the second parameter, while the value to be assigned to the location is the first parameter. The following illustrates how these functions could be used to swap the first and final elements of a vector:

```
Object first = aVec.elementAt(0); // store first position
aVec.setElementAt(aVec.lastElement(), 0); // store last in location 0
aVec.setElementAt(first, aVec.size()-1); // store first at end
```

19.4.2 Using a Vector as a stack

A *stack* is a data structure that allows elements to be inserted and removed at one end, and always removes the last element inserted. A stack of papers on a desk is a good intuitive picture of the stack abstraction.

Although the Java standard library includes a Stack data abstraction (see Section 19.5), it is easy to see how a Vector can be used as a stack. The characteristic operations of a stack are to insert or remove an item from the top of the stack, peek at (but do not remove) the topmost element, test the stack for emptiness. The following shows how each of these can be performed using operations provided by the vector class:

push an value on the stack	aVec.addElement (value)
peek at the topmost element of the stack	aVec.lastElement()
remove the topmost element of the stack	aVec.removeElementAt(aVec.size() - 1)

As we will note in Section 19.5, the Stack abstraction is slightly more robust, since it will throw more meaningful error indications if an attempt is made to remove an element from an empty stack.

19.4.3 Using a Vector as a queue

A *queue* is a data structure that allows elements to be inserted at one end, and removed from the other. In this fashion, the element removed will be the *first* element that was inserted. Thinking about a line of people waiting to enter a theater provides a good intuition.

In a manner analogous to the way that the vector can be used as a stack, the vector operations can also be used to simulate a queue:

push an value on the queue	aVec.addElement (value)
peek at the first element in the queue	aVec.firstElement()
remove the first element of the queue	aVec.removeElementAt(0)

There is one important difference between this abstraction and the earlier simulation of the stack. In the stack abstraction, all the operations could be performed in constant time, independent of the number of elements being held by the stack.[1] Removing the first element from the queue, on the other hand *always* results in all elements being moved, and is therefore always requires time proportional to the number of elements in the collection.

19.4.4 Using a Vector as a set

A *set* is usually envisioned as an unordered collection of values. Characteristic operations on a set include testing to see if a value is being held in the set, and adding or removing values from the set. The following shows how these can be implemented using the operations provided by the Vector class:

see if value is held in set	aVec.contains(value)
add new element to set	aVec.addElement(value)
remove element from set	aVec.removeElement(value)

The equals method is used to perform comparisons. Comparisons are necessary to determine whether or not a value is held in the collection, and whether a value is the element the user wishes to delete. For user defined data values the equals method can be overridden to provide whatever meaning is appropriate (see Section 11.4).

Operations such as union and intersection of sets can be easily implemented using a loop. The following, for example, places in setThree the union of the values from setOne and setTwo.

```
Vector setThree = new Vector();
setThree = setOne.clone(); // first copy all of set one
    // then add elements from set two not already in set one
for (Enumeration e = setTwo.elements(); e.hasMoreElements(); ) {
    Object value = e.nextElement();
    if (! setThree.contains(value))
        setThree.addElement(value);
    }
```

For sets consisting of positive integer values, the BitSet class (Section 19.6 is often a more efficient alternative.

[1] The assertion concerning constant time operation of stack operations is true with one small caveat. An insertion can, in rare occasions, result in a reallocation of the underlying vector buffer, and thus require time proportional to the number of elements in the vector. Thus is usually, however, a rare occurrence.

19.4.5 Using a Vector as a list

Characteristic operations of a *list* data abstraction are the abilities to insert or remove elements at any location, and the ability to find the location of any value.[2]

first element	aVec.firstElement()
last element	aVec.lastElement()
insert to front of list	aVec.insertElementAt(value, 0)
insert to end of list	aVec.addElement(value)
see if value is in collection	aVec.contains(value)
remove first element	aVec.removeElementAt(0)
remove last element	aVec.removeElementAt(aVec.size() - 1)
find location of element	aVec.indexOf(value)
remove value from middle	aVec.removeElementAt(index)

Again, this use of the Vector abstraction to simulate a list differs from the classical description of the *list* abstraction in the algorithmic execution time of certain operations. In particular, the insertion or removal from the front of the collection may result in the entire set of values being moved, thereby requiring time proportional to the size of the collection. This is only a critical concern when the size of the collections is large or when this is a frequent operation. A more direct implementation of a list is described in Section 19.9.

19.5 The Stack collection

Section 19.4.2 described how a stack could be simulated using a Vector. However, the Java standard library also provides the Stack as a data value. The names of the methods used to operate on this data type are slightly different from the Vector operations described in Section 19.4.2, and the structure is slightly more robust. Stack operations can be described as follows:

create a new stack	Stack aStack = new Stack()
push an value on to the stack	aStack.push (value)
peek at the topmost element of the stack	aStack.peek()
remove the topmost element of the stack	aStack.pop()
number of elements in the collection	aStack.size()
test for empty stack	aStack.empty()
position of element in stack	aStack.search(value)

The pop operation both removes and returns the topmost element of the stack. Both the pop and the peek operations will throw an EmptyStackException if they are applied to an empty stack.

[2]The list referred to here is the traditional data abstraction known by that name. The Java library unfortunately uses the class name List to refer to a graphical component that allows the user to select a value from a series of string items.

The search method returns the index of the given element starting from the top of the stack; that is, if the element is found at the top of the stack search will return 0, if found one element down in the stack search will return 1, and so on. Because the Stack data structure is built using inheritance from the Vector class it is also possible to access the values of the stack using their index. However, the positions returned by the search operation do not correspond to the index position. To discover the index position of a value the Vector operation indexOf can be used.

In Chapter 10 we described some of the advantages and disadvantages of creating the stack using inheritance from class Vector.

19.6 The BitSet collection

A BitSet is abstractly a set of positive integer values. The BitSet class differs from the other collection classes in that it can only be used to hold integer values. Like an array or a Vector, each element is given an index position. However, the only operations that can be performed on each element are to set, test or clear the value. A BitSet is a compact way to encode either a collection of positive integer values, or a collection of boolean values (for example, on/off settings).

A create a BitSet the user can specify the number of positions the set will represent. However, like the Vector, the bit set is extensible, and will be enlarged automatically if a position outside the range is accessed.

```
    // create a BitSet that initially contains 75 elements
BitSet bset = new BitSet(75);
```

The following table summarizes the operations used to set or test an individual position in the bit set:

set a bit position	bset.set (index)
test a bit position	bset.get (index)
clear a bit position	bset.clear (index)

The get method returns a boolean value, which is true if the given bit is set, and false otherwise. Each of these operations will throw an IndexOutOfBoundsException if the index value is smaller than zero.

A BitSet can be combined with another BitSet in a variety of ways:

Form bitwise union with argument set	bset.or(setTwo)
Form bitwise intersection with argument set	bset.and(setTwo)
Form bitwise symmetric difference with argument set	bset.xor(setTwo)

The method toString returns the string representation of the collection. This consists of a comma-separated list of the indices of the bits in the collection that have been set.

19.6.1 Example Program: Prime Sieve

A program that will generate a list of prime numbers using the sieve of Erasthones can be used to illustrate the manipulation of a bit set. The constructor for the class Sieve (Figure 19.2) takes an integer argument n, and creates a bit set of n positions. These are initially all set to one, using the member function set. The sieve algorithm then walks through the list, using get to find the next set value. A loop then walks through the remainder of the collection, throwing out (via the clear() member function) values which are multiples of the earlier value. When we are finished, any value not crossed out must be prime.

The remaining two functions illustrate how a new enumeration can be created. The value index will maintain the current "position" in the list, which will change as values are enumerated. The function hasMoreElements loops until a prime value is found, or until the size of the bit set is exceeded. The first results in a true value, the latter a false one. The method nextElement simply makes an object out of the integer value. A small test method is also included in the class, to illustrate how this class could be used.

19.7 The Dictionary interface and the Hashtable collection

A *dictionary* is an indexed collection, similar to an array or a Vector. However, unlike an array, the index values need not be integer. Instead, any object type can be used as in index (called a *key*), and any object value can be stored as the element selected by the key. To place a new value into the collection the user provides both the key and value. To access an element in the collection the user provides a key, and the associated value is returned.

In the Java library the class Dictionary is an abstract class that defines the behavior of the *dictionary* abstraction, but does not provide an implementation. This interface can be described as follows:

retrieve value associated with given key	dict.get(key)
place value into collection with given key	dict.put(key, value)
remove value from collection	dict.remove(key)
see if collection is empty	dict.isEmpty()
return number of elements in collection	dict.size()
return enumeration for collection values	dict.elements()
return enumeration of collection keys	dict.keys()

The get method will return null if the given value is not found in the collection. Otherwise, the value is returned as an Object, which must then be cast into the appropriate type. The remove method returns the value of the association being deleted, again returning null if the key is not a legal index element. There are two enumeration generating methods, one to return an enumeration of keys, and one to return an enumeration of values.

The class Hashtable provides an implementation of the Dictionary operations. A hash table can be envisioned as an array of collections, called *buckets*. To add an element to

```
import java.util.*;

class Sieve implements Enumeration {
    private BitSet primes;
    private int index = 2;

    public Sieve (int n) {
        primes = new BitSet(n);
            // first set all the bits
        for (int i = 1; i < n; i++)
            primes.set(i);
            // then erase all the non-primes
        for (int i = 2; i * i < n; i++)
            if (primes.get(i))
                for (int j = i + i; j <= n; j += i)
                    primes.clear(j);
        }

    public boolean hasMoreElements () {
        index++;
        int n = primes.size();
        while (! primes.get(index))
            if (++index > n)
                return false;
        return true;
        }

    public Object nextElement() { return new Integer(index); }

        // test program for prime sieve algorithm
    public static void main (String [ ] args) {
        Sieve p = new Sieve(100);
        while (p.hasMoreElements())
            System.out.println(p.nextElement());
        }
}
```

Figure 19.2: Prime Sieve program

the collection, an integer value, called the *hash value*, is first computed for the given key. The method hashCode is used for this purpose. This method is defined in class Object, and is therefore common to all object values. It is overridden in various classes to provide alternative algorithms. Using this integer, one of the buckets is selected and the key/value pair inserted into the corresponding collection.

In addition to the methods matching the Dictionary specification, the hash table provides the method clear(), which removes all values from the container, contains(value), which determines whether an element is contained in the collection, and containsKey(key), which tests to see if a given key is in the collection.

The default implementation of the hashCode method, in class Object, should be applicable in almost all situations, just as the default implementation of equals is usually adequate. If a data type that is going to be used as a hash table key overrides the equals method, it is a good idea to also override hashCode, so that two objects that test equal to each other will also have the same hash value.

19.7.1 Example Program: A Concordance

A concordance is a listing of words from a printed text, each word being followed by the lines on which the word appears. A class that will create a concordance will illustrate how the Dictionary data type is used, as well as how different collection classes can be combined with each other.

In the program shown in Figure 19.3, the primary data structure is a dictionary, implemented using the Hashtable class. The keys for this dictionary will be the individual words in the input text. The value associated with each key will be a set of integer values, representing the line numbers on which the word appears. A Vector will be used to represent the set, using the techniques described in Section 19.4.4.

The method readLines reads the input line by line, maintaining a counter to indicate the line number. The method readLine, provided by the class DataInputStream, returns a null value when end of input is encountered, at which time the method returns. (This method is also the potential source for the IOException, which can be thrown if an error occurs during the read operation. In our program we simply pass this exception back to the caller). Otherwise, the text is converted to lower case, using the method toLowerCase provided by the String class, then a StringTokenizer is created to split the text into individual words. A StringTokenizer is a form of Enumeration, and so an enumeration loop is used to enter each word into the concordance.

The private method enterWord is used to place each new word in the concordance. First, the value associated with the key (the word) is determined. Here the program handles the first of two exceptional conditions that might arise. If this is the first time the word has been seen, there will be no entry in the dictionary, and so result of calling get will be a null value. In this case a new and empty Vector is created, and inserted into the dictionary using the word as key. Using the Vector in the fashion of a set, the method contains is invoked to determine if the line has already been placed in the collection. (This is the

```
import java.util.*;
import java.io.*;

class Concordance {
    private Dictionary dict = new Hashtable();

    public void readLines (DataInputStream input) throws IOException {
        String delims = " \t\n.,!?;:";
        for (int line = 1; true; line++ ) {
            String text = input.readLine();
            if (text == null) return;
            text = text.toLowerCase();
            Enumeration e = new StringTokenizer(text, delims);
            while (e.hasMoreElements())
                enterWord ((String) e.nextElement(), new Integer(line));
            }
        }

    public void generateOutput (PrintStream output) {
        Enumeration e = dict.keys();
        while (e.hasMoreElements() ) {
            String word = (String) e.nextElement();
            Vector set = (Vector) dict.get(word);
            output.print (word + ": ");
            Enumeration f = set.elements();
            while (f.hasMoreElements())
                output.print (f.nextElement() + " ");
            output.println (" ");
            }
        }

    private void enterWord (String word, Integer line) {
        Vector set = (Vector) dict.get(word);
        if (set == null) {    // word not in collection
            set = new Vector(); // make new set
            dict.put (word, set);
            }
        if (! set.contains(line)) set.addElement(line);
        }
}
```

Figure 19.3: The class Concordance

second exceptional condition, which will occur if the same word appears two or more times on one line.) If not, the line is then added to the list.

Finally, once all the input has been processed, the method generateOutput is used to create the printed report. This method uses a doubly-nested enumeration loop. The first loop enumerates the keys of the Dictionary, generated by the keys method. The value associated with each key is a set, represented by a Vector. A second loop, using the enumerator produced by the elements method, then prints the values held by the vector.

An easy way to test the program is to use the system resources System.in and System.out as the input and output containers, as in the following:

```
static public void main (String [ ] args) {
    Concordance c = new Concordance();
    try {
        c.readLines(new DataInputStream(System.in));
    } catch (IOException e) { return; }
    c.generateOutput (System.out);
}
```

19.7.2 Properties

The Java run-time system maintains a special type of hash table, termed the *properties list.* The class Properties, a subclass of Hashtable, holds a collection of string key/value pairs. These represent values the describe the current executing environment, such as the user name, operating system name, home directory, and so on. The following program can be used to see the range of properties available to a running Java program:

```
public static void main (String [ ] args) {
    Dictionary props = System.getProperties();
    Enumeration e = props.keys();
    while (e.hasMoreElements()) {
        Object key = e.nextElement();
        Object value = props.get(key);
        System.out.println("property " + key + " value " + value);
        }

}
```

19.8 Why are there no ordered collections?

If one considers the "classic" data abstractions found in most data structures textbooks, a notable omission from the Java library are data structures that maintain values in sequence. Examples of such abstractions are ordered lists, ordered vectors, or binary trees. Indeed, there is not even any mechanism provided in the Java library to sort a vector of values. Rather than being caused by oversight, this omission reflects some fundamental properties of the Java language.

All of the Java collections maintain their values in variables of type Object. The class Object does not define any ordering relation. Indeed, the only elements that can be compared using the < operator are the primitive numeric types (integer, long, double, and so on). One could imagine defining in class Object a method lessThan(Object), similar to the method equals(Object). However, while there is a clear default interpretation for the equality operator (namely, object identity), it is difficult to imagine a similar meaning for the relational operator that would be applicable to all objects. Certainly it could not provide a total ordering on all objects. What, for example, would be the result of comparing the String `"abc"` and the integer 37? In short, ordered collections are not found in the Java library because there is no obvious general mechanism to define what it means to order two values.

One could imagine that an alternative to placing the method lessThan in class Object would be to create an Ordered interface, such as the following:

```
interface Ordered {
    public boolean compare (Ordered arg);
    }
```

One could then create a collection in which all values need to implement the Ordered interface, rather than simply being Object. However, there are two major objections to this technique. The first is that since the argument is only known to be an object that implements the Ordered interface, one must still decide how to compare objects of different types (a Triangle and an Orange, for example). The second problem is that by restricting the type of objects the collection can maintain to only those values that implement the Ordered relation, one severely limits the utility of the classes.

Another possibility is to imagine an interface for an object that is used to create comparisons. That is, the object takes both values as arguments, and returns their ordering. Such an interface could be written as follows:

```
interface ComparisonObject {
    public boolean Compare (Object one, Object two);
}
```

To manipulate an ordered collection, one would then create an implementation of this interface for the desired elements. The following, for example, would be a comparison class for Integer objects:

```
class IntegerComparison implements ComparisonObject {
    public boolean Compare (Object one, Object two) {
        if ((one instanceof Integer) && (two instanceof Integer)) {
            Integer ione = (Integer) one;
            Integer itwo = (Integer) two;
            return ione.intValue() < itwo.intValue();
        }
        return false;
    }
}
```

The following program illustrates how such an object could be used. The static method sort is an implementation of the insertion sort algorithm. The main method creates a vector of integer values, then creates a comparison object to be passed as argument to the sort algorithm. The sorting algorithm orders the elements in place, using the comparison object to determine the relative placement of values.

```
class VectorSort {
    public static void sort (Vector v, ComparisonObject test) {
            // order a vector using insertion sort
        int n = v.size();
        for (int top = 1; top < n; top++) {
            for (int j = top-1; j >= 0 &&
                test.Compare(v.elementAt(j+1), v.elementAt(j)); j--) {
                    // swap the elements
                Object temp = v.elementAt(j+1);
                v.setElementAt(v.elementAt(j), j+1);
                v.setElementAt(temp, j);
            }
        }

    }

    public static void main (String [ ] args) {
        Vector v = new Vector();
        Random r = new Random();
        for (int i = 0; i < 10; i++)
            v.addElement(new Integer(r.nextInt()));
```

```
            // sort the vector
        sort (v, new IntegerComparison());

        for (Enumeration e = v.elements(); e.hasMoreElements(); )
            System.out.println(e.nextElement());
        }
}
```

19.9 Building your Own Containers

Although the containers in the Java library are flexible, they nevertheless cannot handle all situations in which a collection class is needed. It is therefore sometimes necessary to create new collection classes. We will illustrate how this can be done by creating a class that implements the idea of a linked list. The major advantage of the linked list over a vector is that insertions or removals to the beginning or the middle of a linked list can be performed very rapidly (technically, in constant time). In the vector these operations require the movement of all the elements in the collection, and can therefore be much more costly if the collection is large.

The LinkedList class abstraction is shown in Figure 19.4.[3] The actual values are stored in instances of class Link, which is a nested inner class. In addition to a value, links maintain references to the previous and next element in the list. A private internal value firstLink will reference the first link. A link with an empty value is used to mark the end of the list. The private internal value lastLink points to this value.

Values can be inserted either to the front or the back of the list. An enumeration value can also be used to insert new elements into the middle of a list. The value is inserted immediately before the element referred to by the enumeration. All three methods make use of a common insertion routine provided by the inner class Link. This method is also used to maintain the count of the number of elements in the list. The classes Link and ListEnumeration are shown in Figure 19.5.

The class ListEnumeration implements the Enumeration protocol, and is used for iterating over list elements. Note that the Enumeration protocol assumes that the methods hasMoreElements and nextElement will work in tandem, and does not specify which of the two will actually advance the internal reference to the next element. Implementations of the Enumeration protocol use a variety of different schemes. This is why, for example, one should never invoke nextElement twice without an intervening call on hasMoreElements. In the LinkedList class, however, we assume that having examined the current value (the value

[3]It would have been preferable to call this class List. However, has we noted in an earlier footnote, the Java library already has a List class, that implements a graphical component used for selecting one string item out of many alternatives.

```
class LinkedList {
    private Link firstLink;
    private Link lastLink;
    private int  count = 0;

    public LinkedList ()
        { firstLink = lastLink = new Link(null, null, null); }

    private class Link { ... }

    private class ListEnumeration implements Enumeration { ... }

    public boolean isEmpty () { return firstLink == lastLink; }

    public int size () { return count; }

    public Object firstElement () { return firstLink.value; }

    public Object lastElement () { return lastLink.prev.value; }

    public void addFront (Object newValue) { firstLink.insert(newValue); }

    public void addBack (Object newValue) { lastLink.insert(newValue); }

    public void addElement (Enumeration e, Object newValue) {
        ListEnumeration le = (ListEnumeration) e;
        le.link.insert (newValue);
    }

    public Object removeFront () { return firstLink.remove(); }

    public Object removeBack () { return lastLink.prev.remove(); }

    public Object removeElement (Enumeration e) {
        ListEnumeration le = (ListEnumeration) e;
        return le.link.remove ();
    }

    public Enumeration elements () { return new ListEnumeration(); }
}
```

Figure 19.4: The Linked List Class

```
class LinkedList {
    private class Link {
        public Object value;
        public Link next;
        public Link prev;

        public Link (Object v, Link n, Link p)
            { value = v; next = n; prev = p; }

        public void insert (Object newValue) {
            Link newNode = new Link (newValue, this, prev);
            count++;
            if (prev == null) firstLink = newNode;
            else prev.next = newNode;
            prev = newNode;
        }

        public Object remove () {
            if (next == null)
                return null; // cannot remove last element
            count--;
            next.prev = prev;
            if (prev == null) firstLink = next;
            else prev.next = next;
            return value;
        }
    }

    private class ListEnumeration implements Enumeration {
        public Link link = null;

        public boolean hasMoreElements () {
            if (link == null) link = firstLink;
            else link = link.next;
            return link.next != null; }

        public Object nextElement () { return link.value; }
    }
    ...
}
```

Figure 19.5: The Inner Classes in LinkedList

yielded by nextElement) the programmer may wish to either insert a new value or remove the current value. Thus, in this case the task of advancing to the next value is given to the method hasMoreElements.

Chapter Summary

In this chapter we have described the classes in the Java library that are used to hold collections of values. The simplest collection is the array. An array is a linear, indexed homogeneous collection. A difficulty with the array is that the size of an array is fixed at the time the array is created. The Vector class overcomes this restriction, growing as necessary as new values are added to the collection. Vectors can be used to represent sets, queues, and lists of values. The Stack datatype is a specialization of the vector used when values are added and removed from the collection is a strict first-in, first-out fashion. A BitSet is a set of positive integer values. A Dictionary is an interface that describes a collection of key and value pairs. The HashTable is one possible implementation of the Dictionary interface.

The lack of any ordered collections is a reflection of the problem that there is, in general, no way to construct an ordering among all Java values.

The chapter concludes by showing how new collection classes can be created, using as an example a Linked List container.

Study Questions

1. What are collection classes used for?
2. Because the standard library collection classes maintain their values as an Object, what must be done to a value when it is removed from a collection?
3. What is a wrapper class?
4. What is an enumerator?
5. What is the protocol for the class Enumerator? How are these methods combined to form a loop?
6. How is the Java array different from arrays in other languages?
7. What does it mean to say that the Vector data type is expandable?
8. How does the use of the Stack data type differ from the use of a Vector as a stack?
9. What concept does the class BitSet represent?
10. What is the relationship between the classes Dictionary and Hashtable?
11. Why are there no ordered collections in the Java library?

Exercises

1. Assume two sets are implemented using vectors, as described in Section 19.4.4. Write a loop that will place the intersection of the two sets into a third set.

2. Assume two sets are implemented using vectors, as described in Section 19.4.4. Write a loop that will place the symmetric difference of the two sets into a third set. (The symmetric difference is the set of elements that are in one or the other set, but not both).

3. Add the following methods to the LinkedList class described in Section 19.9:

setElement(Enumeration e, Object v)	change value at given location
includes(Object v)	test whether value is in collection
find(Object v)	return enumeration if value is in collection, or null

4. Modify the LinkedList class of Section 19.9 so that linked lists support the cloneable interface. (The cloneable interface is described in Section 11.3.1).

5. Write an OrderedList class. This class will be like a linked list, but will maintain a comparison object, as described in Section 19.8. Using this object, elements will be placed in sequence as they are inserted into the container.