Introduction to Data Structures

David Hughes

CONTENTS

1 ARRAYS

CHAPTER OBJECTIVES

- Explain the difference between one-dimensional, two-dimensional and higher dimensional arrays.
- Describe the two standard traversal patterns for two-dimensional arrays—row-major and column-major.
- Explain how arrays may be parameters to and results of methods.
- Choose and apply arrays for representing information when appropriate.
- Choose and apply the appropriate technique for storing information in arrays—right-sized and variable-sized.
- Apply appropriate array traversal in solving a problem.
- Explain the derivations of the mapping functions for contiguous array storage.
- Differentiate between contiguous allocation and array-of-arrays allocation of higher dimensional arrays.

It is common in programming to need to represent collections of things such as data measurements in an experiment or students in a course. If this collection cannot be processed in sequential order from first to last—as we do when processing information in a file—we will need a way to store the entire collection in memory. For this we need what are called collections or arrays.

We have seen other kinds of collections: `Pictures` and `Sounds`. Arrays share many of the properties of other collections having elements (like a `Picture` has `Pixels`) that can be accessed and processed. Arrays also have dimensions, such as a `Sound` is a one-dimensional collection of `Samples` and a `Picture` is a two-dimensional collection of `Pixels`.

Unlike other collections, arrays are built-in to the programming language rather than being defined as a class. They are thus more general and it is possible to use an array to represent a collection of any kind of value.

1.1 USING ARRAYS

In Java an array is a collection of items (values, objects) all of the same type, stored under a single name. The individual items are called **elements** and the type is known as the **element type**. Arrays are similar to other collections such as `Pictures` and `Sounds` and many of the concepts apply to arrays as well as `Collection` objects. The primary difference between arrays and `Collection` types is that arrays are not defined as a class but rather are built-in in the language.

An array is declared using the following notation:

```
type[]… name;
```

where *`type`* is any type including primitive types such as `int` and class types such as `Student`. This specifies the element type of the array. *`name`* is the variable identifier used to reference the array.

Arrays, like collections, may be one-dimensional, two-dimensional or have more dimensions. The number of pairs of brackets following the type indicates the number of dimensions. Such a declaration can be used anywhere a variable declaration can be used including instance variable declaration, parameter declaration, local variable declaration and as the return type for a function method. For example:

```
int[] a;
```

declares a one dimensional array of `int`s called `a`.

Arrays are classified as reference types (like object references). This means that the array variable is a reference to the actual array, just as an object variable is a reference to the actual object. Like objects, arrays must be created before they can be used. An array creation expression has the form:

```
name = new type [ expression ]…
```

Like object creation, array creation is initiated by the keyword `new`. This is followed by the element *`type`* and then a number of *`expression`* each in brackets. The number of expressions specifies the number of dimensions and each expression is the **length**—the number of elements—of the dimension. The result is a reference to a newly created array which can then be assigned to an array reference variable (*`name`*). The notation is deceptively similar to object creation. In object creation parentheses are used and the expressions in the parentheses are parameters to the constructor. In array creation, brackets are used and the expression is the size of the array. However both create a new item (array or object) and produce a reference to it. In this section we will consider one-dimensional arrays for simplicity.

The array creation:

```
a = new int[10];
```

creates a new one dimensional array of `int`s and stores its reference in `a`. The memory model is shown in Figure 1.1. Storage capable of storing 10 `int` values is allocated and then its reference is stored in `a`. The value of each individual element (`int`s) is not specified.

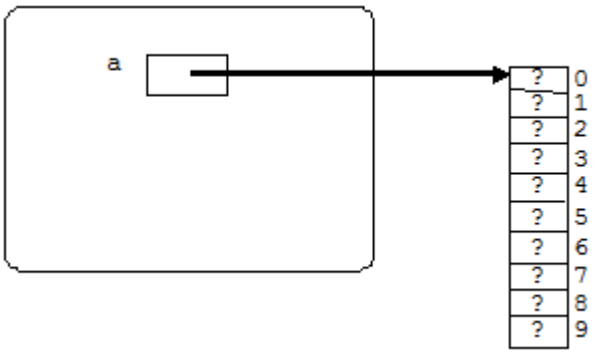


Figure 1.1 Array Memory Model

Assignment compatibility for arrays requires that the array being assigned (right-hand side) have the same number of dimensions and the same element type as the variable (left-hand side). Note that the

length of the dimensions of the array is irrelevant. This means that an array variable may reference different arrays of various lengths over the execution of the program. However, the number of dimensions and the element type are fixed by the declaration. This makes arrays in Java a bit more flexible than in many languages where even the number of elements is fixed by the declaration. Like for object reference variables it is the reference that is assigned—no copy of the array is produced.

There are few operations available for arrays. Arrays may be compared for equality using == and `!=`. As for object references this is **reference equality**, that is, do the expressions reference the same array? Arrays are also considered to have a single attribute representing the length of the array. This attribute is accessed via the expression:

```
name.length
```

resulting in the number of elements in the array referenced by *`name`*. For example, with the array `a` declared and created above, `a.length` will be 10.

Since arrays represent collections of things, their primary purpose is to group the elements together so they can be conveniently processed. Most of the actual processing involves the individual elements of the array. To access the individual elements of the array, a subscripted variable is used. The form of a subscripting expression is:

```
name[expression]…
```

The *`name`* must be an array variable to which an array reference has been assigned, or else a `NullPointerException` occurs. The *`expression`*s must be integer expressions that evaluate to a value between 0 and the length of the corresponding dimension of array minus 1, or else an `ArrayIndexOutOfBoundsException` occurs. Each *`expression`* is known as **subscript** or **index**. For a one-dimensional array, the index indicates a particular element within the array with 0 being the first element, 1 being the second, and `a.length-1` being the last element. Java uses **zero-based subscripting**, where 0 is the first element. Many languages use one-based subscripting where the first element is numbered 1.

A subscripting expression may be used anywhere a simple variable identifier may be used. When used as an expression—either on the right-hand side of an assignment or as an actual parameter—the value is the value of the element designated by the subscript. When used as the left-hand side of an assignment, the value of the element designated by the subscript is replaced. The type of a subscripted array expression is the element type of the array. For example, the code:

```
a[3] = 5;
a[7] = a[3] + 4;
```

results in a change to elements `3` and `7` of the array `a` as shown in Figure 1.2. Note that the values of all other elements are unaffected.

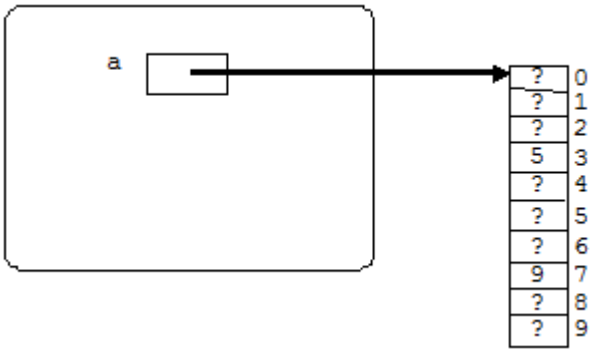


Figure 1.2 Accessing Array Elements

1.2 ARRAY PROCESSING

Arrays can be used whenever we have a collection of related items of the same type such as marks on a test or students in a course. They are necessary whenever the collection must be processed in non-sequential order. This is the case when all the processing required cannot be done when the item is first encountered, but must wait until later items of the collection have been processed.

Array processing involves: (1) the declaration of an array variable, (2) the creation of the array, (3) initialization of some or all elements, and (4) processing of some or all elements. To declare the array, all we need to know is the type of the elements. To create the array, however, we need to know the number of elements involved. Sometimes we know this *a priori*—either the number is fixed or can be computed from information available when the array is created. Sometimes we do not know the number of elements involved, as when the amount of data is unknown until it has all been read. This gives rise to two different ways of using arrays; we designate them as **"right-sized" arrays** (when the size is known) and **"variable-sized"** arrays (when the size is unknown). Both ways will be discussed in the following sections.

The most common form of processing an array is to perform some set of operations on each element of the array. This process is called **traversal**. Similar to `Collection` types, traversal can be done using either a for-each loop or an iterative for loop. The for-each loop can be used whenever every element of the array must be processed in order by index value and the elements are not going to be changed. The iterative for is more general and supports control over the order of processing, processing of portions of the array and change of element values.

RIGHT-SIZED ARRAYS

Right-sized arrays can be used whenever we know, *a priori*, how many elements are involved. For example, consider a program that determines which months of the year have above average monthly rainfall. The data consists of 12 measurements indicating the rainfall for the months of the year. The data cannot be processed sequentially since it is impossible to determine if a month's rainfall is above average until the average is known, and the average cannot be determined until the rainfall for each month has been accessed. The solution is to use an array to store the rainfall values. Since there are always 12 months in the year, we know that the array will have 12 elements, so a right-sized array can be used. Figure 1.3 shows the program.

The rainfall data (`rainfall`) is declared as a one-dimensional array of `double`s, in which the month is the dimension (line 17). Before the data can be read into the array, the array must be created. Since we know the size, the array creation can use the constant `12` (line 23). Loading the rainfall data is an example of array traversal—processing each element to load a value. Since we are loading (changing) the values of the elements, an iterative for is used (lines 25–28). The sum of the rainfall over the months is computed (lines 24 & 27) at the same time as reading the data rather than doing it in a separate loop.

The for loop index variable (`i`) is used as the array subscript (lines 26 & 27) and must range over all the elements of the array. This means that the initial value for `i` must be `0` and the last time through the loop `i` must be `rainfall.length-1`. This gives us the initial and test values for the loop (line 25). We could have used 12 for the test value instead of `a.length` since there will always be 12 months in a year. However, it is good practice to use the `length` attribute of the array instead of a literal since even quantities considered as constant at the time the program is written might change.

```
1  package Rainfall;
2
3  import BasicIO.*;
4
5  /** This program lists the months of the year with above average rainfall.
6    * @author D. Hughes
7    * @version 1.0 (Jan. 2014)                                              */
8
9  public class Rainfall {
10
11     private ASCIIDataFile   in;   // file with rainfall data
12     private ASCIIDisplayer  out;  // displayer for output
13
14     /** The constructor displays the months with above average rainfall.   */
15     public Rainfall ( ) {
16         String    year;      // year
17         double[]  rainfall;  // rainfall for each month
18         double    totRain;   // total rainfall for the year
19         double    aveRain;   // average monthly rainfall
20         in = new ASCIIDataFile();
21         out = new ASCIIDisplayer();
22         year = in.readString();
23         rainfall = new double[12];
24         totRain = 0;
25         for ( int i=0 ; i<rainfall.length ; i++ ) {
26             rainfall[i] = in.readDouble();
27             totRain = totRain + rainfall[i];
28         };
29         aveRain = totRain / rainfall.length;
30         writeHeader(year,aveRain);
31         for ( int i=0 ; i<rainfall.length ; i++ ) {
32             if ( rainfall[i] > aveRain ) {
33                 writeDetail(i+1,year,rainfall[i]);
34             };
35         };
36         in.close();
37         out.close();
38     }; // constructor
```

```
39
40      /** This method writes the header for the rainfall display.
41        * @param year     year
42        * @param aveRain  average rainfall.                                    */
43      private void writeHeader ( String year, double aveRain ) {
44          out.writeString("Average monthly rainfall for "+year+": ");
45          out.writeDouble(aveRain,5,2);
46          out.newLine();
47          out.newLine();
48          out.writeString("Months with above average rainfall:");
49          out.newLine();
50      }; // writeHeader
51
52
53      /** This method writes the deatil line for the rainfall display.
54        * @param month     month number
55        * @param year      year
56        * @param rainfall  rainfall amount.                                    */
57      private void writeDetail ( int month, String year, double rainfall ) {
58          out.writeString(month+"/"+year);
59          out.writeDouble(rainfall,5,2);
60          out.newLine();
61      }; // writeDetail
62
63      public static void main ( String[] args ) { Rainfall r = new Rainfall(); };
64
65  } // Rainfall
```

Figure 1.3 Example—Above Average Rainfall

Note the two different uses of the subscripting expression. In line 26 it is used as a destination (left-hand side of an assignment) indicating the element whose value is to be replaced. In line 27, it is used as an expression (right-hand side of an assignment) to indicate the element from which the value is to be obtained to accumulate the value into the sum.

Once the rainfall data has been read and the average computed, a second array traversal is used to determine which months have above average rainfall. Again the iterative for is used (lines 31–34) since we wish to display the month number (corresponding to the loop index variable) on the line along with the amount of rain (line 33). Since we normally number months from 1 for January, the month number is `i+1`.

Figure 1.4 shows the basic right-sized array traversal algorithm using an iterative for loop over the array *`name`*. The loop index (*`index`*) ranges from `0` to *`name`*`.length-1`. A subscripting expression with *`index`* as the subscript is used to access each element in turn.

```
for ( int index=0 ; index<name.length ; index++ ) {
        :
    process name[index]
        :
};
```

Figure 1.4 Right-sized Array Traversal using an Iterative `for` Loop

Figure 1.5 shows the same algorithm using a for-each loop. *type* is the element type of *name*. The loop variable *elt* is assigned the value of each of the elements of name in turn. Note that in Figure 1.4, the array subscripting expression (*name*[*index*]) can be used on the left-hand side of an assignment statement to change the value of an element. However, using the variable *elt* on the left-hand side of an assignment would only change the value of the index variable *elt*, leaving the corresponding array element unchanged. This is why the for-each loop is only used when the array elements are not being modified.

```
for ( type elt : name ) {
        :
    process elt
        :
};
```

Figure 1.5 Right-sized Array Traversal using a for-each Loop

VARIABLE-SIZED ARRAYS

More often than not, the number of elements that we need in an array is unknown when the program is being written and cannot even be computed before the array must be created. This requires another approach to array processing—what we call variable-sized arrays. Note that this technique is the only one possible in languages where the length of an array must be specified in the declaration.

Let's consider a program to produce a report summarizing the results in a term test (or other piece of work) for a course including the average and standard deviation. Standard deviation is a measure of how close the individual marks cluster around the average (mean). The smaller the standard deviation, the tighter the marks cluster around the mean. A formula to compute standard deviation is:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where N is the number of values (students), x_i is the mark for the i^{th} student and x-bar is the mean (average). To compute the standard deviation, we need to know the difference between each student's mark and the average. However to compute the average we need to obtain each student's mark. Thus we cannot do this in a single sequential pass over the data, and we will need to use an array.

It is likely that this program is part of a system of programs that support the recording and reporting of course grades at a university. In such a system, there would likely be a `Student` class that would represent all relevant information about a student in a course.

Let us assume that the `Student` class provides, among other things, a constructor to read student objects from an `ASCIIDataFile` and accessor methods to obtain a student's student number, name and term test mark. The `Student` class might look something like Figure 1.6.

```
public class Student {
       :
     public Student ( ASCIIDataFile from ) {…
       :
     public String getStNum ( ) {…
       :
     public String getName ( ) {…
       :
     public double getTestMark ( ) {…
       :
}  // Student
```

Figure 1.6 `Student` Class—Partial Specification

Figure 1.8 is a program to display term test statistics. Since the number of students is not known until all the data has been read, we cannot use a right-sized array. Instead we use a technique we will call "variable-sized" array. The idea here is that we create (line 32) an array of some fixed size (constant `MAX_STD` declared on line 13) that is expected to be big enough to handle any course. We then use only part of the array to store data (`Student` objects in this case). The remainder of the elements of the array remain unitialized. We maintain a variable (`numStd`) which is the count of the number of students stored in the array and ensure, when processing the array, that we only process elements in the range `0…numStd-1`. Figure 1.7 shows the memory model for the array after the loop that loads the data (lines 35–42) has been executed.

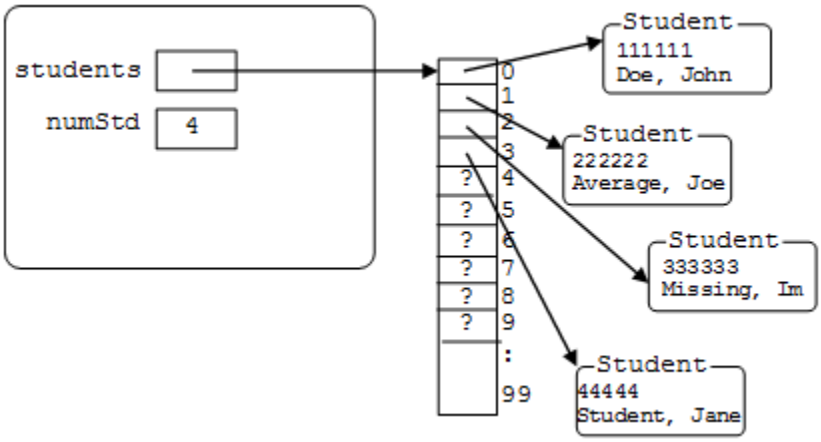


Figure 1.7 Memory Model of a Variable-sized Array

Note that the loop that loads the student objects has two conditions for termination (line 37): end of file and `numStd` reaching `MAX_STD`. This ensures that, if the file contains more than `MAX_STD` students, the program doesn't crash on an `ArrayIndexOutOfBounds` exception. Instead the program will only process the first `MAX_STD` students in the data file. It would be good form to display a message in this case, but for simplicity this was not done in this example. The choice of upper limit on class size is arbitrary and should be easy to change. Using a constant means that the only line that needs to be changed to accommodate larger class sizes in the constant declaration (line 13).

```
1  package ClassAve;
2
3  import BasicIO.*;
4  import static java.lang.Math.*;
5  import static BasicIO.Formats.*;
6
7  /** This class is a program to calculate a class average and standard deviation.
8    * @author David Hughes
9    * @version 1.4 (Jan. 2014)                                                    */
10
11 public class ClassAve {
12
13     private static final int  MAX_STD = 100;  // max number of students
14     private ReportPrinter  report;   // printer for report
15     private ASCIIDataFile  stData;   // data file for student marks
16
17     /** The constructor generates the report, reading the student data,
18       * computing the average and standard deviation and displaying the results. */
19
20     public ClassAve ( ) {
21         String     course;    // course name
22         Student[]  students;  // the class of students
23         Student    aStudent;  // one student
24         int        numStd;    // number of students
25         double     sum;       // sum of marks
26         double     ave;       // average mark
27         double     std;       // standard deviation
28         report = new ReportPrinter();
29         stData = new ASCIIDataFile();
30         course = stData.readString();
31         setUpReport(course,"Term Test");
32         students = new Student[MAX_STD];
33         numStd = 0;
34         sum = 0;
35         for ( ; ; ) {
36             aStudent = new Student(stData);
37         if ( stData.isEOF() | numStd >= MAX_STD ) break;
38             students[numStd] = aStudent;
39             numStd = numStd + 1;
40             sum = sum + aStudent.getTestMark();
41             writeDetail(aStudent);
42         };
43         ave = sum / numStd;
44         std = computeStd(students,numStd,ave);
45         writeSummary(ave,std);
46         report.close();
47         stData.close();
48     };  // constructor
```

```
49
50      /** This method computes the standard deviation of the marks for the students.
51        * @param  students  students in the class
52        * @param  num       number of students
53        * @param  ave       average mark
54        * @return  double  the standard devisation of the marks                    */
55      private double computeStd ( Student[] students, int numStd, double ave ) {
56          double  sum;    // sum of squares of deviations
57          double  aMark;  // a student mark
58          sum = 0;
59          for ( int i=0 ; i<numStd ; i++ ) {
60              aMark = students[i].getTestMark();
61              sum = sum + pow(aMark-ave,2);
62          };
63          return sqrt(sum/numStd);
64      };  // computeStd
65
66      /** This method sets up the report, adding all fields.
67        * @param  courseName  name of the course
68        * @param  workName    name of the piece of work                            */
69      private void setUpReport ( String courseName, String workName ) {
70          report.setTitle(courseName,workName);
71          report.addField("stNum","Student #",10);
72          report.addField("name","Name",20);
73          report.addField("mark","Mark",getDecimalInstance(1),5);
74      };  // setUpReport
75
76      /** This method generates a report detail line.
77        * @param  aStudent  a student                                              */
78      private void writeDetail ( Student aStudent ) {
79          report.writeString("stNum",aStudent.getStNum());
80          report.writeString("name",aStudent.getName());
81          report.writeDouble("mark",aStudent.getTestMark());
82      };  // writeDetail
83
84      /** This method generates the report summary.
85        * @param  average  average mark in course
86        * @param  std      standard deviation                                      */
87      private void writeSummary ( double average, double std ) {
88          report.writeString("stNum","Average");
89          report.writeDouble("mark",average);
90          report.writeString("stNum","Std. Dev.");
91          report.writeDouble("mark",std);
92      };  // writeSummary
93
94      public static void main ( String[] args ) { ClassAve c = new ClassAve(); };
95
96  }  // ClassAve
```

Figure 1.8 Example-Class Average and Standard Deviation

The method `computeStd` (lines 55-64) exemplifies processing of variable-sized arrays. As can be seen, arrays can be passed as parameters to a method (line 55), just as objects can. As for objects, what is passed is a reference to the array. This means that the parameter `students` in

`computeStd` is a second reference to the array referenced by the variable `students` in the constructor. There is only one array, and both names reference it. Secondly, since with variable-sized arrays only part of the array is used, the number of students (`numStd`) must also be passed as a parameter.

The loop (lines 59-62) is an example of the traversal algorithm for variable-sized arrays. Since we should not process all of the elements of the array since some are potentially uninitialized, using a for-each loop for traversal isn't an option. Secondly, the length attribute of the array (`students.length`) is the number of elements (100 in this case) not the number of students, so it cannot be used as the test value on the iterative for. Instead, the iterative for (line 59) runs from `0` through `numStd-1`, ensuring we only process meaningful array elements. This is the reason that `numStd` must also be passed as a parameter. Figure 1.9 shows the basic algorithm for traversal of a variable-sized array where *`index`* is the loop index variable used as the array subscript, *`name`* is the array and *`numberOfElements`* is the variable containing the number of elements used within the array.

```
for ( int index=0 ; index<numberOfElements ; index++ ) {
        :
    process name[index]
        :
};
```

Figure 1.9 Variable-sized Array Traversal Algorithm

As a final point, note the subtle difference between the creation of the array of `Student` references (line 32) and the creation of an individual `Student` object (line 36). Syntactically the only difference is the use of brackets *vs* parentheses. However, the result is very different. In array creation, `Student` is the element type and the array length (`MAX_STD`) is written within brackets. The result is a reference to an array of `MAX_STD` elements that may contain `Student` references. No constructor is executed. In object creation, `Student` is the class (object type) and `stData` is an argument for the constructor call. The result is a reference to a single `Student` object which has been initialized by the call to the `Student` constructor.

1.3 ARRAYS AND METHODS

Like any other type, array references may be passed as parameters to a method and a method can return an array reference as its result. Different techniques must be used for right-sized arrays and variable-sized arrays.

ARRAY PARAMETERS

As we have seen, like any other type, arrays may be passed as parameters to a method. Since array variables are reference variables, what is passed is the reference to the array. The formal parameter becomes another reference to the same array. Within the method, the array elements can be modified, but the array itself remains the same. As an example of array parameters, consider a modification to the example of Figure 1.3 to use a method to read the rainfall data. Lines 23–28 in Figure 1.3 would be replaced by:

```
23          rainfall = new double[12];
24          readRain(rainfall);
25          totRain = 0;
26          for ( int i=0 ; i<rainfall.length ; i++ ) {
27              totRain = totRain + rainfall[i];
28          };
```

After creating the array (line 23), the method `readRain` is called to read the rainfall data into the array `rainfall`. It is now unnecessary to read the rainfall data within the for loop, so that step is omitted. Note that to pass the array reference, the array name is used without a subscript.

Figure 1.10 shows the method `readRain`. At the method call (line 24 above), the parameter `rain` is initialized to a reference to the same array as `rainfall` in the constructor. Since the array exists, it has a length attribute which can be referenced as `rain.length` (line 43). Lines 43–44 implement the traversal of a right-sized array (see Figure 12.4). Within this loop (line 44), assignment to element `i` of `rain` sets the value of the i^{th} element of the array referenced by both `rain` and `rainfall`. When the method returns (after line 24 above) the array `rainfall` has been loaded.

```
40      /** This method reads the rainfall data.
41        * @param  rain  array for rainfall data                    */
42      private void readRain ( double[] rain ) {
43          for ( int i=0 ; i<rain.length ; i++ ) {
44              rain[i] = in.readDouble();
45          };
46      };  // readRain
```

Figure 1.10 Example—Input Method for Above Average Rainfall

Note that the array must be created in the calling code, it cannot be created in the method. Since an array reference is passed, the parameter is a copy of the reference. If an assignment to the parameter is made the argument is unchanged. For example, if

```
rain = new double[12];
```

is inserted before line 43 above, when the method returns, `rainfall` in the calling method will still reference the original uninitialized array created in line 23. The array created in the method will be garbage collected since there are no longer any references to it.

A similar technique can be used for variable-sized arrays, with slight modification. Let us consider modifying the program in Figure 1.8 to use an input routine. In addition to filling in the array elements, the method will have to determine the number of elements placed in the array since this value (`numStd` in Figure 1.8) will be needed in the calling code. Since a method is working with a copy of an argument, attempting to return the value by changing the parameter won't have any effect. The answer is to write a function method to return the number of elements read as its result. Figure 1.11 shows an input method for reading the student data for a modified version of Figure 1.8. Lines 32–42 of Figure 1.8 would be replaced by:

```
31          students = new Student[MAX_STD];
32          numStd = readStudents(students);
33          sum = 0;
34          for ( int i=0 ; i<numStd ; i++ ) {
35              sum = sum + students[i].getTestMark();
36              writeDetail(students[i]);
37          };
```

After the array has been created (line 31), it is passed as a parameter to `readStudents`. The method fills the array and returns the number of students read as its result (line 32). The processing loop (lines 34–37) is now just a variable-sized array traversal algorithm, accessing the student objects within the array.

```
45      /** This method reads the student objects into an array and returns the number
46        * of students read.
47        * @param  stds  the array of students
48        * @return  int  number of students read                                    */
49      private int readStudents ( Student[] stds ) {
50          Student  aStudent;  // a student
51          int      nStd;      // number of students
52          nStd = 0;
53          for ( ; ; ) {
54              aStudent = new Student(stData);
55          if ( stData.isEOF() | nStd >= stds.length ) break;
56              stds[nStd] = aStudent;
57              nStd = nStd + 1;
58          };
59          return nStd;
60      };  // readStds
```

Figure 1.11 Example—Input Method for Class average and Standard Deviation

In the method, the parameter `stds` is a copy of the reference to the array provided as the argument (`students`), so the method modifies the elements of the array. The `length` attribute of the parameter can be used to determine the physical length of the array—the limit on the number of students that can be read. When the data has been read, and the students counted, the count is returned as the result of the method.

The technique of passing an array as a parameter can be used whenever a method requires access to an array, not just for reading values into the array. When the array is right-sized, it is sufficient to simply pass the array since the length of the array can be determined from the `length` attribute. When the array is variable-sized, although the physical length can be determined from the `length` attribute, the actual number of relevant elements needs to be known, so this must be passed as an additional parameter in a method header such as in the `computeStd` method in Figure 1.8.

ARRAYS AS RESULTS OF FUNCTION METHODS

Arrays may also be returned as the result of a method. When an array is passed as a parameter, the method is working with a copy of the reference to the array and so cannot replace it with a new array. The only way a method can produce a new array is to return it as the method result. The data input method in Figure 1.10 can be rewritten as in Figure 1.12 and called by:

```
rainfall = readRain();
```

replacing lines 23 & 24 since the array creation in the constructor will no longer be necessary.

```
39      /** This method reads the rainfall data.
40        * @return  double[]  array for rainfall data                    */
41      private double[] readRain (  ) {
42          double[] rain;
43          rain = new double[12];
44          for ( int i=0 ; i<rain.length ; i++ ) {
45              rain[i] = in.readDouble();
46          };
47          return rain;
48      };  // readRain
```

Figure 1.12 Example—Input Method as a Function

Within the method, a new array is created with reference stored in the local variable `rain` (lines 42 & 43). This array is then filled with data as before. When the method is complete, the array referenced by the local variable is returned; that is the reference to the array is returned. This reference is then stored in the variable `rainfall` in the calling code, producing the desired result. Note that, the array name is used without a subscript to return a reference to the array. If a subscript is used, the value of the element is returned.

Variable-sized arrays cannot effectively be returned as function results since the method would have to return two values (the array reference and the number of elements) and this is not possible.

1.4 MULTIDIMENSIONAL ARRAYS

Often data is presented in tabular form, for example, university enrollment statistics across universities and departments (Figure 1.13) or rainfall by month and year. In these cases, the data is said to have two dimensions because each piece of data has two attributes: university and department or month and year. Sometimes there can be additional dimensions as well. For example, we might we consider the university enrollments over years as well as universities and departments. If we wish to represent such information as a whole, as opposed to processing the individual values sequentially, we need arrays of higher dimension, or multidimensional arrays.

As we saw in Section 12.1, arrays of higher dimension can be declared, created and indexed by including additional dimensions as additional sets of brackets. For example, an array to represent the enrollment statistics for five departments at four universities is declared, created and accessed via the statements similar to:

```
int[][]  enrol;
       :
enrol = new int[5][4];
       :
… enroll[i][j]…
```

Enrolment Report
Jan 29, 2014

	Acadia	Brock	McMaster	Laurentian	Total
Math	162	15	237	35	449
Business	836	182	987	243	2,248
Comp. Sci.	263	91	321	110	785
Biology	743	432	648	0	1,823
French	42	59	117	57	275
Total	2,046	779	2,310	445	5,580

Figure 1.13 Enrollment Statistics

The array `enrol` consists of five rows (the departments) and four columns (the universities), each element at the intersection of a row and column being an integer, the enrollment for the department at the university. The arrangement representing the data of Figure 1.13 looks like Figure 1.14.

	0	1	2	3
0	162	15	237	35
1	836	182	987	243
2	263	91	321	110
3	743	432	648	0
4	42	59	117	57

Figure 1.14 Array of Enrollment Data

In Java, the numbering of both the rows and columns begins at 0, as for one-dimensional arrays. An individual element of the array is referenced via an array access with two subscripts, for example the reference:

```
enrol[3][2]
```

indexes the element in the fourth row, third column, which is the enrollment in Biology at McMaster University (648 students).

Like one-dimensional arrays, multidimensional arrays can be processed as right-sized arrays, or variable-sized arrays. When a right-sized two-dimensional array is used (like `enrol` above), the number of rows in the array is given by the length attribute (`enrol.length`, 5). For any row, the number of columns in the row is given by the length attribute for that row (`enrol[2].length`, 4).

Note: The way the `enrol` array was created ensures that each row has the same number of columns, as we would expect for a table, so it doesn't really matter which row we use to determine the number of columns. However, in Java, it is possible to have arrays in which the rows have different numbers of columns. In this case it is critical to reference the length attribute for the correct row. We will be careful to write the code so that the correct length attribute is accessed to ensure correctness in all cases.

If a two-dimensional array is to be variable-sized, we would fill elements in the first rows and the first columns of each row—the top-left corner of the array. We would maintain two auxiliary variables: one indicating the number of rows that contain data and the other the number of columns of those rows containing data. Processing would be similar to the one-dimensional case—the auxiliary variables would be used to bound the loops used in accessing the array instead of the length attributes.

PROCESSING TWO-DIMENSIONAL ARRAYS

Like one-dimensional arrays, two-dimensional arrays are processed in either a sequential or random manner. Random access is typically used when the array is a lookup table. For example, we use a lookup table to answer the question "What is the enrollment in Biology at McMaster University?". Here the row index (3) and column index (4) are known since they are obtained from input or they can be directly determined. Sequential access usually occurs when the entire array must be processed. Since the array is two-dimensional, there are two natural traversal orders: row-by-row or column-by-column.

Figure 1.15 is the algorithm for row-by-row (**row-major**) traversal of a right-sized array. The algorithm for a variable-sized array is similar. The index `i` sequences through the rows while `j` sequences through the columns. The order of access is: `a[0][0], a[0][1], a[0][2],…` `a[1][0], a[1][1], a[1][2],… a[2][0], a[2][1], a[2][2],….` Note that if we look at the subscripts as digits of a number, the resulting numbers: `00, 01, 02,… 10, 11, 12,… 20, 21, 22,…` are in numeric order. When information is processed in such an order, we say it is processed in **lexicographic order**. Depending on the use of the algorithm, there may be processing required before or after each row or both. If the rows have no particular significance in the algorithm, those steps are omitted.

```
for ( int i=0 ; i<a.length ; i++ ) {
    preprocessing for row i
    for ( int j=0 ; j<a[i].length ; j++ ) {
        process a[i][j]
    };
    postprocessing for row i
};
```

Figure 1.15 Row-major Array Traversal of a Right-Sized Array

Figure 1.16 is the algorithm for column-by-column (**column-major**) traversal of a right-sized array with the variable-sized traversal being similar. Again, the index `i` sequences through the rows and `j` through the columns. However, since `j` is in the outer loop, `i` sequences through all of its values for each value of `j`, giving the order of access: `a[0][0], a[1][0], a[2][0],… a[0][1], a[1][1], a[2][1],… a[0][2], a[1][2], a[2][2],….` Note the use of `a[0].length` as limit on the outer loop. The pattern assumes the array is **regular**—that each row has the same number of columns. Only regular arrays can be processed in column-major order. If the array was not regular, it would be impossible to know how many columns to process. For this reason, row-major processing is preferred unless the array is known to be regular and the data must be processed column-by-column.

```
for ( int j=0 ; j<a[0].length ; j++ ) {
    preprocessing for column j
    for ( int i=0 ; i<a.length ; i++ ) {
        process a[i][j]
    };
    postprocessing for column j
};
```

Figure 1.16 Column- major Array Traversal of a Right-Sized Array

COMPILING UNIVERSITY ENROLLMENT STATISTICS

Figure 1.17 is a program that demonstrates array processing. It reads a file containing university enrollment data and produces the summary table of Figure 1.13. The data file consists of the number of universities (`int`), the number of departments (`int`), followed by the names of the universities (`String`s) and the names of the departments (`String`s). The raw enrolment data follows as `int`s in row-major (by department) order. The program creates a right-sized array (`enrol`) to hold the raw data (lines 27 & 39). The two one-dimensional arrays `univ` and `dept` are created as arrays of `String`s (lines 25, 26, 37 & 38) to hold the names of the universities and departments, respectively. The two one-dimensional arrays `uTotals` and `dTotals` are declared as arrays of `int`s (28, 29) to hold the university and department totals, respectively. Finally, `total` (`int`) will hold the grand total (line 30).

The helper methods are shown in Figure 1.18. The method `readStats` reads the university and department names and the enrollment data. It is passed the previously created arrays `univ`, `dept` and `enrol`. As discussed in Section 1.3, as an input routine, the arrays must be created prior to the call and passed to the method. Since the arrays are right-sized, no other parameters are necessary.

The method `sumbyDept` produces the department (row) totals within the parameter `stats`. It creates a right-sized array (`sums`) as the number of rows in stats. Using the row-major traversal algorithm, it computes the sums for each row. The row preprocessing involves the initialization of the current row sum (`sums[i]`) to zero. The element processing involves accumulating the element into the current row sum. The method then returns the resulting one-dimensional array.

```
1  package UStats;
2
3  import java.util.*;
4  import BasicIO.*;
5  import static BasicIO.Formats.*;
6
7  /** This class inputs enrolment statistics from a number of departments at a
8    * number of universities and produces a summary table with row, sum and grand
9    * totals.
10   * @author D. Hughes
11   * @version 1.0 (Jan. 2014)                                                  */
12
13 public class UStats {
14
15     private ASCIIDataFile   dataFile; // file for input
16     private ReportPrinter   report;   // file for report
17     private ASCIIDisplayer  msg;      // displayer for messages
18
19     /** The constructor reads the enrolment stats and generates and displays the
20       * summaries.                                                             */
21
22     public UStats ( ) {
23         int       nUniv;    // number of universities
24         int       nDept;    // number of departments
25         String[]  univ;     // university names
26         String[]  dept;     // dept names
27         int[][]   enrol;    // enrolment stats
28         int[]     uTotals;  // University totals
29         int[]     dTotals;  // Department totals
30         int       total;    // grand total
31         dataFile = new ASCIIDataFile();
32         report = new ReportPrinter();
33         msg = new ASCIIDisplayer();
34         msg.writeString("Processing.....");
35         nUniv = dataFile.readInt();
36         nDept = dataFile.readInt();
37         univ = new String[nUniv];
38         dept = new String[nDept];
39         enrol = new int[nDept][nUniv];
40         readStats(univ,dept,enrol);
41         uTotals = sumByUniv(enrol);
42         dTotals = sumByDept(enrol);
43         total = sumAll(enrol);
44         initReport(univ);
45         writeStats(dept,enrol,dTotals,uTotals,total);
46         msg.writeString(".....complete");
47         msg.newLine();
48         dataFile.close();
49         report.close();
50         msg.close();
51     }; // constructor
```

Figure 1.17 Example—Producing Enrollment Statistics

```
52
53      /** This method reads the enrollment statistics for the universities and
54        * departments.
55        * @param univ  university names
56        * @param dept  department names
57        * @param enrol the array to read into.                                    */
58      private void readStats ( String[] univ, String[] dept, int[][] stats ) {
59          for ( int j=0 ; j<univ.length ; j++ ) {
60              univ[j] = dataFile.readString();
61          };
62          for ( int i=0 ; i<dept.length ; i++ ) {
63              dept[i] = dataFile.readString();
64          }
65          for ( int i=0 ; i<stats.length ; i++ ) {
66              for ( int j=0 ; j<stats[i].length ; j++ ) {
67                  stats[i][j] = dataFile.readInt();
68              };
69          };
70      }; // readStats
71
72      /** This method sums the enrollments by department (row).
73        * @param  stats  the enrollment statistics.
74        * @return  int[]  the department totals.                                  */
75      private int[] sumByDept ( int[][] stats ) {
76          int[]  sums;
77          sums = new int[stats.length];
78          for ( int i=0 ; i<stats.length ; i++ ) {
79              sums[i] = 0;
80              for ( int j=0 ; j<stats[i].length ; j++ ) {
81                  sums[i] = sums[i] + stats[i][j];
82              };
83          };
84          return sums;
85      }; // sumByDept
86
87      /** This method sums the enrollments by university (column).
88        * @param  stats  the enrollment statistics.
89        * @return  int[]  the university totals.                                  */
90      private int[] sumByUniv ( int[][] stats ) {
91          int[]  sums;
92          sums = new int[stats[0].length];
93          for ( int j=0 ; j<stats[0].length ; j++ ) {
94              sums[j] = 0;
95              for ( int i=0 ; i<stats.length ; i++ ) {
96                  sums[j] = sums[j] + stats[i][j];
97              };
98          };
99          return sums;
100     }; // sumByUniv
```

```
101
102     /** This method computes the grand sum of the enrollment statistics.
103       * @param  stats  the enrollment statistics.
104       * @return  int  the grand total.                                    */
105     private int sumAll ( int[][] stats ) {
106         int  sum;
107         sum = 0;
108         for ( int i=0 ; i<stats.length ; i++ ) {
109             for ( int j=0 ; j<stats[i].length ; j++ ) {
110                 sum = sum + stats[i][j];
111             };
112         };
113         return sum;
114     }; // sumAll
115
116     /** This method initializes the enrolment report.
117       * @param univ  university names                                     */
118     private void initReport ( String[] univ ) {
119         report.setTitle("Enrolment Report",getDateInstance().format(new Date()));
120         report.addField("dept","",10);
121         for ( int i=0 ; i<univ.length ; i++ ) {
122             report.addField("",univ[i],getIntegerInstance(),10);
123         };
124         report.addField("total","     Total",getIntegerInstance(),10);
125     };  // initReport
126
127
128     /** This method displays the enrollment stats in a tabular format with labels
129       * and totals for each row and column and a grand total.
130       * @param  dept   department names
131       * @param  stats  the enrollment statistics.
132       * @param  rSums  the row sums.
133       * @param  cSums  the column sums.
134       * @param  sum    the grand sum.                                     */
135     private void writeStats( String[] dept, int[][] stats,
136                              int[] rSums, int[] cSums, int sum ) {
137         for ( int i=0 ; i<stats.length ; i++ ) {
138             report.writeString(dept[i]);
139             for ( int j=0 ; j<stats[i].length ; j++ ) {
140                 report.writeInt(stats[i][j]);
141             };
142             report.writeInt(rSums[i]);
143         };
144         report.writeString("Total");
145         for ( int j=0 ; j<cSums.length ; j++ ) {
146             report.writeInt(cSums[j]);
147         };
148         report.writeInt(sum);
149     }; // writeStats
150
151     public static void main ( String[] args ) { UStats u = new UStats(); };
152
153 } // UStats
```

Figure 1.18 Example—Producing Enrollment Statistics-Helper Methods

The method `sumByUniv` similarly produces the university (column) totals. The array is assumed to be regular. The column sum array (`sums`) is also created right-sized as the number of columns in the first row of `stats`. It computes the sums using the column-major traversal algorithm. The column sum (`sums[j]`) is initialized to zero as the column preprocessing. The elements are accumulated into the column sum as the element processing. Again, the totals are returned by the method as an array.

The method `sumAll` produces the grand total enrollment in all departments at all universities. The order of processing makes no difference because all elements must be accumulated into the sum, so the row-major traversal is used. This algorithm doesn't require the regularity assumption. The sum is initialized prior to the algorithm because we only want to do it once. The elements are accumulated into the sum, which is returned by the method.

The table is produced by the method `writeStats`. This method uses row-major traversasl to print the report row-by-row. The headings are the university names set up in the method `setUpReport` (lines 118-125). The row preprocessing writes the department names (`dept`). The element processing writes the element values (`stats`). Finally, the row post-processing writes the department totals (`rSums`). As the report summary, the university totals (`cSums`) are written, followed by the grand total (`sum`).

This example is a bit contrived since it would be possible to read, produce all the row, column and grand totals and display the table in a single pass over the array (after initialization). In fact, since the processing is sequential, it was not even necessary to use arrays (except for the column totals—can you see why?). Also, the `readStats` method could have returned the two-dimensional array as a result or the row and column sum methods could have been passed and array to fill. If the number of universities and departments were not known in advance, variable-sized arrays could have been used, passing the number of departments (rows) and number of universities (columns) as parameters as necessary.

*1.5 ARRAY REPRESENTATION

The memory model given in Figure 1.1 is a reasonably accurate depiction of how one-dimensional arrays are represented within main memory. Consider that main memory is a series of cells, each of which can hold a value, and is referenced by an integer called an address. This sounds very similar to an array: a series of elements, each of which can hold a value and is referenced by an integer, called the subscript. Really, memory is a big array of bytes. When the compiler generates code for a program, it allocates to each variable a sequence of consecutive bytes in memory. For example, 4 for `int`, 4 for a reference variable, 8 for `double`. Similarly, when an object or array is created via `new`, the Java run-time allocates a sequence of consecutive bytes. This is the total of the storage requirements for the instance variables of an object or for all the elements of the array. The value stored into the reference variable is the address of the first byte of the allocated storage. This contiguous allocation pattern is used because, in general, it is most efficient and only one address is needed to reference it.

For higher dimensional arrays, a number of different storage allocation schemes are possible. The most common, and the one used by most imperative languages, is contiguous allocation of the array. All of the elements are stored in consecutive memory locations. The other scheme used by Java and some other languages, is called array-of-arrays allocation where the rows of the array are allocated

separately. Each scheme has advantages and disadvantages. Contiguous allocation is often more efficient while array-of-arrays allocation is more flexible.

CONTIGUOUS ALLOCATION

One-dimensional arrays are always stored contiguously or else it would be necessary to have references to each element. Things get a bit more complicated for multidimensional arrays. Since memory is one-dimensional—that is, each cell is referenced via a single address. We need a way of mapping a two-, or higher-dimensional array onto one-dimension. If we wish to maintain the contiguity of the array, for a two-dimensional array, there are two possibilities. Keep the elements of the rows contiguous and allocate the rows, one after another—called **row-major order**. Alternately, keep the elements of the columns contiguous and allocate the columns, one after another—called **column-major order**. For the array `a`, with three rows of four columns, the row-major ordering is shown in Figure 1.19 and the column-major ordering in Figure 1.20. Row-major ordering, also known as lexicographic ordering, is most common and we'll limit our discussion to it.

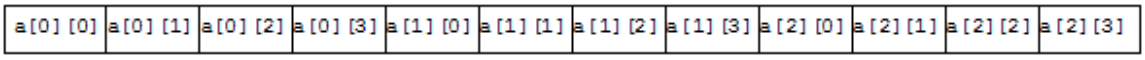


Figure 1.19 Row-major Storage Allocation

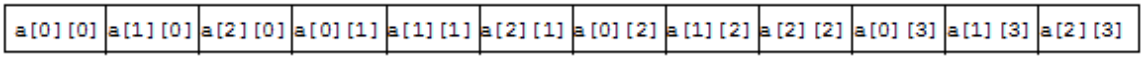


Figure 1.20 Column-major Storage Allocation

If we wish to retain the advantage of storing a single reference to the first byte of an array, we need to be able to compute the address of any element of the array, given the address of the start of the array and an index. That is, when we use the subscripted variable `a[i]` we need to be able to determine the address of element `i` of the array. The formula for this determination is called a **mapping function**.

For a one-dimensional array, things are quite simple. If we assume for the moment that each element of the array occupies one memory cell (i.e. 1 byte) the mapping function, assuming zero-based indexing as in Java, is:

```
address(a[i]) = address(a) + i
```

That is, element 0 is zero bytes from the start of the array, element 1 is one byte from the start of the array, and so on. Since the elements of an array usually occupy more than one byte, the distance from the start of the array must be multiplied by the size of the element type, designated `s`. For example, `s=4` for `int`, `s=4` for reference, `s=8` for `double`. This gives the mapping function:

```
address(a[i]) = address(a) + i × s
```

Most languages require the lower bound for an index to be 1 and some languages allow it to be any arbitrary integer. In these cases, if `l` is the lower bound for the index and `u` the upper bound, then the mapping function is:

$$address(a[i]) = address(a) + (i-l) \times s$$

That is, skip over `i-l` elements from the start of the array to get to element `i`.

The easiest way to determine the two-dimensional row-major mapping function for a regular array is to consider the array to be a one-dimensional array of rows where each "element" is a row. This gives the mapping function:

$$address(a[i]) = address(a) + i \times S$$

where `S` is the size of a row. Since, in Java, a row consists of `a[0].length` elements of size `s`:

$$S = a[0].length \times s$$

This gives us the start of the i^{th} row. Now the mapping function within the i^{th} row is:

$$address(a[i][j]) = address(a[i]) + j \times s$$

substituting for `address(a[i])` gives the complete mapping function for zero-based subscripting:

$$address(a[i][j]) = address(a) + i \times S + j \times s$$

For the general case, where the lower bound on the row index is l_r, the upper bound on the row index is u_r, the lower bound on the column index is l_c and the upper bound on the column index is u_c, the mapping function is:

$$address(a[i][j]) = address(a) + (i-l_r) \times S + (j-l_c) \times s$$

where

$$S = (u_c-l_c+1) \times s$$

For higher dimensional arrays, the mapping function can be extended in a similar manner, by considering the array to be a one-dimensional array of elements, each of which is an array of the next lower dimension.

Since row-major order is the standard representation for arrays of higher dimension, it is usually more efficient to access the array in row-major order since this accesses memory cells that are close together.

ARRAY-OF-ARRAYS ALLOCATION

For multi-dimensional arrays, we might relax the constraint that the entire array must be in contiguous storage, rather requiring only that elements of the rows be contiguous. This is actually the way Java handles arrays. An array with three rows of four elements each would be represented as shown in Figure 1.21. The array is represented as an array of row references each pointing to

separately allocated arrays of elements. Since the representation is an array of references to arrays of elements, it is often called an **array-of-arrays**, or row-of-rows, representation. Additional dimensions can be accommodated by additional rows of references pointing to the rows of references, etc.

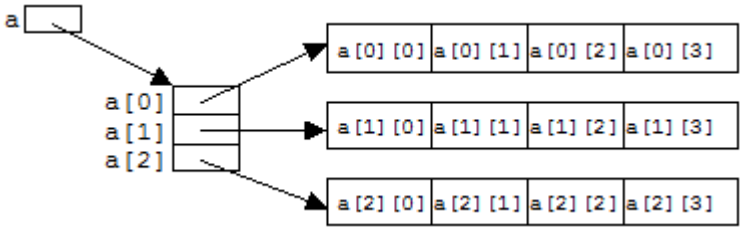


Figure 1.21 Array-of-arrays Storage

With this representation, an array access consists of repeatedly applying the one-dimensional mapping function for each index in turn, and using the result as the address of the next segment. For example, the mapping function for a two-dimensional array-of-arrays is:

```
address(a[i][j]) = contents(address(a)+i×4) + j × s
```

where `contents(…)` indicates the contents of the memory location specified by the address. This is itself an address—the address of the i^{th} row. Note that the multiplier for `i` is `4`, the size of a reference or address. As with row-major contiguous allocation, accessing an array-of-arrays is most efficient if done in row-major order since only the elements of a row are contiguous.

Since in this allocation each row of a two-dimensional array is a one-dimensional array, it is possible to have an array where each row has a different number of elements. In Java, it is not necessary to create the entire two-dimensional array in a single array creation expression, but rather each row can be created independently. For example an array with three rows, the first having three elements, the second two and the third four can be created with the following code:

```
int[][]     a;
      :
a = new int[3][];
a[0] = new int[3];
a[1] = new int[2];
a[2] = new int[4];
```

The result, which is sometimes called a **ragged array**, would look like Figure 1.22. The first assignment statement assigns to `a` a reference to a new array containing three references to arrays of `int`s. However, as yet these have no value, just as `a` had no value before the first assignment. The next three assignment statements create three one-dimensional arrays and assign the references to them into the appropriate elements of the array of references created in the first assignment. Once the array has been completely created, the `length` attributes return the appropriate values. That is, `a.length` is 3, indicating three rows, `a[0].length` is 3, `a[1].length` is 2 and `a[2].length` is 4. The row-major processing pattern will work appropriately with the ragged array, however, the column-major pattern will not. It is not possible, in Java, to create an array with columns of different size.

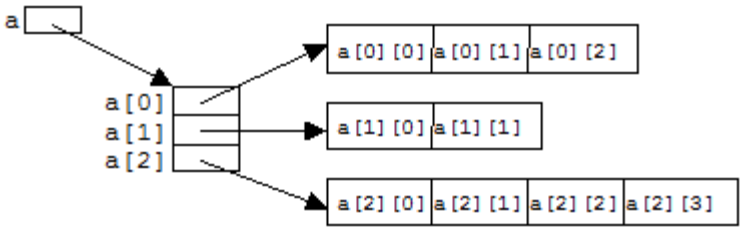


Figure 1.22 Ragged Array in Java

*1.6 SPECIAL ARRAY FORMS

Sometimes two- or higher- dimensional data doesn't occur in rectangular form, or it isn't efficient to store it that way. Java's ragged arrays (see Section 1.7) can sometimes be useful, however they still require that each row be contiguous from index 0, which often isn't the case.

Many areas of Computer Science, the physical sciences and Engineering make use of matrices to represent physical values and properties and linear algebra to represent processes on these values. Matrices can be represented as two-dimensional arrays. Sometimes these matrices are very large, say 100×100 or 1000×1000 or higher but consist of a high proportion—sometimes most—of zero entries. If the matrix is large, it may be unreasonable to store all of it in memory. Consider that a 1000×1000 matrix of doubles occupies 8,000,000 bytes! However, it may be effective to store only the non-zero entries. We will look at four kinds of matrices and consider how they might be stored to conserve space. It should be noted that—as is often the case in Computer Science—there is a **space-time tradeoff**. That is, the amount of space can be reduced at the expense of more time-consuming, processing of the elements.

The following sections are only examples, there are many other possibilities and representations. The most important thing to realize is that for many kinds of information, although one representation is usually predominant other representations are often possible and should be considered in special cases.

DIAGONAL MATRICES

A **diagonal matrix** (Figure 1.23) is a square matrix in which all of the elements, other than those on the main diagonal, are known to be zero. For a `1000×1000` matrix, this means that only `1000` (0.1%) are non-zero and `999,000` (99.9%) are zero. If only the non-zero elements were stored, the data would occupy 8,000 bytes instead of 8,000,000 bytes.

4	0	0	0	0
0	6	0	0	0
0	0	2	0	0
0	0	0	5	0
0	0	0	0	7

Figure 1.23 Diagonal Matrix

If the matrix is n×n, the main diagonal has n elements. These elements are the elements in the original matrix where the row and column indices are equal (i.e. a[0][0], a[1][1],… a[n][n]). The main diagonal can be represented as a row of n elements (call it d) with the value for a[i][i] stored in element d[i] (Figure 1.24).

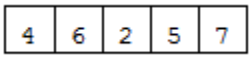


Figure 1.24 Diagonal Matrix as a Row

The mapping function for this new representation would be:

```
address(a[i][i]) = address(d) + i × s
```

all other elements a[i][j], i≠j, are zero. In Java, this could be implemented as a class as shown in Figure 1.25. The constructor would allocate a one-dimensional array (d) with n elements. The method getElt would return the element d[i] if i=j and 0 otherwise. The method setElt would store v in element d[i] if i=j and do nothing (or generate an error) otherwise.

```
public class DiagMatrix {
    public DiagMatrix ( int n ) {…
    public double getElt ( int i, int j ) {…
    public void setElt ( int i, int j, double v ) {…
        :
};  // DiagMatrix
```

Figure 1.25 DiagMatrix Class

Figure 1.26 compares the code when the matrix is declared as a two-dimensional array with the code using the DiagMatrix class. Instead of declaring a as an array, it is declared as a DiagMatrix. The class constructor is used instead of the array constructor. To set the value of an element of the array, setElt is used and to access the element of the array, getElt is used.

```
int[][] a;                          DiagMatrix  a;
  :                                     :
a = new int[1000][1000];            a = new DiagMatrix(1000);
  :                                     :
a[i][j] = …                         a.setElt(i,j,…);
  :                                     :
… = a[i][j] …                       … = a.getElt(i,j) …
```

Figure 1.26 Use of DiagMatrix Class

TRIANGULAR MATRICES

In a **triangular matrix**, all the elements below (**upper-triangular**) or above (**lower-triangular**) the main diagonal are known to be zero (Figure 1.27). Again the matrix is square.

1	0	0	0	0
2	3	0	0	0
4	5	6	0	0
7	8	9	10	0
11	12	13	14	15

Figure 1.27 Lower-triangular Matrix

The total number of non-zero elements is `1+2+…+(n-2)+(n-1)+n = n(n+1)/2`. Essentially 50% of the array has non-zero elements. The easiest way to represent this is to place the `n` partial rows side by side, without the zero elements, as in Figure 1.28.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figure 1.28 Lower-triangular Matrix as a Row

The first row (row `0`) has 1 element, the second, 2 and so on. The i^{th} row has `(i+1)` elements. The first term in the usual mapping function for a two-dimensional array (`i×S`) can be rewritten as `i×n×s` substituting `a.length×s` for `S` and using `n` for `a.length`. This represents the number of elements in the previous rows (`i×n`) times the size of one element (s). For a lower-triangular matrix the rows are not all the same length so we need a different equation for the number of elements in the previous rows. The first row has 1 element, the second 2 and so on. The number of elements in the previous rows is the sum of the numbers from 1 to `i`. For example, for the third row (`i=2`), it is `1+2=3` or `i(i+1)/2`, in general. The second term of the usual equation (`j×s`) is the number of elements from the beginning of the row to the desired element times the size of one element. Since our rows all start at `j=0`, this is still the same. This gives us the modified mapping function, for valid `i` and `j`:

```
address(a[i][j]) = address(d) + i×(i+1)/2 × s + j × s
```

Again, we could define a class `LowerTriangMatrix` that provides this representation and methods similar to that in Figure 12.25. A similar, though a bit more complicated, mapping function and class can be derived for an upper-triangular matrix.

In Java we do have another option for a lower-triangular matrix. A ragged array with the i^{th} row having `i` elements would, of course, serve. There would be nothing extra required. Unfortunately, for an upper-triangular matrix, since the rows do not all start at `j=0`, the ragged array technique is not effective.

TRI-DIAGONAL MATRICES

Another common form of matrix is the **tri-diagonal matrix**. This is a matrix similar to the diagonal matrix, however it has non-zero elements not only on the main diagonal, but also on the diagonals above and below the main diagonal (Figure 1.29).

1	2	0	0	0
3	4	5	0	0
0	6	7	8	0
0	0	9	10	11
0	0	0	12	13

Figure 1.29 Tri-diagonal Matrix

Since the upper and lower diagonals have one less element than the main diagonal, an `n×n` tri-diagonal matrix has `3n−2` non-zero elements. Thus, for a `1000×1000` matrix `2998` out of `1,000,000` or 0.3% of the matrix is non-zero. The matrix can be represented by placing the `n` partial rows, without the zero elements, end-to-end consecutively into memory (Figure 1.30).

1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 1.30 Tri-diagonal Matrix as a Row

Essentially, the rows have been offset from each other by `2` positions and then merged with the non-zero elements taking precedence over the zero elements (Figure 1.31).

1	2	0	0	0								
		3	4	5	0	0						
				0	6	7	8	0				
						0	0	9	10	11		
								0	0	0	12	13

Figure 1.31 Merging Rows

This gives the mapping function, for valid indices `i` and j:

```
address(a[i][j]) = address(d) + i × 2 × s + j × s
```

That is, each row (`i`) starts 2 entries from the last and then each column (`j`) accounts for 1 entry. Again, a class such as Figure 1.25 could be defined to make it easier to use this representation.

SPARSE MATRICES

A final example of an array with a large proportion of zero entries is the **sparse matrix**. This is a matrix that is regular, but not necessarily square, in which a very high percentage of the elements are zero. However the non-zero elements are not distributed in any regular way. That is, the elements are “randomly” placed (Figure 1.32).

0	0	0	1	0
0	0	0	0	0
0	2	0	0	0
0	0	0	3	0
4	0	0	0	0

Figure 1.32 Sparse Matrix

Unfortunately a mapping function cannot be effective here, since such a function captures the regularity of occurrence of the elements and here there is no such regularity. The matrix can be represented in the usual way, if we can afford the space. However, there is an alternative representation using linked structures.

CASE STUDY: FINITE STATE MACHINE

A **Finite State Machine** (FSM) is a mathematical model of simple computation. It is defined as a computational device which at any given time is in a particular state called the **current state**. When it receives input (or when an event occurs), it transitions to another state depending only on the current state and the input (or event) that occurred. At the beginning of the computation, the FSM is in its initial or **start state**. The computation halts either when the input is exhausted or there is no transition defined for the input in the current state. When the process halts, if the input is exhausted and the machine is in one of its **goal**, **accepting** or **final states**, the computation was successful. FSMs can be used to model—or even implement—many processes such as the behavior of a subway turnstile or a vending machine, recognizing patterns and modeling neurological systems.

FSMs are typically described using finite state diagrams. A **finite state diagram** is a diagram consisting of circles representing states and arcs representing transitions from the state at the tail of the arc to the state at the head of the arc. The arcs are labeled by the inputs (events) for which the transition is defined. The start state is designated by an arc with no state at its tail and final states are marked with a double ring.

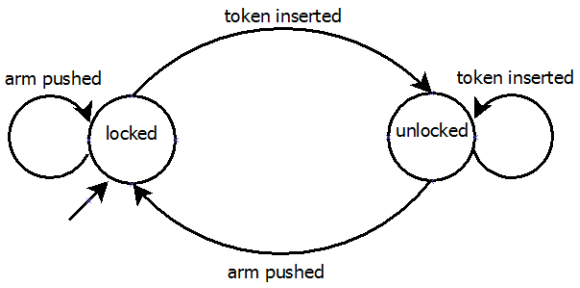


Figure 1.33 State Diagram for a Subway Turnstile

Figure 1.33 shows a state diagram for a simple FSM that models the behavior of a subway turnstile[1]. The turnstile is initially locked. Pushing on the locked turnstile has no effect. Inserting a token transitions to the unlocked state. Inserting a token in the unlocked state has no effect; however pushing on the turnstile allows the individual through and transitions to the locked state. In modeling this behavior, there is no end to the input since the turnstile is in service indefinitely and there is no final state.

An alternative specification for a finite state machine is a state transition table. A **state transition table** is a table listing the states across the top and the inputs (events) down the side with the intersection showing the result state (transition) for the input in the state. The start state is marked *. Figure 1.34 is a state transition table for the FSM in Figure 1.33.

state→ / event↓	locked*	unlocked
token	unlocked	unlocked
push	locked	locked

Figure 1.34 State Transition Table for a Subway Turnstile

FSMs are useful in describing or recognizing patterns. For example, consider a definition of a `double` literal. It is a sequence of characters beginning with an optional sign (+, -) and then one or more digits followed by a decimal point and zero or more digits. A finite state diagram describing this is shown in Figure 1.35. The states have been numbered with state `0` being the start state and state `3` the final (accepting) state (marked by double ring).

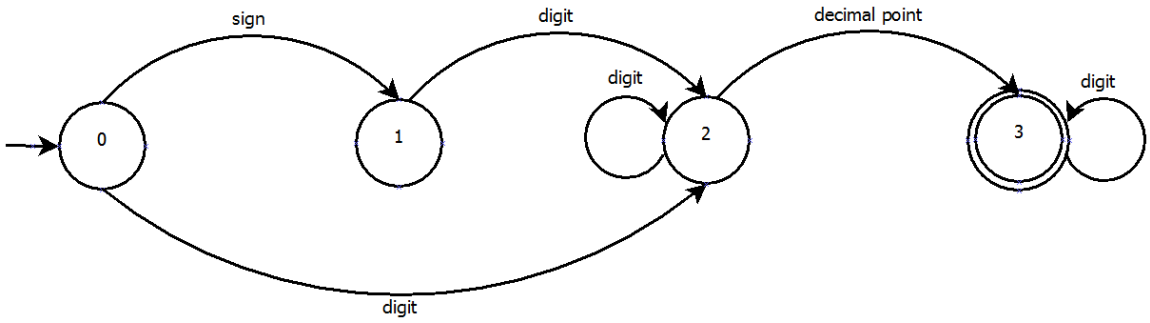


Figure 1.35 State Diagram – `double` Literal

[1] Koshy, Thomas; *Discrete Mathematics with Application;* Academic Press (2004); p. 762; ISBN 0124211801

Note that not all inputs have transitions in all states. For example, in state `1` only `digit` has a transition. Any other symbol encountered in state one would cause the machine to halt without reaching a final state (`3`).

The corresponding state transition table for the state diagram in Figure 1.35 could look like Figure 1.36. The final state (`3`) is marked with `†`. Instead of leaving blank the entries where no transition is possible (such as a `decimal point` in state `1`), the transition is marked with `E` indicating an error state.

	0*	1	2	3†
sign	1	E	E	E
digit	2	2	2	3
decimal point	E	E	3	E

Figure 1.36 State Transition Table – `double` Literal

A finite state machine can be used as a recognizer or validator for a pattern. For example, if a Java compiler was being written, it would have to recognize `double` literals within the program text. If there was an implementation of the finite state machine described in Figures 1.35 and 1.36, this could be used to recognize the literal. In fact the lexical symbols or tokens in a programming language—symbols such as identifier, `double` literal, left-parenthesis, etc. that make up a program—are typically defined by regular expressions and can be recognized by a FSM.

The implementation of a FSM can be based on an array representing the state transition table. The columns of the array are indexed by the current state and the rows indexed by the input symbol. The element type of the array is the new state. The states can be represented by integers (`int`) corresponding to state numbers (e.g. in Figure 1.36). Since the states are numbered from `0`, the error state (`E`) can be represented by `−1`. For pattern recognition in text (e.g. program text), the input symbols are characters (`char`). Since, in Java, `char` is automatically converted to `int`, the input symbols can be used directly to index the array. The array can have 128 rows (since standard Latin text characters are represented in 7-bit ASCII) and as many columns as there are states.

The algorithm records the current state (initially the start state) and repeatedly inputs an input symbol and uses the input symbol and current state to index the array for the new value for the current state—the transition. As long as there are additional input symbols and the error state has not been encountered, the process continues. When the process halts, if the current state is one of the final (accepting) states, the recognition was successful.

SUMMARY

An array represents a collection of data values or object references. The individual values are accessed via subscripting, and a subscripted variable may be used wherever a variable of the element type of the array may be used. Arrays allow a program to collect the data to be processed in one place and then process the data in non-sequential order. Arrays may be one-dimensional (e.g. a sequence, list, or vector) or two-dimensional (a table) or of even higher dimension.

In Java, an array is similar to an object in that an array variable is a reference variable, pointing to the actual array. Like objects, an array must be created using `new` and the size of the array must be supplied at creation time. Once created, an array's size is fixed. Array elements are indexed by integers with the first element indexed by 0. Arrays may be passed as method parameters and returned as method results, just as object types.

For one-dimensional arrays, two styles of array processing arise: right-sized arrays and variable-sized arrays. For right-sized arrays, the size must be known *a priori* or computable at the time of creation. In this case, the array is created with the required number of elements and processing uses the length attribute (`a.length`). If the size is not known at creation time, an array of "large enough" size is created and only the first part—elements from 0—is used to store data. An additional variable is used to record the number of elements in use in the array, and array processing is based on this variable. This technique is called a variable-sized array.

Similar to one-dimensional arrays, two-dimensional arrays can be right-sized or variable-sized. In a variable-sized array, the top left corner of the array is filled, that is, the rows from 0 and the columns from 0. Two additional variables record the number of rows and the number of columns occupied. For two-dimensional arrays, two processing patterns occur: row-major and column-major. In row-major processing, the elements of the array are processed row-wise, processing across row 0, then row 1, etc. In column-major processing, the elements are processed column-wise, down column 0, then column 1, etc.

Since computer memory is a one-dimensional sequence of bytes addressed by integers from 0, arrays must be mapped to memory locations. For one-dimensional arrays, a contiguous mapping is used where the consecutive elements follow each other in memory. For two- and higher-dimensional arrays both contiguous and non-contiguous or array-of-array mappings are possible. Contiguous mappings may be in row-major or column major order, however, row-major is most common. Java uses the array-of-arrays mapping.

Sometimes the standard mappings are not space efficient, especially if many of the array elements are no of interest or known to be zero, such as in diagonal and triangular matrices. In these cases it is possible to use different mapping functions to conserve space.

REVIEW QUESTIONS

1. T F In Java, the elements of the array must all be of the same type.

2. T F Summing the elements of an array involves a traversal.

3. T F A variable-sized array changes length to suit the amount of data being processed.

4. T F Array variables are reference variables.

5. T F Processing in row-major order is only possible in regular arrays.

6. T F Column-major order is also known as lexicographic order.

7. T F An array may not be returned by a function method.

8. T F Java uses array-of-arrays allocation for higher dimensional arrays.

9. T F A mapping function can be used to compress a sparse array.

10. In the following code, which is the last element of the array?

```
int[]  a;
a = new int[5];
```

a) `a[1]`
b) `a[5]`
c) `a[a.length]`
d) `a[4]`

11. In processing a right-sized array:

a) every element should contain data
b) the length attribute should be used for traversal
c) the length of the array must be known
d) all of the above

12. What is the value of `s` after the following code?

```
int   i;
int[] a[];
make(a,5);
s = 0;
for ( i=0 ; i<a.length ; i++ ) {
      s = s + a[i];
};
:
private void make ( int a[], int s ) {
      int   i;
      a = new int[s];
      for ( i=0 ; i<a.length ; i++ ) {
            a[i] = i;
      };
};    // make
```

a) 0
b) 5
c) 15
d) none of the above

13. Which of the following is the access pattern for lexicographic order?

    a) `a[1][2], a[1][1], a[1][0], a[0][2], a[0][1], a[0][0], …`
    b) `a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2], …`
    c) `a[0][0], a[1][1], a[2][2], a[3][3], a[4][4], a[5][5], …`
    d) `a[0][0], a[1][0], a[0][1], a[1][1], a[0][2], a[1][2], …`

14. If `a` is declared as an array of integers with 3 rows and 4 columns and is stored at address 1024, what is the address of `a[1][2]`, assuming zero-based subscripting, contiguous, row-major allocation and that an integer occupies 4 bytes?

    a) 1027
    b) 1036
    c) 1048
    d) 1060

15. When passing a variable-sized array as a parameter:

    a) the formal parameter must indicate the array length
    b) the array length is passed as an additional parameter
    c) the array length can be determined using the `length` attribute of the formal parameter
    d) none of the above

16. Which of the following is an appropriate mapping function for a tri-diagonal matrix?

    a) `address(a[i][i]) = address(d) + i × s`
    b) `address(a[i][j]) = address(d) + i×(i+1)/2 × s + j ×s`
    c) `address(a[i][j]) = address(d) + i × 2 × s + j ×s`
    d) none of the above

EXERCISES

1. Write a method that computes the dot product of two vectors stored in right-sized arrays. The method would have header:

```
private double dotProd ( double[] a, double[] b )
```

   The dot-product is the sum of the products of the corresponding elements of the vectors. For example if `a = {1,2,3,4}` and `b = {2,4,6,8}`, the dot product would be `60 = 1*2 + 2*4 + 3*6 + 4*8`. You may assume that the two arrays are the same size. Write a main class to test this method.

2. The sieve of Erostosthenes is an efficient process for determining all the prime numbers up to a specific limit. The process uses an array of `boolean` values, each indicating whether or not the element's index is a prime. For example, if the array is called `sieve`, then `sieve[2]` would be `true` since `2` is a prime while `sieve[4]` would be `false` since `4` is not a prime. The array is initialized so all elements are `true` then, starting from position `2`, the next `true` element is repeatedly located. The element at each multiple of this index value is set to `false`. Then the next `true` element is located and the multiples set to `false`. This process continues until the

search for `true` values reaches the half-way point in the array. The `true` elements indicate the primes.

Write a program that uses the sieve method to find and list (to an `ASCIIDisplayer`) all primes up to the limit entered by the user from and `ASCIIPrompter`.

3. At Broccoli University, many departments use multiple-choice tests for evaluation of students. Marking these by hand is tedious and error-prone so a computer program to perform this task is desired. The Computation Center has purchased a mark-sense form reader that will read answer sheets and produce a data file containing the students' answers. You have been contracted to produce the program that marks the tests and generates a report.

   A multiple-choice test consists of a number of questions for which responses are a choice from five possible answers (denoted: `A`, `B`, `C`, `D`, `E`). There is an answer key giving the correct responses. A student's mark is computed as the number correct minus 25% of the number incorrect and reported as a percentage by dividing by the number of questions.

   The mark-sense reader produces an `ASCIIDataFile` file consisting of one line for each student containing a student number followed by the letters corresponding to the responses on the form in tab-delimited format. For example, if there were five questions on the test, the line for one student from this file might be:

```
111111          A     B     C     D     E
```

   corresponding to student `111111` answering `A` to question 1, `B` to question 2 etc. At the beginning of the file is a line giving the number of questions on the test followed by the correct answers, again in tab-delimited format. For example,

```
5     A     B     C     D     E
```

   indicates that there are five questions and the correct answer for question 1 is `A`, for 2 is `B`, etc. You may assume that the responses are always one of `A`, `B`, `C`, `D`, or `E` and the number of responses given for the key and each student are correct.

   The program is to produce a report (to an `ASCIIReportFile`) that gives, for each student, the student number, answers and the mark. The mark is a percentage, but note that it could be negative. In addition, the report is to give a summary indicating the number of correct responses for each question—the number of students who answered correctly. The report is to also give the percentage correct—number correct divided by number of students. For example, the report might look like the following:

```
 St #   1   2   3   4   5  Mark(%)
111111 A   B   C   D   E   100.0
222222 A   E   C   E   E    50.0
333333 B   C   D   E   A   -25.0
#Cor     2   1   2   1   2
%Cor   66% 33% 66% 33% 66%
```

Note: In Java, the primitive type `char` can store a single ASCII character. A `char` literal has the form `'c'` where *c* is any ASCII character. `ASCIIDataFile` has a method `readChar` that reads a single `char` from the tab-delimited file. Arrays of `char` can be declared.

2 ANALYSIS OF ALGORITHMS

CHAPTER OBJECTIVES

- Understand that different algorithms for solving the same problem may differ in efficiency.
- Differentiate between the three common mechanisms for determining the efficiency of algorithms using asymptotic complexity: big-O, big-Ω, and big-Θ.
- Determine the big-O order of basic algorithms expressed in a programming language.
- Understand the significance of the complexity classes as determined by big-O order.
- Determine the performance of an algorithm empirically using algorithm timing.

In subsequent chapters of this book we will consider different representations for data and thus devise many algorithms to approach the same problem. How do we decide which algorithm—and hence which representation—to use? To make that decision we need a way of comparing algorithms. As we saw in Section 1.8, algorithms typically have to make a time-space tradeoff, that is, a faster algorithm can often be designed if more space is used to store the information, or a more compact representation may be chosen at the cost of slower processing. We need a way of estimating how long an algorithm will take to execute and how much memory it will require. If different algorithms are to be compared, this must be done in a general way, not just looking at a particular set of data. The study of algorithms and their complexity in time and space is called **Computability Theory** and particularly **Complexity Analysis**. We will, in a fairly informal way, consider time complexity. Many other sources (for example Knuth[2]) give more formal and detailed discussion of this topic.

2.1 ASYMPTOTIC COMPLEXITY

To compare algorithms that will be used on a variety of different data and run on a variety of different computers, we need a measure that is independent of the actual data and machine used. However, we know that the execution time of an algorithm will usually be affected by the amount of data—the "size" of the problem—so our measure will be dependent on the size of the problem. The common way to achieve machine independence is to count not the actual execution time, but rather the number of steps taken to complete the problem. Since not all steps are equal in time, the most significant step or sequence of steps is chosen for the count. For example, as we will see in Chapter 10, the significant step in searching is usually considered as the comparison of the key of the record with the key being searched for. Thus the analysis of search algorithms is done in terms of the number of key comparisons.

When counting the number of steps in an algorithm, the result is generally a polynomial function in `n`, the size of the problem. For example `n` would be the number of records in the collection to be searched when analyzing a search algorithm. The resulting function might be:

$$g(n) = n^2/2 + 5n/2 - 3$$

---

[2] Knuth, D. E; *Searching & Sorting: The Art of Computer Programming*; vol. 3; Addison-Wesley, Reading, MA; 1973.

Without substituting for n, how would we compare this function with some other function in n? One approach is to use an approximation for g(n) that approaches the actual value of g(n) as n increases in size. Such an approximation, when used as a measure of complexity, is called **asymptotic complexity**.

Consider Table 2.1 that shows the values of each of the terms of the polynomial g(n) for different values of n. When n is small, for example 0, the value of g(n) is the value of the lowest order term of the polynomial, −3. This term is really $-3n^0$, having the lowest power of n, namely 0. As n gets larger, for example 3, the value of the function is essentially the value of the second term, 5n/2. But as n continues to increase, the function value tends to the value of the first or highest order term. This is demonstrated by the last column, which is the quotient of the function value (g(n)) and the first term ($n^2/2$), and tends to 1.

n	$n^2/2$	5n/2	−3	$n^2/2+5n/2-3$ = g(n)	g(n) / ($n^2/2$)
0	0	0	−3	−3	∞
1	0.5	2.5	−3	0	0
3	4.5	7.5	−3	9	2
5	12.5	12.5	−3	22	1.76
10	50	25	−3	72	1.44
100	5000	250	−3	5247	1.0494
1000	500,000	2500	−3	502,497	1.004994
10,000	50,000,000	25,000	−3	50,024,997	1.00049994

Table 2.1 Convergence of Function to High-order Term

This pattern is true for any polynomial. As n gets large, the lower-order terms contribute, relatively, less and less to the function value and we say the function is dominated by the high-order term. Thus for large n, we can approximate the function by just the first term. In fact, the constant on the term (½ in the above) isn't even significant so the expression can be approximated by just n^2.

BIG-O NOTATION

The most commonly used notation for asymptotic complexity is the so-called **big-O notation**. The **order of an algorithm** (designated by O) is the order of the high-order term of the polynomial defining the number of steps in the algorithm. The algorithm whose defining function is g(n) above, thus has order n-squared or $O(n^2)$.

Strictly speaking, g(n) is O(f(n)) if, for some positive numbers c and N, $g(n) \leq cf(n)$ for all $n \geq N$. There are infinitely many choices for c and N as well as f(n). However, what the definition means is that if g(n) is O(f(n)) then, for reasonably large n, the function g(n) grows no faster than f(n).

Thus $f(n)$ is a worst case approximation for $g(n)$. Note that if $g(n)$ is $O(n^2)$, it is also $O(n^3)$ and $O(n^4)$, and so on. If we are going to do comparisons, we want to use the lowest-order such function.

BIG-Ω NOTATION

An alternative notation for asymptotic complexity is big-Ω notation (big omega notation). By definition, $g(n)$ is $\Omega(f(n))$ if, for some positive numbers c and N, $g(n) \geq cf(n)$ for all $n \geq N$. Note that this is just the same as big-O, except that the inequality is reversed. It means is that if $g(n)$ is $\Omega(f(n))$ then, for reasonably large n, the function $f(n)$ grows no faster than $g(n)$. Thus $f(n)$ is a best case approximation for $g(n)$. Again, there are infinitely many choices for c and N as well as $f(n)$ and we would want to use the highest-order function $f(n)$ for comparison. Note that if $g(n)$ is $O(f(n))$ then $f(n)$ is $\Omega(g(n))$.

BIG-Θ NOTATION

A final notation is big-Θ notation (big theta notation). By definition, $g(n)$ is $\Theta(f(n))$ if, for some positive numbers c_1, c_2 and N, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$. This means that, for large n, $g(n)$ and $f(n)$ grow at the same rate. It is also true that if $g(n)$ is $\Theta(f(n))$ then $g(n)$ is $O(f(n))$ and $g(n)$ is $\Omega(f(n))$ at the same time. Thus, if we can determine that lowest big-O for $g(n)$ is the same as the highest big-Ω for $g(n)$, we know the big-Θ for $g(n)$.

COMPARISON OF BIG-O, BIG-Ω AND BIG-Θ

Figure 2.1 shows the relationship between these three notations for asymptotic complexity. Here, $g(n)$ grows no faster than $f(n)$ so $g(n)$ is $O(f(n))$. Similarly, $g(n)$ grows at least as fast as $h(n)$ so $g(n)$ is $\Omega(h(n))$. If $f(n)$ and $h(n)$ were the same function f, $g(n)$ would be $\Theta(f(n))$.

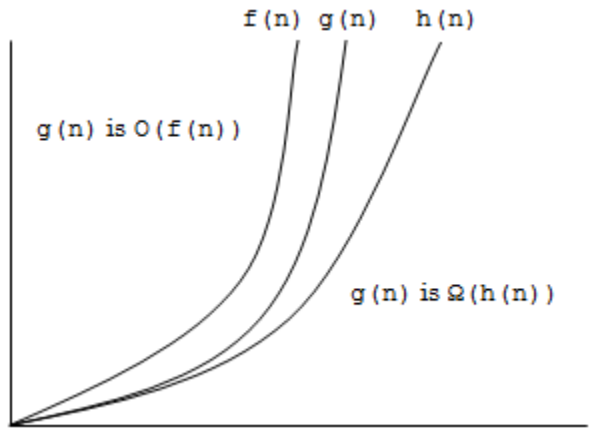


Figure 2.1 Big-O, Big-Ω and Big-Θ

Most often algorithm analysis is done using big-O. Since big-O represents a worst case—the performance of the algorithm is no worse than big-O—it gives us a conservative estimate for algorithm comparison. Of course, there are infinitely many functions for which any function is of that big-O, so our goal would be to determine the lowest order one we can, to make the estimate the best.

Finding big-Θ would give us a tightest estimate, but it usually isn't necessary and is often more difficult to find. We will restrict our discussion in this, and subsequent chapters, to using big-O, and use the term the order of the algorithm, to mean big-O.

2.2 COMPLEXITY CLASSES

Algorithms can be classified by their order into complexity classes (Table 2.2). Algorithms in the same complexity class have comparable behavior. For example, if the problem size (n) for a linear algorithm doubles, the time doubles, while for quadratic algorithms, the time quadruples. The growth rates of the various complexity classes are shown in Figure 2.2. Note that, as the order gets higher, the curve rises more quickly. Since the function is an estimate of the time taken by the algorithm, the more quickly the curve rises, the smaller the problem size (n) that can be handled in reasonable time. Algorithms above $O(n^3)$ are very slow and those of $O(2^n)$ and higher are only tractable for small n. Actually, these algorithms' functions are not even represented by polynomials and are thus called non-polynomial or n-p algorithms.

class	order
constant	$O(1)$
logarithmic	$O(\lg n)$
linear	$O(n)$
n log n	$O(n \lg n)$
quadratic	$O(n^2)$
cubic	$O(n^3)$
	$O(n^4)$
	$O(2^n)$

Table 2.2 Complexity classes

DETERMINING COMPLEXITY CLASS

How do we determine the order of an algorithm? Generally it requires rigorous mathematical analysis. However, for a given algorithm expressed in a programming language, we can estimate the order fairly easily. Consider the following piece of code:

```
some other steps
the significant step
some other steps
```

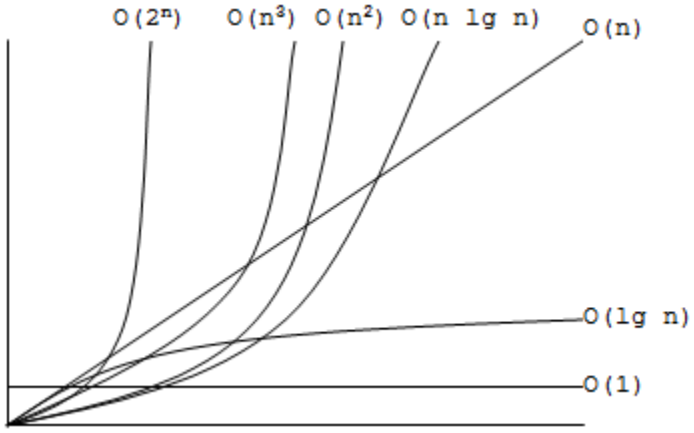


Figure 2.2 Orders of Algorithms

Here, the first and last steps do not involve the significant step we are counting. Clearly, this code has constant time (`O(1)`). That is, the process is independent of the size of the problem. An example of an `O(1)` algorithm would be accessing the first item in a collection, such as the first element of an array or the first record in a file. The process is independent of the size of the array or file.

Now consider:

```
for ( int i=0 ; i<n ; i++ ) {
    some other steps
    the significant step
    some other steps
};
```

Each time the loop is executed, the significant step is executed once. Since the number of times the loop executes is proportional to `n`, the number of times the significant step is executed is proportional to `n`, so the code is linear (`O(n)`). This gives us the **nesting rule**: if a piece of code of order `O(q)` is nested within a loop of order `O(p)`, the order of the resulting code is `O(p×q)`. In this case the nested code was `O(1)` and the loop `O(n)`, so we have `O(1×n)` or `O(n)`.

Finally consider:

```
some other steps
the significant step
some other steps
for ( int i=0 ; i<n ; i++ ) {
    some other steps
    the significant step
    some other steps
};
```

Here the total number of steps is clearly the number of steps before the loop plus the number of steps represented by the loop. In this case, `1+n`. Since only the highest order term is significant, the order would be `O(n)`. In general, if we have two consecutive pieces of code, the resulting polynomial would have terms of the first part added to the terms of the second part. In such a case, the resulting

polynomial would have, as the highest-order term, the highest order of each of the two parts. This is gives us the **sequence rule**: if a piece of code of order `O(p)` is followed by a piece of code of order `O(q)`, the resulting code is of order `O(max(p,q))`. In this case the first is `O(1)` and the second `O(n)` so the result is `O(max(1,n))` or `O(n)`.

The only control structures that affect the order are sequences and loops. A decision structure such as an if simply provides alternate paths in the algorithm. The order of the whole is the maximum of the parts (just like the sequence rule) since we are looking for "worst case". Similarly, a method call simply represents the execution of the method, so it is of the order of the method itself.

The order of a loop is the number of times its body is executed, proportional to `n`. We will consider only for loops, however, other loops can also be analyzed. If neither of the bounds, that is the initial value or the value in the test, of the loop, nor the increment, are a function of `n`, the loop is `O(1)`. That is, the number of times it is executed is independent of `n`. If one bound is a function of `n`, for example, `n-1`, `2×n` or n^2, and the other and the increment are independent of `n`, the loop is the order of this function. Other cases require careful analysis. For example:

```
for ( int i=5 ; i<2*n ; i=i+5 ) {
        :
};
```

is `O(n)` since the initial and increment values are independent of `n` and the test is a linear function of `n`. Now:

```
for ( int i=n*n ; i>0 ; i-- ) {
        :
};
```

is $O(n^2)$ since the test and increment are independent of `n` and the initial value is a quadratic function of `n`. Finally:

```
for ( int i=1 ; i<n ; i=i*2 ) {
        :
};
```

is `O(lg n)`. In this case, although the initial value is independent of `n` and the test is a linear function of `n`, the increment part is not an increment at all. If we consider the values `i` will assume we have: `1`, `2`, `4`, `8`, `16`, `32`, …—consecutive powers of `2`. The loop will stop at the first power of `2` that is `>=n`. This value is $2^{\log_2 n}$ and the number of times through the loop is $\log_2 n$. That is, the first time through the loop `i` is 2^0, the second 2^1, the third 2^2 and the last $2^{\log_2 n-1}$. Thus the loop is `O(lg n)`. Note that the base of the log is irrelevant. Functions involving any base of a log are of the same order.

The order of an algorithm can be estimated by considering each loop containing a significant step either directly or indirectly via a method call, determining its order and applying the nesting and sequence rules.

2.3 TIMING ALGORITHMS

While the order of the algorithms can serve to differentiate between algorithms of different order, there will be cases when there are a variety of algorithms to choose from that are of the same order. Other times we may be able to implement the same algorithm in a number of ways that are of the same order and want to gain some idea about their relative efficiency. In these cases, we may wish to perform a timing test on the implementation of these algorithms themselves.

What we desire is a measure of the amount of time the algorithm takes to complete, however this clearly depends on the data used. Certain sets of data may favor certain implementations. To make the timing test effective, we need to choose a representative set of data and probably run the test on more than one set of data. The representative set should be representative of the live data, if we know what it is likely to look like. Otherwise randomly generated data can be used.

To time the algorithm, we need a way of determining the current time before and after the algorithm is executed, and subtract the two to get the elapsed time. In Java, the `System` class provides a method `currentTimeMillis` that returns the current time as known by the computer in milliseconds. You will remember that the `System` class provides a number of useful facilities, including the stream `out` that allows display of debugging information on the system console. This is just another of these features.

The basic mechanism is to include the piece of code to be tested between two calls to `currentTimeMillis` and subtract the two values to get the elapsed time in milliseconds. The pattern would be:

```
long        t1, t2, elapsed;
  :
t1 = System.currentTimeMillis();
code to be timed here
t2 = System.currentTimeMillis();
elapsed = t2 - t1;
```

A program was written to demonstrate the timing and behaviour of algorithms of different order. The program has four methods: `lgNMethod`, `linearMethod`, `nLgNMethod` and `quadraticMethod` representing `O(lg n)`, `O(n)`, `O(n lg n)` and $O(n^2)$ algorithms, respectively. Figure 2.3 shows one such method. Each of these methods simply repeats a step an appropriate number of times. Since they are not real algorithms, but rather just examples, the step is replaced by a statement that simply delays the program execution some amount of time (`DELAY`). The details of this statement are beyond the scope of the text.

```
137     private void nLgNMethod ( int n ) {
138
139         int  i, j;
140
141         for ( i=1 ; i<=n ; i++ ) {
142             for ( j=1 ; j<=n ; j=j*2 ) {
143                 try { Thread.sleep(DELAY); } catch ( InterruptedException e ) { };
144             };
145         };
146
147     }; // nLgNMethod
```

Figure 2.3 Example—`O(n lg n)` Algorithm to be Timed

The constructor times repeated calls (`NUM_TESTS`) of each of the four methods for a given problem size (`SIZE`) as shown in Figure 2.4. It then repeats the process using a larger problem size (increased by `FACTOR`). A table of results is produced.

```
59          t1 = System.currentTimeMillis();
60          for ( int i=1 ; i<=NUM_TESTS ; i++ ) {
61              nLgNMethod(SIZE);
62          };
63          t2 = System.currentTimeMillis();
64          out.writeLong(t2-t1,8);
```

Figure 2.4 Example—Timing the `O(n lg n)` Algorithm

The results are shown in Figure 2.5. The column for `O(n)` shows a approximate doubling of time as the problem size doubles. The column for $O(n^2)$ shows a approximate four-fold increase when the problem size doubles. The column for `O(lg n)` shows an increase less than that for `O(n)` and the column for `O(n lg n)` shows an increase between that for `O(n)` and $O(n^2)$. These results seem to be consistent with the expected performance.

Care must be taken in producing timings of algorithms. Firstly, consider that computers are relatively fast. Even if the algorithm being measured is of higher order, it might complete quite quickly. It might be necessary to either use a large `n` or repeat the algorithm a number of times to make the results significant. Secondly, the timing method (`currentTimeMillis`) has a measuring accuracy. If the error in time measurement is large compared to the timing of the algorithm, the results will be meaningless. Thirdly, consider that a method call involves some execution time. If the method call is measured along with the algorithm being timed, it contributes to the time being measured. If the algorithm fast, the method call overhead might dominate the timing. An alternative would be to do the timing within the method and have the method return, as its result, the time the algorithm took, which can be accumulated outside of the method.

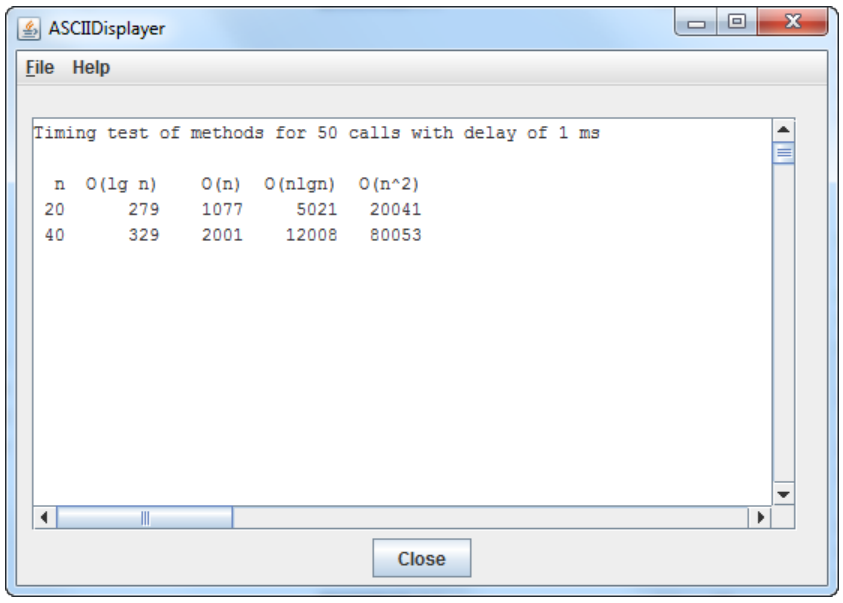


Figure 2.5 Results of Timing Test

SUMMARY

In the design of a solution to a problem, there are often different representations and algorithms that can be used. The choice of the representation and/or algorithm is likely to be based, at least in part, on the efficiency of the representation in space and the algorithm in time. Often this represents a trade-off.

To compare two algorithms, we can analyze them to determine the number of steps they take relative to the problem size. Using `big-O` analysis, we determine a simple function in `n`—the problem size—that serves as an upper bound on the actual number of steps the algorithm will take. This function classifies the function into one of the complexity classes designated as `O(1)`, `O(lg n)`, `O(n)`, `O(n lg n)`, `O(n`^{2}`)`, etc. For large `n`, algorithms of one order are faster than algorithms of a higher order.

To compare algorithms of the same order, or to compare algorithms for which we are unable to determine a good bound on the order, we can resort to timing tests. In a timing test, we measure the actual time it takes the different algorithms to perform the process on selected, or random data. Care must be taken to ensure that we are measuring the actual cost of the algorithms and that the measuring accuracy and other overhead does not distort the results.

REVIEW QUESTIONS

1. T F Asymptotic complexity is valid only for large `n`.

2. T F The sequence rule states that if two pieces of code of `O(p)` and `O(q)` follow each other, the order of the result is `O(p+q)`.

3. T F Non-polynomial algorithms are only effective for small `n`.

4. T F The big-Θ of an algorithm is always of lower order than the big-O.

5. T F The method `System.currentTimeMillis` returns the elapsed time in milliseconds.

6. Which of the following shows the complexity classes in increasing order of complexity for linear (L), quadratic (Q), cubic (C) and logarithmic (Lg)?

   a) L, Q, C, Lg
   b) C, Q, Lg, L
   c) L, Lg, Q, C
   d) Lg, L, Q, C

7. The order of the following code is:

```
for ( int i=0 ; i<2*n ; i=i+2 ) {
        :
};
for ( int i=1 ; i<n ; i=i*2 ) {
        :
};
```

   a) O(lg n)
   b) O(n)
   c) O(n lg n)
   d) $O(n^2)$

8. The order of the following code is:

```
for ( int i=1 ; i<n ; i++ ) {
      for ( int j=i ; j>0 ; j=j/2 ) {
              :
      };
};
```

   a) O(lg n)
   b) O(n)
   c) O(n lg n)
   d) $O(n^2)$

9. The order of the following code is:

```
for ( int i=0 ; i<n*n ; i=i+2 ) {
      :
};
j = n;
while ( j > 0 ) {
      :
    j = j/3;
};
```

a) O(lg n)
b) O(n)
c) O(n lg n)
d) $O(n^2)$

10. The algorithm for computing the average mark for students in a course would be:

a) O(lg n)
b) O(n)
c) O(n lg n)
d) $O(n^2)$

3 LINEAR LINKED STRUCTURES

CHAPTER OBJECTIVES

- Recognize the limitations of static data structures.
- Explain the advantages and disadvantages of dynamic data structures.
- Describe the representation of sequentially-linked structures.
- Explain the fundamental operations on sequentially-linked structures.
- Apply sequentially-linked structures in a problem.
- Describe the variations of sequentially linked structures such as circular structures and header and sentinel nodes, and situations where they might profitably be used.
- Describe other linear linked structures such as symmetrically-linked structures, list-of-lists and multi-lists, and situations where they might profitably be used.

Much of the data processed in computer systems is dynamic in nature. Students register and leave the university. Employees are hired and retire. Customers make additional purchases and make payments on their credit cards. When a computer system has to keep track of such information over time, the amount of data maintained changes.

An array is known as a **static data structure**—once created, the amount of information it can hold is fixed or unchangeable. We adapted arrays to suit dynamic situations using the variable-sized array technique. However, the array itself does not actually change in size, we just change which part of the array we are using. This comes at a cost. When the array is created with 1000 entries and we only use 100 of them, we are wasting 900 entries. In Java where array elements are usually references to objects, this isn't too bad we only waste 3600 bytes. However in many languages the array elements are the entities themselves and the waste is much higher. For example if we are storing student records, the waste would be 900 × the storage for a student record. Even in Java if the disparity between the created size of the array and amount used is large, there is much waste.

There is a second problem with static data structures. Rearrangement of items within the structure is generally costly, at least $O(n)$. Consider if we wish to add an item to the front of an array. To maintain the relative ordering of the items, we have to move all the other items over to make room. This is an $O(n)$ operation. Similarly if we wish to remove an item from an arbitrary position within the array, unless we are willing to leave a "hole" we have to move about half or $O(n)$ items over to fill the gap.

A **dynamic data structure** is one that can change in size over time. Unlike a static data structure, the amount of space used by a dynamic data structure is proportional to the number of entities it holds. That is, if there is little information in the structure little storage is used. Similarly dynamic structures generally allow addition and deletion of items at arbitrary locations without moving items. Once the position is located these operations can be $O(1)$.

The most common dynamic data structures are **linked structures**. A linked structure is one in which items—typically called **nodes**—are connected together by references—also called pointers or **links**. The nodes themselves are created as needed when a new entity is to be added to the structure, and

are recovered (garbage collected), when the entity they represent is no longer needed. This means that the number of nodes is proportional to the number of entities.

In an array, proximity in memory indicates the ordering. That is, `a[i]` and `a[i+1]` are considered to be neighbors in the ordering because they are neighbors in memory. This is an advantage since, as we saw in Section 1.7, random element access is possible involving only a simple computation using a mapping function. In linked structures, instead of relying on proximity in memory to indicate ordering, a reference indicates the next node in the ordering. Since it is relatively inexpensive to change a reference, as opposed to moving entire entities around within the array, linked structures are dynamic at sub-linear cost. However, the ability to use a mapping function is lost and so linked structures are not accessible randomly.

Of the linked structures, **linear linked structures** and trees are the most common. This chapter discusses linear linked structures, often called linked-lists or lists. We will use the term linear-linked structures to discuss the data structure to avoid confusion with the ADT called a list. We will not discuss trees in this text.

3.1 SEQUENTIALLY-LINKED STRUCTURES

A **sequentially-linked structure** consists of a series of nodes, each node containing a reference to the next or successor node in the sequence (see Figure 3.1). The first node in the sequence is indicated by a reference, typically stored in a variable, say `list`. The last node in the sequence has no successor, so its successor reference is `null`. This is indicated in the figure by a slash. Note that the successor references are explicit unlike in arrays where the successor is implicit.

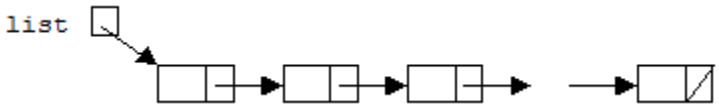


Figure 3.1 Sequentially-linked Structure

Each node in the structure represents a single item in the collection, for example, a student within a course. A sequentially-linked structure may consist of any number of such nodes. If the structure is empty the structure reference—the variable referencing the structure—is `null`.

REPRESENTATION

In Java, objects may be created dynamically and referenced via references. This means that the nodes of a linked structure should be objects. Since the items that will be in the list—for example, students—are likely going to be represented by objects as well, it would appear that these could be the nodes. For example, to create a sequentially-linked structure of students to represent a course, we could add a field `nextStudent` to the `Student` class and use it to link together the students:

```
public class Student {
      :
    private Student  nextStudent;  // next student in course
      :
}  // Student
```

This has two serious drawbacks. It requires that the `Student` class specification "knows" that it can be part of a course since it must contain the next student reference. Although this might in some cases be appropriate, consider that a student may be part of a number of courses, requiring a number of such fields. The second drawback is that it requires a change to the `Student` class to allow a linked structure to be used. Since the `Student` class is probably fundamental to the university's information system, it is undesirable to require the class to be changed just because for one system the students are to be placed in a linked structure as opposed to, say, an array. Any change to the `Student` class may require recompilation of many other systems and, of course, any bug introduced by the change would be propagated to all these systems as well.

THE `Node` WRAPPER CLASS. A better technique is to use a separate object for the node and have it reference the `Student` object. Such a class, defined solely to add functionality to or change the type of another class without modifying that class, is called a **wrapper class** or sometimes a **mixin**. A node wrapper class for a `Student` class to allow it to be placed on a sequentially-linked structure is shown in Figure 3.2.

```
class Node {

    public Student  item;  // the wrapped student
    public Node     next;  // next node in sequence

    public Node ( Student i, Node n ) {
        item = i;
        next = n;
    };  // constructor

}  // Node
```

Figure 3.2 `Node` wrapper class

The `Node` class has two fields: a reference to a `Student` object—the "wrapped" student—and a reference to another `Node`—the next node in the sequence. The constructor creates a new `Node`, filling in the fields. Note that the `Node` class itself isn't declared `public`. The expectation is that the class would be part of a package that manipulates `Student` objects in a linked structure. As such, the `Node` class would only be known to those classes, since it is for implementation only. Note also that the fields of the class are `public` instead of the usual `private`. Since the `Node` class isn't `public`, only classes in the package can reference it and its `public` fields. Making the fields `public` in this context means that accessor and updater functions are not required since the fields can be accessed directly. This makes working with the wrapper class easier.

Figure 3.3 shows the structure that would connect a number of `Student` objects into a sequentially-linked structure. Each `Node` object references a `Student` of the collection. The `Node` objects also reference the next `Node` in the sequence. The `Student` objects are referenced indirectly via the `Node` objects. For example, if `p` is a `Node` reference variable declared as `Node p`, then `p.item` refers to the `Student` object wrapped by the `Node` and `p.next` refers to the subsequent `Node` in the structure. It is tedious to draw a diagram such as in Figure 3.3. In terms of manipulating linked structures the content—the `Student` in this case—is irrelevant. Thus when we are describing the

algorithms for list manipulation, we will use a diagram such as in Figure 3.1 instead, recognizing that what we really mean is what is shown in Figure 3.3.

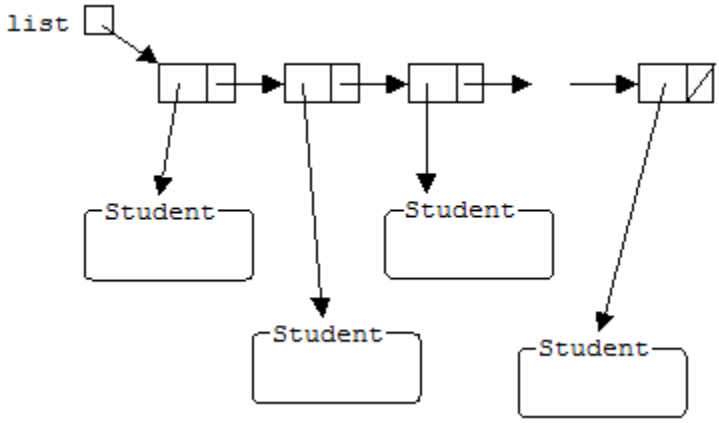


Figure 3.3 Sequentially-linked Structure using a Wrapper Class

Like any object, a linked structure must be referenced by some reference variable such as `list`. Since the object being referenced is a `Node` object, `list` is of type `Node`. Since a sequentially-linked structure may consist of any number of nodes, there is the possibility that the structure will be empty. The structure reference variable normally references the first node. If the structure is empty there is no first node, therefore the structure reference variable should be `null`. Since a sequentially-linked structure is dynamic, its size (number of nodes) will grow and shrink. Initially, there will be no nodes, so the initial state of a sequentially-linked structure is empty. Thus the structure must be initialized via:

```
list = null;
```

before any other operation is performed. This initial or empty state is drawn as in Figure 3.4.

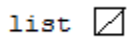


Figure 3.4 Sequentially-linked Structure: Initial (empty) State

OPERATIONS

As with any data structure, there are a number of common operations that are performed with sequentially-linked structures. In this section we will consider these operations in general, without considering the object type—such as `Student`—in the collection or the actual application—such as representing a class list.

A sequentially-linked structure is a collection of entities, which starts out empty and grows and shrinks as the application requires. Clearly two of the operations are the addition of a new entity to the collection (called **insertion**) and removal of an existing item from the collection (called **deletion**). The collection exists for some reason within the application. Commonly it is necessary to process all or most of the items in the collection, for example to compute final marks of all students in

the course. This operation is called **traversal** and is similar to traversal of arrays (see Section 1.2). Finally, it is often necessary to locate a particular entity within the collection. This operation is called searching and will be covered in Chapter 10.

INSERTION AT THE FRONT. A sequentially-linked structure begins in the empty state (Figure 3.4). Items are inserted one at a time into the structure so that it represents a collection. If we consider the general case (Figure 3.1), we see a number of options for insertion: at the front of the sequence, at the end of the sequence or within the sequence according to some criterion, for example in sorted order. We will consider insertion at the front of the sequence first, as it is the easiest.

Insertion at the front requires that the node referencing the inserted item become the first node—the one referenced by the structure reference variable `list`. At the same time the node referencing the original first item should become the second, referenced by the new first node. The algorithm is given in Figure 3.5 where *`item`* is the reference to the item being inserted.

```
list = new Node(item,list);
```

Figure 3.5 Algorithm for Insertion at the Front of a Sequentially-linked Structure

Here a new `Node` is created which references the item being added to the collection and the prior first node in the collection, previously referenced by `list`. The structure reference variable `list` is changed to refer to the new first node. Figure 3.6 demonstrates this operation. New nodes and links are drawn using coarse dashed lines. Prior values are drawn with finely dashed lines. The algorithm also works for an empty structure—where list is `null`. The result is a new `Node` that references the item and has a `null` successor pointer. It is the only node in the list. Clearly this algorithm is `O(1)` as it is independent of the length of the list (`n`). Note that repeated use of this algorithm for insertion builds the collection with the items in the reverse order of their insertion. Try it!

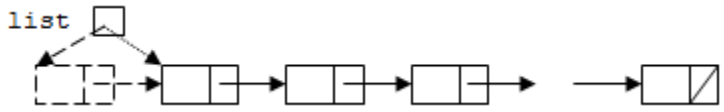


Figure 3.6 Insertion at the Front of Sequentially-linked Structure

DELETION AT THE FRONT. Now let us consider deletion of an item. Again there are a number of possibilities including: deletion of the first entity, deletion of the last entity and deletion of an entity matching some criterion, such as a given student number. In any event, deletion is not always possible. If the structure is empty, or the item cannot be located, the deletion will fail. At this point, we will simply indicate failure. In Chapter 4 we will see a mechanism in Java for signaling failure—exceptions.

Again we will consider the easiest first: deletion of the first item. When the first entity in the structure is deleted, the result must be that the second entity becomes the first. Thus the structure reference (`list`) must reference the second node in the structure. What becomes of the first node and item? Clearly the node is no longer of use, so it should be a candidate for garbage collection. This means it should no longer be referenced by any variable. What happens to the item depends on the application. Likely it is still of some use within the application—for example, to be added to some other structure. We will store a reference to it in the variable labeled `item` in the algorithm. The

algorithm is shown in Figure 3.7. If the goal is simply to remove the reference to the item and the item might become garbage, assignment to `item` is omitted.

```
if ( list == null ) {
    delete fails
}
else {
    item = list.item;
    list = list.next;
};
```

Figure 3.7 Algorithm for Deletion at the Front of a Sequentially-linked Structure

Figure 3.8 shows this operation. Again, previous links and nodes are drawn in fine dashed lines and new values in coarse dashed lines. This pattern works for all structures, including the case where the structure contains only one item. In deletion with one item, `list` will become `null` since the `next` field of the first and only node is `null`. `list==null` indicates that the structure is empty, the desired result. The algorithm is again `O(1)` since it is independent of the length of the list.

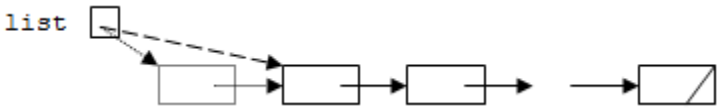


Figure 3.8 Deletion at the Front of a Sequentially-linked Structure

TRAVERSAL. Once the structure has been filled with some items, it may be traversed. The goal of a traversal algorithm is to perform some operation on each item within the structure. The nature of a sequentially-linked structure is sequential since the only way to get to the n^{th} entity is by processing the previous `n-1` entities first. This is much the same as processing records in a sequential file or an array traversal. Thus a traversal algorithm will process the items in sequential order—the order they occur within the structure.

Traversal begins at the first node of the list. Since we don't want to modify the structure reference `list` and lose access to the front of the list, a temporary variable (`Node p`), called a **traveling pointer** will be used to refer to the current node. This variable must sequence through all the nodes, starting with the first (`list`). To get to the next node from the current one, `p` is updated to `p.next`. When must the algorithm stop? Clearly it must process the last entity, whose `next` reference is `null`. After processing this node, `p` will be updated to `null`. This then is the condition on which to terminate. The algorithm is shown in Figure 3.9.

```
p = list;
while ( p != null ) {
    process p.item
    p = p.next;
};
```

Figure 3.9 Algorithm for Traversal of a Sequentially-linked Structure

Since the number of times through the loop is equivalent to the number of entities in the list, the algorithm is `O(n)`. Note that this algorithm performs correctly even if the list is empty. In this case, `p` is `null` from the start and the loop is not executed, processing the zero items.

It is interesting to compare this algorithm with that for array traversal (Figure 1.4). If we substitute the equivalent while loop for the for loop in Figure 1.4, we get the comparison in Figure 3.10.

```
p = list;                          i = 0;
while ( p != null ) {              while ( i < a.length ) {
    process p.item;                    process a[i]
    p = p.next;                        i = i + 1;
};                                 };
```

Figure 3.10 Comparison of Traversal Algorithms

This indicates a general algorithm for sequential traversal of a structure as shown in Figure 3.11.

```
start at first entry;
while ( more entries ) {
    process entry;
    on to next entry;
};
```

Figure 3.11 Generalized Sequential Traversal Algorithm

INSERTION AT THE END. As we have seen, repeated insertion at the front of a sequentially-linked structure leads to the items being in reverse order. To have them in the order of insertion, we need to perform the insertion at the end of the list. Unfortunately, a sequentially-linked structure only has a reference to the first node since all others are reachable from there via traversal. This means that we have to locate the end of the list before the insertion can occur. The last node in the list is the one whose successor reference is `null`. In the traversal algorithm, we terminated the loop when the traveling pointer (`p`) became `null`—falling off the end of the list. We can use a traversal here, as long as we can remember which node was the last one visited. To do this we maintain a pair of traveling pointers, `p` and `q`, where `q` always references `p`'s predecessor. Initially, `p` references the first node. Since this node has no predecessor, the appropriate initial value for `q` is `null`. The algorithm is shown in Figure 3.12.

```
q = null;
p = list;
while ( p != null ) {
    q = p;
    p = p.next;
};
if ( q == null ) {
    list = new Node(item,null);
}
else {
    q.next = new Node(item,null);
};
```

Figure 3.12 Algorithm for Insertion at the End of a Sequentially-linked Structure

The loop is a traversal (Figure 3.9) with no processing except updating `q`. In order to maintain the relationship between `p` and `q`, `q` is updated to `p` before `p` moves to the next node. When the loop terminates, `p` is `null` and there are two possibilities. If `q` is `null` then there was no initial node—the list was empty—so insertion is the same as at the front of the list (see Figure 3.5), except that this is also the end of the list so the `next` reference is `null`. If `q` is not `null`, `q` references the last node in the list—the one with the `null next` reference from which `p` became `null`—and the new node is created and linked to that node as shown in Figure 3.13. Since the algorithm contains a loop which sequences through all the entities in the list, the entire algorithm is `O(n)`.

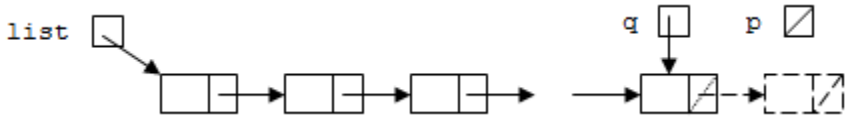


Figure 3.13 Insertion at the End of a Sequentially-linked Structure

DELETION AT THE END. Deletion of the last node in a sequentially-linked structure requires a similar arrangement. To delete the last node, it must first be found. However, in deleting the last node of the list, the second last node needs to be modified. It becomes the last node, so its successor reference must be `null`. This means that we need a reference to the last and second last nodes. This can still be done with two traveling pointers if we consider that there must be at least one node in the list if deletion is to occur at all. That being the case, `p` can never be `null` at the start and it is legitimate to use `p.next == null` rather than `p == null` as the termination condition—terminating one node earlier than the regular traversal. The algorithm is shown in Figure 3.14.

```
if ( list == null ) {
    delete fails
}
else {
    q = null;
    p = list;
    while ( p.next != null ) {
        q = p;
        p = p.next;
    };
    item = p.item;
    if ( q == null ) {
        list = null;
    }
    else {
        q.next = null;
    };
};
```

Figure 3.14 Algorithm for Deletion at the End of a Sequentially-linked Structure

Since the loop terminates when `p.next == null`, `p` references the last node and `q` the second last. There are now two cases. If `q` is `null`, there is no predecessor to `p` and `p` is the first, last and only node. Therefore the list becomes empty by setting `list` to `null`. Otherwise there are at least two nodes in the list, and `q` is the second last one. `q`'s successor reference must be set to `null`. This is shown in Figure 3.15. Again, since the loop traverses through all the nodes to locate the end of the structure, the algorithm is `O(n)`.

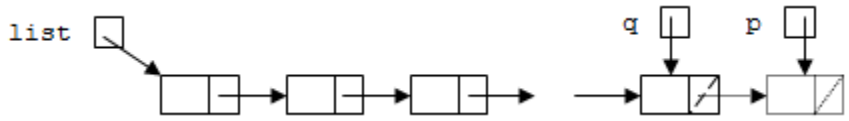


Figure 3.15 Deletion at the End of a Sequentially-linked Structure

INSERTION IN SORTED ORDER. It is sometimes necessary to maintain the structure in some particular order, for example sorted by key. Ordering a sequentially-linked structure after it is created is a very messy proposition, so it is much better to build it in order. This requires an insertion algorithm that inserts items at arbitrary positions, as opposed to at the front or rear, according to sorted order.

The algorithm in Figure 3.14 shows sorted insertion in ascending order by key. Changing the loop termination condition could support other orders or insertion criteria. The algorithm first searches for the insertion position—the loop—and then does the actual insertion. The search is for the first item whose key is greater than the key of the entity to be inserted. The insertion then occurs between that item's predecessor and the item itself. Since that item is the first one with a key greater than the one being inserted, the predecessor must have a key less than or equal to the insertion key so this is the insertion point.

```
q = null;
p = list;
while ( p != null &&
        insertion key is greater or equal to key of p.item ) {
    q = p;
    p = p.next;
};
if ( q == null ) {
    list = new Node(item,p);
}
else {
    q.next = new Node(item,p);
};
```

Figure 3.16 Algorithm for Sorted Insertion into a Sequentially-linked Structure

Again, a pair of traveling pointers is used with `p` referencing the node and `q` its predecessor. The insertion will be between q and `p`. If no item has a key greater than the insertion key, the loop will terminate with `p == null` and `q` will point to the last node. Otherwise the loop will terminate at the first item with key greater than the insertion key. Note the careful ordering of the conditions. If `p` is `null`, we cannot access `p.item`.

After the search loop, there are two possibilities. If `q` is `null`, there is no predecessor to `p` and the new item has a key less than all others. It is inserted at the front of the list (as in Figure 3.6). If not, `q` references the predecessor and `p` the successor, and the insertion occurs between them. This is shown in Figure 3.17. Note that this code works equally well if the insertion key is the largest and insertion is at the end of the list. Since `p` would be `null`, the new node will have a `null` successor reference, marking it as the last node. Since the loop will on average traverse half the list, the algorithm is `O(n)`.

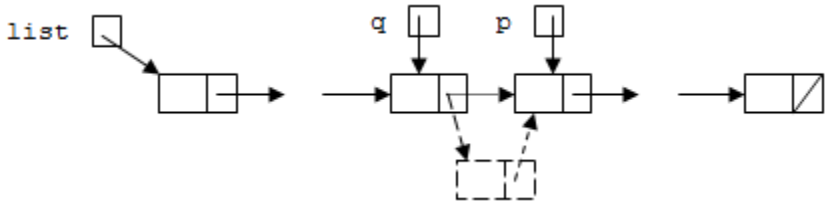


Figure 3.17 Sorted Insertion into a Sequentially-linked Structure

DELETION OF A NODE BY KEY. Finally we will consider deletion of an arbitrary item within the structure. There will be a criterion to determine which item to delete, such as matching a particular key. If the item cannot be located, an error results; otherwise, the item is deleted. To delete the item, it must first be located and a reference to its predecessor found. Like insertion in sorted order, this involves a traversal with a pair of traveling pointers.

Figure 3.18 gives the algorithm for deletion of the item matching the search key. Other criteria could be substituted for the loop termination condition. Again, the termination condition of the search loop is carefully crafted to ensure that upon reaching the end of the list—`p==null` and the key has not

been found—the algorithm doesn't attempt to access `p.item`. Not finding the item could be considered an error as indicated below. However a common variation of the algorithm deletes the item only if it is present. In this case, nothing need be done when `p==null`.

```
q = null;
p = list;
while ( p != null &&
        deletion key not equal to key of p.item ) {
    q = p;
    p = p.next;
};
if ( p == null ) {
    delete fails
}
else {
    entity = p.item;
    if ( q == null ) {
        list = p.next;
    }
    else {
        q.next = p.next;
    };
};
```

Figure 3.18 Algorithm for Keyed Deletion from a Sequentially-linked Structure

When the search loop terminates there are two possibilities. If `p` is `null`, the desired item has not been located and deletion is not possible. Otherwise, `p` references the item to be deleted and there are two possibilities. If `q` is not `null`, it references the entity's predecessor and it should have `p`'s successor as its successor as shown in Figure 3.19. If `q` is `null`, there are no nodes preceding `p` in the list, and `p`'s successor should become the first item in the list. In any event, since the loop will traverse on average half the nodes, the algorithm is `O(n)`.

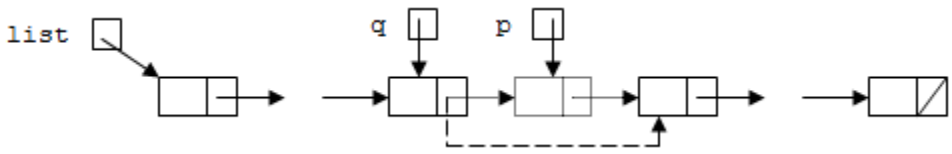


Figure 3.19 Keyed Deletion from a Sequentially-linked Structure

INSERTION AT THE END USING A TAIL REFERENCE. With a minor change in representation it is possible to improve the performance of insertion at the end of the structure. The reason this insertion is `O(n)` is that the algorithm must first locate the end of the list. If we maintain a reference (`tail`) to the last node in the list, this is not necessary and the insertion algorithm is `O(1)`. Figure 3.20 shows this representation.

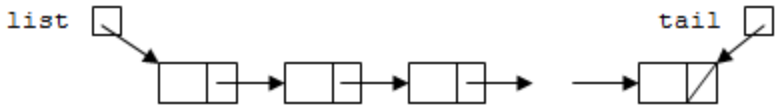


Figure 3.20 Sequentially-linked Structure with Tail Reference

An empty structure is represented by the structure reference (`list`) being `null`. When the structure is empty there is also no last node, so `tail` should also be `null`. This means that whenever the first node is inserted into an empty structure or the last node is deleted from a structure, care must be taken to adjust both `list` and `tail`. This is true in all insertion and deletion algorithms when this representation is used. The new algorithm for insertion at the end of a structure using this representation is shown in Figure 3.21.

```
if ( tail == null ) {
    list = new Node(item,null);
    tail = list;
}
else {
    tail.next = new Node(item,null);
    tail = tail.next;
};
```

Figure 3.21 Algorithm for Insertion at the End of a Sequentially-linked Structure with a Tail Reference

There are two possibilities: an empty structure and a structure containing at least one entity. If `tail` is `null` there is no last node and the list is empty. The new node is both the first and the last. When `tail` is not `null` there is a last node and the new node is grafted on after `tail`. `tail` is updated to reference the new node. The latter case is shown in Figure 3.22. Since there is now no loop in the algorithm, it is `O(1)`.

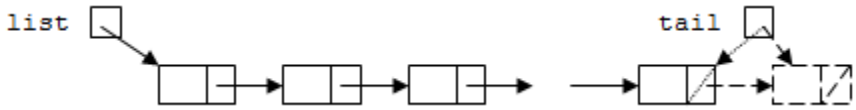


Figure 3.22 Insertion at the End of a Sequentially-linked Structure with a Tail Reference

As an example of a modified deletion algorithm for this representation, consider deletion of the first node of the structure. As long as there is at least one node left in the structure, nothing changes. However, when the last node is deleted, the `tail` reference must also be updated. The modified algorithm is shown in Figure 3.23. Compare this with Figure 3.7.

```
if ( list == null ) {
     delete fails
}
else {
     item = list.item;
     list = list.next;
     if ( list == null ) {
          tail = null;
     };
};
```

Figure 3.23 Deletion at the Front of a Sequentially-linked Structure with a Tail Reference

In general, the original representation and implementations would be preferred unless insertion at the end of the structure is very common or is the only kind of insertion. In this case the additional storage for the tail reference and the added complexity to all of the algorithms is acceptable. Since the changes to the other algorithms involve insertion of code of `O(1)` not in a loop, it does not affect the order of the other algorithms.

Unfortunately, there is no way to improve the performance of the algorithm for deletion of the last node (Figure 3.14). Maintaining a reference to the last node doesn't help here since what is needed is the location of the second last node. Even keeping a reference to the second last node doesn't help. After the deletion, this reference would have to be updated to the previous third last node, for which there is no reference. The only way this could work would be to maintain a reference to each node of the list. Of course, this representation would essentially be an array, and all benefits of linked representation would be lost.

COMPARISON OF ARRAYS AND SEQUENTIALLY-LINKED STRUCTURES

Table 3.1 compares the use of arrays and sequentially-linked structures for representing collections of information. The best-case results are presented, including the use of a tail reference for the sequentially-linked structure. Remember, since arrays are static, there is always the additional problem of goodness of fit. The array will often be too big—resulting in wasted space—or too small—requiring modification of the program and recompilation. The insertion and deletion entries that show `O(n)` performance for arrays involve `O(n)` moves of items. In languages that store the item within the array instead of a reference as in Java, these are very expensive operations and outweigh the `O(n)` reference traversals of the equivalent sequentially-linked structure algorithms.

Searching is covered in Chapter 10, but search times are included here for completeness. As we will see, there is an algorithm called binary search for searching a sorted array in `O(lg n)` time. Although searches in a sequentially-linked structure are restricted to sequential search, which is `O(n)`, there is a linked structure called a binary search tree which has `O(lg n)` search performance. Binary search trees are beyond the scope of this text.

operation	array	sequentially-linked structure
insertion at front	O(n)	O(1)
insertion at rear	O(1)	O(1)
sorted insertion	O(n)	O(n)
deletion at front	O(n)	O(1)
deletion at rear	O(1)	O(n)
keyed deletion	O(n)	O(n)
traversal	O(n)	O(n)
search (unsorted)	O(n)	O(n)
search (sorted)	O(lg n)	O(n)

Table 3.1 Comparison of Array and Sequentially-linked Structure Algorithms

3.2 OTHER LINEAR LINKED STRUCTURES

There are a number of other linear linked structures. We will briefly consider a few. Once the concept of linking nodes together via references has been established, only the imagination limits the number of ways linked structures can be built. We will look at three common variations: symmetrically-linked structures, circular linked structures and lists-of-lists. We will also describe a variation for all linked structures—header nodes—which make certain operations more convenient.

*SYMMETRICALLY-LINKED STRUCTURES

When we were looking at deletion in sequentially-linked structures we needed to access the node's predecessor. This made it impossible to delete the last node in the structure without `O(n)` steps. Similarly, the only possible traversal order is from front to rear of the structure. The problem is that although each node "knows" its successor, it doesn't know is predecessor. Symmetrically-linked structures remedy this problem.

NODE STRUCTURE. In a **symmetrically-linked structure**—also known as a **doubly-linked structure**—each node maintains a reference to its predecessor as well as its successor. The node specification is given in Figure 3.24 and Figure 3.25 shows the resulting structure. Since the first node has no predecessor, its `prev` reference is `null`. As can be seen from the diagram the structure is more symmetric, hence its name.

```
class Node {

    public Node  prev;  // previous node in sequence
    public type  item;  // the wrapped item
    public Node  next;  // next node in sequence

    public Node ( Node p, type i, Node n ) {
        prev = p;
        item = s;
        next = n;
    };  // constructor

}     // Node
```

Figure 3.24 Symmetrically-linked `Node` Wrapper Class

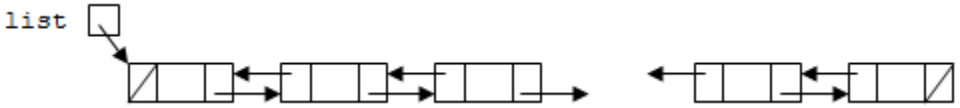


Figure 3.25 Symmetrically-linked Structure

If only the `next` reference is considered, this structure is a sequentially-linked structure and most of the algorithms are similar. Specifically the traversal, and hence search algorithms are the same. If a reference is maintained to the last node in the list, similar to the technique used in sequentially-linked structures, traversal can also occur in the reverse direction using the `prev` reference rather than the `next` reference. This also allows insertion and deletion at either end of the list in `O(1)` time. Insertion and deletion at the rear are just like insertion and deletion at the front, except using the `prev` reference instead of the `next` reference. Of course, all insertion and deletion algorithms must be modified to account for the extra reference in each node. Since a node references both its predecessor and successor, only one travelling pointer is generally required.

INSERTION IN SORTED ORDER. For example, consider the algorithm for insertion in sorted order into a symmetrically-linked structure without a tail reference as shown in Figure 3.26.

```
q = null;
p = list;
while ( p != null &&
        insertion key is greater or equal to key of p.item ) {
    q = p;
    p = p.next;
};
if ( q == null ) {
    list = new Node(null,item,p);
    if ( p != null ) {
        p.prev = list;
    }
}
else {
    q.next = new Node(q,item,p);
    if ( p != null ) {
        p.prev = q.next;
    }
};
```

Figure 3.26 Algorithm for Sorted Insertion into a Symmetrically-linked structure

Compare this to the algorithm in Figure 3.16. The code to locate the insertion point is the same, following the `next` pointers from the start of the structure. The changes occur at the insertion. If the node is at the front of the list (`q` is `null`), its `prev` reference must be set to `null`. If the list wasn't empty, `p`'s `prev` reference is to the new node (`list`). If the insertion is not at the front, the node is inserted after `p`'s predecessor (`q`), with `prev` referencing `q`. If it is at the end of the list (`p` is `null`), its `prev` reference is `q` and its `next` reference is `null` (so is `p`). Otherwise its `prev` reference is `q` and its `next` reference is `p` and `p`'s `prev` reference is to the new node (`q.next`). The general case, where insertion is within the structure is shown in Figure 3.27.

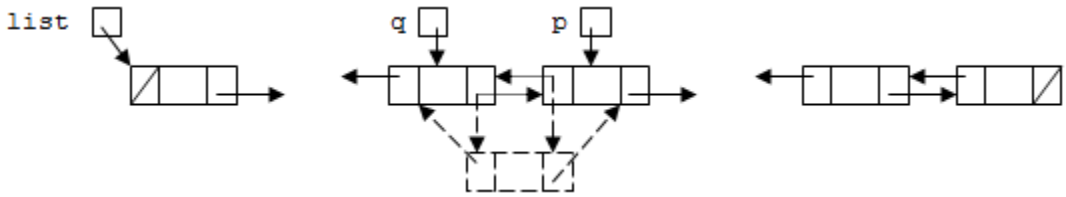


Figure 3.27 Sorted Insertion into a Symmetrically-linked Structure

DELETION OF A NODE BY KEY. Similarly, in deletion it is necessary to ensure that the structure is correctly linked back together, in both directions. Consider the algorithm for keyed deletion in a symmetrically-linked structure without tail reference as shown in Figure 3.28 and compare it to the algorithm in Figure 3.18. Since each node references its predecessor, we don't need two traveling pointers. We needed them in the insertion algorithm above in case `p` fell off the structure.

```
p = list;
while ( p != null &&
        key of p.item not equal to deletion key ) {
    p = p.next;
};
if ( p == null ) {
    delete fails
}
else {
    item = p.item;
    if ( p.prev == null ) {
        list = p.next;
    }
    else {
        p.prev.next = p.next;
    };
    if ( p.next != null ) {
        p.next.prev = p.prev;
    };
};
```

Figure 3.28 Algorithm for Keyed Deletion from a Symmetrically-linked Structure

The search is essentially the same, except using only one traveling pointer. If `p` falls off the structure, deletion cannot occur since the item wasn't found. If the item being deleted is the first node (`p.prev` is `null`), the next node becomes the first. Otherwise the previous node (`p.prev`) must reference the next node (`p.next`) as its successor. If the deleted node is the last, the previous node's successor should be `null`. This is handled by the earlier code since `p.next` will be `null`. Otherwise, the successor to `p` must reference as its predecessor `p`'s predecessor. The general case is shown in Figure 3.29.

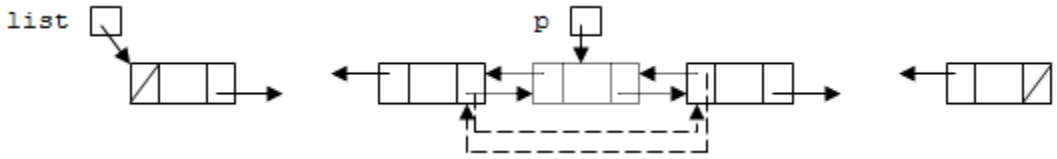


Figure 3.29 Keyed Deletion from a Symmetrically-linked Structure

There is a symmetrically-linked equivalent of each of the sequentially-linked algorithms. With the exception of deletion at rear with a tail pointer which can be done in `O(1)`, the orders of the symmetrically-linked algorithms are the same as the sequentially-linked ones. However, there is an extra reference in each node. This increases the space overhead for linking by 50%, from two references to three, and adds complexity to most of the algorithms. The complexity is in the difficulty of coding the algorithms dealing with two pointers and hence increases the probability of bugs. The added code is `O(1)` and not in a loop, so it doesn't change the order of the algorithms. Symmetrically-linked structures should only be used when the benefits, such as bi-directional traversal and `O(1)` insertion at rear, warrant.

*CIRCULAR LINKED STRUCTURES

In the linear linked structures we have seen so far, the last node contains a `null` reference indicating no successor. In symmetrically-linked structures, the same is true for the first node indicating no predecessor. Sometimes it is desirable to be able to traverse from any node to any other node within a structure. Other times, although the same relative order is desirable, it is useful to be able to consider a different node as the front or current node of the structure. In these cases, the `null` reference(s) can be used to reference the front and/or end of the list as shown in Figures 3.30 and 3.31, creating a **circular linked structure**.

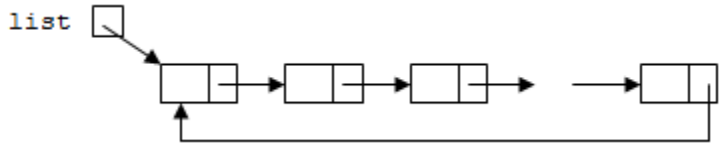


Figure 3.30 Circular Sequentially-linked Structure

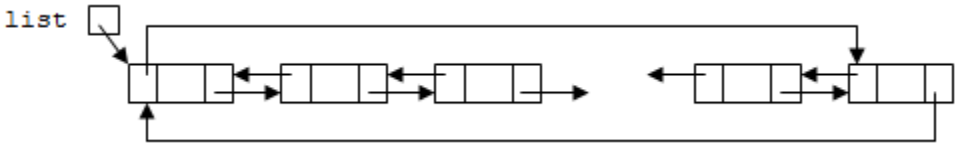


Figure 3.31 Circular Symmetrically-linked Structure

The empty structure is still be represented by a `null` list reference however care should be taken with a structure consisting of only one node. Here the node should reference itself as shown in Figure 3.32.

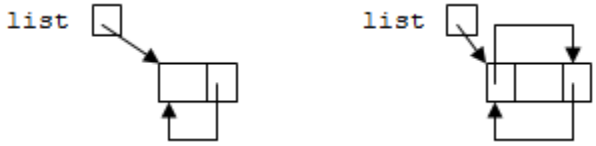


Figure 3.32 Single Node Circular Structures

This variation might be used when it is desirable to treat each node in the structure as the front of the list in succession. An example is round-robin scheduling in an operating system. Each task, as front of the task list, in turn receives some CPU cycles and then control passes to the next task or node. The step from node to node is simply the code:

```
list = list.next;
```

Since in this case the sequencing is intended to go on indefinitely there is no problem that the structure is circular. However in the more usual case unending repetition must be avoided otherwise the algorithm will be in an infinite loop. In a circular structure a traversal or search algorithm must now terminate, not when it reaches the node with a `null` reference, but rather when it has finished

processing the node that references back to the first. Figure 3.33 shows such a traversal as a variation of the algorithm shown in Figure 3.9.

```
if ( list != null ) {
    p = list;
    do {
        process p.item;
        p = p.next;
    } while ( p != list );
};
```

Figure 3.33 Algorithm for Traversal of a Circular Sequentially-linked Structure

Of course, to make the structure circular and maintain it as such care must be taken in insertion and deletion. In a circular sequentially-linked structure, when the list gains its first node the `next` reference must be made circular—pointing to itself. Similarly when the head or tail node within a list is deleted, a check must be made in case its `next` reference refers back to the node itself. In this case, the `list` reference must be set to `null`. Additionally if the first node is deleted, or a new first node inserted, the last node's `next` reference must be updated. This makes these algorithms `O(n)` unless there is a tail reference or the structure is symmetric. Of course, these problems are doubled for a symmetrically-linked structure since both the first and last node have circular references.

Since the algorithms are more complicated and in some cases of higher order, circular structures are only used in special cases where circularity is necessary and a higher order algorithm is not a problem.

HEADER AND SENTINEL NODES

Many of the algorithms for linear linked structures are complicated by the necessity to handle the first node of the structure differently from the rest. For example, insertion at the front of the structure requires modifying the structure reference while other insertions do not. The same is true for deletion of the first node in the structure. If it could be guaranteed that the first node of the list is always there—thet every node has a predecessor—these special cases would not be necessary.

HEADER NODE. A special node, placed at the front of a linear linked structure and never removed is called a **header node**. Since the node is neither inserted nor removed, it is not considered part of the collection and thus does not reference an item. The same node structure, such as in Figures 3.2 and 3.24, is used. Figure 3.34 shows a sequentially-linked structure with a header node and Figure 3.35 shows the symmetrically-linked version. The item reference in the header node is shaded to show it doesn't reference an item. Its actual value is `null`.

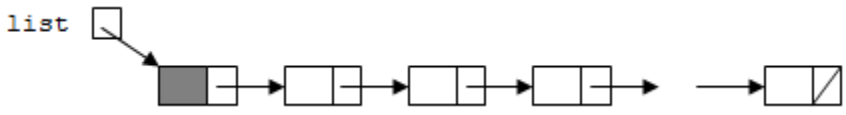


Figure 3.34 Sequentially-linked Structure with a Header Node

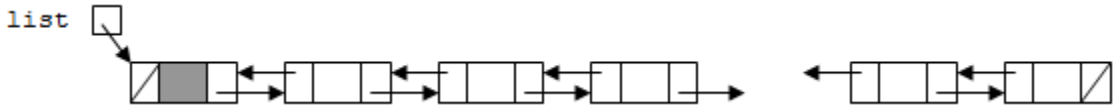


Figure 3.35 Symmetrically-linked Structure with a Header Node

The empty structure always has one node in it—the header node—as shown in Figure 3.36. The initial or empty state is established by the code shown in Figure 3.37.

Figure 3.36 Empty Linear linked Structures with Header Nodes

```
list = new Node(null,null);        list = new Node(null,null,null);
```

Figure 3.37 Initial (empty) States of Linear linked Structures with Header Nodes

Traversals must begin at the second node—the first real item—since the first node has no associated item.

INSERTION IN SORTED ORDER. The use of a header node makes insertion and deletion easier. Figure 3.38 shows the algorithm for sorted insertion into a sequentially-linked structure with a header node as a variation of the algorithm in Figure 3.16. The search is essentially the same except it starts at the second node (`list.next`). When the search is complete, `q` always references some node—either the header in the case of the insertion occurring at the head of the list or some other node otherwise—so the test for `q == null` is unnecessary. A similar modification to the algorithm in Figure 3.26 would provide the algorithm for a symmetrically-linked structure.

```
q = list;
p = list.next;
while ( p != null &&
        insertion key is greater or equal to key of p.item ) {
    q = p;
    p = p.next;
};
q.next = new Node(item,p);
```

Figure 3.38 Algorithm for Sorted Insertion into a Sequentially-linked Structure with a Header Node

DELETION OF A NODE BY KEY. Figure 3.39 shows the algorithm for keyed deletion from a sequentially-linked structure with a header node. Again the search begins at the second node and when complete, `q` is known to reference a node—possibly the header—so the actual deletion becomes a single case. Again a similar modification to algorithm in Figure 3.28 would provide deletion in a symmetrically-linked structure.

```
q = list;
p = list.next;
while ( p != null &&
        deletion key not equal to key of p.item ) {
    q = p;
    p = p.next;
};
if ( p == null ) {
    delete fails
}
else {
    item = p.item;
    q.next = p.next;
};
```

Figure 3.39 Algorithm for Keyed Deletion from a Sequentially-linked Structure with a Header Node

TAIL REFERENCE. In the symmetrically-linked structure the header node has two references available. Since the header node's predecessor reference would otherwise always be `null`, it can be used to reference the last node of the list gaining the advantages of a tail reference without requiring an additional variable!

CIRCULAR STRUCTURES WITH A HEADER NODE. Of course a header node can be used in a with circular structure. In this case the empty structures would consist of the header node whose references refer to the header node itself (Figure 3.40). Care has to be taken to avoid the header during a circuit around the structure. Sometimes making the last node reference the logical first node, instead of the header, can be a solution although this complicates insertion and deletion algorithms. In a circular symmetrically-linked structure, the header winds up referencing both the first and last nodes so it can essentially serve as a header in either direction.

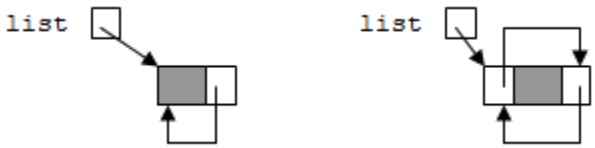


Figure 3.40 Empty Circular linked Structures with Header Nodes

SENTINEL NODE. It is also possible to add an extra node to the end of a linked structure. Like a header node, a **sentinel node** does not contain a reference to an item and is referenced by the last node in the structure. It simplifies any algorithm that has a special case for the last node of the structure. For example, by placing a reference to a dummy entity containing a high (or low) key value, the not found case can be eliminated in a sorted insertion algorithm (Figure 3.16 and Figure 3.26). This technique is similar to the technique used in a file merge algorithm using a sentinel record. Actually, the advantages here are slim. More interestingly, in a circular symmetrically-linked structure the header node also serves as a sentinel. In this case, every node has a predecessor and successor and insertion and deletion are simpler. It is even possible to delete a node when it is not even known which list it is in, as long as it is on a circular symmetrically-linked structure with a

header. This situation might occur when a node occurs on a number of lists simultaneously, a structure called a multi-list (see next section).

COST. The cost of using a header and/or sentinel node is the space for the additional node(s) and, the complication in traversal algorithms to deal with the header or sentinel. Their use simplifies some algorithms, although it doesn't change their order.

*LIST-OF-LISTS AND MULTI-LISTS

It is possible that the items in a linear linked structure are not simple items, but rather collections in their own right. We can combine arrays with lists yielding arrays of lists and lists of arrays. Similarly, we can have lists of lists. Figure 3.41 shows one such structure: the list-of-lists. The structure reference (`list`) references a first sequentially-linked structure (the left-hand column) whose nodes, in turn, reference sequentially-linked structures. Note the similarity between this structure and the array-of-arrays structure as shown in Figure 1.21.

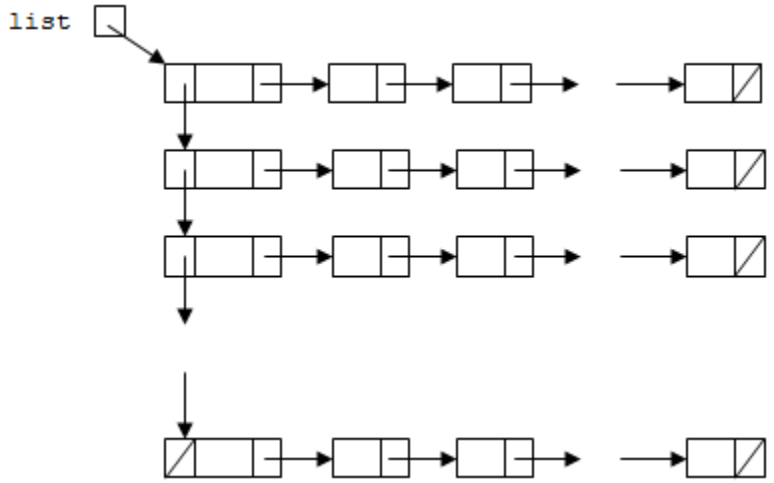


Figure 3.41 List-of-lists Structure

The nodes referenced directly by `list` (the left–hand column) differ from the rest of the nodes since they contain a list reference as well as the link to the other nodes in the list. The structure for these nodes is shown in Figure 3.42. Typically the list nodes would contain some kind of identifying information for the list to which they point.

```
class ListNode {

    public ListNode  next;
    public type      info;
    public Node      list;

    public ListNode ( ListNode n, type i, Node l ) {
        next = n;
        info = i;
        head = l;
    };  // constructor

}  // ListNode
```

Figure 3.42 List Node Structure

The list nodes themselves form a sequentially-linked structure, using their `next` field. Insertion, deletion, traversal and search of the lists would use the same algorithms as described in Section 3.1. Similarly, each sub-list is a sequentially-linked structure referenced using the `list` field of the appropriate list node. Again, operations on these lists are essentially the same as the sequentially-linked structures.

An example of the use of this kind of structure might be to represent the classes in a university. There is an indefinite number of classes and new classes may be added as needed. Each class contains an indefinite number of students who may add and drop classes. The sub-lists represent the class lists themselves as a list of students and the list nodes might contain the class identifier—for example "COSC 1P03 S1" for section 1 of COSC 1P03—as the content to identify the class.

MULTI-LISTS. Finally, there is the concept of a **multi-list**. Here each node may be part of more than one list at the same time. If there are a specific number of lists that each node must be on, the `Node` class can be modified to contain links to successors for each list. If the number of lists the node is on may vary, there are other variations such as wrapping the node with multiple wrappers, one per list.

An example of multi-lists is a representation for sparse matrices (see Section 1.8). Here, a node represents a non-zero matrix element. The elements in the same row are collected onto a list representing the row. Similarly the elements of a column are collected onto a list for the column. The row lists are collected into a list of rows, and likewise for the columns. The structure reference includes a reference to the row list and also the column list. This structure is shown in Figure 3.43, representing the matrix shown in Figure 1.32.

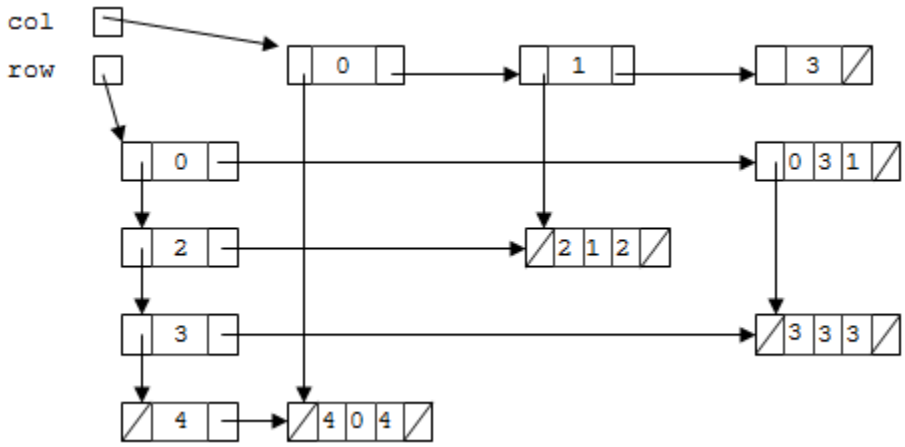


Figure 3.43 Sparse Matrix as a Multi-list

The row (left side) and column (top) header nodes contain their row (column) index as identifying information. The individual element nodes contain both the row and column indices in addition to their value. These are all placed in a single object since they are part of the representation of the individual matrix element. In this case there is no great advantage in using a wrapper node to separate the link information from the data, since the data is not an object. If the array is sparse enough, having a very low percentage of non-zero elements, this representation is reasonably space efficient. Traversal (row or column major) is efficient (still `O(n)`), however random access involves a traversal so is `O(n)` rather than `O(1)` as in an array.

3.3 WORKING WITH LINKED STRUCTURES

Sections 3.1 and 3.2 have described some variations on linked structures and given common algorithms for their manipulation. However, the structures and their corresponding algorithms need to be tailored to the particular application. Sometimes the algorithms will be embedded within methods corresponding to operations of the application. Sometimes the algorithms will be written as standalone methods supporting many application operations. Various techniques are used depending on how the algorithms are used.

It is necessary to take care when writing algorithms that manipulate linked structures as methods. The natural approach might be to write a method such as:

```
private void insert ( Node list, Item item ) {
```

where `list` is the list reference and `item` is the item to be added as a new item on the list. Unfortunately this might not work. In parameter passing, a copy of the argument is made and used by the method. If the method modifies the parameter (i.e. its copy), the argument does not change and the calling method does not see any effect. Since many of the linked structure algorithms—such as insertion at the front (Figure 3.5)—modify the list reference (`list`), passing the list reference as a parameter in this way is not effective.

One solution is to change the method definition so that it returns the new value of the list reference if it is changed or the original value if not. The method would look like:

```
private Node insert ( Node list, Item item ) {
```

and would be called such as:

```
myList = insert(myList,newItem);
```

This technique doesn't work well if the method would otherwise return a value. For example a delete method might change the list reference (deletion at the front) but the method may also need to return the item deleted. It is not possible to return two values from a function.

Since a header node is never deleted or replaced and is referenced via the list reference, this is not a problem for structures with a header node. The list reference can be passed as a parameter since it is never going to be changed. Insertion or deletion at the front actually inserts or deletes after the header node. Of course, the header node has to be created when the list is created so, if the create operation is to be written as a method, it would have to be a function that returns the list reference as its result.

If the algorithms are embedded in other code, the list reference will likely be a local or instance variable and can thus be referenced and modified directly (as opposed to being a parameter). This technique can also be used if the algorithm is written as a method, with the method referencing the list reference as an instance variable. Of course, this limits the algorithm to working with only one list, the one defined as the instance variable.

This last case is not that uncommon, however. As we will see in subsequent chapters, a linked structure may be used as the implementation of some kind of collection object such as a stack or queue or for that matter a picture. In this case, the object being implemented has a single list reference to the linked structure as an instance variable which can be directly referenced by the methods. If there are multiple objects, each has its own list reference to its own linked structure and the list reference need not be passed as a parameter. This is the natural approach when developing container objects such as stacks, queues and lists as we will see in Chapters 5, 8 and 9.

CASE STUDY: CAR RENTAL AGENCY

A car rental agency owns a number of cars that they rent to customers. The agency maintains a record for each car including its licence plate number (string), its current mileage (integer) and its car class (integer: 0=Economy, 1=Full Size, 2=Van and 3=SUV). When a customer arrives at the agency, a car is selected for the customer. When a car is returned, the charge is computed based on the difference between the mileage at return and the mileage at rental and the mileage rate for the class of car rented.

The company needs a program to keep track of their fleet of cars. This can be done using two lists: one of cars available for rent and one for cars currently rented but not yet returned. When a car is rented, it is removed from the available list and added to the rented list. Similarly, on return a car is removed from the rented list and added to the available list. At any time, the agency should also be able to display the two lists to track their fleet.

The system can be implemented representing the cars by a `Car` class and using sequentially-linked structures with items of type `Car` to track the fleet. One structure would represent the `available`

list and one the `rented` list. At any time, each `Car` is on exactly one of the two lists. `Car` objects are deleted from one list and added to the other.

VERSION 0. As a proof of concept, implement the system using sequentially-linked structures for the available and rented lists. The fleet information should be read from a text file and used to create the initial available list. The initial rented list will be empty. When a car is to be rented, the first car from the available list is removed and placed at the front of the rented list. When a car is returned, the first car from the rented list is removed, the charge computed and mileage updated, and placed at the front of the available list. When a list of the fleet is required, the two lists are traversed in turn, displaying the licence numbers to the screen. An appropriate GUI should be used to handle the rental and return of cars. It should display the licence plate and, for return should handle the input of mileage and display of the charge for the rental.

VERSION 1. The agency has accepted the proof of concept so it is time to implement the complete system. When a car is to be rented, the customer specifies a car class and the first car in the available list that matches that class is removed, rented and added to the front of the rented list. When a car is returned, the rented list is searched for the car with the supplied licence plate. This item is removed from the rented list, the charge computed and mileage updated and the car is added to the end of the available list. The GUI and listing of the fleet information is the same as in version 0.

SUMMARY

In many problems the amount of data to be represented varies over time. Static data structures, such as arrays, are ineffective in representing the such information. Dynamic data structures, such as linear linked structures, have the advantage that the amount of storage used is proportional to the amount of data. Generally, insertion and deletion of items is also less expensive that in static data structures.

The most common linear linked structure is the sequentially-linked structure in which each node contains a link to the next or successor node in the collection. Typically the nodes themselves are wrapper nodes, containing a reference to the item actually on the list. Insertion involves creating a new wrapper node and linking it onto the list. Deletion involves modifying the links in the nodes to bypass the node being deleted. These operations are `O(1)`, however finding the point for insertion or the node to be deleted may still be `O(n)`.

Alternative linear linked structures include symmetrically-linked structures, list-of-lists and multi-lists. In a symmetrically-linked structure, each node has a link, not only to its successor, but also its predecessor. This structure allows traversal in both directions and the possibility of deletion without the need to traverse the list. The fundamental algorithms are essentially the same as for the sequentially-linked case, however they are complicated by the necessity of updating a pair of pointers instead of only one. In list-of-lists there is a list of references to the sub-lists. Each sub-list is a sequentially-linked structure of nodes. Operations typically involve finding the appropriate sub-list by traversing the list and then performing the operations on the appropriate sub-list. In multi-lists each node occurs on more than one list and may be accessed via traversal of any of the lists. Operations are complicated by the need to involve each of the lists on which a node occurs.

Variations on linked structures include circular structures and header and sentinel nodes. In a circular structure the last node points back to the first node. This allows traversal from any node to any other node and may be useful when there really isn't a first node, but rather each node is

considered the current node in turn. With circular structures care must be taken in traversal to avoid an infinite loop. A header node is a special node at the front of the list that is never deleted and doesn't contain a reference to an item. Header nodes can simplify the algorithms that must special case operations at the front of the structure. Similarly sentinel nodes—a special node at the end of the structure—can simplify algorithms that must special case operations at the end of the list.

REVIEW QUESTIONS

1. T F A wrapper class is a class that references another class to add functionality or change its type.

2. T F If a sequentially-linked structure contains `Student` objects, printing a transcript for a particular student would take `O(1)` time.

3. T F Deletion at the end of a sequentially-linked structure can be done in `O(1)` time.

4. T F A circular list allows traversal from any node to any other node in the list.

5. T F A header node is used to reduce the order of an algorithm that performs operations at the head of a linked-structure.

6. T F Listing all the students in a course would involve a traversal of the course data structure.

7. T F If a sequentially-linked structure contains `Student` objects ordered by student number, locating a particular `Student` object (by student number) would take `O(n)` time.

8. T F Deletion at the end of a symmetrically-linked structure with a header node can be done in `O(1)` time.

9. T F A sequentially-linked structure allows traversal from any node to any other node in the structure.

10. T F In a sequentially-linked structure with a header node, traversal starts at the second node.

11. Which of the following is a problem of static data structures?:

    a) may occupy too much space
    b) may be too small
    c) are expensive to rearrange
    d) all of the above

12. Using the object itself as a node in a linked structure is a good idea because it:

    a) requires fewer classes
    b) requires less operations to access the node
    c) is easy to add new links
    d) it is not generally a good idea

13. Repeated application of the code

```
list = new Node(list,new Student(in));
```

(assuming `list` is initially `null` and `in` is an input stream) would result in:

a) a circular list
b) a list of students in reverse order
c) an exception
d) list of students in sorted order

14. After the search part of the sorted insertion in a sequentially-linked structure algorithm, the condition `q==null` implies:

a) the list was empty
b) insertion occurs in front of the first node
c) the key was not found
d) a and b

15. A search in a sorted sequentially-linked structure is:

a) $O(1)$
b) $O(\lg n)$
c) $O(n)$
d) none of the above

16. Which of the following is an advantage of a sequentially-linked structure over an array?

a) requires less space
b) space requirements are proportional to the number of elements
c) operations are of lower order
d) all of the above

17. Creating a sorted sequentially-linked structure of n nodes from the empty state would be:

a) $O(n^2)$
b) $O(n \lg n)$
c) $O(n)$
d) none of the above

18. An object that is on three independent lists would be wrapped in how many wrapper objects?

a) 0
b) 1
c) 2
d) 3

19. After a search in a sequentially-linked structure, the condition `p==null` implies:

    a) the list was empty
    b) the item was the last node
    c) the item was not found
    d) a or c

20. In a sorted (ascending order) insertion in a sequentially-linked structure, the condition `q==null` implies:

    a) the key already exists
    b) the list is empty
    c) the new key is the smallest
    d) b or c

EXERCISES

1. Write a program that manipulates sequentially-linked structures to maintain lists of athletes for the Snowboarding event during the Sochi Winter Olympics. In snowboarding, there are two runs of competitors to determine the gold, silver and bronze medallists. In the first run, all athletes registered in the event will compete. For the second run, only the top half of the competitors from the first run will compete. The score of the athletes after the final (second) run is the sum of the scores from the first and second runs. The athlete with the highest score is the winner (gold medallist).

   You will need to maintain four lists for the status of the athletes as follows:

   - `competitors`: list of all athletes in the snowboarding event, by competitor number
   - `firstRun`: list of all athletes with score after the first run, in descending score order
   - `secondRun`: list of the top-half of the athletes from the first run, in ascending order of first run score
   - `finalist`:list of athletes from the second run, in descending order of total score

   Three files will be used in the application simulating the data entry in the event. The first file, with athletes in no specific order, will be used to create the original competitor list. Each competitor has a name (`String`) and country (`String`). The competitors are also assigned competitor numbers (`int`) in the order they are entered. The competitor list should be built according to order of arrival, that is by their competitor number. The second file simulates the entry of the results of the first run, with competitors in random order. Each record contains the competitor number (`int`, as assigned in processing the first file) and the score for the run (`double`). This file is used to create the first-run list by adding competitors in descending order according to score. The competitors are not to be removed from the competitor list. Each will now be on two lists. The top-half of the entries—those with the highest first run scores—should be removed from the first-run list and placed into a new list—the second-run list—in reverse score order, from lowest to highest. The third file simulates the data entry for the second run in which the competitors compete in reverse score order, the same order as the second-run list. Competitors are removed from the second-run list and placed on the finalist list in descending order according to total score. (Total score = run 1 score + run 2 score).

Your output will include printing the various lists as follows:

1. print the competitors list (competitor number, name and country, in competitor number order) prior to the first run.
2. print the results of first run, that is, the first-run list (competitor number & score, in score order) before the second run.
3. print the second-run list (competitor number & score, in reverse score order) before the second run.
4. print the final results, that is, the finalist list (competitor number & total score, in score order) after the second run.
5. finally, print a summary of results of all competitors, that is, the competitors list (competitor number, name, country and score) in competitor number order.

The program should use `ASCIIDataFile` for the three input files and an `ReportPrinter` for the output report.

Hint

Define a class (`Athlete`) to represent a competitor including the registration number, name, country and score (either first run or total depending on the situation). The item field in the `Node` class will be `Athlete`.

2. A polynomial such as:

$$3x^3 + 5x - 4$$

is a sum of terms of the form:

$$cx^i$$

where `c` is the coefficient (integer, either positive or negative) and `i` is exponent (integer, non-negative). The example above consists of the 3 terms:

$$3x^3,\ \ 5x^1,\ \ -4x^0$$

A polynomial can be represented as a sequentially-linked structure with each node representing a term and the structure representing a sum of terms. Each node consists of the pair: coefficient and exponent and the nodes should be in descending order by exponent. Terms with zero coefficients are not represented.

Write a program that manipulates polynomials represented as sequentially-linked structures. The program should implement reading a polynomial from an `ASCIIDataFile`, addition and subtraction of polynomials, evaluation of a polynomial for a given value of `x` (`double`) and writing a polynomial to an `ASCIIDisplayer`. The program should input two polynomials from an `ASCIIDataFile`. Each polynomial begins on a new line with an integer giving the number of terms (`n`). Following this are `n` pairs of values, an integer being the coefficient and an integer being the exponent. The program should then create two new polynomials being the sum and difference of the two polynomials input. It should then read a value for `x` and write the two

original polynomials and their sum and difference, each followed by the evaluation of the polynomial for the input `x`. The format of the output of the polynomial should be:

```
cXi + cXi + …
```

in order from highest exponent to lowest. The above example would be displayed as:

```
3X3 + 5X1 + -4X0
```

Hint

Since a polynomial is a sum of terms, the sum of two polynomials is a form of merge between the two polynomials. The algorithm will be similar to a file merge. Common terms, those with the same exponent, must be combined by adding the coefficients. Remember that terms with a zero coefficient are not included. Subtraction is just addition with the coefficients of the subtrahend being negated.

3. In some applications, such as Number Theory, even the Java type `long` doesn't provide large enough integer values. What is needed is an integer type with an unlimited number of digits, just like integers in Mathematics. One possible representation is to use a linear-linked structure where each node represents a digit of the number. Since we can create new nodes to represent more digits, until we run out of memory, the size of number we can represent is unlimited.

   Write a program to demonstrate addition of positive integers. The program should read two integer values from an `ASCIIDataFile`. Each number begins on a new line with the number of digits (n) followed by `n` digits (integers). It should then print the two numbers and their sum.

   Consider addition of integers. We normally proceed from right to left, adding the digits together. On the other hand, when we write out the number, we need to write the digits from left to right. We need to traverse both ways, so a symmetrically-linked structure could be used. Each node represents one digit (0-9) and the digits should be in order from highest power of 10 to lowest (that is, the 1s digit). Since we need a reference to both the head and tail of the list to do the traversals, we can use a header node and use the `prev` reference to point to the tail.

   With addition of positive integers, we proceed right to left adding the digits and the carry from the previous digit position. This produces a digit and a carry (0 or 1). When we run out of digits in one addend we use 0 for the digit, until we are out of digits in the other addend. If the final carry is 1 we have one more digit (the 1).

4 ABSTRACT DATA TYPES

CHAPTER OBJECTIVES

- Explain and recognize data abstraction.
- Apply data abstraction in a problem.
- Explain the difference between a data type, data structure and abstract data type.
- Describe advantages of information hiding.
- Apply information hiding in the development of an abstract data type.
- Describe and apply the Java facilities for data abstraction: packages, interfaces, classes and exceptions.
- Describe the three types of exceptions in Java.
- Explain the throwing, propagation and catching of exceptions.

Abstraction—the emphasis of the relevant and the de-emphasis of unnecessary details—is the basis of all modeling. We have repeatedly used procedural abstraction where what is modeled is an activity or operation and the abstraction is hiding the implementation of the operation within the method body. The emphasis is on the purpose of the operation. That is we are interested in what the method does rather than how it does it. We have seen a second form of abstraction—data abstraction—in which we used a class to model an entity in a real-world or hypothetical system. We specified how the object participated in the system, ignoring details of the object that didn't concern the system.

Software engineering can be considered as a modeling exercise in which we model the entities in an existing (or hypothetical) system as classes within a computer system. The resulting classes are abstractions of the real-life entities, emphasizing the objects and the operations they can perform and de-emphasizing the representation of the entities and the implementation of the operations.

In this chapter we will look at data abstraction in more detail and consider the features in Java that support this form of abstraction.

4.1 DATA ABSTRACTION

The object-oriented modeling of a class as an abstraction of an entity is a case of a general concept called data abstraction. **Data abstraction** is the specification of a set of potential objects and the accompanying operations those objects may perform. Data abstraction predated object-orientation as a mechanism in software development and object-orientation extended the notion of data abstraction in defining classes.

The core of data abstraction is the notion of an **abstract data type** (**ADT**). An ADT is the specification of a set of possible data items or objects and the operations or methods defined on those items. An abstract data type is a formal notion. It specifies a set of operations and rules that serve to define the abstraction and allow properties of the abstraction to be determined. In this book, we will be more interested in the representation of an abstraction in a programming language.

The term **data type** is used to specify the types of data or objects that can be represented in a programming language. This includes primitive types such as `int`, which are abstractions of hardware concepts; and user-defined types or classes, which are abstractions for which the programmer supplies a realization. The term **data structure** is used to describe a way of organizing or structuring information to represent some abstraction. For example, in using an array of `Student` objects to represent the abstraction of a class within a university system—the array is a data structure.

Fundamental to the effective use of data abstraction is the concept of **information hiding**. In a large system consisting of many classes, it would be easy to get bogged down in the details, making it impossible to complete the development task. Information hiding is a technique in which each part of the system—a class or component—exposes only the minimum amount of information about its inner workings to other parts of the system. The advantage is threefold. First, when working on one part of the system, a programmer doesn't have to be concerned about details of another part. This is **reduction of complexity**. Second, when writing a component the programmer is assured that other parts of the system cannot reference the hidden parts of the code and so may make simplifying assumptions. This provides **security**. Third, a component may be replaced by another component with a different or modified representation or implementation, as long as it provides all the exposed parts in the same way. This improves **modifiability**.

In programming large systems, there are two roles. The **supplier** or implementer is the programmer who implements an abstraction. She selects a representation for the entity and implements the operations specified. The **client** or user makes use of the abstraction to support his own programming effort, of which he is the implementer. Abstraction and information hiding assist both of these roles.

An ADT can be considered to have two parts: a definition or **specification** and a **realization**—the representation and implementation. The specification serves as a contract between the supplier and the client, specifying all that the client needs to know to use the abstraction and all that the supplier requires to implement it. As long as the client doesn't violate the contract—using only what is specified and providing appropriate parameters, etc. —he is free to use the abstraction as he wishes. Similarly, as long as the supplier fulfils all of the requirements of the definition—supplying all the methods and using only information specified in the definition—she is free to implement the abstraction in any reasonable way. This style of programming is often called **programming-by-contract.**

Programming language support is valuable in making effective use of data abstraction and information hiding in system development. The language support must include a mechanism for definition of an abstraction or contract. It must enforce information hiding by limiting the client code to using the resources specified in the definition and also ensuring the supplier code implements the requirements as specified in the definition. Most object-oriented languages provide some level of such language support.

Many of the common and well-studied abstract data types are collections. A **collection** or **container type** is any type that represents or contains a group of objects or data values. In that sense, an array and the `Picture` type are collection types. We will consider a number of the most important collection abstract data types—stack, queue and list—in subsequent chapters of this book.

4.2 DATA ABSTRACTION IN JAVA

As we have already seen, a class defines a new data type specifying the operations available as `public` methods and hiding the representation via `private` instance variables. The class provides the definition, representation and implementation of an abstraction. The compiler enforces restriction of access to methods and data through scope rules, especially enforcing `public` and `private`. Thus a class can serve as a mechanism for providing ADTs that enforces information hiding.

Although a class may serve as a data abstraction mechanism, it has one serious drawback. The definition and implementation are both part of the same file containing the class declaration. This makes it more expensive to change an implementation since such a change requires recompilation of the class declaration. Since the class declaration also serves as the specification, the compiler is potentially required recheck all client code against the changed specification. Actually, smart compilers will recognize that the changes don't affect the specification and may avoid this. However, this does make implementation of the compilers more complicated.

More importantly, using only class declarations makes it impossible to provide alternative realizations of the same abstraction, since only one realization can be included in the class declaration. These realizations might use different data structures and/or algorithms. As we saw in Section 1.8, there are often many different choices for representations and there is not always one best choice. This is especially true when it is not known in advance to what problem the abstraction will be applied.

Java provides a construct called an interface, which solves this problem. An **interface** serves as the specification of an abstract data type without providing an implementation. A class—**implementation class**—may then be used to provide the implementation.

Information hiding relies on enforcement of the contract between the client and the supplier. Language constructs can handle most of the specification by defining what methods are available and what their parameters are. However, there is no complete notation for specifying requirements on the use of a method such as specifying in which states of the object the method may be used and what restrictions the object's state places on the method parameters. This means that these aspects of the specification must be given as comments in the specification, and leaves it up to the implementer to include code to check and enforce the restrictions. If an error in the use of the abstraction is found by the implementation, there must be a way of reporting this error. In Java, the **exception** mechanism provides this capability.

Finally, many abstractions cannot be completely defined and implemented in a single file. If an interface is used for the specification and a class for the implementation, there are at least two files—the interface and an implementation class. Often the implementation class requires support from additional classes that are not part of the specification. These are called local or **support classes**. A construct that allows grouping of classes into a component with different visibility rules within and between components, is a valuable tool. In Java, the **package** serves this purpose.

The `Date` Abstraction

As a simple example of data abstraction, we will develop an ADT representing a date. To keep it simple, we will consider only dates in the current year. A date is a set of values—365 in the year 2014—which are the days of the year. Dates are typically used to label information. For example, assignments have due dates and birthdays occur on a specific date. Some manipulation of date information is possible. For example, if an assignment is due in two weeks, we want to know the date two weeks or 14 days from now. Also, we may want to know how many days there are between dates. For example, how many days is it until your mother's birthday? Finally, we may want to know relationships between dates, for example asking the question "Is today the assignment due date?" or "Is my assignment in Mathematics due before my assignment in Computer Science?"

4.3 PACKAGES

Since the representation of dates will require a number of classes and interfaces, we will package them together as a Java package. One advantage is that a package can be placed in a library and used in many applications. A second advantage is that we can make only certain parts of the package—certain classes—visible to clients while keeping others solely for use in implementing the abstraction.

In Java, a package is a collection of classes and interfaces. All classes and interfaces are part of some package. A class declares itself as part of a package placing a package declaration (Figure 4.1) prior to its import clause, if any.

```
package name ;
```

Figure 4.1 Package Declaration

The *`name`* is a package name and may be simple name, such as `BasicIO`, or multi-part name, such as `java.awt`. The significance of the package name is outside the language definition. It typically corresponds to a directory, library entry or URL. Many Java development environments, such as the JDK from Oracle, assume that a package corresponds to a directory and all the Java bytecode (`.class`) files are placed in a directory with the package name. For our purposes, we will use simple package names.

If a class does not have a package declaration, it is considered to be part of the **unnamed package**. Again, the significance of this depends on the environment—the operating system and compiler. However, it typically means that classes without package declarations that are in the same directory are all part of the same, unnamed, package.

We have seen that a class declaration may be preceded by a number of import declarations. We have said that these referred to libraries from which we are using resources. Strictly speaking they are referring to packages that are in libraries. When we write:

```
import BasicIO.*;
```

we are importing from the `BasicIO` package. The `BasicIO` package consists of a number of classes and interfaces such as `ASCIIDataFile`. The use of the notation `.*` following the package name indicates that we may use any of the public classes or interfaces from the package.

The class declaration itself begins with some optional modifiers. For a class, the only modifier of significance is the modifier `public`. If a class is `public`, it is visible in any class that imports the package. If there is no `public` modifier, the package is said to be **package private** and is visible only to other classes in the same package. This is one way to affect information hiding at the class level in Java. Note that only public methods and variables of a visible class may be accessed, so there is a secondary level of information hiding available.

The issue of visibility of packages is outside the language definition. It is typically controlled by the operating system via accessibility to directories in which the packages reside. There is usually some mechanism to inform the compiler where to look within the operating system directory structure for packages.

STYLE TIP

It is a good idea for all classes and interfaces to explicitly declare they belong to a package instead of relying on the "unnamed" package. When classes that have been independently developed without explicit package membership are used together there is the possibility of name clashes—two classes with the same name being used together—and unexpected visibility problems—methods that are meant to be visible to some classes but not others. Using explicit package names avoids this problem.

It is sometimes difficult to decide in which package to place a class. If the class is part of an ADT or a library, then it obviously belongs to the package for the ADT or library. If the class is an implementation class in a system, then it probably should be part of the package for the system. Sometimes a class might be used within many systems. For example a `Student` class in a university registration system may be reused in other university systems dealing with students. In this case, such classes might be grouped together into a special library of in-house support classes and placed into a package for that library.

4.4 INTERFACES

Like a class an interface defines a new type by specifying the resources—specifically constants and methods—that objects of the type provide. Unlike a class, an interface does not specify how the objects are represented nor how the operations are implemented. It is strictly a definition. In some languages such a definition is called a **pure abstract class**. The form of an interface declaration is shown in Figure 4.2.

```
modifier… interface name {
    fieldDeclaration ; …
    methodHeader ; …
}
```

Figure 4.2 Interface Declaration

An interface is a compilation unit and as such, can be preceded by a package declaration and import clauses, and compiled by the compiler. Like a class, the only modifier for an interface is `public`. If an interface is `public`, it is visible in any class that imports the package. If there is no `public` modifier, the interface is package private and visible only to classes in the same package. *`name`* is the name (identifier) of the interface and is used as a type name elsewhere. The interface body consists of a number of *`fieldDeclaration`*s and *`methodHeader`*s. A *`fieldDeclaration`* must declare a constant. A *`methodHeader`* is a method declaration without the method body. Although modifiers are allowed on the field and method declarations in an interface, every item declared is considered public so only the `public` modifier should be used.

The `Date` Interface

The Figure 4.3 shows an interface declaration for the abstract data type `Date`. It is part of the `Dates` package, which will also include implementation classes and exceptions. It defines four methods for `Date` objects. It specifies a method to compare two dates to determine if they are the same date or if comes before or after another. It provides basic date "arithmetic" calculating the date some number of days in the future (or past) and determining how many days there are between two dates. It also defines a method that produces the day number within the year of the date. This method is not intended for general use but to support interoperability between implementation classes as we shall see later.

Each of these has been specified by a method header giving the return type, method name and parameters. Note that all methods and the interface itself have been declared `public`. This would be the norm since the purpose of an interface is to specify the public view of a type.

The documentation indicates that the `Date` type is immutable. An **immutable type** is one in which, once an object is created it does not change its state. That is a `Date` object always represents the same date. All operations on `Date` objects—such as `advance`—produce a new `Date` object as a result rather than modifying the existing `Date` object. Most object types (e.g. `Student`) are **mutable**—that is they change state. In fact that is how they have effect within systems. However there are some object types that represent "values" and these should be immutable. We are used to values being immutable. The value 5 is still 5, even after it has been added to the value 7 producing a new value 12. Arithmetic on values doesn't change the values themselves, rather it produces new values. To model this, we write the type as immutable where operations could affect an object produce new objects as a result (i.e. are functions).

STYLE TIP

When designing a class, seriously consider making it immutable. Immutable classes are much easier to define, use and debug. Don't just automatically write `set` methods. Of course, if the objects have to change state, they cannot be immutable. Essentially, when a type is immutable, the objects behave like values not true objects and really represent data not entities. In general, all value types should be implemented as immutable (see Bloch[3], Item 13). Care must be taken with immutable objects. Since their definition uses interfaces and classes, they are still reference types and variables point to the values. This means that the == and `!=` operators, even for immutable types, mean "same object" not

[3] Bloch, J.; *Effective Java™*: *Programming Language Guide*; Addison-Wesley, Reading MA; 2001

"same value". Classes that define value types should implement a method `equals` to determine value equality and/or implement a method `compareTo` supporting both equality and relational operations.

```
1  package Dates;
2
3  /** This interface defines an abstract data type representing a date in the current
4    * year (2014). Date objects are immutable.
5    * @author  D. Hughes
6    * @version 1.0 (Feb. 2014)                                                    */
7  public interface Date {
8
9      public static final String YEAR = "2014";  // the year of the date object
10
11     /** This method returns the day number within the year (from Jan 1 = 1). It is
12       * intended solely for use by implementation classes for interoperability.
13       * @return  int  day number within the year (Jan 1 = 1)                    */
14     public int getDays ( ) ;
15
16     /** This method compares this date object to other. It returns negative it
17       * this date precedes the other, zero if they represent tha same date and
18       * positive if this date follows the other.
19       * @param  other  other date for comparison
20       * @return  int  <0 if this precedes other, 0 if same date, >0 if this follows
21       *               other                                                      */
22     public int compareTo ( Date other ) ;
23
24     /** This method returns the date days following this date for positive days
25       * values and the date days predeeding this for negative days values.
26       * @param  days  the number of days to advance. Negative values produce
27       *               preceeding dates.
28       * @return  Date  new date before (days<0) or after (days>0) this date.
29       * @throws  InvalidDateException  if new date would not be in the current
30       *                                year                                     */
31     public Date advance ( int days ) ;
32
33     /** This method returns the numebr of days between this date and other. If
34       * other precedes this, the result is negative.
35       * @param  other  the date for difference computation
36       * @return  int  the number of days between this and other (<0 if other
37       *               precedes this)                                           */
38     public int between ( Date other ) ;
39
40 }  // Date
```

Figure 4.3 Example—`Date` Interface

Interface Types

An interface defines a type and the interface name, like a class name, is used to declare variables. However, since an interface does not include variable declarations or method bodies it cannot used to create objects. The actual representation of the objects and the implementation of the methods must be supplied by some class—an implementation class.

It is good practice to use the interface type to declare variables within client code rather than use the class type of an implementation class. For example we would—within the client code—write code such as the following:

```
private void meth ( Date  d1, Date  d2 ) {
    Date  d3;
    int   d;
       :
    d = d1.bewteen(d2);
    d3 = d2.advance(10);
    if ( d3.compareTo(d2) > 0 ) { … };
       :
};  // meth
```

Once the `Date` interface has been compiled, the compiler can verify that this code is valid, even if there isn't an implementation class written yet. Since the variables are declared using the interface type, we can choose any implementation type and we know it will remain valid. This provides interchangeability—we can change implementation types in the future without fear of breaking existing code. Of course, we do have to commit to an implementation class at some point to create objects in an executable system.

A class specifies that it implements an interface—that is, is an implementation class—by including the implements clause:

```
… implements interfaceName …
```

after the class name in the class declaration header. When a class implements an interface, it is declaring that it implements all the methods that the interface defines. It must include corresponding method bodies for each method header listed in the interface. It may define additional methods, both `public` and `private`. Note that `Serializable`—which we use to define persistent classes—is an interface. It doesn't declare any methods, so a class implementing it isn't required to provide anything extra. It simply serves as a type to limit binary I/O to specific classes.

When a class implements an interface, it is considered a subtype of the interface type. This means that objects of the class type may be assigned to variables of the interface type. This is how objects of the type defined by the interface are created. Any number of classes may implement the same interface, as long as they define all the methods included in the interface. Objects of these class types may be assigned, interchangeably, to interface type variables. If we have two classes: `JulianDate` and `GregorianDate` which implement the `Date` interface, the following code is valid:

```
Date  j, g;
    :
j = new JulianDate();
g = new GregorianDate();
meth(j,g):
j = g;
    :
```

The first assignment statement creates a new `JulianDate` object and assigns it to the `Date` variable `j`. Since `JulianDate` is a subtype of `Date`, this is upcasting and is a valid assignment. The same is true for the second assignment. In the method call, `j` and `g` are compatible arguments for the `Date` parameters `d1` and `d2` since `JulianDate` (and `GregorianDate`) are assignment compatible to `Date`. Again upcasting occurs automatically. In the third assignment, the `GregorianDate` object is assigned to `j`, replacing the reference to the original `JulianDate`. `j` and `g` now refer to the same `GregorianDate` object. This is again valid since both `j` and `g` are of type `Date` (no upcasting here) and are thus assignment compatible.

As discussed above, the interface type is normally used to declare variables and parameters. The only place the actual implementation class name is used is in the `new` clause when creating an object.

STYLE TIP

It is good practice to design a class by writing an interface even if you do not intend to have multiple implementations. As discussed above, code written using interface types for declarations instead of class types is much more easily modified. Although you might only think of one implementation when you are designing, things may change and another implementation become required. By using an interface hardly anything needs to be changed—only the object creation. The drawback to using interfaces for each class is the number of additional compilation units—two per class instead of one—and the need to come up with unique names for the class and the interface. (See also Bloch[4], Items 25 and 24.)

4.5 EXCEPTIONS

When a method of a class is called by a client and the method determines that it is being used improperly and cannot do what is asked, what can it do? A class implementation should be written without knowledge of any particular client class since it could be used by a number of different clients. This is especially true if the class is part of a library. Thus the class knows nothing about the client and cannot, for example call a method of the client to deal with the problem.

One solution to this problem is to write the method returning a `boolean` or `int` result—often called an **error code**. The result is used to indicate whether or not the method was able to do its job. This is a solution often used in languages such as C. One drawback of this technique is that the method cannot then return any other value. A probably more important drawback is that the client must test the result value of the method each place the method is called and deal with the result. This causes a much more complex control structure in the client.

Some languages, Java included, provide a mechanism called **exception handling**. An **exception** is an unusual, but not necessarily unexpected, event that may occur during execution. In Java exceptions are divided into three categories: unrecoverable errors, program errors and recoverable exceptions.

An **unrecoverable error** is an exception essentially out of the control of the program such as running out of memory or loss of a component such as hard disk. Unrecoverable errors could occur at any point in the program and there is little the program can do about them except terminate gracefully.

---

[4] Bloch, J.; *Effective Java™: Programming Language Guide*; Addison-Wesley, Reading MA; 2001

A **program error** is an error within the code of the program that is detected at execution time. Examples of program errors are division by zero and indexing out of the bounds of an array. These exceptions are indications of a bug in the program and should not occur during production. They should be detected during testing and rectified in the debugging phase.

A **recoverable exception** is an event that can occur only at particular points in the program and for which there is a well-defined response. An example is reaching end-of-file when reading a file. We know that when we try to read, eventually there will be no more data and end-of-file will occur. However, most of the time the read is successful and only one time will it fail. The exception can be dealt with and processing can continue to complete the tasks required after the file has been read. Standard Java I/O uses the exception mechanism to deal with this case. In `BasicIO`, the library code handles the exception and clients use the `isEOF` method to check for EOF.

A complete discussion of exception handling is beyond the scope of this text. However we will use the exception mechanism to allow a supplier class to signal an error in the client class. Such an error is a result of the client making incorrect use of the supplier method and is a program error, just like indexing out of an array's bounds. As a program error, there is no expectation that the client class will handle it. Rather it will cause the program to crash during testing and the error can then be debugged.

In Java, an exception is an instance (object) of an exception class. To define a new kind of program error, we must define a new exception class. This turns out to be very simple. The code:

```
package Dates;
public class InvalidDateException extends RuntimeException { }
```

defines a program error exception (a `RuntimeException`) as a new class called `InvalidDateException` as part of the package `Dates`. This is just a class declaration with an empty class body. The additional clause (`extends`) is used to specify that `InvalidDateException` is a subclass of `RuntimeException`—the class in Java that defines program error exceptions. As a subtype of `RuntimeException`, `InvalidDateException` may be used wherever a `RuntimeException` is expected. The difference between `implements` and `extends` is that the new class automatically has default implementations of the methods in the superclass and does not have to re-implement them. This is why the class body can be empty. We will not discuss the extends clause further in this text but simply use it in defining program error exceptions.

Throwing Exceptions

Now that we have defined an exception type, we need a way to cause an exception to happen. In Java this is called **throwing** an exception. The form of the `throw` statement is given in Figure 4.4. The *`expression`* must produce an object that is a kind of exception. The result of a throw statement is to cause execution of the method in which it occurs to cease, much like a return statement. The calling method is allowed the opportunity to handle the exception—called **catching** the exception in Java. If the calling method does not handle the exception, it is likewise terminated and its caller is allowed to catch the exception, and so on. This is called **exception propagation**. Finally the `main` method is reached. If it doesn't catch the exception, the program crashes. Since our purpose for using

exceptions is to signal an error in a client class—causing the program to crash so it can be debugged—we will not discuss catching exceptions here.

```
throw expression ;
```

Figure 4.4 Throw Statement

When the supplier method detects a situation that shouldn't occur according to the ADT definition, it uses the throw statement to signal an exception. For example the supplier code might look like:

```
if ( day < 1 | day > DAYS_IN_YEAR ) {
    throw new InvalidDateException();
};
```

This indicates that, if the day number (`day`) is not between 1 and the number of days in the year, it is not a valid date. Assuming that the client supplied the day number, this is a client error. A new `InvalidDateException` object is created and thrown, terminating the supplier method. If the exception is not caught, the client code terminates as well crashing the program. Typically the run-time environment will display the exception and the location at which it was thrown as an aid to debugging.

STYLE TIP

There are three things a supplier code can do in the event of an invalid value being supplied. First, it can signal the violation as an exception, forcing the client to debug the code. This is the best solution. Second, it can make an assumption about what the value should have been, that is "fix" the problem, and continue. This may be acceptable. Finally it can ignore the violation and allow things to progress as they will. This is undesirable since it will likely lead to another exception occurring later in execution when it will be hard to determine the real cause.

4.6 IMPLEMENTATION CLASSES

An implementation class is just a class which implements some abstraction defined by an interface—there is nothing else special about it. A class may implement as many interfaces as it wishes by listing the interface names in its implements clause and supplying all the appropriate method bodies. This means that a class can provide the resources specified in as many interfaces as is appropriate, and is substitutable for any of the interface types. For example, if we want to be able to write `JulianDate` objects to a binary file, we would implement the `Serializable` interface in addition to the `Date` interface. The class header would be:

```
public class JulianDate implements Date, Serializable
```

We will consider two different implementations of the `Date` interface corresponding to the two different representations of dates commonly used. In a Gregorian date, the date is considered as a day within a month within the year for example, May 24, 2014. In a Julian date, the date is considered as a day within the year for example 125/2014. We will represent each as an implementation class as part of the `Dates` package. A client class can declare a `Date` variable and then choose the best representation by using either a `GregorianDate` or `JulianDate` constructor.

An implementation class must provide appropriate constructors for object creation, all the methods of the interface and any other methods considered appropriate. Since from the point of view of the interface type, only the methods defined in the interface are visible, declaring other public methods in an implementation class isn't very useful. However a class would likely define private helper methods supporting the implementation.

Remember one of the responsibilities of the implementation class is to verify that it is being correctly used. When it is not it should throw an exception so that the error in the client code can be debugged. The exception classes are part of the package providing the ADT.

There is one additional method often implemented by a class—the method `toString`. All classes are subtypes of the predefined class `Object`. `Object` defines a number of useful methods for all objects. They are available in all classes, even if we don't write them. One is the method `toString` which gives a printable (`String`) representation of the object. It is useful for quickly displaying the object for debugging or other purposes. The method header is:

```
public String toString ( )
```

If the method `toString` is not implemented in a class, a default implementation that returns the class name followed by a unique number for the specific object is provided. The number can be used to differentiate between objects in the same class. When a more useful printable representation is desired the method may be implemented. It is not necessary to declare `toString` in the interface since it is always there for all classes.

JULIAN DATE IMPLEMENTATION

Figure 4.5 shows the Julian (`JulianDate`) implementation of a date. The class is part of the `Dates` package (line 1) and implements the `Date` interface (line 10). It implements the four methods defined in the interface, `toString` and three constructors. So that `JulianDate` objects can be part of a persistent object, it also implements `Serializable` (line 10) and defines `serialVersionUID` (line 12). Since the number of days in the year changes in a leap year, this value is declared as a constant (line 13) to make modifying the code easier. The representation of the date is simply a day number within the year stored in the instance variable `dayNum` (line 14).

```
1  package Dates;
2  import java.io.*;
3  import java.util.*;
4
5  /** This class defines an implementation of the abstract data type Date
6    * representing a date in the current year (2014) using the Julian
7    * representation (day/year). Date objects are Serializable and immutable.
8    * @author  D. Hughes
9    * @version 1.0 (Feb. 2014)                                                 */
10 public class JulianDate implements Date, Serializable  {
11
12     public static final long serialVersionUID = 9876500021;
13     private static final int DAYS_IN_YEAR = 365;  // days in year 2014
14     private int dayNum;  // ths day number within the year
15
16     /** This constructor creates a Julian date object in the current year for the
17       * given day number. If the day number does not lie within the valid range an
18       * exception will be thrown.
19       * @param  day  day number
20       * @param  InvalidDateException  if day does not fall within the current
21       *                               year.                                   */
22     public JulianDate ( int day ) {
23         if ( day < 1 | day > DAYS_IN_YEAR ) {
24             throw new InvalidDateException();
25         }
26         dayNum = day;
27     };  // constructor
28
29     /** This constructor creates a Julian date object representing today in the
30       * current year.                                                         */
31     public JulianDate ( ) {
32         this(Calendar.getInstance().get(Calendar.DAY_OF_YEAR));
33     };  // constructor
34
35     /** This constructor creates a Julian date object representing the date
36       * specified by the parameter. The format of the string must be day/year each
37       * part being an integer. The year must be the current year and the day an
38       * integer between 1 & the number of days in the year.
39       * @param  s  the year as a string in format day/year.
40       * @throws  InvalidDateException  if the year or day part is invalid.    */
41     public JulianDate ( String s ) {
42         String[]  part;
43         int       d;
44         part = s.split("/");
45         if ( ! part[1].equals(YEAR) ) {
46             throw new InvalidDateException();
47         };
48         d = Integer.parseInt(part[0]);
49         if ( d < 1 | d > DAYS_IN_YEAR ) {
50             throw new InvalidDateException();
51         };
52         dayNum = d;
53     };  // constructor
```

```
54
55      /** This method returns the day number within the year (from Jan 1 = 1). It is
56        * intended solely for use by implementation classes for interoperability.
57        * @return  int  day number within the year (Jan 1 = 1)                    */
58      public int getDays ( ) {
59          return dayNum;
60      };  // getDays
61
62      /** This method compares this date object to other. It returns negative it
63        * this date precedes the other, zero if they represent tha same date and
64        * positive if this date follows the other.
65        * @param  other  other date for comparison
66        * @return  int  <0 if this precedes other, 0 if same date, >0 if this follows
67        *               other                                                      */
68      public int compareTo ( Date other ) {
69          return dayNum - other.getDays();
70      };  // compareTo
71
72      /** This method returns the date days following this date for positive days
73        * values and the date days predeeding this for negative days values.
74        * @param  days  the number of days to advance. Negative values produce
75        *               preceeding dates.
76        * @return  Date  new date before (days<0) or after (days>0) this date.
77        * @throws  InvalidDateException  if new date would not be in the current
78        *                                year                                     */
79      public Date advance ( int days ) {
80          return new JulianDate(dayNum + days);
81      };  // advance
82
83      /** This method returns the number of days between this date and other. If
84        * other precedes this, the result is negative.
85        * @param  other  the date for difference computation
86        * @return  int  the number of days between this and other (<0 if other
87        *               precedes this)                                             */
88      public int between ( Date other ) {
89          return dayNum - other.getDays();
90      };  // between
91
92      /** This method returns a string representation of this date suitable for
93        * display in the format day/year.
94        * @return  String  the string representation of this date.                */
95      public String toString ( ) {
96          return dayNum + "/" + YEAR;
97      };  // toString
98
99  }  // JulianDate
```

Figure 4.5 Example—`JulianDate` Implementation Class

One of the tasks in the design of an implementation class is to decide what constructors should be supplied. Typically there is more than one. However one is often the most “natural”. In this case, a Julian date is naturally defined as a day number within the year so a constructor taking a day number as a parameter is declared (lines 22-27). Sometimes such a constructor is called the **natural constructor**.

In the implementation of the constructor it would be unsafe to assume that the client will provide a valid day number. Therefore the constructor verifies that the day number is valid, lying between 1 and `DAYS_IN_YEAR` (line 23). If it is not it throws an `InvalidDateException` (line 24). Otherwise it sets the day number to the one supplied.

A **default constructor**—the one with no parameters—is automatically defined for any class and does no initialization. A class should always provide an implementation for the default constructor so that all objects are created in a valid state. In this case (lines 31-33) the implementation is that the default constructor would create an object representing the current day. The Java class `Calendar` (from the `java.util` library) provides access to calendars. The expression:

```
Calendar.getInstance().get(Calendar.DAY_OF_YEAR)
```

produces the day number within the year for "today"

Consider line 32. We have seen that the reserved word `this` when used as a "variable" refers to the object itself. It can also be used as a "method name" to call another constructor within the class. This is called **constructor chaining**. The constructor with the appropriate parameters is executed. Constructor chaining can only be used in a constructor within the class and it must be the first statement of the constructor. In our case the default constructor is supposed to create "today". It does this by chaining to the constructor that creates a `JulianDate` based on day number.

Typically, there is one constructor that can be thought of as the most general, and others can be defined in terms of it. By using constructor chaining in all the other constructors, we can avoid duplication of code. Code duplication has a number of drawbacks. It results in more code to write and debug but more importantly, it creates maintenance headaches. If code is duplicated and only one copy of it is noted and changed during maintenance, the code becomes inconsistent. Constructor chaining avoids this maintenance problem. Constructor chaining is not always possible since it must occur as the first statement in the constructor. Another way to avoid code duplication is to place the code in a local method, and invoke it from all constructors.

The third constructor (lines 41-53) is defined to support text input. If a `JulianDate` value is to be read, the easiest approach is to read a `String` and then create a `JulianDate` object by parsing this string. This constructor uses the `String` method `split` to split the string apart at the slash. It converts the day number into an `int` and validates the day and year numbers, throwing an `InvalidDateException` if not. Note that this constructor does not use constructor chaining since the constructor call cannot be written as the first statement in the body.

The methods: `getDays` (lines 58-60) and `toString` (lines 95-97) are straightforward. `getDays` returns the day number, which is the actual representation for a Julian date. The method `toString` which concatenates the day number to the year number with an intervening slash, producing a printable result.

The method `advance` (lines 79-81) is defined in the interface to return a `Date` object. Since `Date` is an interface, we cannot create a `Date` object, instead we create a `JulianDate` object. Since `JulianDate` is a subtype of `Date`, it is compatible with the return type of the method and automatic upcasting occurs. Of course, we could create any kind of `Date` object (e.g. `GregorianDate`). However as the author of the `JulianDate` class, we don't necessarily know

what other implementations of `Date` exist so this way makes the most sense. The method uses the natural constructor to create the new object. Since the constructor does validity checking, it is not necessary to do it in the `advance` method. If the resulting date is invalid, the constructor will throw an `InvalidDateException`. This will cause the `advance` method to terminate, propagating the exception to the client code.

The methods `compareTo` (lines 68-71) and `between` (lines 88-90) both work with respect to another date—the parameter `other`. In each case, what is needed is the day number of the other date. Within a method of a class, full access is possible to the instance variables, including `private` ones, of any object of that same class type. The notation is *`object.name`* (e.g. `other.dayNum`). Unfortunately this is of no help here. The type of `other` is not `JulianDate` but `Date`. The type `Date` doesn't have any instance variables since it is an interface type. Although we could try to downcast `other` to `JulianDate` using `((JulianDate) other).dayNum`, we cannot be certain that `other` is actually a `JulianDate`. There may be other `Date` implementations and this code could cause a run-time exception. Our answer is to use methods of the interface to get at the desired information, in this case the method `getDays`. This situation was considered when the `Date` class was defined and `getDays` was provided for interoperability between implementation classes.

`between` is supposed to return the number of days between two dates, returning negative if `this` comes before `other`. This is simply the difference between the day numbers. `compareTo` is to return negative if `this` precedes `other`, `0` if they are the same date and positive if `this` follows `other`. Again this can be accomplished by simple subtraction of the day numbers.

GREGORIAN DATE IMPLEMENTATION

Figure 4.6 shows a second implementation of the `Date` interface using the Gregorian representation. Like `JulianDate`, `GregorianDate` is part of the `Dates` package and implements the `Date` and `Serializable` interfaces. This means the `Dates` package provides two alternative implementations. The `GregorianDate` class implements the four methods defined in the interface, `toString` and three constructors. The representation of the date is a month number (`month`) and a day number (`day`) within the month.

The constructor (lines 28-45) with two integer parameters (`m` and `d`) is the natural constructor. It verifies that the values are valid and sets the month and day numbers appropriately. Note that the checking is a bit different here. If the month is not positive, it is an error. If the day number is negative the constructor adjusts the month and day numbers accordingly, backing up to previous months. Similarly, if the day number is beyond the end of the month, it adjusts the day and month numbers, advancing to successive months. This allows the creation of Feb. 42 (`GregorianDate(2,42)`) which will be interpreted as Mar. 12. Using a constant array `DAYS_IN_MONTHS` which contains the number of days in each month, it checks the day number versus the number of days in the month. Since, months start at 1 while array indices start at 0, the array is declared with 13 elements, leaving the first (0^{th}) unused. Now, as long as the month is still valid, the date is valid. This implementation of the constructor makes the `advance` method easier to write as we will see later.

```
1  package Dates;
2  import java.io.*;
3  import java.util.*;
4
5  /** This class defines an implementation of the abstract data type Date
6    * representing a date in the current year (2014) using the Gregorian
7    * representation (day/month/year). Date objects are Serializable and immutable.
8    * @author  D. Hughes
9    * @version 1.0 (Feb. 2014)                                                    */
10 public class GregorianDate implements Date, Serializable {
11
12     public static final long serialVersionUID = 987650001L;
13     private static final int FEB = 28;  // days in February in 2014
14     private static final int[] DAYS_IN_MONTH =
15                                         {0,31,FEB,31,30,31,30,31,31,30,31,30,31};
16                                        // days in each month
17     private int  month;  // month number
18     private int  day;    // day in month
19
20     /** This constructor creates a Gregorian date object in the current year for
21       * the given month and day. If possible, the constructor will adjust the month
22       * and day numbers to lie within the valid range. If not, an exception will be
23       * thrown.
24       * @param  m  month number
25       * @param  d  day number
26       * @param  InvalidDateException  if m & d cannot be adjusted to fall within
27       *                               the current year.                          */
28     public GregorianDate ( int m, int d ) {
29         if ( m < 1 ) {
30             throw new InvalidDateException();
31         };
32         while ( m >= 1 & d < 1 ) {
33             m = m - 1;
34             d = d + DAYS_IN_MONTH[m];
35         };
36         while ( m <= 12 && d > DAYS_IN_MONTH[m] ) {
37             d = d - DAYS_IN_MONTH[m];
38             m = m + 1;
39         };
40         if ( m < 1 | m > 12 ) {
41             throw new InvalidDateException();
42         };
43         month = m;
44         day = d;
45     };  // constructor
46
47     /** This constructor creates a Gregorian date object representing today in the
48       * current year.                                                           */
49     public GregorianDate ( ) {
50         this(Calendar.getInstance().get(Calendar.MONTH)+1,
51              Calendar.getInstance().get(Calendar.DAY_OF_MONTH));
52     };  // constructor
53
```

```
54      /** This constructor creates a Gregorian date object representing the date
55        * specified by the parameter. The format of the string must be day/month/year
56        * each part being an integer. The year must be the current year, the month an
57        * integer between 1 & 12 and the day between 1 and the number of days in the
58        * month.
59        * @param  s  the year as a string in format day/month/year.
60        * @throws  InvalidDateException  if the year, month or day part is invalid.*/
61      public GregorianDate ( String s ) {
62          String[]  part;
63          int       m, d;
64          part = s.split("/");
65          if ( ! part[2].equals(YEAR) ) {
66              throw new InvalidDateException();
67          };
68          m = Integer.parseInt(part[1]);
69          if ( m < 1 | m > 12 ) {
70              throw new InvalidDateException();
71          };
72          d = Integer.parseInt(part[0]);
73          if ( d < 1 | d > DAYS_IN_MONTH[m] ) {
74              throw new InvalidDateException();
75          };
76          month = m;
77          day = d;
78      };  // constructor
79
80      /** This method returns the day number within the year (from Jan 1 = 1). It is
81        * intended solely for use by implementation classes for interoperability.
82        * @return  int  day number within the year (Jan 1 = 1)                    */
83      public int getDays ( ) {
84          int num;
85          num = 0;
86          for ( int i=0 ; i<month ; i++ ) {
87              num = num + DAYS_IN_MONTH[i];
88          };
89          num = num + day;
90          return num;
91      };  // getDays
92
93      /** This method compares this date object to other. It returns negative it
94        * this date precedes the other, zero if they represent tha same date and
95        * positive if this date follows the other.
96        * @param  other  other date for comparison
97        * @return  int  <0 if this precedes other, 0 if same date, >0 if this follows
98        *               other                                                      */
99      public int compareTo ( Date other ) {
100         return getDays() - other.getDays();
101     };  // compareTo
102
```

```
103     /** This method returns the date days following this date for positive days
104       * values and the date days predeeding this for negative days values.
105       * @param  days  the number of days to advance. Negative values produce
106       *               preceeding dates.
107       * @return  Date  new date before (days<0) or after (days>0) this date.
108       * @throws  InvalidDateException  if new date would not be in the current
109       *                                year                                    */
110     public Date advance ( int days ) {
111         return new GregorianDate(month,day+days);
112     };  // advance
113
114     /** This method returns the numebr of days between this date and other. If
115       * other precedes this, the result is negative.
116       * @param  other  the date for difference computation
117       * @return  int  the number of days between this and other (<0 if other
118       *               precedes this)                                          */
119     public int between ( Date other ) {
120         return getDays() - other.getDays();
121     };  // between
122
123     /** This method returns a string representation of this date suitable for
124       * display in the format day/month/year.
125       * @return  String  the string representation of this date.               */
126     public String toString ( ) {
127         return day + "/" + month + "/" + YEAR;
128     };  // toString
129
130 }  // GregorianDate
```

Figure 4.6 Example—`GregorianDate` Implementation Class

The default constructor (lines 49-52) uses constructor chaining and the `Calendar` class to create an object representing today. The constructor taking a `String` parameter (lines 61-78) splits the string into three parts and converts and validates each part.

The method `getDays` (lines 83-91) is more complicated than its counterpart in `JulianDate` since it is not the natural representation. The days of the months that precede `month` are accumulated and the day within the month (`day`) added giving the days number from the start of the year.

Like in the `JulianDate` class, `advance` (lines 110-112) makes use of the feature incorporated in the natural constructor to handle moving to previous or successive months. If the date is beyond the end of the year, the constructor throws an exception, which is propagated by this method. A `GregorianDate` is created since we cannot assume that any other implementation classes exist.

Since we were required to implement `getDays` to provide the day number within the year and it is the only helpful method we can assume for the parameter `other`, `compareTo` (lines 99-101) and `between` (lines 119-121) are implemented based on day number. Note the call to `getDays` on `this` object to compute the day number of `this` object.

`toString` simply concatenates the day, month and years into a string separated by slashes.

TESTING IMPLEMENTATION CLASSES

Essentially an implementation class is just a class and should be tested in a manner similar to any other class. The test suites should be developed based on the interface specification and essentially the same suites could be used for all implementation classes. However, there may be an advantage to have some tests based on the implementation specifically, for example testing bounds in the implementation. So a mix of common and specialized tests should be used. As usual, the test output should be self-evident and testing should involve all methods and constructors and all boundary cases.

There is one special consideration in testing an ADT—testing exceptions. Situations that would generate an exception should be tested for all methods and constructors that should generate one. The problem is, if an exception crashes a program each exception test would have to be a separate test program. Luckily Java provides a way around this—**exception handling**.

Earlier we mentioned that a program should handle recoverable exceptions. In fact, Java requires this. The Java statement used to handle exceptions is called the try statement shown in Figure 4.7. The block of text in which an exception might be thrown is written as the *body* of the try statement. This is typically one or more statements that include a call to a method that might throw the exception, and is called the "normal" code. A catch clause for the exception is included at the end of the try statement. The catch clause is introduced by the keyword `catch` and identifies the exception it is handling by writing the exception type as the *type* in the parameter list. Finally, the code to handle the exception is written as the *handler* in the catch clause. This is the "exceptional" code.

```
try {
    body
}
catch ( type paramName ) {
    handler
}
```

Figure 4.7 Try Statement

The try statement is executed as follows. The *body* is executed. If an exception is not thrown, the try statement is completed. The result is the same as the *body* written without the try. If an exception is thrown, the *body* is terminated right at the point of the exception being thrown, the rest of the *body* is omitted. The *handler* for the exception is executed instead of the rest of the *body* and the try statement is completed and execution continues after the try.

We will consider this statement only as it allows us to test that an ADT generates exceptions when it should. If the ADT generates exceptions when it shouldn't, the test crashes and debugging can ensue. Our desired goal is to execute a method call that should lead to an exception and note whether or not the exception occurred. And we want to do this without crashing the program! For example, to test that the `JulianDate` constructor generates an exception when a date past the end of the year is created, the following code may be used:

```
out.writeString("Day 375 is ");
try {
    j = new JulianDate(375);
    out.writeString("valid");
}
catch ( InvalidDateException e ) {
    out.writeString("invalid");
};
out.newLine();
```

Here the constructor is passed an invalid parameter. If the exception is thrown by the constructor, the body containing the constructor call is terminated omitting the statement `out.writeString("valid");`. The exception is handled by the handler consisting of the code `out.writeString("invalid");`. If the exception isn't thrown, the constructor call completes, as does the following statement, and the try statement is completed. The result is the message: `Day 375 is invalid` if the exception was thrown and: `Day 375 is valid` if not. It is now easy to verify in the test output.

STYLE TIP

Exception handling should not normally be used for program errors. Instead, the program should be allowed to crash and then debugged. A program that catches a program error and simply ignores it is not a correct program even if it doesn't crash! The only variation might be for the production version of a system where, at an outer level in the program logic all exceptions including program errors, unrecoverable errors and previously unhandled recoverable exceptions are caught. The handler code informs the user, permits her to save her work, and gracefully shuts down the program probably asking the user to report the problem. Such a program is often called "**fail-soft**" as it doesn't crash but rather shuts down gracefully in the event of an error.

4.7 BUILDING LIBRARIES

If an ADT is to be used in more than one application, it makes sense to build it as a library that can then be imported into other applications. In Java, as discussed in section 4.3, a package is the basis for libraries. A single ADT (the `Dates` package) or a group of related ADTs (the `Collections` package discussed in Chapters 6, 8 and 9) can be declared to be part of a package and then compiled into a library.

A compiled library doesn't normally contain an executable program—with the exception maybe of test harnesses. Rather it is a collection of compiled classes (`.class` files) that can be accessed within an application program via the `import` statement. To create the library, it is necessary to combine the `.class` files into a Java Archive file. (`.jar`). A `.jar` file is really just a compressed version of the `.class` files in a directory including an additional file called the manifest which contains some information about the library. The standard Java distribution includes a program to create a `.jar` file from a package and is usually directly available within an IDE.

Since many different programmers in many different organizations may develop libraries, it is necessary to inform the compiler where to look for libraries that are used by an application. The Java compiler uses a list of directories called the **classpath** to locate imported libraries. If an application

uses a library that is not one of the standard ones, the directory containing the `.jar` file must be added to the classpath. Most IDEs have an option for doing this.

When a package is used by programmers other than the author, it is necessary to provide documentation. We have been writing our programs including class comments at the start of each class and method comments at the beginning of each method. In these comments we have included tags such as `@author` and `@param`. The `JavaDoc` tool included with the standard Java release can be used to generate online documentation for a package. It copies the class and method comments and processes the tags to generate a hyper-linked documentation. Figure 4.8 is the generated page for the `Date` interface from the `Dates` package. Typically the IDE has an option for running the JavaDoc processor.

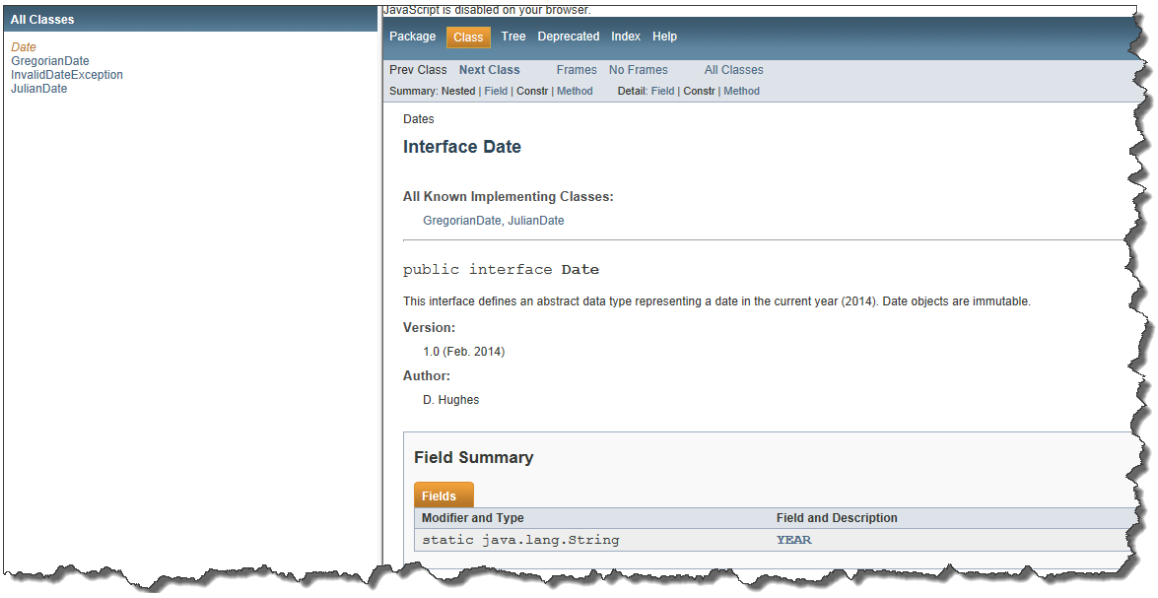


Figure 4.8 Generated Documentation for `Date` Interface

CASE STUDY: THE FRACTION ADT

Although fractions are common in arithmetic, they are not directly supported in most programming languages or by computer hardware. Instead an approximation to the fraction is used—typically `float` or `double` in Java. Although for many computations this is adequate, in some cases an approximation doesn't suffice. Try, for example, adding together 1.0/3.0, 300,000 times using `double` arithmetic. The result is `99,999.9999996892` instead of `100,000.0`!

An ADT for fractions should support fractional arithmetic, comparison and conversion to and from other arithmetic types for mixed-mode arithmetic. The ADT should be part of a library package so that it can be used in any program requiring fractions.

```
1  package Fractions;
2  /** This interface defines a fraction type for manipulation of fractions.
3    * Fractional values are immutable.
4    * @author D. Hughes
5    * @version 1.0 (Jan. 2014)                                              */
6  public interface Fraction {
7
8      /** This method returns the numerator of the fraction. It is for
9        * interoperability and should only be used by implementation classes.
10       * @return int the numerator.                                        */
11     public int getNumerator ( );
12
13     /** This method returns the denominator of the fraction. It is for
14       * interoperability and should only be used by implementation classes.
15       * @return int the denominator.                                      */
16     public int getDenominator ( );
17
18     /** This method returns the double value equivalent of the fraction.
19       * @return double equivalent double value.                          */
20     public double doubleValue ( );
21
22     /** This method returns the integer value equivalent of the fraction.
23       * @return int equivalent integer value.                            */
24     public int intValue ( );
25
26     /** This method returns <0, 0, >0 if this fraction is less, equal or greater,
27       * respectively, than the other.
28       * @param f the fraction for comparison.
29       * @return int <0 for less, =0 for equal, >0 for greater  than f     */
30     public int compareTo ( Fraction f );
31
32     /** This method returns the sum of this fraction and another.
33       * @param f the other addend
34       * @return Fraction this + f                                         */
35     public Fraction add ( Fraction f );
36
37     /** This method returns the difference between this fraction and another.
38       * @param f the subtrahend
39       * @return Fraction this - f                                         */
40     public Fraction sub ( Fraction f );
41
42     /** This method returnsthe product of this fraction and another.
43       * @param f the multiplier
44       * @return Fraction this * f                                         */
45     public Fraction mul ( Fraction f );
46
47     /** This method returns the quotient of this fraction and another.
48       * @param f the divisor
49       * @return Fraction this / f
50       * @throws  ZeroDenominatorException if the result has a zero denominator.  */
51     public Fraction div ( Fraction f );
52
53 } // Fraction
```

Figure 4.9 `Fraction` Interface

Figure 4.9 shows an interface defining a fraction ADT as type `Fraction`. Of course, in designing an ADT there is considerable flexibility in what operations to provide and how they should be provided. We will consider a minimal set of operations to provide reasonable support for arithmetic using fractions so that the problem doesn't get too large. There is a good argument for including many more operations to make the use of the ADT more convenient to use.

Since operations on fractions involve manipulation of the numerator and denominator, methods providing these (`getNumerator` and `getDenominator`) are included for interoperability between representations. A fraction is a numeric entity and will be involved in computations with other numeric entities in mixed-mode arithmetic. For this to happen, conversion to and from other numeric types is required. The methods `doubleValue` and `intValue` provide conversion to other numeric types. In both cases, there can be some loss of information. Certainly converting to `int` is narrowing since both 1/1 and 4/3 would convert to 1. Conversion to `double` also involves possible loss of information since `double` is an approximation. Constructors can be defined to handle conversion from other numeric types into `Fraction`. An implementation class would likely include an implementation of `toString` and possibly other constructors.

Comparison of `Fractions` is supported in the usual way via the method `compareTo`. The four standard arithmetic operations are provided as the methods `add`, `sub`, `mul` and `div`. In each case, the method returns a new `Fraction` object as the result of the operation. If you consider the methods of the `Fraction` interface you will note that, once a `Fraction` object has been created—either by a constructor or as the result of an arithmetic operation—it can never change. Thus `Fraction` is an immutable type.

The values we commonly call fractions are formally called Rational numbers in Mathematics and are defined as a ratio of two integers p and q, with $q \neq 0$. This definition gives us an obvious representation for fractions as a pair of integer values: the numerator and the denominator. This appears straightforward enough however, it is not the complete answer. When choosing a representation for an ADT, it is desirable that there is only one representation for each unique value of the ADT. If we consider the fractions: 1/3, 2/6, 3/9, etc., we realize that each is actually the same value. When we first studied fractions in grade school, we were taught to "reduce the fraction to its lowest terms". We would never write 2/6 but rather the equivalent 1/3.

The choice of a single representation for a unique value of a type is called a **canonical** or **normal** form. We need to choose such a canonical representation for our fractions to avoid potential problems with duplicate representations. Clearly one part of the representation would be the rule that the fraction is always reduced to its lowest terms. We still do not have a canonical representation however. Consider 1/3, -1/3, 1/-3 and -1/-3. These are in their lowest terms, however they represent only two unique fractions: 1/3 and –1/3. The usual rule is that the sign is always associated with the numerator—the denominator is always positive. What about mixed fractions like 1 1/3 or 5 7/8? As rational numbers they would be 4/3 and 47/8, respectively. This presents no new problems. Similarly, the whole numbers such as 2 or 5 would be represented as 2/1 and 5/1. The only remaining problem is the representation for 0. Clearly 0/1, 0/2, 0/3, etc. are all possible representations that are in their lowest terms. To obtain a canonical representation, we must pick one. It seems reasonable to choose the same representation as the rest of the whole numbers, so 0/1 can be used. Finally, the denominator can never be zero since such a thing would not be a rational number. Table 4.1 summarizes these rules:

Rule	Example
reduced to lowest terms	2/3 not 4/6
denominator positive	-1/3 not 1/-3 and n/0 is undefined
zero is 0/1	not 0/2

Table 4.1 Normalization Rules

The constructors for the implementation class should ensure that fraction objects are only constructed in their canonical form. Since fractions are immutable this is sufficient to ensure that all fractions are in normal form. The constructors should also ensure that only valid fractions are created, throwing an exception otherwise (i.e. if the denominator would be zero).

The standard rules for fractional arithmetic are given in Table 4.2. The `add`, `sub`, `mul` and `div` methods implement these definitions. `compareTo` can be implemented based on subtraction since it is defined to return negative if the first value is smaller than the second, zero if they are equal and positive if the first is larger than the second.

addition	$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$
subtraction	$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$
multiplication	$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$
division	$\frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}$

Table 4.2 Fractional Arithmetic

SUMMARY

Data abstraction is a mechanism for reducing complexity in large systems. In data abstraction we concentrate on the set of objects and the operations they provide and de-emphasize the representation of the objects and implementation of the methods. Interfaces in Java provide the mechanism for defining a data abstraction or abstract data type. Classes serve to provide the representation and implementation details. Information hiding is the process by which we hide the details within a class by making the instance variables (representation) private and only those methods defined in the interface public. Such a process provides reduction of complexity, security within the class and modifiability of the system being developed.

A package is a collection of interfaces and classes that are related and can be used to combine the interfaces and classes that make up an ADT. An interface is a collection of constant and method declarations, without the specification of a representation or implementation. They can be used to define an ADT. Interfaces define types so variables can be declared of the interface types. A class can

be declared to implement an interface, guaranteeing that it implements all the methods specified in the interface. A class can be used to provide an implementation of and ADT. Objects of such classes can be assigned to variables of the interface type.

An exception is an unusual, but not necessarily unexpected, event that may occur during execution. In Java exceptions are instances of exception classes. Exceptions can be used to signal error conditions that are detected by an ADT to the client class. Although Java provides a mechanism for handling exceptions, these exceptions indicate program errors and are usually allowed to crash the program during testing to allow debugging to ensue.

When implementing an ADT as defined by an interface, a number of decisions must be made. One decision is how many and which constructors should be provided since this is not specified in an interface. When multiple constructors are defined, constructor chaining should be used to reduce code duplication and the subsequent maintenance problems this would cause. A second decision is to choose a representation for the ADT. Some sets of values of the instance variables may not represent valid instances of the ADT. The method implementations must ensure that such instances to not occur, typically throwing exceptions when they would.

REVIEW QUESTIONS

1. T F A class declared `public` in a package is visible only to other classes of the package.

2. T F An interface defines a type.

3. T F A package is a collection of classes.

4. T F A `RuntimeException` must be caught by the program.

5. T F Constructor chaining is used when the constructor of a class needs to call the constructor of another class.

6. T F An instance of an interface may be substituted for an object of a class which implements the interface.

7. T F Programming-by-contract describes the situation where a programmer is paid to write an implementation class.

8. A set of values and the operations upon them defines:

   a) a data type
   b) a data structure
   c) an abstract data type
   d) a class

9. Information hiding provides which advantage?

   a) modifiability
   b) reduction of complexity
   c) security
   d) all of the above

10. An unusual event that can occur only at particular points during execution is called a:

    a) recoverable exception
    b) program error
    c) unrecoverable error
    d) `RuntimeException`

11. An array of integers is:

    a) a primitive type
    b) a data type
    c) a data structure
    d) an abstract data type

12. An immutable object is one that:

    e) cannot be cast to a different type
    a) cannot change state
    b) cannot be represented as a table
    c) none of the above

13. A canonical form is a:

    a) unique representation for each value
    b) a normal form
    c) used in representing fractions
    d) all of the above

14. The `toString` method:

    a) must be written for every class
    b) converts a `String` into an object
    c) is defined in class `Object`
    d) all of the above

15. The try statement:

    a) is used to test exception generation
    b) is used to handle exceptions
    c) can be used to make a program fail-soft
    d) all of the above

EXERCISES

1. In order to represent playing cards for programs that play card games, an abstract data type (ADT) representing playing cards is to be developed. Define an interface and implementation of the `Card` abstraction as follows.

   Each card has a suit: clubs, diamonds, hearts and spades and a rank: ace (1), 2, 3, ..., 10, jack, queen, king. The methods `getRank` and `getSuit` return the rank, a number from 1 (for ace) to 13 (for king) and suit, a number from 1 (for clubs) to 4 (for spades), respectively. Each card has a

count (or value) which is usually the number of pips on the card, that is 1 for ace, 2 for 2, etc., with the face cards jack, queen and king all having the count 10. The method `getCount` returns this value. In some games a card can have more than one possible count. For example, in blackjack an ace can count either 1 or 11. The method `hasAltCount` returns `true` if the card has an alternate count and `getAltCount` returns the alternate count value or the same value as `getCount` if the card has only one possible value. The method `compareTo` returns `0` if the card represents the same card—the same rank and suit—as the other card negative if the card precedes the other card in the sorted order or positive if the card follows the other card. The order is rank: ace through king, within suit:, clubs, diamonds, hearts, spades. Finally, the method `toString` returns the `String` representation for the card in the form: *`rr_s`*, where *`rr`* is `A`, `2`, `3`, ..., `Q`, `K`; `_` is a space and *`s`* is `C`, `D`, `H`, `S`. The interface defines the constants: `CLUBS`, `DIAMONDS`, `HEARTS` and `SPADES` to make it easier to use these symbolic names for the suits instead of the integers 1 through 4. They would be used in client code as, for example, `Card.CLUBS`. Similarly, it defines the constants: `ACE`, `JACK`, `QUEEN` and `KING` (1, 11, 12, and 13, respectively) for the named cards.

Write a class that implements the `Card` abstraction. Choose a representation for cards and implement the operations specified in the `Card` interface. You should also provide a number of constructors. The constructors should ensure that the card constructed is a valid card (i.e. suit is between 1 and 4 and rank is between 1 and 13). There should be a default constructor, creating some valid card, a constructor that takes a rank and suit and one which takes a `String` representing the card, in the same form that `toString` generates.

Write a test harness to test your card ADT.

2. Develop a library package to provide support for working with time values. Time may be divided into hours, minutes, seconds and milliseconds (thousandths of a second) or may simply be a number of milliseconds. Two times may be added, subtracted or compared and a time may be multiplied or divided by a numeric value, for example, doubling a time or cutting a time in half.

   The interface `Time` specifies this abstraction of time. `Time` objects are immutable. It is possible to determine the total number of milliseconds, or the hours, minutes, seconds and milliseconds components of a `Time`. Adding or subtracting `Time` objects results in a new `Time` object. Multiplying or dividing a `Time` by an integer results in another `Time`. It is possible to compare `Time` objects to determine if they are equal or one is less than the other. Negative `Time` values are not supported.

   A `Time` may be represented as a set of four integers representing the hours, minutes, seconds and milliseconds of the time. The canonical form of a time: `h:m:s.t` would have `h>=0`, `60>m>=0`, `60>s>=0` and `1000>t>=0` where `h` is the hours, `m` is the minutes, `s` is the seconds and `t` is the milliseconds. Creation of a negative time would throw a `NegativeTimeException`.

   As part of a package called `Times`, write a class HMSTime which implements the `Time` interface using the representation above. In addition to the methods of the `Time` interface, include appropriate constructors: at least a default constructor creating time: `0:0:0.0`, one that takes four `int` parameters `h`, `m`, s, `t` and creates the time: `h:m:s.t` and the method `toString` which

should generate a string of the form `h:m:s.t` with `t` always having `3` digits. If a negative time occurs, a `NegativeTimeException` should be thrown.

Write a test harness to test your `HMSTime` implementation. In the test harness, the time objects should be declared of type `Time` and your test harness should test every method and constructor in appropriate situations, especially limiting cases such as zero times, as well as testing exception generation. Each test should clearly indicate what is being tested and the correctness of the test should be clear.

3. When you go to the ATM (automated teller machine), you simply insert your card, answer a few questions and out comes money. Amazing! You can even go to a machine in another country and withdraw money in that country's currency and your account is automatically debited the correct amount in your native currency. How does this all work? Fundamental to the system must be a way of representing currency that is independent of the country in which the ATM is being used and the bank of origin.

   To design a `Currency` abstraction we have to determine what one does with currency, or at least what is done with currency values within the ATM system. Clearly it must be possible to add currency amounts (for a deposit) and subtract currency amounts (for a withdrawal). The system must be able to determine if you have enough currency in the account to make a withdrawal. This means it must be possible to compare currency objects for equality and ordering. To grant interest, a currency amount must be able to be multiplied by a numeric (`double`) value. Since currency amounts must be displayed, there must be a conversion to an appropriate string representation.

   With international transactions a problem occurs. It is necessary to do conversions between different currencies when, for example, you wish to withdraw Euros form your Canadian dollar account. A way this can be handled is that, for international banking purposes, all values are considered to be in an international monetary unit, and essentially converted to and from this unit to allow operations on dissimilar units. This will require a method that returns the equivalent amount in the international units for any currency amount.

   Design an interface called `Currency` that captures the essence of the currency abstraction.

   The `Currency` abstraction is useful. However, real currencies are used in practice, not some imaginary international money. Most countries have their own currency. In computing terms, we need concrete classes for each currency. For example, a `USDollar` class, a `CanDollar` class and a `Euro` class, which implement the `Currency` interface. Each of these will implement the `Currency` methods in their own native way. For example, the string representation for US dollars would be something like `$123.45` while it would be something like `123.456 E` for Euros. And, of course, each actual currency would have a particular conversion factor to international monetary units. When doing computations involving another `Currency` object, the methods cannot assume it is the native currency but can only work with its international monetary unit value.

   Write concrete classes to support US dollars, Canadian dollars and Euros. For the US and Canadian dollar classes, we desire exact representations of dollars and cents so it is easiest to represent the value as an integral number of cents. Use `long` since otherwise the largest value would be `2147483647` cents or only `$21,474,836.47`. The exchange rate for US dollars to

international units is 1.00, for Canadian dollars is 0.9035. For Euros, simply use `double` as the representation. The exchange rate for Euro to international units is 1.25. Each class must implement all the methods in the `Currency` interface and have appropriate constructors.

Write a test harness that tests your `Currency` classes. Be sure to test international transactions, for example, withdrawing Euros from your US dollar account.

4. Develop a library package to provide support for set manipulation. A set is said to cover some elements. For example, we can talk of a set of integers where the set covers the integers or a set of employees where the elements are employees in a company. A set contains certain of the elements that it covers. The individual elements are not physically part of a set but are considered to be included (or not) within the set. For example, the employees to be considered for promotion can be thought of as a set. Each employee to be considered is included in the set and the rest are not included.

   There are a variety of operations that can be performed on sets. It is possible to determine how many of the elements a set contains. This is called the cardinality of the set and is a number between 0 and the total number of possible elements. A set can be empty, in which case it contains none of the elements and its cardinality is 0. It is possible to determine if a particular element is contained in a set. An element can be included into the set, that is, become contained whether or not it was previously contained in the set. The union of two sets is the set that contains elements that are in either of the sets—contained in one or the other or both. The intersection of two sets is the set of elements that are contained in both sets—the elements they have in common. The difference between two sets is the set of elements that are contained in the first set but not contained in the second.

   As part of a package called `Sets`, write an interface `IntSet` that specifies the abstraction of a set which covers (whose elements are) the integers in the range `0...SET_SIZE-1`. `SET_SIZE` is a constant defined by `IntSet`. It includes specification of the operations described above, cardinality, empty, contains, include, union, intersection, difference. `IntSet` is mutable—operations change the actual set rather than producing new sets. The union, intersection and difference methods modify the set object to be the union, intersection or difference between the object and its parameter. That is, `s.union(t)` modifies `s` to include all elements that are in either `s` or `t`.

   A set of integers can be represented by a bitset. A bitset is an array of bits (`boolean`) with one element for possible element in the set. For example, for the set covering the integers from 0 to 31, there would be an array of 32 `boolean` elements. Each element in the bitset indicates whether (`true`) or not (`false`) the corresponding integer is contained in the set. For example, if the set contains 3 and 5, the array would have `false` in elements `0...2`, `4` and `6...31` and `true` in elements `3` and `5`.

   As part of the `Sets` package, write a class `BitIntSet` which implements the `IntSet` interface using a bitset. Include appropriate constructors—at least a default constructor creating an empty set and a constructor which takes an array of integers and creates a set containing those integers. Also implement the method `toString`. If an attempt is made to include an element out of the range `0...SET_SIZE-1` into a set, an appropriate exception should be thrown.

Write a test harness to test your `BitIntSet` implementation. In the test harness, the set objects should be declared of type `IntSet`. Your test harness should test every method and constructor in appropriate situations, especially the limiting cases such as empty sets, as well as test exception generation. Each test should clearly indicate what is being tested and the correctness of the test should be clear.

5. Write a program using the Card ADT from Exercise 1 to play blackjack with the user. The computer will play the part of the dealer.

   Blackjack is a simple game in which the player who is closest to 21, in total card value, but not over is the winner. Each card has a value as given below:

   - ace is 1 or 11 as the need arises
   - 2 through 10 are valued at 2 through 10 respectively
   - jack through king are valued at 10 each

   The game starts with the player and the house (dealer) each being dealt one card face up on the table. The player is then dealt a second card. From this point on the player has a choice of being dealt more cards, trying to get as close to 21 as they can without going "bust" by exceeding 21. The house is then required to take cards, face up, until its count exceeds 16. In other words the house cannot stop with fewer than 17 points and cannot take more cards once they reach 17 points or more. If the player has bust, s/he loses, even if the house also busts. If the house busts, the player wins, otherwise the winner is the one with the highest total. A tie is considered a win for the house.

   The person running the program will be playing against the program as the house. A new deck of 52 cards is created, shuffled and play begins. The player and the house will each be given a card. The player will then be given a second card. At this point the cards, the house's one and player's two, should be displayed. Now the player is given the option to hit (take another card) or stick (satisfied with the total and refuse any more cards) until s/he is satisfied. You may assume s/he will quit as soon as s/he busts. Each card is displayed as it is taken. The program (house) will then take cards, displaying each one, in accordance with the rules given above.

   After the hand has been completed, the program should evaluate the player and house hands, summing the card values, display the totals and determine and display the winner. If there is an ace in a hand, it should be considered as 1 or 11 to maximize the value without exceeding 21, if possible.

   The program will then ask the player whether or not s/he wishes to play again. If s/he plays again, they continue to play with the remainder of the same deck that is neither shuffled nor changed. If, at any time during the play, the deck becomes empty, a fresh deck is created and shuffled and play continues. When the player decides to quit, the program should print out the number of games played and the number of games won by the player.

   User input can be done using an `ASCIIPrompter`. The program output can be done using an `ASCIIDisplayer`. The following is sample output from one game:

```
Dealer's card: 9 C
Your hand: 2 H, 6 C, 7 C, 5 C
Dealer's hand: 9 C, 8 C
Dealer count: 17
Your count: 20
You win
```

Hint

The card deck and the hands can be arrays of cards (52 for the deck and 12 for the hands). The deck array can be initialized with one of each card and then shuffled. A variable, initially 0, can be kept to indicate the index of next card to be dealt, and when this equals the deck size, the array is reloaded with cards and shuffled and the dealing starting again at position 0. Each hand array contains from 0 to 12 cards at any time during play.

Shuffling can be done by, some number of times, choosing two cards at random and exchanging them within the card deck array. The random choice can be done by generating two random numbers between 0 and 51. Doing this twenty or thirty times should leave the deck reasonably well shuffled.

It would probably be useful to have a number of methods to help out the main method. Consider, for example, methods for loading the deck with cards, shuffling and evaluating a hand.

6. Rewrite the solution to Chapter 3 Exercise 2 as a library ADT. Design the interface and then rewrite the code as an implementation class for the interface type. The ADT should support addition and subtraction of polynomials, evaluation of a polynomial for a given value of `x` (`double`). The addition and subtraction methods should modify the polynomial to be the sum (or difference) between the polynomial and the parameter. That is, the `Polynomial` type in **not** immutable.

   The implementation class should provide at least three constructors. The default constructor should produce a zero polynomial (no terms). A constructor taking two `int` parameters (coefficient and exponent) produces a polynomial of a single term. Input of polynomials can be done in client code by repeatedly reading (coefficient, exponent) pairs, creating a one term polynomials and adding them to the result. A `toString` method should be provided to support polynomial output.

   Test your ADT using a modified version of the code from Chapter 3 Exercise 2 as a test harness.

7. Write an ADT `BigInteger` that supports working with integers of indefinite size based on your solution to Chapter 3 Exercise 3. It should support addition and subtraction of `BigInteger` values. The addition and subtraction methods should produce a new `BigInteger` value, so `BigInteger` values are immutable. A `toString` method should produce a string of digits preceded by a sign. Constructors should include the default constructor producing zero, one taking a `long` parameter and one taking a `String` with a sign followed by digits.

   Since subtraction is defined both positive and negative integers must be supported. The sign can be stored as either 1 or -1 in the digit field of the header node.

For addition with signed values, when the signs of the two numbers are different, we can use the nines-complement of the negative number and then add. The nines complement of a number is the number produced by subtracting each digit from 9. For example, the nines-complement of 827 is 172. After the addition, if the final carry is 1, 1 is added to the result in the 1s position, possibly carrying to positions to the left, and the result is positive. If the final carry is 0, the result is negative and the nines-complement must be taken.

As an example, consider the following cases:

```
+182      +182      -182      -182    addends
+246      -246      +246      -246
----      ----      ----      ----
 182       182       817[1]    182    remove sign
 246       753[1]    246       246
----      ----      ----      ----
 428       935      1063       468    add
----      ----      ----      ----
           064[2]    064[3]            carry?
----      ----      ----      ----
+468       -64[2]    +64[3]   -468    apply sign
```

In the second and third cases, since the signs are different, the negative value is converted to nines-complement ([1]). In all cases the addition now proceeds ignoring the sign. In the first and fourth cases, since the signs are the same, the final carry is treated as an extra digit. Since it is 0 in these cases, it is ignored. The sign of the result is the same as the sign of the addends. In the second case, since the signs are different and the final carry is 0, the result is complemented and is negative ([2]). In the third case, since the signs were different and the final carry is 1, 1 is added to the result and the sign is positive ([3]).

Of course, subtraction is just addition after negating of the subtrahend.

Rewrite your code from Chapter 3 Exercise 3 as an implementation class for `BigInteger` and as a test harness to test your ADT. Include tests of negative values and subtraction.

5 STACKS

CHAPTER OBJECTIVES

- Define the stack abstract data type.
- Explain the behavior of a stack.
- Implement a stack using contiguous representation.
- Implement a stack using linked representation.
- Describe the postfix notation for expressions.
- Understand the connection between postfix notation and hardware evaluation of expressions.
- Convert from infix to postfix notation.

The stack is the first of the three fundamental ADTs of Computer Science that we will examine in this book. As was mentioned in Chapter 4, each of these—the stack, queue and list—is a container ADT. A container or collection ADT is one that collects together or contains a number of items or objects of some type. Typically operations to add and remove items are defined along with mechanisms for accessing the objects.

What differentiates the ADTs is the way in which the items are accessed. The three ADTs we will study—stack, queue and list—are all list-oriented collections or lists. A **list-oriented collection** is one in which a positional ordering is defined—there is a first item, a second and so on. Addition, removal and access are defined based on the position within the list. A second kind of ADT is a **keyed collection**. Each item is associated with a key and addition, removal and access are based on the key. We will not discuss keyed-collections in this text.

For any ADT there is a variety of representations and implementations. For each of the three discussed, we will consider two representations: one based on arrays—called a **contiguous implementation** since the memory is allocated as one contiguous piece—and one based on linked structures—called a **linked representation** in which memory is allocated as needed.

5.1 THE STACK ADT

A **stack** is a list of items of some type that is initially empty. Items may be added or **push**ed at one end called the **top** and items may be removed or **pop**ped also from the top.

A common example of a stack is a Pez™ candy dispenser (Figure 5.1). When the top is raised it dispenses the top candy from the package and another is pushed up by a spring to take its place. A new candy can be inserted on top of the rest. Since new items are added on the top, the next candy to be removed from the dispenser must be the last one that was added. Stacks are said to exhibit the **Last-In-First-Out (LIFO)** property.

Figure 5.1 Pez™ Dispenser[5]

Let's look at a stack in operation. For simplicity assume that the items being placed on the stack are integers. Figure 5.2 shows a stack as it is transformed by the operations of pushing the integers 1 through 5 in order onto the stack; four consecutive pop operations removing the integers 5 through 2 leaving only 1 on the stack. Subsequent push of 6, 7 and 8 and then four more pops reduce the stack to the empty state. Note that the items of the stack are removed in the reverse of the relative order that they were pushed—5 before 4 before 3 etc. and 8 before 7 before 6 before 1.

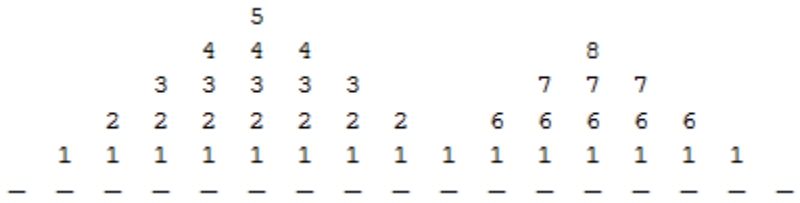


Figure 5.2 A Stack in Action

If we were to look at the Pez dispenser we could tell what the top candy was. But that would be about all. Without removing all the candies, we would not know how many were on the stack nor in what order they occurred. About the only other thing we could tell was whether or not there were any items on the stack at all—whether or not the dispenser was empty.

These observations give us a basis for defining the operations of a stack. Items can be pushed or popped at the top. The top item can be examined. The stack can be checked to see if it is empty. Not all operations have a valid meaning at all times. If the stack is empty an item cannot be popped nor

[5] © Deborah Austin. Used unchanged under CC by 2.0 license (http://creativecommons.org/licenses/by/2.0/legalcode)

can the top item be examined. This situation is called **stack underflow**. Since any representation of the stack will occupy storage and there is only finite memory available, it can happen that another item cannot be pushed onto the stack. This situation is called **stack overflow**.

5.2 THE `CharStack` INTERFACE

We now will consider an interface specifying the stack ADT. The stack is to contain items of some type for example a stack of candies. In our case we will design the stack to be a stack of characters (`char`). The interface declaration of the character stack ADT is given in Figure 5.3. We will see in Chapter 6 that we can generalize our definition so that a common stack interface and classes can be used to create stacks of items of various types.

```
1  package CharStacks;
2
3  /** This interface defines the abstract data type stack representing a LIFO
4    * collection of characters.
5    * @see  OverflowException
6    * @see  UnderflowException
7    * @author  D. Hughes
8    * @version  1.0 (Jan. 2014)                                              */
9  public interface CharStack {
10
11     /** This method adds an item to the stack. Stack overflow occurs if there is no
12       * room to add another item.
13       * @param  item  the item (char) to be added.
14       * @throws  OverflowException  no more room to add to stack.           */
15     public void push ( char item );
16
17     /** This method removes an item to the stack. Stack underflow occurs if there
18       * are no more items left.
19       * @return  char  the item (char) removed.
20       * @throws  UnderflowException  no items available in stack.           */
21     public char pop ( );
22
23     /** This method returns the top item of the stack. Stack underflow occurs if
24       * there are no more items left.
25       * @return  char  the top item (char).
26       * @throws  UnderflowException  no items available in stack.           */
27     public char top ( );
28
29     /** This method returns true if the stack contains no items.
30       * @return  boolean  whether the stack is empty.                       */
31     public boolean empty ( );
32
33 } // CharStack
```

Figure 5.4 Example—Character Stack ADT Interface

As part of the `CharStacks` package the `CharStack` interface defines four methods. The method `push` adds a character (`item`) to the stack. The function `pop` removes the top item from the stack, returning it. The function `top` returns the top item on the stack without removing it. Finally the function `empty` returns `true` if the stack contains no items and `false` otherwise.

The two exceptional conditions are represented by the exceptions (see Section 4.5): `OverflowException` for stack overflow and `UnderflowException` for stack underflow. `push`, `pop` and `top` throw these in exceptional conditions to signal a programming error.

5.3 CONTIGUOUS IMPLEMENTATION OF `CharStack`

If we look at the behavior of a stack, for example Figure 5.2, we see that while the top moves up and down, the base stays rooted in place. This lends itself well to an array implementation based on the variable-sized array technique (see Section 1.2). If we fix the base of the stack at one end of the array, we can add new items towards the other end and remove items from there as well. These operations will not affect the placement of the items already in the stack and so will be `O(1)`. If we maintain an index variable (`top`) that indicates the first open element position in the array—the position just above the top item—we have a simple solution. Such an implementation of a stack is shown in Figure 5.4. The shaded areas represent the items on the stack. top is actually an index (`int`) variable with value 5, although the diagram shows it as an arrow for clarity.

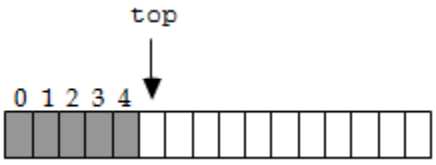


Figure 5.4 Contiguous Stack Representation

The implementation class is shown in Figure 5.5. As part of the `CharStacks` package, the class `ConCharStack` implements the `CharStack` interface. It defines two instance variables: `top`—the top open element index—and elts—the stack items themselves—to represent the ADT. A constructor is provided which allows the client to supply a value (`size`) for the maximum number of items that the stack should contain. This is necessary since arrays must be allocated a size when created. The constructor allocates the array and sets `top` to 0, indicating that the stack is empty—indicating the first element in the array is the first open element. The default constructor sets the stack to a predefined size (100 elements) using constructor chaining.

The operations are implemented in a straightforward manner. `push` first tests to see if there is room for another item in the stack—`top` is less than `elts.length`. If not it throws an `OverflowException`. Otherwise it stores the new `item` at the first available position in the array (`elts`) and increments `top`.

`pop` and `top` are similar. They first test to see if there are any items in the stack—`top` is greater than `0`—and throw a `UnderflowException` if not. If all is OK, they return the item at the top of the stack, with `pop` decrementing `top` to indicate the removal of that item. Note that the item isn't physically removed. The representation is such that it is no longer logically part of the stack. The array element will be reused at a later `push`.

`empty` simply returns `true` when there are no items—`top` is less than or equal to `0`—and `false` otherwise. It does not throw an exception since this operation is always possible. You might also note that with the code as written, `top` could never be less than 0 nor, for that matter, greater than

`elts.length`. However these conditions are tested. This is defensive programming in case the code is modified and this assertion is no longer valid.

Since none of the methods involve loops, they are all `O(1)`.

```
1  package CharStacks;
2  import java.io.*;
3
4  /** This class represents an implementation of the CharStack interface using
5    * contiguous memory (i.e. an array).
6    * @see  CharStack
7    * @see  OverflowException
8    * @see  UnderflowException
9    * @author  D. Hughes
10   * @version  1.0 (Jan. 2014)                                                  */
11 public class ConCharStack implements CharStack, Serializable {
12
13     private int     top;   // next available element
14     private char[]  elts;  // elements of the stack
15
16     /** This constructor creates a new, empty stack capable of holding 100 items */
17     public ConCharStack ( ) {
18         this(100);
19     }; // constructor
20
21     /** This constructor creates a new, empty stack capable of holding a particular
22       * number of items.
23       * @param  size  the number of items for the stack.                        */
24     public ConCharStack ( int size ) {
25         elts = new char[size];
26         top = 0;
27     }; // constructor
28
29     /** This method adds an item to the stack. Stack overflow occurs if there is no
30       * room to add another item.
31       * @param  item  the item (char) to be added.
32       * @throws  OverflowException  no more room to add to stack.              */
33     public void push ( char item ) {
34         if ( top >= elts.length ) {
35             throw new OverflowException();
36         }
37         else {
38             elts[top] = item;
39             top = top + 1;
40         };
41     }; // push
42
```

```
43      /** This method removes an item to the stack. Stack underflow occurs if there
44        * are no more items left.
45        * @return  char  the item (char) removed.
46        * @throws  UnderflowException  no items available in stack.                */
47      public char pop ( ) {
48          if ( top <= 0 ) {
49              throw new UnderflowException();
50          }
51          else {
52              top = top - 1;
53              return elts[top];
54          }
55      }; // pop
56
57      /** This method returns the top item of the stack. Stack underflow occurs if
58        * there are no more items left.
59        * @return  char  the top item (char).
60        * @throws  UnderflowException  no items available in stack.                */
61      public char top ( ) {
62          if ( top <= 0 ) {
63              throw new UnderflowException();
64          }
65          else {
66              return elts[top-1];
67          }
68      }; // top
69
70      /** This method returns true if the stack contains no items.
71        * @return  boolean  whether the stack is empty.                            */
72      public boolean empty ( ) {
73          return top <= 0;
74      }; // empty
75
76  } // ConCharStack
```

Figure 5.5 Example—Contiguous Stack Implementation

5.4 LINKED IMPLEMENTATION OF `CharStack`

A stack can also be implemented using a sequentially-linked structure. The structure will contain the stack items in the order they have been added. Both insertion (`push`) and removal (`pop`) occur at the same end. Since in sequentially-linked structures the easiest end to do insertion and removal is the front, it makes sense to have the front of the list be the top of the stack. We have already noted that with insertion occurring at the front of the structure the items are in reverse order. Since we want LIFO ordering for a stack this is appropriate. The empty stack can be represented by the empty list. This arrangement is shown in Figure 5.6 and its implementation class in the in Figure 5.7.

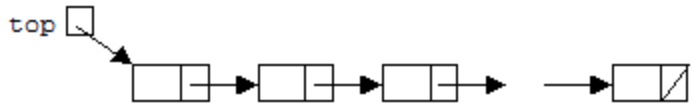


Figure 5.6 Linked Stack Representation

```
1  package CharStacks;
2  import java.io.*;
3
4  /** This class represents an implementation of the CharStack interface using a
5    * sequentially-linked structure. It makes use of the class Node to represent
6    * the list nodes.
7    * @see  CharStack
8    * @see  Node
9    * @see  UnderflowException
10   * @author  D. Hughes
11   * @version  1.0 (Jan. 2014)                                                  */
12 public class LnkCharStack implements CharStack, Serializable {
13
14     private Node  top;  // top element of the stack
15
16     /** This constructor creates a new, empty stack.                          */
17     public LnkCharStack ( ) {
18         top = null;
19     }; // constructor
20
21     /** This method adds an item to the stack. Stack overflow occurs if there is no
22       * room to add another item.
23       * @param  item  the item (char) to be added.
24       * @throws  OutOfMemoryError  no more room to add to stack.               */
25     public void push ( char item ) {
26         top = new Node(item,top);
27     }; // push
28
29     /** This method removes an item to the stack. Stack underflow occurs if there
30       * are no more items left.
31       * @return  char  the item (char) removed.
32       * @throws  UnderflowException  no items available in stack.              */
33     public char pop ( ) {
34         char  i;
35         if ( top == null ) {
36             throw new UnderflowException();
37         }
38         else {
39             i = top.item;
40             top = top.next;
41             return i;
42         }
43     }; // pop
44
```

```
45      /** This method returns the top item of the stack. Stack underflow occurs if
46        * there are no more items left.
47        * @return  char  the top item (char).
48        * @throws  UnderflowException  no items available in stack.                */
49      public char top ( ) {
50          if ( top == null ) {
51              throw new UnderflowException();
52          }
53          else {
54              return top.item;
55          }
56      }; // top
57
58      /** This method returns true if the stack contains no items.
59        * @return  boolean  whether the stack is empty.                            */
60      public boolean empty ( ) {
61          return top == null;
62      }; // empty
63
64  } // LnkCharStack
```

Figure 5.7 Example—Linked Stack Implementation

The stack itself is represented by the reference to the top stack node (`top`) and, indirectly by the nodes linked to it. The default constructor initializes the stack as empty by setting the `top` reference to `null`.

The `push` method creates a new node, inserting it at the front of the structure. Note that unlike the `ConCharStack` implementation it doesn't throw a `UnderflowException`. The only time it is impossible to add a new node to the stack is if there is not enough memory left to create a `Node`. In this case an `OutOfMemoryError` will be thrown by the `new` operator. The `push` code could try to catch this error to throw an `UnderflowException`. However it would be unable to create a new `UnderflowException` object since there is no more memory. Rather, it simply allows the `OutOfMemoryError` exception to propagate. `OutOfMemoryError` is an unrecoverable error that would normally not be caught but does not necessarily indicate a program error.

Again `pop` and `top` are similar. They check for stack underflow (`top==null`) and throw a `UnderflowException` in that case. Otherwiose they return the item contained in the top node (`top.item`). In addition, `pop` removes the front (`top`) node from the structure.

Finally, `empty` returns `true` when the stack contains no nodes (`top==null`) and `false` otherwise.

As described in Chapter 3, the class `Node` (Figure 5.8) defines the nodes of the list. In this case the *`item`* field is a character (`char`). This is the actual character itself not a reference since `char` is a primitive type. If a `LnkStack` is to be `Serializable`, the `Node` class must also implement `Serializable`, since all components of an object must be `Serializable` for the object to be serialized.

```
 1  package CharStacks;
 2  import java.io.*;
 3
 4  /** This class represents a node containing a character in a linked structure.
 5    * @author  D. Hughes
 6    * @version  1.0 (Jan. 2014)                                                  */
 7  class Node implements Serializable {
 8
 9      char  item;  // the item in the node
10      Node  next;  // the next node in the structure
11
12      /** This constructor creates a new node for the linked structure representing
13        * the stack.
14        * @param  i  the item in the node.
15        * @param  n  the next node in the structure.                             */
16      public Node ( char i, Node n ) {
17          item = i;
18          next = n;
19      }; // constructor
20
21  } // Node
```

Figure 5.8 Example—`Node` Class for `LnkCharStack` Implementation

CASE STUDY: POSTFIX NOTATION

When we write arithmetic expressions such as 2πr, we write them in what is called infix notation. In infix notation, the operators—here multiplication implicitly—are written in-between the operands—in this case 2, π and r. We are quite used to this notation and have no trouble realizing that 3x+2y-z means that 2 times y is to be added to 3 times x and then z is to be subtracted from that sum. However, this understanding is based on the notion that different operators have different priority for association—or binding—with operands. That is multiplication has higher priority for binding than addition or subtraction. If we wanted a different interpretation, we would use parentheses to group operands and operators. Thus 3x+2y-z means ((3*x)+(2*y))-z, not (3*(x+2))*(y-z).

The Polish mathematician Jan Lukasiewicz[6] devised a notation for expressions in 1924 which does not require operator priorities nor parentheses for association. In this notation—called **Polish** or **prefix**—operators are written immediately before their corresponding operands. Burks, Warren & Wright[7] proposed **Reverse Polish Notation** (**RPN** or **postfix**) as a notation for automated reduction of logical formulae. RPN can be the basis for efficient automated evaluation of expressions. Table 5.1 shows expressions in infix, prefix and postfix notations.

---

6 Łukasiewicz, J; *Aristotle's Syllogistic from the Standpoint of Modern Formal Logic*.; Oxford University Press. 2nd Edition, enlarged; 1957. Reprinted by Garland Publishing in 1987. ISBN 0-8240-6924-2

7 Burks, AW, Warren, DW and Wright, JB; "An Analysis of a Logical Machine Using Parenthesis-Free Notation," in Mathematical Tables and other Aids to Computation; American Mathematical Society; Apr 1954, pp 53-57

infix	prefix	postfix
x+y	+xy	xy+
x+y-z	-+xyz	xy+z-
x-y-z	--xyz	xy-z-
x*y+z	+*xyz	xy*z+
x+y*z	+x*yz	xyz*+
(x+y)*z	*+xyz	xy+z*
(w+x)*(y-z)	*+wx-yz	wx+yz-*

Table 5.1 Infix, Prefix and Postfix Notations

When an expression is written in postfix it is evaluated by scanning from left to right and, upon encountering an operator, performing the operation on the two immediately preceding operands. Figure 5.9 shows some such evaluations. The indicated operator is the next to be applied.

```
3 4 + 5 *       2 5 3 - *       6 4 - 2 -       6 4 2 - -
    ↑               ↑               ↑               ↑
 (3+4⇒7) 5 *    2 (5-3⇒2) *     (6-4⇒2) 2 -     6 (4-2⇒2) -
7 5 *           2 2 *           2 2 -           6 2 -
    ↑               ↑               ↑               ↑
 (7*5⇒35)       (2*2⇒4)         (2-2⇒0)         (6-2⇒4)
35              4               0               4
```

Figure 5.9 Postfix Evaluation

On traditional hardware the sequence of instructions for performing an operation in the ALU—for example adding two values together—is:

```
1  load the value of x into the ALU
2. load the value of y into the ALU
3. calculate the sum of x and y
```

Notice how this is exactly the order expressed in postfix notation—`xy+` or `load x`, `load y`, `add`. Postfix notation and computer computation are thus directly related. In fact, the Hewlet-Packard™ hand-held calculators of the '70s used reverse Polish notation (RPN) for entering calculations and evaluated them in exactly this way.

If you had been writing computer programs before the advent of high-level language compilers, you would have had to master the process of translating infix notation into postfix notation (either that or learn to use postfix exclusively). Luckily, we now can leave that exercise to a compiler. However, someone has to write compilers so we should at least understand the process for infix to postfix translation.

To do the process manually it is easiest to insert all of the parentheses implied by the operator binding rules—high priority to low priority, then left to right. Next we convert each parenthesized

sub-expression individually, from inside out and treating items within parentheses as operands. Finally remove the now redundant parentheses. Figure 5.10 shows such conversions.

```
x+3*y-(2+z/2)                    ((x+y)*3)/(z/(a-b))
((x+(3*y))-(2+(z/2)))            ((xy+)*3)/(z/(ab-))
((x+(3y*))-(2+(z2/)))            ((xy+)3*)/(z(ab-)/)
((x(3y*)+)-(2(z2/)+))            ((xy+)3*)(z(ab-)/)/)
((x(3y*)+)(2(z2/)+)-)            xy+3*zab-//
x3y*+2z2/+-
```

Figure 5.10 Infix to Postfix Translation

While this procedure works reasonably well manually it is not easy to automate. To come up with a reasonable algorithm for computer solution we need to look at the process more carefully. We want an algorithm that can produce the result in a single left-to-right pass over the expression yielding an `O(n)` solution.

Upon careful examination of Table 5.1 we see some interesting properties. The operands in both the infix and postfix expressions occur in precisely the same order—only the relative positions of the operators change. Secondly, if we are going to do the translation in a single left-to-right pass, the operators must be saved from the point at which they are encountered until they are later placed into the resultant expression. For example, in a simple case such as `x+y`, `x` is encountered and written in the output expression. Then `+` is encountered and saved. Next `y` is encountered and written in the output expression. Finally the `+` can be written in the output expression. When there are two operators in the infix expression, the leftmost is placed into the output expression before the rightmost only if it is of higher or equal priority since operations in postfix are performed left-to-right. If the leftmost operator is of lower priority, it must be saved until after we decide what to do with the rightmost operator. This implies that when operators occur in increasing operator priority, they must be dealt with in a LIFO manner. Here's where a stack comes in.

THE RAIL YARD ALGORITHM

A common mnemonic for the conversion algorithm is to consider the process as shunting rail cars—representing the operators and operands—around in a railway switching yard. Incoming operand cars are shunted straight through to the outgoing track. An operator car is shunted to the siding if the siding is empty or the priority of the operator car is greater than that of the one on the siding. If the priority of the incoming operator car is less than or equal to that of the one on the siding, the car on the siding is shunted to the outgoing track and the process continues with the incoming car and the next one on the siding. Finally, when there are no more incoming cars the cars on the siding are shunted to the outgoing track. Figure 5.11 shows such a process.

Notice that the siding behaves as a stack. This algorithm can be implemented using a stack for the operands on the siding. As simplifying assumptions, we will ignore parentheses, consider only single character operands and only the operators `+`, `-`, `*` and `/`.

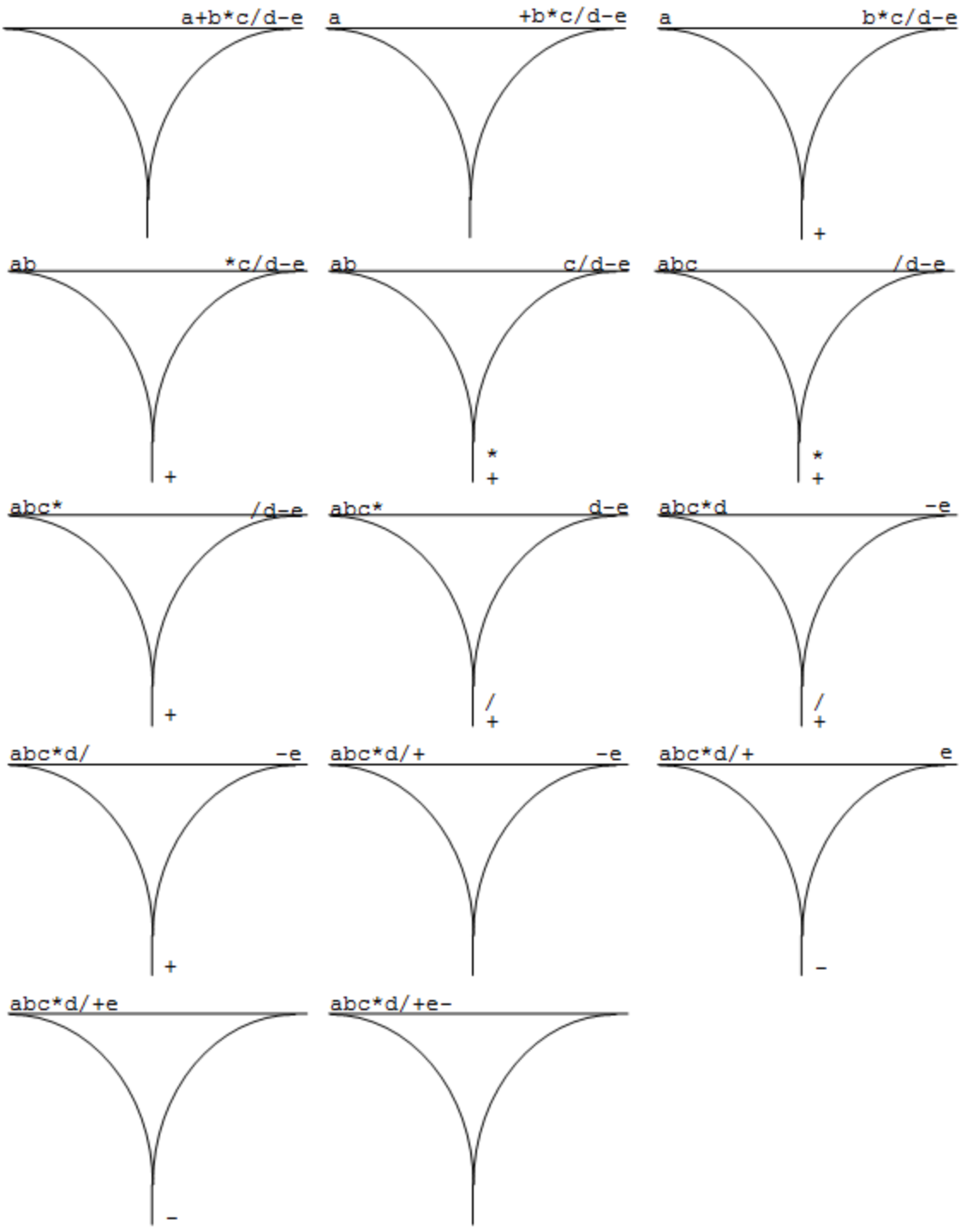


Figure 5.11 The Rail Yard Algorithm

The process involves two special cases: when there is nothing on the siding the operator is always stacked and when the input is exhausted the operators from the stack are popped. We can make the algorithm easier to express by removing these special cases. A special character ($) is be placed on the base of the stack. This is treated like an operator with lowest priority so that it forces all other

operators to be stacked. Similarly, another special character (#) is be added at the end of the infix string. It has a low priority, which forces all other operators to be popped. Both the `$` and the `#` serve as sentinels, marking the base of the stack and the end of the input string, respectively.

The program can be modified to, accommodate an additional operator for exponentiation—`^` with highest priority—and handle parentheses for grouping. The parentheses can be handled by the following rules:

1. an open parenthesis (`(`) in the input is always stacked
2. a close parenthesis (`)`) in the input always pops the operator from the stack
3. an open parenthesis on the stack always causes the operator on the input to be stacked
4. if there is a close parenthesis in the input and an open parenthesis on the stack, the open parenthesis is popped and both the open and close parentheses are discarded.

The exponentiation operator and rules 2 and 3 above can be handled by appropriate setting of priorities. Consider that an open parenthesis is the beginning of a sub-expression and close parenthesis is the end of a sub-expression. Rules 1 and 4 should probably be handled by special cases.

SUMMARY

A stack is a container ADT that exhibits the last-in-first-out (LIFO) property. That is, the item that was inserted into the stack most recently is the one that will be removed next. Operations on a stack are adding and removing items (push and pop), examining the top item (top) and determining whether or not the stack is empty (empty). Exceptional conditions that can arise in processing stacks are stack underflow that occurs on a pop or top operation on an empty stack and stack overflow that might occur on a push operation if there is no more space available for the stack.

The contiguous representation of a stack uses an array, maintaining the base (bottom) of the stack at the front of the array and allowing the top to float up and down as items are pushed and popped. An auxiliary variable is maintained to mark the top of the stack within the array. The linked representation uses a sequentially-linked structure with the front of the structure representing the top of the stack and push and pop being insertion and deletion at the front of the structure.

REVIEW QUESTIONS

1. T F In a list-oriented collection, an item is located based on its key.

2. T F A stack underflow occurs when there aren't any items in the stack.

3. T F In the contiguous representation of a stack, an array of characters is used.

4. T F All operations in the contiguous implementation of a stack are `O(n)`.

5. T F In the linked representation of a stack, an empty stack is represented by a `null` pointer.

6. T F A special character added to the end of a string to mark the end, is called a sentinel.

7. T F In postfix, the priority of `*` is higher than the priority of `+`.

8. LIFO stands for

   a) Last-Index-For-Output
   b) Linked-Item-Final-Object
   c) Last-In-First-Out
   d) none of the above

9. The following set of operations on `s`, a `CharStack` would output?

```
s.push('a'); s.push('b'); s.push('c');
c = s.pop(); c = s.pop(); s.push('d');
System.out.println(s.pop()+" "+s.pop());
```

   a) `a b`
   b) `a d`
   c) `c d`
   d) `d a`

10. The infix expression:

```
a*b+c
```

   is equivalent to which postfix expression?

   e) `ab*c+`
   a) `*ab+c`
   b) `abc*+`
   c) `c+ab*`

11. The infix expression:

```
a*(b+c)*d
```

   is equivalent to which postfix expression?

   d) `bc+a*d*`
   a) `abc+*d*`
   b) `a*(bc+)d*`
   c) `ab*c+d*`

12. The postfix expression:

```
54+3*3-
```

   evaluates to:

   a) `32`
   b) `24`
   c) -8
   d) none of the above

13. The postfix expression

```
2345+-*
```

evaluates to:

a) -15
b) 5
c) -12
d) none of the above

EXERCISES

1. Write a Java program that uses a stack to determine if a string represents a palindrome. A palindrome is a phrase that reads the same forward and backward, e.g. the word "ewe" is a palindrome as is the phrase (attributed to Napoleon) "Able was I ere I saw Elba".

   If a phrase is a palindrome then it is reflected about its middle character—the characters following the middle character are the same as the characters before the middle character, except in reverse order. This hints at the possibility of using a stack.

   Your program should read a string and display it to an `ASCIIDisplayer`. It should then scan the first half of the string placing each character into a stack. When it reaches the middle character, it should compare the remaining characters to the characters in the stack. Once the stack is empty—that is the end of the string reached—if each comparison was equal the string is a palindrome, else it is not. A message indicating whether or not the string is a palindrome should be displayed. You may assume that the entire string is in the same case. Don't forget to correctly handle strings of both even and odd length.

2. In mathematical expressions and computer programs, it is important to be able to match parentheses (`(,)`), brackets (`[,]`) and braces (`{,}`) to determine what the expressions or programs mean. Write a program that will read a text file (`ASCIIDataFile`) line-by-line and display (to an `ASCIIDisplayer`) each line followed by a second line that shows the matching of symbols. Of course, an open parenthesis must match a corresponding close parenthesis, an open bracket matches a close bracket, etc. The symbols may be nested, that is multiple open symbols may occur before a closing symbol, and the matching must take this into consideration. To show the nesting, use a digit under each open and close symbol pair that indicates the nesting level. That is, the first open brace would be level 1 and, if it is followed by an open parenthesis that symbol would be level 2. Consider, for example, the following piece of Java code:

```
public void m ( ) {
              1 1 1
    int  a[];
          22
    return (((a[3]+a[4])*a[2])/(a[1]-a[3]));
           234 5 5  5 54  4 43 3 4 4  4 432
};  // m
1
```

Each open symbol begins a new nesting level and must be matched by the correct closing symbol. If a second open symbol is encountered before the next close symbol, its closing symbol must occur before the closing symbol of the previous open symbol. This indicates a LIFO matching of close symbols to open symbols, and suggests the use of a stack. Whenever an open symbol is encountered, it is pushed on the stack, the nesting level is increased, and the nesting level displayed. Whenever a closing symbol is encountered, it is compared to the top symbol of the stack. If it is the corresponding symbol, the top stack symbol is removed, the nesting level is displayed and then the nesting level is decreased. If the closing symbol is not the corresponding symbol to the top, or there is no top stack symbol, there is an error and a * should be displayed under the offending closing symbol.

6 GENERICS

CHAPTER OBJECTIVES

- Explain the need for generalization of interfaces and classes (generics)
- Describe how generic types are type-safe
- Apply type parameters in generalizing a class
- Apply type arguments in writing a client of a generic class
- Explain autoboxing and autounboxing

While the `CharStack` interface and the `ConCharStack` and `LnkCharStack` implementation classes defined in Chapter 5 are working definitions, they are severely limited. They can only represent stacks of characters! While we could also define an interface and class implementations for an `IntStack` and a `StudentStack` and so on, this would be extremely tedious. The only differences between these interfaces and their implementation classes would be the types of the parameters (`char` *vs* `int` *vs* `Student`), the return values of the methods (`char` *vs* `int` *vs* `Student`) and the representations in the implementation classes (arrays of `char` *vs* `int` *vs* `Student` or nodes containing `char` *vs* `int` *vs* `Student`). The remaining code would be identical. In fact, it should be possible to declare a stack of any type just as we can declare an array of any type. What we want is a mechanism to declare a stack once and be able to use that declaration for any type of element.

The generics facility in Java addresses this requirement. A class can be generalized by abstracting out the type of the components defining what is called a parametric type. Once a parametric type has been defined, it is instantiated by supplying an actual type for the type parameter. For a stack we can abstract out the type of the item in the stack (`char` in Chapter 5) defining a parametric type `Stack`. When we write an application that uses a stack, we instantiate the parametric type by supplying the actual item type (character for the infix to postfix conversion case study in Chapter 5).

6.1 GENERIC INTERFACES

A type can be generalized (made **generic**) by abstracting out the component type resulting in a **parametric type**. The parametric type is then instantiated by supplying an actual type parameter in an application. Figure 6.1 shows the `Stack` interface abstracted over the item type defining the parametric type `Stack`.

If you compare Figure 6.1 with Figure 5.4 you will see only minor changes. Ignoring for the moment the package name, interface name and the names of the exceptions, the only changes are the inclusion of the notation `<E>` after the interface name (line 9) and the substitution of `E` for `char` throughout the code (lines 15, 21 & 27). The `CharStack` interface has been made generic as the interface `Stack` by abstracting out component type (`char`) as the type parameter `E`. A **type parameter** is a variable—an identifier—that stands for a type name just as an `int` parameter is a variable that stands for an `int` value. The `Stack` interface defines a parametric type that is instantiated by supplying a type argument for the type parameter `E`.

```
1  package Collections;
2
3  /** This interface defines the generic data type stack representing a LIFO
4    * collection of items of the same type (E).
5    * @see  NoSpaceException
6    * @see  NoItemException
7    * @author  D. Hughes
8    * @version  1.0 (Jan. 2014)                                          */
9  public interface Stack <E> {
10
11     /** This method adds an item to the stack. Stack overflow occurs if there is no
12       * room to add another item.
13       * @param  item  the item to be added.
14       * @exception  NoSpaceException  no more room to add to stack.     */
15     public void push ( E item );
16
17     /** This method removes an item to the stack. Stack underflow occurs if there
18       * are no more items left.
19       * @return  E  the item removed.
20       * @exception  NoItemException  no items available in stack.       */
21     public E pop ( );
22
23     /** This method returns the top item of the stack. Stack underflow occurs if
24       * there are no more items left.
25       * @return  E  the top item.
26       * @exception  NoItemException  no items available in stack.       */
27     public E top ( );
28
29     /** This method returns true if the stack contains no items.
30       * @return  boolean  whether the stack is empty.                   */
31     public boolean empty ( );
32
33 } // Stack
```

Figure 6.1 Example—Generic Stack Interface

STYLE TIP

By convention type variables are written as single letters in uppercase in Java. However syntactically they are just identifiers. In this case, since `E` is standing in for the type of the elements (items) in the stack we have called it `E` for element.

PARAMETRIC TYPES

The notation `<E>` after the interface name declares `Stack` to be a parametric or generic type. Just as a parametric method is a method that varies depending on the values of arguments that are supplied in the method call, a parametric type is a type that varies depending on the type name provided as type arguments when the parametric type is **instantiated**. To instantiate the parametric type `Stack` to declare a variable `opStack` as a stack of characters we use the following notation:

```
Stack<Character>  opStack;
```

The effect is essentially the same as if we had defined an interface like `Stack` in Figure 6.1 in which every occurrence of `E` was replaced by `Character`. Thus `opStack` has a method called `push` which takes a parameter of type `Character`, a method `pop` that returns a `Character`, and so on. Therefore `opStack` is a `Stack` whose elements are restricted to being `Character`s.

Note that within the same piece of code we can declare:

```
Stack<Student>  stdStack;
```

which declares `stdStack` as a `Stack` whose elements are restricted to `Student`. Like `opStack`, it has methods `push`, `pop`, `top` and empty—only the parameter and return value types would be `Student` instead of `Character`. This demonstrates the expressive power of parametric types. We only need one interface declaration to define all the possible kinds of stacks that might exist.

Figure 6.2 shows the extended form of an interface declaration including type parameters.

```
interfaceDeclaration:
    modifier… interface name typeParameters_opt {
        fieldDeclaration ; …
        methodHeader ; …
    }

typeParameters:
    < paramName, … >
```

Figure 6.2 Interface Declaration

An interface may optionally have *typeParameters*. If not the interface type is not parametric. This is what we have done up to now. When the type is parametric, the *typeParameters* serve to declare each parameter as a *typeVariable*. This is similar to a method declaration that optionally has parameters. The method parameter declarations include a type and a name and declare each parameter as a variable. The scope of a *typeVariable* is the entire interface declaration.

Figure 6.3 shows the extended form of a *type*. Remember a *type* is used to specify the type in three contexts: the declaration of an instance or local variable, the declaration of a parameter of a method and the declaration of the result type for a function method. This is what allows us to declare `opStack` and `stdStack` as above. Note also that within a class or interface body, a *typeVariable* may be used wherever a *referenceType* can be used. An example of this is when we use `E` to define the parameter type for `push` and the return types for `pop` and `top` in the `Stack` interface.

When a parametric type is used in a declaration, *typeArguments* must be supplied for each *typeParameter* of the parametric type. We see this in the declaration of `opStack`, above. A *typeParameter* must be a reference type—it cannot be a primitive type. This is why in our example above we declared `opStack` as a `Stack` of `Character` rather than `char`. Remember that `Character` is the standard wrapper class that wraps a `char` as an object.

```
type:
    primitiveType
    referenceType

referenceType:
    classOrInterfaceType
    arrayType
    typeVariable

classOrInterfaceType:
    classOrInterfaceName typeArguments_opt

typeArguments:
    < referenceType, … >
```

Figure 6.3 Type syntax

THE COLLECTIONS PACKAGE

If you look at the package declaration for the `Stack` class, you will note that `Stack` is part of the `Collections` package. Remember that a package is a collection of class and interface declarations and can be used to provide a library of useful resources, such as `BasicIO`. Since there are a number of collection ADTs (stack, queue, list, etc.) it is useful to include them as a library within one package called `Collections`. This is what we will do in the remaining chapters of the book. We will add additional ADTs to the package in Chapters 8 and 9.

The exceptions in the `Stack` interface are named `NoItemException` and `NoSpaceException` instead of `UnderFlowException` and `OverflowException` as in `CharStack`. Since the exceptional conditions of trying to access a non-existent item from a collection and exceeding the available space in a collection occur in all collection ADTs, we have chosen to use two common exceptions rather than defining six exceptions that are essentially the same.

6.2 GENERIC IMPLEMENTATION CLASSES

The code for `ConStack`, a contiguous implementation of `Stack`, is shown in Figure 6.4. There are minimal changes from the code for `ConCharStack` in Figure 5.5, primarily the substitution of `E` for `char` throughout (lines 15, 34, 49 & 65).

Like the `Stack` interface the class `ConStack` is a parametric type with the type parameter `E` (line 11). As in the interface declaration (Figure 6.2) *`typeParameters`* may optionally follow the class name in the class header. The implements clause includes a list of interface types. Since `Stack` is a parametric type, it must be instantiated by specifying the component type. In this case, the component type is the same as the component type for `ConStack`—`E`. Note that `E` is used in two contexts on line 11. Following the class name it is being declared as a *`typeParameter`* (as in Figure 6.2). In the implements clause it is being used as a *`typeArgument`* (Figure 6.3) to instantiate `Stack`. The meaning is that a `ConStack` of some type implements a `Stack` of that same type, that is a `ConStack<Student>` implements a `Stack<Student>` but not `Stack<Character>`.

On line 13, the type parameter `E` is also used to declare the element type of the array `elts` as part of the representation of `ConStack`. That means that the representation of `ConStack` of some type includes an array of elements of that type.

A type parameter `E` stands in for the type argument when an instance of `ConStack` is created, The effect of the statement:

```
opStack = new ConStack<Character>();
```

is the same as if an instance of the `ConStack` class with `Character` substituted for `E` throughout is created and the representation for `opStack` is an array of `Character`.

```
1  package Collections;
2  import java.io.*;
3
4  /** This class represents an implementation of the Stack interface using contiguous
5    * memory (i.e. an array).
6    * @see  Stack
7    * @see  NoSpaceException
8    * @see  NoItemException
9    * @author  D. Hughes
10   * @version  1.0 (Jan. 2014)                                                    */
11 public class ConStack <E> implements Stack<E>, Serializable {
12
13     private int  top;   // next available element
14     private E[]  elts;  // elements of the stack
15
16     /** This constructor creates a new, empty stack capable of holding 100 items.*/
17     public ConStack ( ) {
18         this(100);
19     }; // constructor
20
21     /** This constructor creates a new, empty stack capable of holding a particular
22       * number of items.
23       * @param  size  the number of items for the stack.                          */
24     public ConStack ( int size ) {
25         elts = (E[]) new Object[size];
26         top = 0;
27     }; // constructor
28
29     /** This method adds an item to the stack. Stack overflow occurs if there is no
30       * room to add another item.
31       * @param  item  the item to be added.
32       * @exception  NoSpaceException  no more room to add to stack.              */
33     public void push ( E item ) {
34         if ( top >= elts.length ) {
35             throw new NoSpaceException();
36         }
37         else {
38             elts[top] = item;
39             top = top + 1;
40         };
41     }; // push
42
```

```
43      /** This method removes an item to the stack. Stack underflow occurs if there
44        * are no more items left.
45        * @return  E  the item removed.
46        * @exception  NoItemException  no items available in stack.                */
47      public E pop ( ) {
48          E  i;
49          if ( top <= 0 ) {
50              throw new NoItemException();
51          }
52          else {
53              top = top - 1;
54              i = elts[top];
55              elts[top] = null;
56              return i;
57          }
58      }; // pop
59
60      /** This method returns the top item of the stack. Stack underflow occurs if
61        * there are no more items left.
62        * @return  E  the top item.
63        * @exception  NoItemException  no items available in stack.                */
64      public E top ( ) {
65          if ( top <= 0 ) {
66              throw new NoItemException();
67          }
68          else {
69              return elts[top-1];
70          }
71      }; // top
72
73      /** This method returns true if the stack contains no items.
74        * @return  boolean  whether the stack is empty.                            */
75      public boolean empty ( ) {
76          return top <= 0;
77      }; // empty
78
79  } // ConStack
```

Example 6.4 Example—Generic contiguous stack implementation

On the other hand, the statement:

```
stdStack = new ConStack<Student>();
```

effectively substitutes `Student` for `E` and the representation for `stdStack` would be an array of `Student`.

In the object creation expression (the `new` operator) the parametric type is instantiated and then an object of that type is created. Thus we have the *`typeArguments`*—`<Character>` or `<Student>`—followed by the arguments for the constructor—in this case `()`.

TYPE COMPATIBILITY OF PARAMETRIC TYPES

The class header specifies that `ConStack<E> implements Stack<E>`. In other words `ConStack<Character>` implements `Stack<Character>` and `ConStack<Student>` implements `Stack<Student>`. This is what ensures type safety using generics. Thus the code:

```
Stack<Character>  opStack;
       :
opStack = new ConStack<Character>();
```

is valid since `ConStack<Character>` implements `Stack<Character>` and is thus a subtype of `Stack<Character>`. On the other hand the code:

```
Stack<Character>  opStack;
       :
opStack = new ConStack<Student>();
```

Is invalid since `ConStack<Student>` does not implement `Stack<Character>` and thus is not a subtype of `Stack<Character>`. This is consistent with the way arrays work. An array of `int` is compatible with an array of `int` variable but is not compatible with an array of `Student` variable.

TYPE CHECKING OF THE TYPE PARAMETER

Within the code for `ConStack` we have statements such as:

```
elts[top] = item;
```

(line 38 in `push`). How is this type checked? If `ConStack` has been instantiated with `Character` then the parameter `item` for `push` is of type `Character`. `elts` is an array of `Character` (line 14) and this statement is type correct since we are assigning a `Character` value to an element of a `Character` array. However, how does the compiler know this? When it compiles `ConStack`, it has no idea what the actual type parameter (`E`) will be!

The rule is simple. Since any reference type may be used as the *`typeArgument`* (Figure 6.3), the only thing the compiler knows about the type parameter is that it represents some object type. Recall that there is a standard type in Java called `Object`. This type represents the common properties of objects of all classes. For example all objects have the method `toString`. All reference types are subtypes of `Object` and thus any reference type can be used wherever an `Object` is required. Thus, while compiling a parametric type, the compiler uses the assumption that the type parameter is `Object`.

The side effect of this assumption is that any variable declared using the type parameter—such as the array `elts`—is considered to be of type `Object`. Within the code for the parametric type, only operations available for `Object` can be used. This includes, of course, assignment—as needed in `ConStack`—and also the methods defined by `Object` such as `toString`, but nothing else.

It is instructive to note that we can declare the `elts` array even though the compiler doesn't know what actual kinds of objects are going to be stacked. Since the element type is assumed to be

`Object`, the array elements are object references. This means that regardless of the actual kind of object being stacked, the amount of storage for each element in the array is just 4 bytes—enough for a reference. Thus the compiler can allocate the storage for the array as just the number of elements times 4 bytes.

Consider the constructor with one parameter where the array `elts` is created (line 25). Note that the `elts` is created as an array of `Object` rather than and array of `E` as we might expect. Since the type of `elts` is `E[]` (line 14), creating a new array of `E` and assigning it to `elts` would seem to be valid. However for technical reasons (see Bracha et al[8]), the Java language specification does not allow a type parameter to be used in an array creation expression. To overcome this restriction, we create an array of `Object` instead and downcast the array to `E[]`. The Java compiler will flag this with an "unchecked cast" warning since the compiler cannot know what types of objects will be assigned to elements of `elts`. However we know that only objects of type `E` are assigned to `elts` so this is valid.

ENABLING GARBAGE COLLECTION

One other change has been made in the `pop` method. After the item is copied from the `elts` array (line 54), the element of the array is set to `null` (line 55). Since `E` is a reference type, the array has object references as its elements. If the element is left unchanged as it is in `ConCharStack` (Figure 5.5), it would still reference the popped item until a new value is pushed, if ever. Since objects can only be garbage collected when there are no references to them, this could defeat the garbage collection process. Setting the element to `null` removes this reference to the object. If the result of the `pop` isn't stored anywhere, the object may be garbage collected. Note that this wasn't necessary in `ConCharStack` since the element type was a primitive type (`char`) and thus garbage collection was not an issue.

GENERIC `NODE` CLASS

We will not show the generic version of the linked representation of `Stack` (`LnkStack`) since most of the differences from `LnkCharStack` are similar to the differences between `ConCharStack` and `ConStack`. However there is a need to talk about the `Node` class.

In the linked representation the items are stored in nodes. Since the type of the item is unknown when the code is written—and it needs to be the same as the actual type parameter of the `LnkStack`—the `Node` class will also have to be parametric. The code for the parametric `Node` class is given in Figure 6.5.

---

[8] Bracha, G., Odersky, M., Stoutamire, D., & Wadler, P.; *Making the future safe for the past: Adding Genericity to the Java™ Programming Language*; Proceedings of the SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98), Vancouver, BC, Canada, Oct. 1998

```
1  package Collections;
2  import java.io.*;
3
4  /** This class represents a node in a sequentially-linked structure representing a
5    * collection of objects of type E.
6    * @author  D. Hughes
7    * @version  1.0 (Jan. 2014)                                                    */
8  class Node <E> implements Serializable {
9
10     E        item;  // the item in the stack
11     Node<E>  next;  // the next node in the list
12
13     /** This constructor creates a new node for the sequentially-linked structure
14       * representing a collection.
15       * @param  i  the item in the node.
16       * @param  n  the next node in the list.                                     */
17     public Node ( E i, Node<E> n ) {
18         item = i;
19         next = n;
20     }; // constructor
21
22 } // Node
```

Example 6.5 Example—Generic Node class

The `Node` class is parametric in some type `E` (line 8). Note that the name of the type parameter does not have to be `E`, it could be anything. Each node contains two fields: `item` (line 10)—a reference to an item in the stack of type `E`—and `next` (line 11)—a reference to another `Node` with the same type parameter (`E`). The constructor takes an item of type `E` and a reference to another `Node` with the same type parameter as parameters.

In the `LnkStack` class, assuming the type parameter is called `E`, `top` would be declared as:

```
private Node<E>  top;
```

When a new `Node` is created to hold the item being `push`ed, the code would be

```
top = new Node<E>(item,top);
```

This creates a new node with the same type parameter as `LnkStack`, filling in the item and link to the current top of the stack. `item` is the parameter to `push` of type `E` and `top` is a reference to a `Node` with type argument `E`, so this is valid.

6.3 THE CLIENT CLASS

Within a client class, the use of a generic type is reasonably straightforward. Whenever a variable referencing an object of the generic type is declared the generic type must be instantiated by providing a type argument. For example, to declare a stack of characters the generic type `Stack` would be instantiated with the type argument `Character`:

```
Stack<Character>  opStack;
```

Note that, since type arguments must be reference types, the wrapper class `Character` must be used instead of the primitive type `char`. Such an instantiation can be used wherever a type is expected: as an instance variable, as a local variable or as a method parameter.

Creation of an instance of the generic type includes both instantiation of the generic type and supplying parameters to the constructor of the generic type. For example, to create a stack of up to 10 characters, the following is used:

```
opStack = new ConStack<Character>(10);
```

Here `Character` is the type argument for the generic instantiation of `ConStack` and `10` is the argument for the `ConStack` constructor. The result is of type `ConStack<Character>` which is assignment compatible with `Stack<Character>` since `ConStack<Character>` implements `Stack<Character>` and is thus a subtype.

Other than this, objects of the generic type behave in the same way as other objects and are subject to usual type checking. Two objects of a generic type are of the same type if they have both been declared with an instantiation of the generic type with the same type argument. For example in:

```
Stack<Character>  s1;
Stack<Character>  s2;
Stack<Student>    s3;
```

`s1` and `s2` are of the same type while `s1` and `s3` are not of the same type.

AUTOBOXING AND AUTOUNBOXING

Java defines wrapper classes for the primitive types. These classes simply wrap a value of the primitive type in an object and provide a constructor taking a primitive type value as a parameter creating a wrapped version of the value and a method to obtain the wrapped primitive value from the object.

The wrapped objects allow a primitive type value to be used where a reference type (i.e. object) is required, such as in generic types where the type argument must be a reference type. The generic type is instantiated with the wrapper type as the type parameter. When the primitive type value is to be passed as a parameter where the wrapper type is expected, the primitive type value must be wrapped. When it is necessary to access the primitive type value in the wrapper, it must be unwrapped.

For convenience Java provides two automatic conversions for wrapped primitive types: autoboxing and autounboxing. **Autoboxing** wraps a primitive type within a wrapped object whenever a primitive value is used where a wrapped object is required. Similarly **autounboxing** unwraps the primitive value from a wrapper object whenever the wrapper object is used where the primitive type is required.

Consider the following code:

```
Stack<Character>  opStack;
char              c;
  :
opStack.push(c);
  :
c = opStack.pop();
```

`push` takes a parameter of type `Character` but the method call supplies a value of type `char`. Since `Character` wraps a value of type `char`, autoboxing is applied. The compiler generates code as if the statement was written as:

```
opStack.push(new Character(c));
```

Similarly `pop` returns a value of type `Character` but it is assigned to a variable of type `char`. This time autounboxing occurs and the compiler generates code as is the statement had been written:

```
c = opStack.pop().charValue();
```

SUMMARY

A generic container ADT is one which can be used for any (or many) kind of items. Java provides a second kind of class or interface called a parametric class or interface. This allows the type of a component in a type to be generalized as a type parameter on the interface and class declaration and specified as a type argument when a class or interface is instantiated. This allows, for example, stacks of any reference type to be created while providing type safety such that only objects of the declared type may be pushed onto the stack. To make it easier to use generic classes with the primitive types, Java provides the new automatic conversions autoboxing which automatically wraps a primitive type within its wrapper object type and autounboxing which automatically unwraps the primitive type value from within the wrapper object.

REVIEW QUESTIONS

1. T F A parametric type allows the use of any type as al type argument.

2. In Java, generics can be used:

   a) on an interface
   b) on a class
   c) to create type-safe collections
   d) all of the above

EXERCISES

1. As part of the `Collections` package, implement a parametric version of the linked representation of a stack as the class `LnkStack`.

2. Rewrite the code for Exercise 1 in Chapter 5 using the generic `Stack` interface and `LnkStack` class.

3. Rewrite the code for Exercise 2 in Chaper 5 using the generic `Stack` interface and the `ConStack` class.

4. Rewrite the solution to the Case Study in Chapter 5 using the generic `Stack` interface and the `ConStack` class.

5. Write a program to simulate an RPN calculator. The calculator has 26 registers (designated `a` through `z`) into which intermediate results may be placed. The calculator can perform addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) on integer values. In addition, the result of a computation can be displayed (`D`) or stored (`S`) in a register. The computation itself is specified by an expression written in reverse polish notation (RPN) including the operators (`+`, `-`, `*`, `/`), constants (single digits `0-9`) and registers (`a-z`).

   Initially the value in each register is `0`. The calculator repeatedly reads an operation and executes the operation until end of file. An operation starts with the command to perform (i.e. `S` for store, `D` for display). For store, the command is followed by a register name and then an expression each separated by tabs. For display, the operation is followed only by an expression separated by a tab. Each operation is on a separate line and the expression contains no spaces. The calculator evaluates the expression—using the current values of the registers—and either stores the result in the register specified (store) or writes out the result (display). For example, the following operation line

```
D	234*+
```

   would display the value `14`, that is, `2+3*4`. You may assume that the input is valid. An `ASCIIDataFile` should be used for input and an `ASCIIDisplayer` for output.

   Hint

   RPN expressions are evaluated left to right with each operator binding to the two operands immediately preceding it. This means that they may be evaluated with the use of a stack to hold the operands. In a left to right pass through the expression, when an operand—constant or variable—is encountered its value is stacked. Note this is the value currently stored in the register or the numeric value represented by the literal. When an operator is encountered, it is applied to the top two operands on the stack. That is, the two operands are popped, the operation is performed, and the result is pushed. In a well-formed expression, the stack will have at least two operands on it whenever an operator is encountered and, at the end of the expression, there will be exactly one value left on the stack, which is the result of the computation.

6. In this exercise we will use stacks to accomplish a technique called backtracking. Consider the problem of walking through a maze. We think of the maze as blocked out in squares each of which can be a space, somewhere you can walk, or a wall, where you cannot walk. At any time while we are walking through the maze, we will be in some square. From this square there are four possible directions we can move: west, north, east or south. We choose one direction, say west, and try to find a path from that direction. If we cannot, we try the next direction, e.g. north,

etc. If we exhaust all directions, we give up on this square and backtrack (backup) to wherever we came to this square from.

Stacks can be used to handle backtracking. Basically, we maintain a stack of the choices we have made in the past. Whenever we make a new choice, we push it on the stack. Whenever we eliminate all possibilities for a choice, we pop it from the stack, exposing a prior choice we made. For the maze walk, the stack would be the positions within the maze from which we can choose to walk to another position. We make a choice, for example to move into the position to the west, and push that position onto the stack. When we backtrack to the current position after discovering there is no path going west, we try the next choice, for example north, pushing it onto the stack. Again, when we backtrack, we try east and then south. If we backtrack again to this position we have run out of possibilities, so we pop this possibility off the stack and continue with the exposed position underneath.

This leads to an algorithm like the following:

```
push initial position on the stack
while the stack is not empty and the top position isn't the exit
                                                    (goal) square
      considering the top stack position
      if there is another choice left
            make another choice
            push it onto the stack
      otherwise
            pop current choice from the stack (i.e. backtrack)
```

Although as humans we can look ahead to see if a square is a wall or a space, this is a bit harder in a computer program. It is easier to allow the algorithm to enter a wall square and immediately backtrack, as if it immediately runs out of choices in the wall square. Thus the wall square choice would be pushed onto the stack and, the next time through the loop would immediately be popped off. Positions outside of the maze could be handled in the same manner as walls.

This algorithm will work as long as we don't wind up going back and forth between squares, or go in circles. The easiest way of handling this is to mark each square when we first get to it and then treat it like a wall any other time we get to it.

When the algorithm completes, there are two possibilities. If the stack is empty the algorithm didn't find a path through the maze. If the stack isn't empty it contains, in reverse order, the positions on the path. To put them into correct order, we could pop them from one stack pushing them onto another and then pop the resulting stack. Remember pushing items onto a stack and then popping them gives us the items in reverse order.

Write a program using this backtracking method to find a path through a maze. The maze can be stored as an array of characters. The maze is provided in a text data file with the character `'X'` representing a wall space and a space character (`' '`) representing an open space. The file has one line per row of the maze containing the characters representing the spaces and walls in that row. Preceding the maze in the file are two integers representing the number of rows and the number of columns in the maze. Following the maze are two pairs of integers (row, column) representing the starting square and the finishing square for the path. The rows of the maze are

numbered, starting at 0 from top to bottom and the columns from left to right. Use `readC` to read the maze.

Once the maze has been read, perform the path search algorithm. If a path was found reverse the path in the stack and display the positions as pairs (row, column). Otherwise print a message that no path exists.

Use generic Stacks. The items on the stack will be a class representing the choices/positions. The item will have to at least keep track of its position—its (row,column) —nd an indication of the choice made—west, north, east, south. Basically, there are four choices for each position, so an integer 1–4 can represent the choice. When the position is considered, the choice number can be used to determine the next square to go to and the choice number incremented. When the choice number is >4, there are no more choices and backtracking occurs.

7 RECURSION

CHAPTER OBJECTIVES

- Explain the effectiveness of a recursive definition.
- Recognize a recursive method.
- Explain the mechanism for storage allocation for method invocation.
- Demonstrate that a recursive algorithm terminates.
- Describe the conversion of a recursive algorithm into an iterative one.
- Apply recursion in the solution of a programming problem.

A recursive definition is one in which the entity being defined is defined in terms of itself. In natural language we do not accept recursive definitions. Defining a word using the same word in the definition is not useful. However in many areas, especially Mathematics, recursive definitions can be very useful. A recursive definition can formally define an infinite concept with a finite definition.

We will see in this chapter that recursion can be a powerful tool in Computer Science.

7.1 RECURSION IN MATHEMATICS AND COMPUTING

Consider the definition of n factorial (written `n!`). When we first encountered the concept, it was probably defined in the following way:

```
n!  =  1 × 2 × … × (n-1) × n
```

Although this sufficed at the time, it is an informal definition as it relies on the reader being able to determine the pattern implied by the ellipsis (…). In this case since the pattern is very simple this is not an unreasonable assumption, but it might not always be so. The formal definition is:

```
n! =  1            if n = 0
      n × (n-1)!   if n > 0
```

This definition is clearly recursive. In the second line `!` is used to define `!`. There is however, no ambiguity in the definition. `n!` can be determined for any value of `n` (≥0) from the definition. In fact, there is no precise definition for `n!` that doesn't involve recursion. Such functions are called recursive functions in Mathematics.

As another example of a recursively defined entity, consider the Fibonacci numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, …
```

Here the pattern is not as obvious. Each number in the sequence is the sum of the two numbers preceding it. The formal definition is:

$$f_n = \begin{cases} n & \text{if } n = 0,1 \\ f_{n-1} + f_{n-2} & \text{if } n > 1 \end{cases}$$

clearly capturing the definition previously expressed in words. Again this is a recursive definition.

In both of these cases, the recursive definition has been able to define an infinite set of possibilities, for example `0!`, `1!`, `2!`, ..., using a finite definition.

We have already seen the use of recursion in our study of Computer Science! If you consider the definition of a *`statement`* in Java, you will see the following:

```
statement:
    :
    whileStatement
    :

whileStatement:
    while ( expression ) statement
```

The rules have been abridged with ellipses indicating parts omitted for clarity. A *`statement`* is defined in terms of a *`whileStatement`*, which is defined in terms of a *`statement`*. Although neither definition is directly recursive—where the right hand side includes a use of the left hand side—the pair is indirectly recursive. Ultimately, a *`statement`* is defined in terms of a *`statement`* via the definition of a *`whileStatement`*. There is no lack of clarity here. It clearly indicates that a *`whileStatement`* can contain any *`statement`*—including another nested *`whileStatement`*. Although the definition is recursive, in any practical application of the definition—in a particular program—the number of levels of nesting would be finite. It is through the use of recursion that the finite set of syntax rules can describe an infinite number of possible programs.

7.2 RECURSIVE METHODS

In Computer Science, a **recursive method** (procedure, algorithm) is one that directly or indirectly calls itself. This can be a very effective way to write a method. Figure 7.1 shows a recursive function method to compute `n!` based on the mathematical definition above.

```
29      /** This method computes n! using recursion.
30        * @param  n  value to compute n!.                    */
31      private long factorial ( int n ) {
32          long  result;
33          if ( n == 0 ) {
34              result = 1;
35          }
36          else {
37              result = n * factorial(n-1);
38          };
39          return result;
40      }; // factorial
```

Figure 7.1 Example—Recursive Factorial Method

The method derives directly from the recursive definition. There are two cases: `n=0` and `n>0` (remember factorial is not defined for `n<0`). When `n` is zero, `0!` is simply `1`. Otherwise when `n>0`, the result of the function is computed as `n` times the result of the function applied to `n-1`. Note that the return type is `long`. Factorial grows so quickly that even `13!` exceeds the capacity of an `int` variable.

There is nothing new syntactically. The call to `factorial` is just a method call—the difference is that the method being called is the same method as is making the call. This makes it recursive. Semantically, there is also little difference. The execution of the current method (`factorial`) is suspended and the new method (also `factorial`) is called. When that method returns with a result, execution of the current method (`factorial`) continues where it left off.

Figure 7.2 shows a method to determine the n^{th} Fibonacci number based on the mathematical definition above. As in the `factorial` method there are two cases: `n=0`, `1` and `n>1`. The first case simply sets the result to `n`. The second case involves two recursive calls to `fibonacci` for f_{n-1} and f_{n-2}, respectively.

```
29      /** This method computes the nth Fibonacci number using recursion.
30        * @param  n  Fibonacci number to compute.                          */
31      private int fibonacci ( int n ) {
32          int  result;
33          if ( n == 0 | n == 1) {
34              result = n;
35          }
36          else {
37              result = fibonacci(n-1) + fibonacci(n-2);
38          };
39          return result;
40      }; // fibonacci
```

Figure 7.2 Example—Recursive Fibonacci Method

Again the usual semantics for method calls hold. The current method execution (`fibonacci`) is suspended and the new method call (to `fibonacci` with `n-1`) is made. When that returns, another new method call (to `fibonacci` with `n-2`) is made. When this returns, the two return values are summed and assigned to `result`.

As these examples show, recursion can be an effective way to express an algorithm.

7.3 IMPLEMENTATION

Although it is clear that recursion is possible in Java and other languages, what exactly are the details of the semantics? From our knowledge of method invocation we know that the calling method is suspended when it calls another method. The arguments are passed to the called method and the called method executes to completion. When the called method returns, the calling method is unsuspended and continues its execution where it left off. This still holds for recursive methods. However, there is the question about local variables and parameters. Does each invocation of a method use the same storage—and hence same values—for its locals and parameters or is there different storage for each invocation? Although the former is possible, it would make programming

recursive methods very awkward since a called method would affect the variables of the calling method. This would violate the notion of abstraction, since the details of the called method would have to be understood to use it. Most programming languages that support recursion specify that each method invocation has its own memory for locals and parameters. This is true for Java.

LOCAL STORAGE FOR METHODS

To get an understanding of how recursive methods operate we need to analyze how memory is managed for method invocation. Our earlier memory model was adequate for non-recursive methods but it isn't sufficient to explain recursive methods. What is described here is the actual implementation for local storage for methods including both recursive and non-recursive methods.

When a method is invoked it requires storage for its parameters and its local variables. The method must also remember where it must return to, since a method can be called from more than one place. Finally there is other housekeeping information required by the run-time environment. Although it would be possible to set aside a region of storage for this purpose for each method, this would not be efficient. In a large system, with hundreds of classes and thousands of methods, not all methods are active at once. In fact many methods will not be called at all during a single execution of a system! To set aside memory for each method—whether or not it will be active—would occupy far too much memory. Instead, memory is allocated for a method only when it is called. Of course memory for the code (instructions) for the method must also be allocated. What we are considering here is the memory for the parameters and local variables.

Each method invocation requires some storage to be allocated. This block of storage is called an **activation record** (AR) since it serves as a record of the method's activation or invocation. The activation record includes storage for the parameters, local variables, the return address—where to return to—and other housekeeping information required by the language implementation. The amount of storage required can be determined at compile-time from the local variable declarations etc. and doesn't vary from invocation to invocation. Only the point in time at which the allocation is actually done varies.

What happens when the method completes? It would be possible to retain the AR for use next time the method is called. However, this would be just as inefficient as allocating the storage before execution since not all methods that are called, are called twice. Thus it makes sense for the AR to be deallocated after the method returns. This is what is done. The complete steps for method execution are shown in Figure 7.3.

1. storage for the AR is allocated
2. actual parameters are evaluated L→R and copied to storage for formal parameters in the AR
3. return address is copied to the AR
4. method body is executed, accessing the AR for parameters and locals
5. return value (if any) is evaluated
6. return address is accessed from the AR
7. the AR is deallocated
8. calling method continues at the return address using the return value (if any)

Figure 7.3 Steps in Method Execution

This explains something we have ignored so far. When a method is invoked for a second time the value of its local variables is indeterminate. That is the local variables do not necessarily have the values they had when the method last completed. This is due to the fact that the storage was deallocated and new storage is allocated for the new invocation, likely at a different location.

THE ACTIVATION RECORD STACK

How is memory allocation for activation records handled? Although it could be handled the same way as allocation of storage for objects, there is a better solution. Let's examine the pattern of method invocation. When a method `A` calls another method `B`, the execution of `A` is suspended. If `B` calls a further method `C`, `B` is suspended and `C` begins execution. If `C` in turn calls another method `D`, `C` is suspended and `D` executes. There are now four methods in a state of execution: `A`, `B`, `C` and `D`. Each of these has an AR and the ARs were allocated in order: `A`, `B`, `C`, `D`. Now what about deallocation? Only `D` can complete its execution since the others are suspended. `D` returns (to `C`) and `C` is unsuspended. Eventually `C` must complete its execution since `A` and `B` are still suspended and `D` is finished. `C`'s AR is deallocated and it returns to `B`. `B` will now execute to completion, its AR deallocated and will return to `A`. Finally `A` can complete and its AR is deallocated. The pattern of call and AR allocation is:

```
A→B→C→D
```

and the pattern of return and AR deallocation is:

```
D→C→B→A
```

These are reverse order! In fact, method call/return and hence AR allocation and deallocation, is LIFO since clearly only the last method called can return. Consider also that since only the last method invoked is executing only it can reference variables. The variables it references are in its AR, which is the last AR allocated. This behavior is exactly like a stack—LIFO allocation and access only to the last (top) item on the stack. Thus a stack can be used for AR allocation.

We have seen in Section 5.1 that there is an representation for stacks using arrays. We also know that memory is essentially one big array. Thus it makes good sense that AR allocation and deallocation be done as a stack in main memory. This is so common that most computer hardware—the CPUs—provides direct support for manipulation of memory as a stack.

Typically memory for a program's execution is divided into three areas: code (the program machine code), stack (the AR stack) and heap (the region from which dynamic storage allocation of objects is done). This is shown in Figure 7.4.

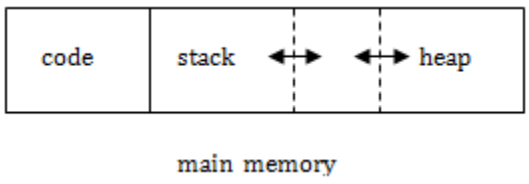


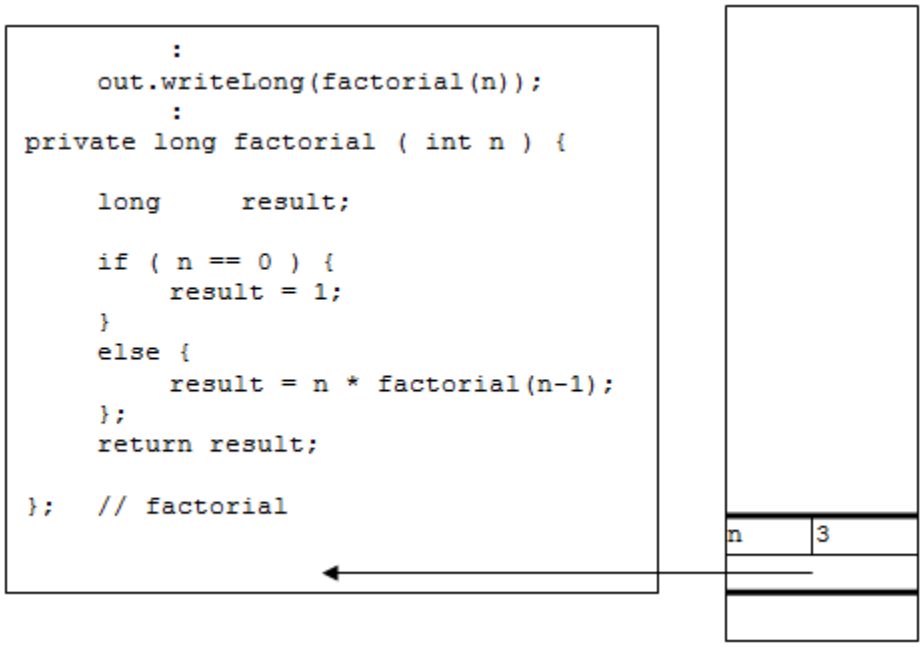
Figure 7.4 Memory Allocation

The code segment is fixed in size. The base of the stack begins at the end of the code segment and grows upwards in memory or it can be at the top end and grow downwards, depending on the hardware. The heap begins at the other end of memory and can be increased in size, if needed, growing towards the top of the stack. As long as the top of the stack and the end of the heap do not overlap, memory isn't exhausted and the program can continue executing.

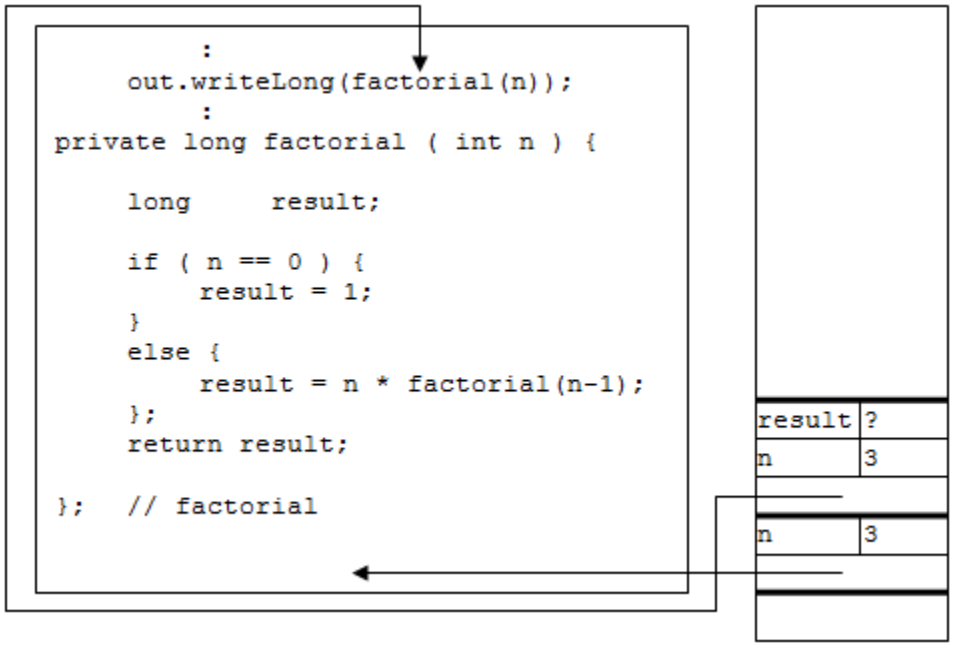
While the ARs have a well-defined allocation pattern (LIFO) and hence can be implemented efficiently, objects allocated on the heap have no predictable allocation pattern and hence the more expensive garbage collection mechanism must be used. This is why ARs are not allocated as objects.

The only difference in the AR stack when recursive methods are involved is that there can be more than one AR for the same method on the stack. When a method calls itself recursively, the process is just that of Figure 7.3. A new AR is allocated for the new invocation and the previous invocation is suspended. We have to be careful to use the term method invocation here rather than method to avoid confusion. Just as for non-recursive methods, there is one AR on the stack for each method invocation that has begun execution but not yet completed, all but the last of which is suspended.

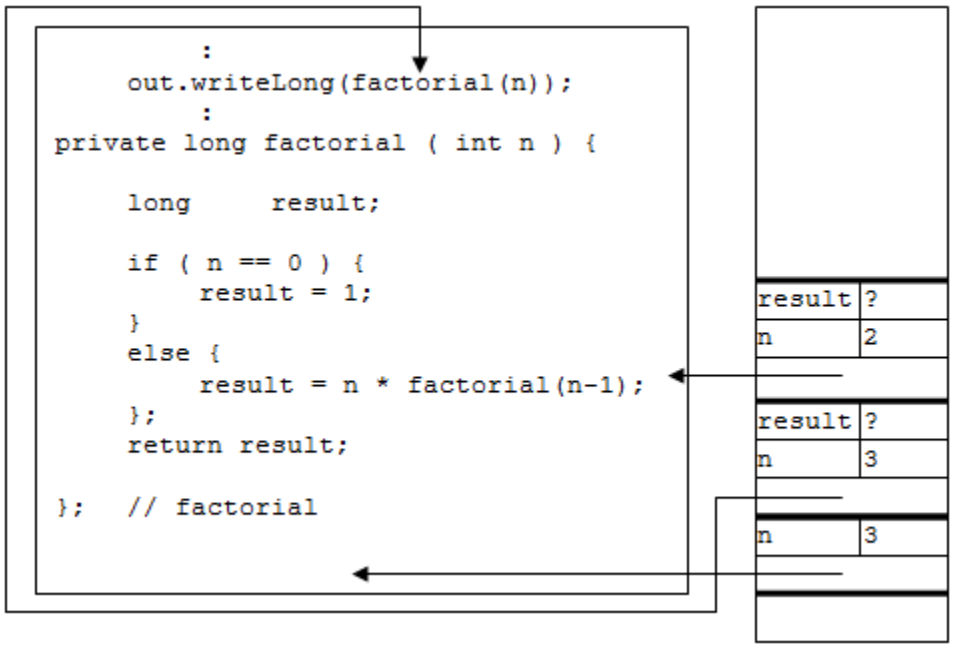
Let us look at an example. Figure 7.5 shows the behavior of the AR stack during the execution of the `factorial` method from Figure 7.1. In each part (a–j), the box on the left represents the code segment and the box on the right the AR stack. Within the AR stack, the bold lines separate the individual activation records and the internal boxes represent the storage for the formal parameters and local variables. The bottom box in each AR is the return address represented by an arrow to the point of call.



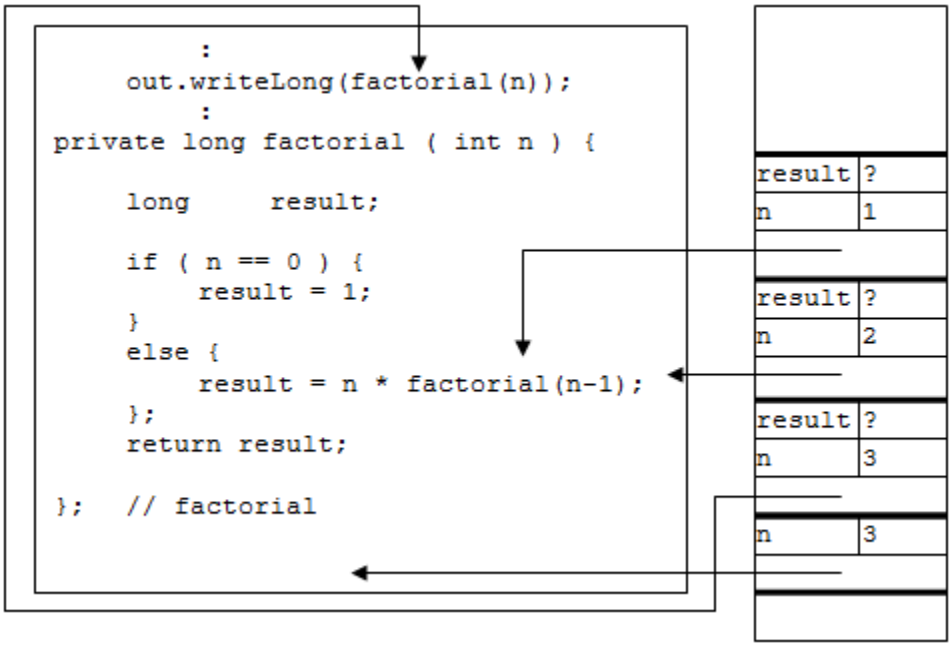
(a)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
result ?
n 3
n 3

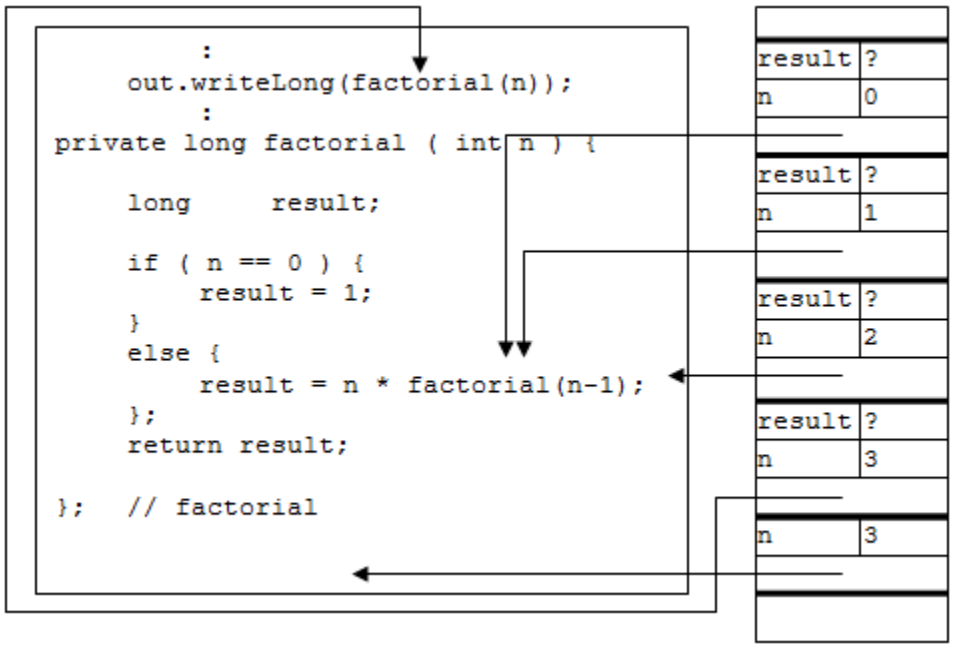
(b)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
result ?
n 2
result ?
n 3
n 3

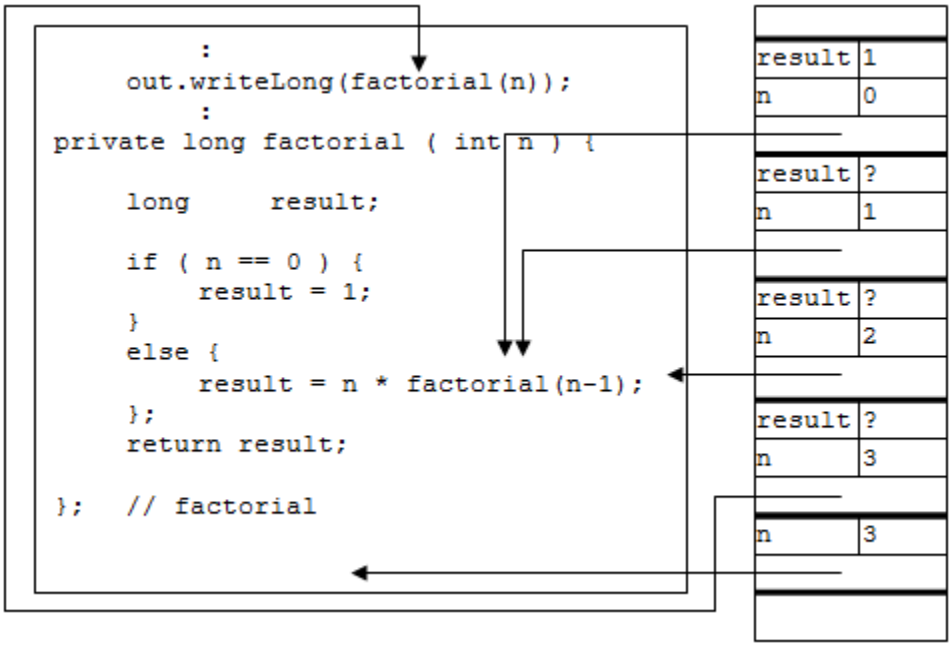
(c)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
result ?
n 1
result ?
n 2
result ?
n 3
n 3

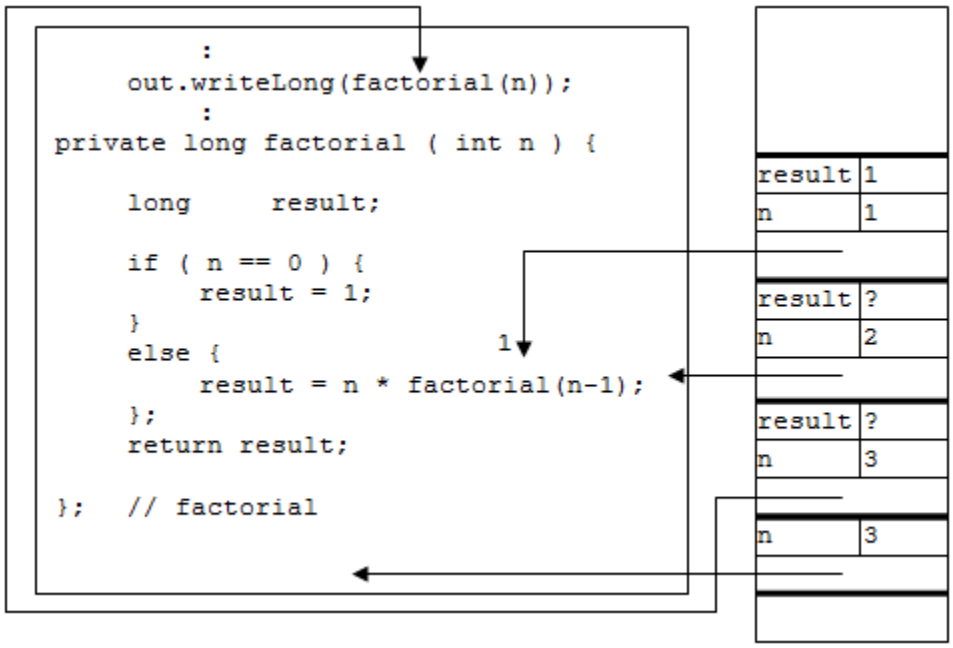
(d)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
result ?
n 0
result ?
n 1
result ?
n 2
result ?
n 3
n 3

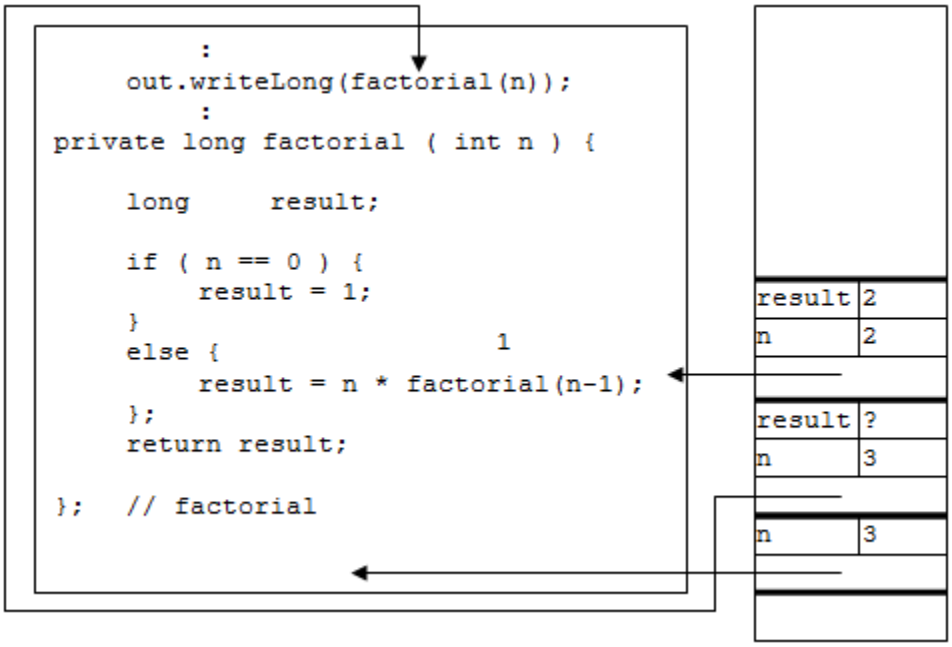
(e)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
result 1
n 0
result ?
n 1
result ?
n 2
result ?
n 3
n 3

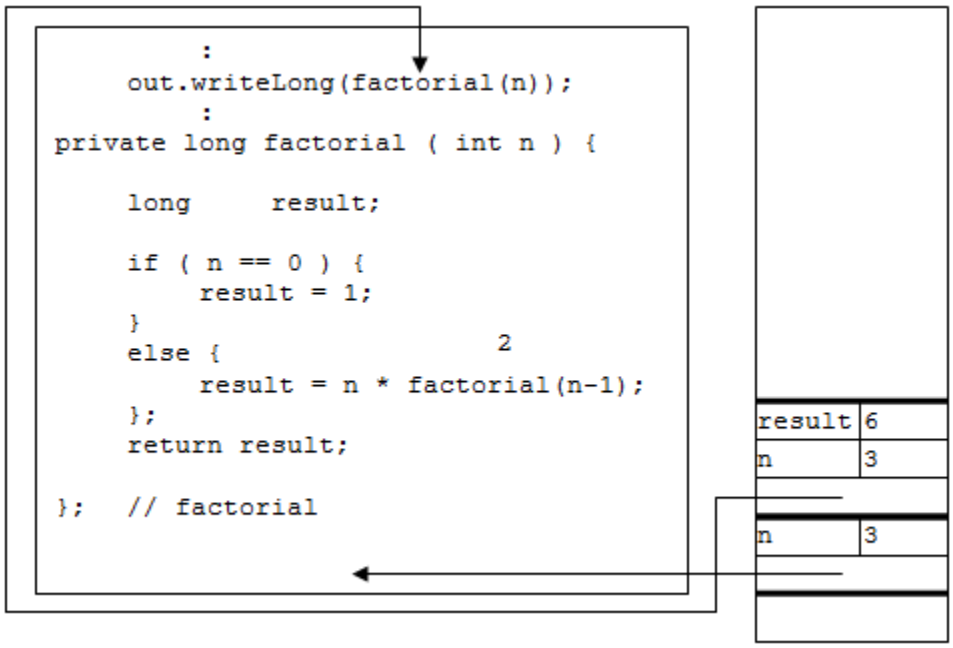
(f)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
1
result 1
n 1
result ?
n 2
result ?
n 3
n 3

(g)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
1
result 2
n 2
result ?
n 3
n 3

(h)


:
out.writeLong(factorial(n));
:
private long factorial (int n) {
long result;
if (n == 0) {
result = 1;
}
else {
result = n * factorial(n-1);
};
return result;
}; // factorial
2
result 6
n 3
n 3

(i)

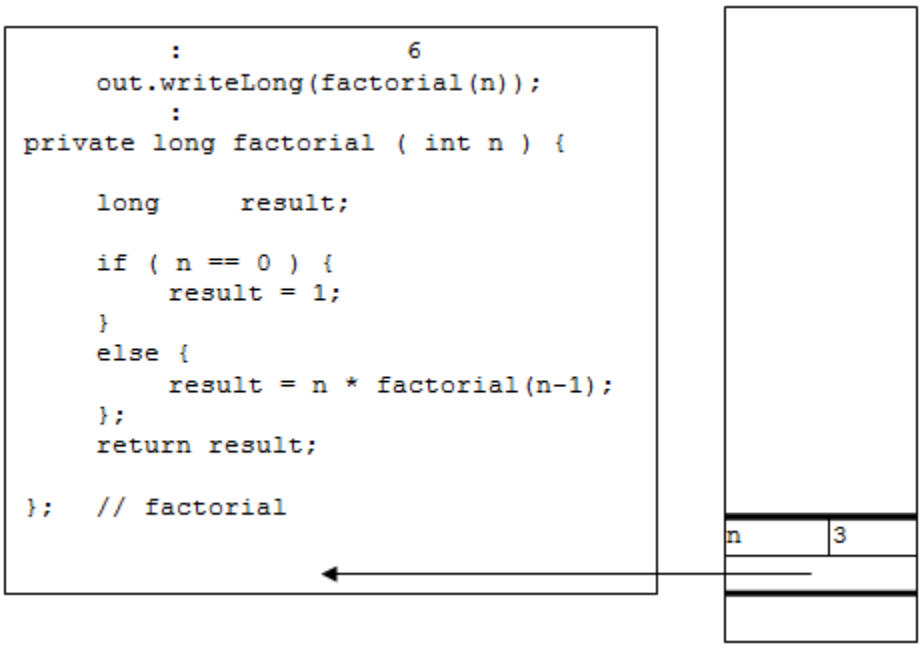


Figure 7.5(j) Recursive Method Execution

In (a), the AR is that of the calling method, the one that contains the code:

```
out.writeLong(factorial(n));
```

This AR has storage for `n`, let's say with the value `3`, and a return address to wherever it was called. At (b), the first invocation of `factorial` occurs. The parameter `n` has been copied from the argument `n` (`3`) in the calling method. The local variable `result` as yet has no value (designated `?`). The return address is to the call: `factorial(n)`. Since (`n=3`)>`0`, the else-part of the if is executed. Part (c) shows the effect of this first recursive call. A new AR for `factorial` has been created with parameter `n` being `2`, the value copied from the argument `n-1` (`3-1`→`2`) from the previous AR. Again `result` is undefined and the return address is to the recursive call in `factorial`. Parts (d) and (e) show the next two recursive calls. In each case, the parameter is copied from the argument `n-1` from the previous top of stack AR—`1` and `0` respectively. And again, `result` is undefined and the return address is to the recursive call.

Part (f) shows the state just before the first return—from the invocation with `n=0` to the invocation with `n=1`. This time, since (`n=0`)==`0`, the then-part of the if is executed. `result` is set to `1` and the return is made. The value of `result`— `1` from the top AR—is returned as the return value of the method. Part (g) now shows the state before the completion of the invocation with `n=1`. The value `1` was returned from the recursive call (shown as a `1` above the method call). `result` is computed as `n` (`1`) times this return value (`1`) yielding `1`, which is stored in the top AR. Now this invocation also returns with return value `1`. Part (h) is at the end of the next invocation. Now when `result` is computed, `n` (`2`, from the top AR) is multiplied by the return value (`1`) yielding `2`. The invocation returns with the value `2`. Part (i) shows the end of the first invocation, with `n=3`. Here the return value (`2`) is multiplied by `n` (3) yielding `6` as the value stored in `result`. This value is then returned

as the result of the method to the non-recursive call in the invoking method. Finally, part (j) shows the state after completion of the calls. The invoking method has received the return value (`6`), which it will now print. The stack has returned to the state it was in prior to the first call—at (a).

Although this kind of analysis allows us to understand how, and why recursion works, it doesn't really serve to help us understand any particular recursive method. Without substituting specific values for parameters for a call, the process described here goes on forever. What is needed is to use abstraction just as we do for a non-recursive method. To understand the `factorial` method, we examine its code. It is responsible for computing `n!`. In the case where `n=0`, the result is clearly `1`, which is correct. Any other case results in a recursive call. If this were not a recursive call we would use abstraction and assume that the method called did what it advertised without examining that code in detail. There is no reason to change strategy. Using abstraction we assume that this call will return `(n-1)!`. By multiplying this by `n`, we get the desired result—`n!`.

7.4 RECURSIVE ALGORITHMS

Recursion can be an effective tool in implementing algorithms. However how do we know when it is appropriate? What are the requirements for a recursive algorithm to be effective? When might a recursive algorithm be employed? These are questions we must answer if we wish to use recursion as a tool in software development.

REQUIREMENTS FOR TERMINATION

For an algorithm to be effective it must terminate. What is required to ensure a recursive algorithm terminates? Consider that each recursive call involves execution of the algorithm another time—essentially like a loop—so there must be some time in which there is no recursive call. An algorithm will have a number of paths, each supporting the way of handling one case of the solution. For example an if-then-else statement provides two paths. If the method is recursive at least one of the paths involves a recursive call. To ensure the method terminates at least one path must be non-recursive. We call this a **trivial case**.

Having a trivial case only makes termination possible. We must ensure that it occurs. The non-recursive cases must lead to a reduction of the problem. **Reduction** is a simplification "closer" to the trivial case. Finally we must establish that repeated application of the reduction will lead to a trivial case.

In many ways this is like an inductive proof in Mathematics. There we prove the statement for some number of simple, or trivial, cases where the proof can be expressed directly. We then express a general solution that reduces the problem towards the trivial cases. Finally we prove that the reduction will lead to a trivial case. This proof is often implicit.

PROOF OF TERMINATION

Consider the recursive factorial method in Figure 7.1. The trivial case is `n=0` where the result can be expressed without recursion since `0!=1`. The general case—for `n>0`—involves recursion and a reduction of the problem from `n!` to `(n-1)!` This is a reduction since `n-1` is closer to `0` than `n`, for any positive `n`. Finally, applying the reduction repeatedly subtracts `1` from `n`, guaranteeing that `n` will eventually reach the trivial case of `n=0`.

Similarly, consider the recursive fibonacci method shown in Figure 7.2. Here the trivial cases are `n=0` and `n=1`—expressed as the single path `n==0 | n==1`. Here $f_n=n$. The general case where `n>1` involves recursion twice. Each recursive call is a reduction, as both `n-1` and `n-2` are closer to `0` and `1` than `n`, for `n>1`. Finally, repeated subtraction of `1` and/or `2` from `n` will lead to either `0` or `1`. It could not bypass these values unless `3` or more was subtracted.

APPLYING RECURSION

When might we consider applying recursion? Clearly there must be a number of cases involved where more complicated cases can be reduced to simpler cases. There must be some cases where the solution is direct, that is easily expressible without using reduction.

*7.5 EXAMPLES

The algorithms we have considered so far can all easily be expressed using loops (iteratively). If this is always the case, why do we have recursion in programming languages? Let us consider some algorithms for which a recursive solution is more easily determined than an iterative one.

THE KOCH CURVE

In computer graphics, there are a number of interesting shapes that can be defined easily using recursion. One of the most useful is a fractal which is used extensively for representing natural phenomena. Others are the so-called space-filling curves. Let us consider one of the simplest of these space filling curves—the Koch curve.

A Koch curve is a transformation of a straight line. It is defined by an order and a length. The 0^{th} order Koch curve of length `l` is just a straight line of length `l`. An n^{th} order Koch curve consists of four connected $(n-1)^{st}$ order Koch curves each of length `l/3`. The first is in the original direction, the second at an angle of $\pi/3$ from the first, the third at $-2\pi/3$ from the second and the last at $\pi/3$ from the third—the original direction. Figure 7.6 shows Koch curves of various orders.

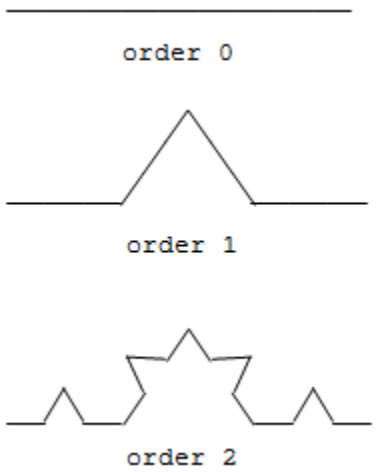


Figure 7.6 Koch Curves

This definition of the curve leads easily to a recursive algorithm for drawing an n^{th} order Koch curve of length `l`. The trivial case is order `0` where the curve is just a straight line. The recursive case involves four curves at appropriate angles, each of one lower order (reduction). Since each reduction decreases the order by one, the trivial case will ultimately be reached.

Figure 7.8 shows a method `koch` that draws a Koch curve of specified order and length. Drawing an equilateral triangle with each side being a Koch curve of order 3 draws a Koch snowflake as shown in Figure 7.7. The program uses the `TurtleGraphics` library and a `Turtle` named `yertle`.

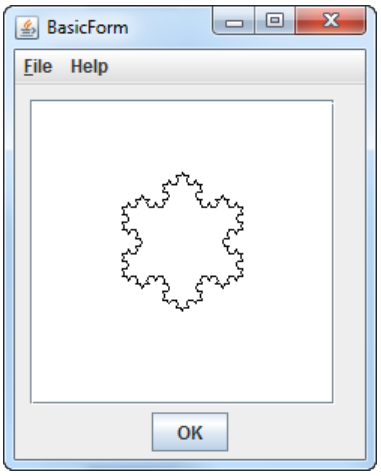


Figure 7.7 Koch snowflake

```
36      /** This method draws a Koch curve of specified order and length.
37        * @param  order  order of curve
38        * @param  len    length of curve.                                    */
39      private void koch ( int order, double len ) {
40          if ( order == 0 ) {
41              yertle.forward(len);
42          }
43          else {
44              koch(order-1,len/3);
45              yertle.left(PI/3);
46              koch(order-1,len/3);
47              yertle.right(2*PI/3);
48              koch(order-1,len/3);
49              yertle.left(PI/3);
50              koch(order-1,len/3);
51          };
52      }; // koch
```

Figure 7.8 Example—Method to Draw a Koch Curve

The `koch` method takes the `order` and `length` as parameters and then generates the Koch curve. The trivial case (order `0`) results in a straight line. The general case involves four Koch curves of one lower order with appropriate rotations between.

IMAGE SCAN

Computers are applied in many interesting areas. One of these areas is imaging—using the computer to analyze an image to detect or recognize things. This is used in areas such as robotics for robot vision systems and medical imaging such as CT (computerized tomography) and MRI (magnetic resonance imaging) scans.

As a simple example of this kind of application, consider a program that will scan an image produced by CT or MRI to determine the size and extent of a dark mass such as a tumor. To make things simple, we will assume that the image has been transformed into an array of cells—each cell being either light, represented by a space character, or dark, represented by an `x` character. What is desired is to determine the size—number of cells—of the dark mass around a specified location. A sample image is shown in Figure 7.9.

```
x   xxx   x x    x x
   xxxxx   x    xxxx
    xxxx   x      x
x   xxxx       xxxxx
     xx          x
        xxxxx
 xx      xxx       x
xxx       x       xx
 x                xxxx
      xx            x
```

Figure 7.9 Image to be Scanned

The problem here is to determine what is a single mass. We are not simply trying to count the total number of dark cells in the image but rather a set of contiguous cells. One approach is the following. A dark cell is part of a mass consisting of itself, all of its neighbors that are dark cells, all of their neighbors that are dark cells, etc. The total size of a dark mass can be determined by a cell enquiring of each of its neighbors what is the size of the mass including it. As long as we can ensure that no cell is counted more than once, the total mass size would be the sum of these sizes, plus 1 for the cell itself.

One way to ensure that cells are not counted more than once is to mark them when they are first counted. We can do that by changing the cell from an `x` to another symbol, say `*`. We now have the basis of a solution. If a cell is a space, or has already been counted (`*`), it contributes nothing (`0`) to the size of the mass. If the cell is dark (`x`), the mass surrounding it is one greater than the sum of the masses detected by the eight surrounding cells:

```
1 2 3
4 x 5
6 7 8
```

There is one problem left to handle—the border of the image. Cells on the periphery have fewer neighbors. To avoid having to handle these specially, the cells outside the image can be considered as blank space and contribute zero to the mass.

Figure 7.10 shows a method `scan` which given a specified row and column position scans image to report the size of the mass surrounding the position.

```
66      /** This method does the actual scan around the cell [r,c]. It is recursive and
67        * marks cells already counted with *s. It essentially counts the size of the
68        * remaining area from each of its neighbors and sums these to get the total
69        * area.
70        * @param  r  row coordinate of point.
71        * @param  c  column coordinate of point.                                    */
72      private int scan ( int r, int c ) {
73          if ( r<0 | r>=image.length | c<0 | c>=image[0].length ) {
74              return 0;
75          }
76          else {
77              if ( image[r][c] == ' ' | image[r][c] == '*' ) {
78                  return 0;
79              }
80              else {
81                  image[r][c] = '*';
82                  return scan(r+1,c-1) +
83                         scan(r+1,c  ) +
84                         scan(r+1,c+1) +
85                         scan(r  ,c-1) +
86                         1             + // (r,c)
87                         scan(r  ,c+1) +
88                         scan(r-1,c-1) +
89                         scan(r-1,c  ) +
90                         scan(r-1,c+1);
91              }
92          }
93      }; // scan
```

Figure 7.10 Example—Image Scan Method

The method is an implementation of the algorithm described earlier. If either the row or column is beyond the extent of the image, the method returns zero as if the cell were a space. Similarly, if the cell is a space or has already been counted (`*`), it returns zero as there is no mass including this cell that has not already been counted. Finally, there is the case of a dark cell (`x`). The method first marks the cell as counted (`*`)—this method invocation is counting it. It then recursively checks each of the neighbors to get the size of the mass including that neighbor. It sums these and adds `1` for the cell itself to get the total mass size.

7.6 RECURSION *vs* ITERATION

Consider computing `n!`. If we use the informal definition for `n!` (see Section 7.1) as a guide, we would likely come up with an iterative algorithm like that shown in Figure 7.11.

```
29      /** This method computes n! using iteration.
30        * @param  n  value to compute n!.                                    */
31      private long factorial ( int n ) {
32          long result;
33          result = 1;
34          for ( int i=2 ; i<=n ; i++ ) {
35              result = result * i;
36          };
37          return result;
38      }; // factorial
```

Figure 7.11 Example—Iterative Factorial Method

Each time through the loop we multiply by the next larger integer until we multiply by `n`. The algorithm terminates and produces `n!` just like the recursive version. Which one is better? Clearly the iterative version is `O(n)`. In the recursive version we execute the body first for `n`, then recursively for `n-1`, then `n-2`, etc. until we reach `0`. Clearly this is also `O(n)`. There is not much to choose between the two versions. Both are relatively easy to express and understand.

What about computing the Fibonacci numbers? Again we can consider the informal definition that says that each Fibonacci number is the sum of the previous two Fibonacci numbers. Considering this, if we start with the zeroth and first Fibonacci numbers (`0` and `1`), we can compute the second. From the first and second, we can compute the third, and so on. This leads to the algorithm shown in Figure 7.12.

```
29      /** This method computes the nth Fibonacci number using iteration.
30        * @param  n  Fibonacci number to compute.                           */
31      private int fibonacci ( int n ) {
32          int  fi;   // ith fibonacci number
33          int  fi1;  // (i-1)st fibonacci number
34          int  fi2;  // (i-2)nd fibonacci number
35          if ( n == 0 | n == 1 ) {
36              return n;
37          }
38          else {
39              fi1 = 0;
40              fi = 1;
41              for ( int i=2 ; i<=n ; i++ ) {
42                  fi2 = fi1;
43                  fi1 = fi;
44                  fi = fi1 + fi2;
45              };
46              return fi;
47          }
48      }; // fibonacci
```

Figure 7.12—Iterative Fibonacci Method

In each execution of the for loop, we are computing the `i`th Fibonacci number. At the start of the loop, `fi` and `fi1` are the `i`th and `(i-1)`th Fibonacci numbers, respectively, from the last time through the loop. `fi2` is updated to be the `(i-2)`th number—which was the `(i-1)`th number last time through

the loop. Similarly `fi1` is updated then and `fi`—the i^{th} number—is computed as the sum of the previous two. This algorithm is $O(n)$. In the recursive version the execution of the body for `n` makes two recursive calls for n-1 and `n-2`. These each make two recursive calls—for `n-2` and `n-3`, and `n-3` and `n-4`, respectively. These four calls each make 2 more, etc. This rises as a power of 2 and the algorithm is $O(2^n)$. This is much worse than the iterative version so we would choose the iterative version, even though it was more difficult to design and program.

CONVERTING RECURSION INTO ITERATION

As we have seen, recursion can be an effective tool in designing an algorithm. In some cases an alternative iterative algorithm is reasonably obvious. In other cases an iterative algorithm is not obvious. Is it always possible to write an iterative version of a recursive algorithm?

Consider that on traditional hardware, the only machine language control instructions are branch instructions—an instruction that changes the address from which the next instruction is to be fetched. Thus the only form of control structure at the machine language level is a loop. Therefore it must be possible to express a recursive algorithm iteratively or recursive algorithms wouldn't be able to execute on sequential machines.

It can be proven that any algorithm that can be written recursively can also be written iteratively. The body is placed in a loop which terminates when the trivial case is reached. The current state of the computation—the values of all variables affected by the body—is pushed on the stack at the top of the loop. The return from the recursive path is replaced by popping the stack restoring the values of the variables and continuing in the loop. If you think about how recursion is implemented by pushing an activation record at the method call and popping it on the return, you see that this is essentially an implementation of this constructive proof.

An algorithm created by such a process is correct, however it is typically very complex. Usually an iterative algorithm derived from scratch would be preferable. However this is not always forthcoming.

Since the generation of an iterative algorithm from a recursive algorithm is tedious and generally produces a complex algorithm, the best approach for writing a non-recursive algorithm is usually to consider the problem in a different light. When the factorial computation is viewed using the informal definition, the iterative solution is forthcoming.

APPLYING RECURSION

Given that for any recursive algorithm there exists an iterative algorithm—and incidentally any looping algorithm can be rewritten recursively—why do programming languages provide recursive methods and how do we decide when to use a recursive method?

Sometimes the recursive version of an algorithm is much simpler to express than the iterative version. The recursive Fibonacci algorithm is quite simple. Consider also an iterative version of the Koch curve algorithm. When a recursive algorithm seems apparent it is often worth looking for an iterative solution as well. This might involve looking at the problem from a different angle, rather than trying to translate a recursive algorithm into an iterative one. If an iterative solution of the same or lower order as the recursive one is found, and it is and not unduly more complex to express, the

iterative solution would be preferred. Otherwise, or if no iterative solution is forthcoming, the recursive solution would be used.

CASE STUDY: RECURSIVE STRUCTURES

A **recursive structure** (**recursive data structure**) is a container that can include values that are instances of the same container. This is like the matryoshka (or nesting) dolls often bought as souvenirs from Russia. Many structures in Computer Science can be considered as recursive structures.

Consider sequentially-linked structures described in Section 3.1. When we look at a diagram of such a structure (Figure 3.1), we see an arrow pointing to a node. Within the node there is an arrow pointing at a node, and so on. The structure can be considered to have a head—which is the item in the node—and a tail—which is a sequentially-linked structure. We could capture this idea of a list of integers through the class `List` shown in Figure 7.13.

```
… class List {
     int   head;
     List  tail;
      :
}
```

Figure 7.13 Example—Recursive List Definition

This is really just a renaming of the `Node` class (Figure 3.2) but the use of the word `List` makes it more obvious that the definition is recursive. This leads to a recursive view of a sequentially-linked structure and, implicitly, to recursive definitions for algorithms on the structure. The view of lists as recursive structures is the basis of the language LISP (LISt Processing language) whose primary data structure is a list or sequentially-linked structure and whose primary control structure is recursion.

Consider operations on lists. Here the cases are lists of various lengths (n). The trivial case is usually the empty structure. The reduction involves applying the operation to the item at the head of the structure and then processing the rest of the structure—the tail. Since the tail is part of the whole, the tail is shorter and thus the reduction leads to the empty structure.

With a definition of `List` similar to Figure 7.13, we could write a collection of methods that implement operations on lists similar to those in Lisp. For example, consider a method:

```
… int length ( List aList )
```

that returns the length—number of nodes in—the list `aList`. Clearly an empty list has zero nodes and a non-empty list has one more node than its tail. This gives a recursive implementation.

Table 7.1 describes a number of methods that might be implemented.

`load`	returns a list of items in order reading values from a file
`print`	prints the items in the list in order
`printReverse`	prints the items in the list in reverse order
`length`	returns the length (number of nodes) in the list
`sum`	returns the sum of the items in the list
`max`	returns the maximum value of any item in the list
`contains`	returns true if the list contains the item
`copy`	returns a copy of the list (all new nodes)
`plusOne`	returns a new list with each item being one greater than the original item
`append`	returns a list with the second list appended to the first
`reverse`	returns a new list which is the list in reverse order

Table 7.1 List Methods

CASE STUDY: RECURSIVE-DESCENT PARSING

As seen in Section 7.1, the syntax of a programming language is typically expressed as a set of recursive rules. A compiler must examine (parse) the structure of a program to determine how it matches the syntax rules to generate machine code. Since the rules are recursive, one technique for such parsing is to use recursive methods that parse each rule. This technique is called **recursive-descent parsing**.

Expressions in infix notation are typically part of a programming language. Recursive-descent parsing can be used to process expressions so we can use recursive-descent parsing to convert infix expressions into postfix. Instead of generating machine code as a compiler would, we can simply generate a postfix character string.

Figure 7.14 shows a set of rules describing simple infix expressions involving addition, subtraction, multiplication, division, parentheses and variables. In this notation, things enclosed in braces (`{,}`) may occur zero or more times. Things separated by vertical bar (`|`) are alternatives—only one may occur. Parentheses (`(,)`) can be used for grouping. Words in italics refer to other rules and things in quotations (“,”) are part of the language (tokens) and written as indicated. The rules in Figure 7.14 are mutually recursive—*`expr`* is defined in terms of *`term`* which is defined in terms of *`factor`* which is defined in terms of *`expr`*.

The infix to postfix conversion would consist of methods for each of the rules—*`expr`*, *`term`* and *`factor`*—which parse the input (the infix expression) to match the rule and generate the corresponding postfix expression. The body of the method for a rule is based on the syntax *viz* to parse an *`expr`* first a *`term`* is parsed. Then, repeatedly, if a + or – occur, another *`term`* is parsed. While parsing the rule, the method builds a new string consisting of the string(s) for the *`term`*s and operators in appropriate order for a postfix expression (e.g. *`term term op`*). The result of the method is the resulting string.

```
expr:
   term { ( "+" | "-" ) term }

term:
   factor { ( "*" | "/" ) factor }

factor:
   "(" expr ")" | variable
```

Figure 7.14 Syntax Rules for Expressions

SUMMARY

Recursion is a powerful technique in Mathematics and Computing for specifying an infinite set of possibilities with a finite definition. Recursion can also be a powerful programming tool.

A recursive method is a method that directly or indirectly calls itself. Each invocation of a method allocates a region of storage called an allocation record to store the parameters, local variables, return address and other information. When the method is called an AR is created and when the method returns, the AR is deallocated. Since method call/return is LIFO, the ARs are allocated on a stack. With recursive methods there can be ARs for many different invocations of the same method on the stack at once, each representing a different distinct invocation.

For a recursive method to terminate there must be at least one non-recursive path—called a trivial case—through the method. Each recursive call must involve a use of the method on a case closer to the trivial case (reduction). It must be possible to show that repeated application of the reduction will eventually lead to a trivial case to demonstrate algorithm termination.

It can be shown that any algorithm that can be written recursively can also be written iteratively. In general, a recursive method is a bit less efficient than a similar iterative method, so an iterative method is preferred. However, some algorithms are much easier to express recursively and, if they are not of a higher complexity order, they might be preferred since the maintenance costs would be lower for the simpler algorithm. In general, it is best to look for an iterative algorithm first.

REVIEW QUESTIONS

1. T F The syntax rules for Java are recursive.
2. T F A recursive method must call itself.
3. T F A method's code is stored in the heap.
4. T F Recursion can always be replaced by iteration.
5. T F A recursive algorithm for a problem is always of higher order than an iterative algorithm.
6. T F There is always a recursive version of any iterative algorithm.

7. T F The following method will terminate for all positive `n`.

```
private int f ( int n ) {
    if ( n == 0 ) {
        return n;
    }
    else {
        return n + f(n/3);
    }
};  // f
```

8. An activation record contains:

   a) storage for local variables
   b) return address
   c) storage for parameters
   d) all of the above

9. Which of the following occurs at method call?

   a) parameter evaluated
   b) AR created
   c) return address accessed
   d) none of the above

10. Which of the following occurs at method return?

    a) parameter evaluated
    b) AR created
    c) return address accessed
    d) none of the above

11. Which is not required for an effective recursive algorithm?

    a) generalization
    b) trivial case
    c) reduction
    d) all are required

12. With the code:

```
f(2);
  :
private int f ( int i ) {
    if ( i <= 0 ){
        return 1;
    }
    else {
        return f(i-1)*f(i-2);
    };
};  // f
```

how many times is the method `f` called?

a) 1
b) 3
c) 5
d) `f` doesn't terminate

13. The comparative orders of the recursive (R) and iterative (I) versions of the Fibonacci function are:

a) R: $O(n)$, I: $O(n)$
b) R: $O(n)$, I: $O(2^n)$
c) R: $O(2^n)$, I: $O(n)$
d) R: $O(2^n)$, I: $O(2^n)$

14. A recursive version of an algorithm would be preferred over an iterative version when it is:

a) of the same complexity order
b) simpler to express
c) both a and b
d) either a or b

EXERCISES

1. Suppose you need to find the maximum value stored in an array of integers. We know an iterative solution, what about a recursive solution?

a. One solution would be to consider the problem in the following way: the maximum is the larger of the last (n^{th}) item and the maximum value of the first `n-1` items. This leads to a recursive solution. What are the trivial cases? Don't forget about the possibility that the array is empty to start, in which case the result should be `Integer.MIN_VALUE`. What is the reduction? Does the reduction lead to a trivial case? Write a method:

```
private int max1 ( int[] a, int n )
```

which finds the maximum value of the `n` integers in `a[0]...a[n-1]` using the recursive process described above.

b. An alternative viewpoint would be to consider that the maximum of `n` values is the larger of the maximum of the first half of the numbers and the maximum of the second half of the numbers. This gives us an alternative recursive solution using a divide-and-conquer technique. What are the trivial cases? Don't forget about the possibility that the array is empty to start, in which case, the result should be `Integer.MIN_VALUE`. What is the reduction? Does the reduction lead to a trivial case? Write a method:

```
private int max2 ( int[] a, int lb, int ub )
```

which finds the maximum value of the integers in `a[lb]...a[ub]` using the recursive process described above.

c. Write a test class to test the two methods developed in parts a and b. Fill an array with 10,000 random integers, try each method and time the performance of the methods. What is the complexity class of each method?

2. Write a method that uses recursion to determine if a string represents a palindrome. A palindrome is a phrase that reads the same forward and backward, e.g. the word "ewe" is a palindrome as is the phrase (attributed to Napoleon) "Able was I ere I saw Elba". The method should return a `boolean` result.

   If a phrase is a palindrome then the first and last characters must be the same, and the string consisting of the second through second last characters must be a palindrome. This suggests a recursive solution. What are the trivial cases? Remember that the method must handle phrases of even and odd length. What is the reduction? Does the reduction lead to a trivial case?

   Write a test class to test your method. It should repeatedly read a string and display it to an `ASCIIDisplayer`. It should then use the method to determine whether the string is a palindrome and display message indicating the result. You may assume that the entire string is in the same case.

3. Rewrite the program for Exercise 6 in Chapter 6 that determines a path through a maze, using recursion. Like the previous solution, this method uses backtracking. However, the process is easier to express using recursion.

   The method for finding a path through the maze using recursion is related to the image scan shown in Section 7.4. Essentially the method tries to find a path from the specified co-ordinates (parameters) to the goal co-ordinates. If the square is a space, it repeatedly tries each of its four neighbors for a path until it finds one that works. If it cannot find a path from any neighbor, it returns indicating no path was found (backtracking). To make sure that the method doesn't circle around on itself, the method should mark the current square before going to any of its neighbors and then treat this mark like a wall. If it cannot find a path, it unmarks the square before returning. When it reaches the ending square, the trail of marks, like the breadcrumbs in Hansel & Gretel, will mark the path.

4. Say you are planning a road trip for reading week. Typically there are a number of routes (roads) you can take to get from your home—say St. Catharines, Ontario—to your destination—say

Orlando, Florida. On the trip you go through a number of cities, by driving the road between them. If you are interested in getting to your destination as quickly as possible, you want to take the shortest route.

This is a specific case of a general problem in Mathematics called the shortest path problem. We consider the cities as vertices in a graph and the roads connecting them as edges connecting the vertices. The edges have a weight associated with them being the distance between the cities. The shortest path is the set of edges connecting the origin—the city you are starting from—and the destination—the city you are going to—that has the smallest total weight.

The shortest path algorithm can be expressed recursively. The shortest path from a city F to a city T is the shortest of the paths from the neighboring cities to T, plus the edge from F to that neighbor. The length of the shortest path is the smallest of the lengths of the paths from the neighboring cities to T, plus the length of the edge from F to that neighbor. Of course, the length of the path from T to T is 0. This gives an algorithm for determining the length of the shortest path.

There is only one problem with the algorithm, the possibility of going in a loop such as going from St. Catharines to Buffalo to Syracuse to St. Catharines. Although we need to try all the cities on this route since there may be a shorter path through one of them, we don't want to retrace our route taking any road more than once in any path we are trying. However, the road could be part of another path we try later. The easiest way to handle this is to temporarily remove the road (edge) we are using to get from the city (vertex) we are trying to its neighbor. After we've found the shortest path from the neighbor to the destination, we can put the road (edge) back in, before trying anything else.

The last problem is to represent the map—the graph of cities and their connections (roads) to other cities. One way to do this is to use a 2-dimensional array, indexed by city number on each dimension. The entries contain the distance between the two cities—the length of the road connecting the cities. Two cities that are not directly connected by a road, where you have to go through another city are not connected, and a value of 0 is entered in the element. Since distances must be positive, we can remove a connection by changing the entry for the pair of cities to negative before following the road and back to positive after following it. Note that there are two entries for the connection between `a` & `b` one at `[a][b]` and one at `[b][a]` since any road can be followed in either direction. Both entries have to be removed and replaced together.

The method should return the length of the shortest path found between the two cities that are its parameters. If there is no path, `Integer.MAX_VALUE` should be returned. If the two cities are the same city, a value of `0` should be returned. The method will return the length of the shortest path. We won't worry about trying to display the actual path, although we could modify things to determine it.

Write a program that reads the number of cities followed by a 2-dimensional array of distances between cities and finally a pair of city numbers for the origin and destination from an `ASCIIDataFile` It should print to an `ASCIIDisplayer` the length of the shortest path between the cities. The distances are all integers.

8 QUEUES

CHAPTER OBJECTIVES

- Define the queue abstract data type.
- Explain the behavior of a queue.
- Implement a queue using contiguous representation.
- Implement a queue using linked representation.
- Apply the queue ADT in a problem

The second list-oriented collection we discuss is the queue. Like all collection ADTs it represents a collection of items of some kind and supports operations of adding and removing items from the collection. A queue is similar to a stack except it supports a different queuing discipline—a different order of insertion and removal.

We will define a queue as an ADT and then consider both a contiguous and a linked implementation.

8.1 THE QUEUE ADT

A **queue** is a list of items of some type that is initially empty. Items may be added at one end called the **rear** and items may be removed from the other end called the **front**. Unlike the stack, the terminology for the operations is not consistent. Insert, enter and add are all used for addition and delete, leave and remove for removal.

A common example of a queue is a waiting line in a coffee shop or movie theater (Figure 8.1). Here new customers arrive and enter the end of the line. When they reach the front they are served. Customers are served in the order in which they arrive. Queues are said to exhibit the **First-In-First-Out (FIFO)** queuing discipline and because of this queues are typically used where "fair" treatment is intended.

Typically a queue is defined with a number of operations: addition (enter) and removal (leave), access to the first item in the queue (front), determining if the queue is empty (empty) and often determining the number of entries in the queue (length). Not all operations have a valid meaning at all times. Specifically, if the queue is empty an item cannot be removed nor can the front item be accessed. This situation is called **queue underflow**. Since any representation of a queue will occupy storage and there is only finite memory available, it can happen that another item cannot be added to the queue. This situation is called **queue overflow**.

Figure 8.1 Coffee Waiting Line[9]

8.2 THE `Queue` INTERFACE

The queue ADT is defined by an interface shown in Figure 8.2. Unlike when we first encountered stacks, we will consider only the generic definition.

The support for queues is included in the `Collections` package as we discussed in Section 6.1. The exceptions `NoItemException`—representing queue underflow—and `NoSpaceException`—representing queue overflow—are the same as we discussed for stacks.

The `Queue` interface defines five methods. The method `enter` adds a item (`item`) to the queue. The function method `leave` removes the front item from the queue and returns it. The function `front` returns the front item from the queue without removing it. The function `length` returns the length or number of items in the queue. Finally the function `empty` returns `true` if the queue contains no items and `false` otherwise. The method `enter` signals overflow and `leave` and `front` signal underflow by throwing `NoSpaceException` and `NoItemException`, respectively.

[9] © Jeffery Simpson. Used unchanged under CC BY-NC-SA 2.0 license (http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode)

```
1  package Collections;
2
3  /** This interface defines the generic data type queue representing a FIFO
4    * collection of objects of the same type (E).
5    * @see  NoSpaceException
6    * @see  NoItemException
7    * @author  D. Hughes
8    * @version  1.0 (Jan. 2014)                                              */
9  public interface Queue < E > {
10
11     /** This method adds an item to the end of the queue. Queue overflow occurs if
12       * there is no room to add another item.
13       * @param  item  the item to be added.
14       * @exception  NoSpaceException  no more room to add to queue.          */
15     public void enter ( E item );
16
17     /** This method removes the front item from the queue. Queue underflow occurs
18       * if there are no more items left.
19       * @return  E  the item removed.
20       * @exception  NoItemException  no items available in queue.            */
21     public E leave ( );
22
23     /** This method returns the front item of the queue. Queue underflow occurs if
24       * there are no more items left.
25       * @return  E  the front item
26       * @exception  NoItemException  no items available in queue.            */
27     public E front ( );
28
29     /** This method returns true if the queue contains no items.
30       * @return  boolean  whether the queue is empty.                        */
31     public boolean empty ( );
32
33     /** This method returns the number of items currently in the queue.
34       * @return  int  the number of items in the queue.                      */
35     public int length ( );
36
37 } // Queue
```

8.2 Example—Generic `Queue` Interface

8.3 CONTIGUOUS IMPLEMENTATION OF `Queue`

The obvious start for a contiguous implementation of a queue is a variable-sized array. As items are added to the queue they are placed in increasing element positions of the array. The index `rear` indicates the next available position in the array for a queue item. Figure 8.3(a) shows an initially empty queue after 4 insertions. Unlike the stack, when items are removed from the queue they are removed from the opposite end—the front. This means that the queue will shift from left to right in the array as items are inserted and removed unless we wish to move the elements to the left on a remove, making it `O(n)`. Figure 8.3(b) shows the queue after three more insertions and two removals. To keep track of where the front is we maintain a second index (`front`) indicating the position of the front queue item.

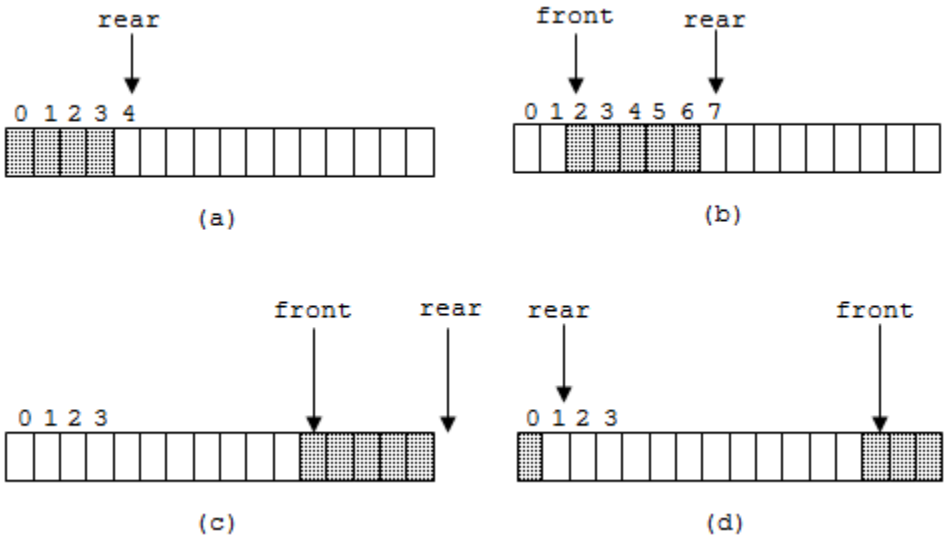


Figure 8.3 Contiguous `Queue` Representation

Things work fine until the queue has shifted all the way to the right end of the array as shown in Figure 8.3(c). At this point further insertions have no place to go and an overflow situation would result, However, there is plenty of room for new items at the front of the array! We need to reuse the positions at the front of the array for further insertions.

This is possible if we think of the array as being bent around on itself with position `0` immediately following the last position. If the array being used is called `elts`, the last position is at `elts.length-1`. Modular arithmetic on the index (modulo `elts.length`) implements this view. For example, if `elts.length` is `10` and `rear` is `9` then `(rear + 1) % 10` ⇒ `0`. If we do our increment of the index position modulo the length of the array, we get the effect of the array wrapping around on itself. Using this principle, Figure 8.3(d) shows the queue after one more insertion and two removals.

The implementation class (`ConQueue`) will bear similarity to the `ConStack` class. Like for `ConStack`, the natural constructor will take one int parameter to indicate the maximum size of the queue. And, of course, there should be an implementation of the default constructor

The methods are implemented in a straightforward manner. `enter` throws a `NoSpaceException` if the array is full. `leave` and `front` throw `NoItemException` if there are no items in the queue. Like `ConStack`, the element from which an item is removed should be set to `null` to allow the garbage collection of the item if necessary.

While it is possible to compute `length` from `front` and `rear`, it is tricky. If `front == rear` we cannot decide if the queue is empty or full! By maintaining a count as an additional instance variable, `length` (and `empty`) can be easily implemented at the minor expense of keeping the count updated on insertion and removal.

The actual implementation is left as an exercise.

8.4 LINKED IMPLEMENTATION OF `Queue`

A queue can also be implemented using a sequentially-linked structure (`LnkQueue`) with the existing generic `Node` class. The structure must contain the queue items in the order they have been added. Insertion and removal occur at opposite ends and in sequentially-linked structures the easiest end to do removal is at the front. It makes sense to have the front of the list be the front of the queue. To keep insertion from being `O(n)`, a reference to the last node of the list is also maintained (`rear`) as discussed in Section 3.1 (Figure 3.20). This arrangement is shown in Figure 8.4. The empty queue is represented by the empty list.

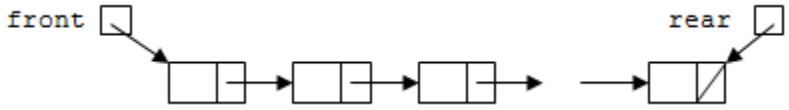


Figure 8.4 Linked `Queue` Representation

The operations are all basic sequentially-linked structure operations. In order to make the `length` operation `O(1)` an additional instance variable (`count`) can be maintained similar to `ConQueue`.. The actual implementation is left as an exercise.

All operations for both the contiguous and the linked implementations are `O(1)`. There is not much to choose between them. Of course, the space utilization for the linked implementation is superior as the size of the queue grows or if there are multiple queues, which then share the space on the heap.

CASE STUDY: SHORTEST PATH

Say you are planning a road trip for reading week. Typically there are a number of routes (roads) you can take to get from your residence—say St. Catharines, Ontario—to your destination—say Orlando, Florida. On the trip you go through a number of cities, by driving the road between them. If you are interested in getting to your destination as quickly as possible, you want to take the shortest route.

This is a specific case of a general problem in Mathematics called the shortest path problem. We consider the cities as vertices in a graph and the roads connecting them as edges connecting the vertices. The edges have a weight associated with them being the distance between the cities. The shortest path is the set of edges connecting the origin—the city you are starting from—and the destination—the city you are going to—that has the smallest total weight.

To represent the map—the graph of cities and their connections (roads) to other cities—we can use an adjacency matrix. An **adjacency matrix** is a 2-dimensional array (call it `dist`) indexed by vertex (city) number on each dimension. The entries contain the distance between the two vertices—the length of the road connecting the cities. When two cities are not directly connected by a road—where you have to go through another city—the distance value is ∞ (`Double.MAX_VALUE`). Note that there are two entries for the connection between cities `a` & `b`, one at `dist[a][b]` and one at `dist[b][a]` since a road can be followed in either direction. The matrix is symmetric across the diagonal.

A solution to this problem is to consider all possible paths through the graph from source to destination using what is called a **breadth-first search**. We start at the source and consider each of the neighboring vertices. From each of these we consider their neighbors, etc. until we run out of vertices to consider. We fan out from the source considering all vertices one hop away, then all two hops away, etc.

To keep track of our progress, we maintain an array (call it `dTo`) that records the shortest path found so far to each vertex. Initially all entries are ∞ (`Double.MAX_VALUE`). The distance to the source vertex is set to zero. A queue of vertices to consider is initialized with the source vertex. Then, as long as there are vertices in the queue to consider, we remove the first vertex (call it `v`). We consider each neighbor and, if the path through v to the neighbor is shorter than the shortest so far found to that neighbor, we update the distance to the neighbor and add the neighbor to the queue for future consideration. When we run out of vertices in the queue, the distance recorded to the destination vertex is the shortest path.

Note that we don't have to worry about going in a circle (or back and forth along one edge). For example, if we follow a path from `a` to `b` to `c` to `b`, this path must be longer than the path from `a` to `b` alone and thus will not be considered.

This algorithm is similar to Dijkstra's algorithm for finding the shortest path in a graph. However Dijkstra's algorithm uses a sorted list of vertices instead of a queue (it is not a breadth-first search) and does not visit any node more than once.

SUMMARY

A queue is a container ADT that exhibits the first-in-first-out (FIFO) property—the first item added to the container is the first item to be removed. Operations on a queue include adding and removing items, examining the first item, determining the number of items in the queue and determining whether or not the queue is empty. Exceptional conditions that can arise are queue underflow if an attempt is made to examine or remove an item from an empty queue and queue overflow if an attempt is made to add an item to the queue when there is no more space available.

The contiguous representation of a queue uses an array to contain the items and two indices indexing the front and rear items. Items are added at the rear index and removed at the front index. In each case the index is incremented modulo the array size so that the lower index positions in the array are reused as the queue moves from left to right, reaching the end of the array. A count of the number of items in the queue is maintained to simplify processing `O(1)`.

The linked representation of a queue is as a sequentially-linked structure with a tail pointer to allow insertion at the end in `O(1)`. The head of the structure is the front of the queue and the tail of the structure is the rear of the queue. A count of the number of elements in the queue can be used to make the queue size operation `O(1)`.

REVIEW QUESTIONS

1. T F A queue is a list-oriented collection in which the front of the queue is position `1` and the rear is position `n`.

2. T F A queue exhibits the FIFO property.

3. T F Queues are common in computing because they represent "fair" treatment.

4. T F When removing an item from the linked representation of a queue, it is necessary to set the item reference in the `Node` to `null` to allow garbage collection.

5. T F Using the type parameter to define the parameter for `enter` in `Queue` guarantees that the client will not accidentally place items of an incorrect type into the queue.

6. T F The operation `front` is not necessary for queues since it can be implemented in terms of the `leave` and `enter` operations.

7. What is the value of the following expression when `i=5`?

```
(i+3)%5
```

a) `0`
b) `3`
c) `5`
d) `8`

8. When removing an item from the contiguous implementation of a queue, it is necessary to:

a) decrement the count
b) increment the rear index
c) decrement the front index
d) all of the above

9. The comparative orders of the `leave` operation in the contiguous (C) and linked (L) implementations of a queue are:

a) C: `O(1)`, L: `O(1)`
b) C: `O(1)`, L: `O(n)`
c) C: `O(n)`, L: `O(1)`
d) C: `O(n)`, L: `O(n)`

10. In the linked implementation of a queue, which of the following is true when the queue (`q`) is empty?

a) `front == null`
b) `rear == null`
c) `q.length()` returns `0`
d) all of the above

EXERCISES

1. As part of the `Collections` package, implement the class `ConQueue` as described in section 8.3. Write a test harness to test your implementation.

2. As part of the `Collections` package, implement the class `LnkQueue` as described in section 8.4. Write a test harness to test your implementation.

3. A deque (double-ended queue) is an ADT similar to a queue except that additions and removals may occur at both ends, that is the front and rear.

   As part of the `Collections` package, write an interface `Deque` that describes the deque ADT. It will be similar to the interface `Queue` (Section 8.2) except that it will have additional methods for insertion at the front, deletion from the rear and examination of the rear item.

   Write an implementation of `Deque` called `ConDeque` that implements a deque using an array. The implementation will be similar to the contiguous implementation of a queue (Section 8.3) except that insertion and removal can occur at either end so the front and read indices will move both upwards and downwards in the array rather than just upwards as in the queue. The implementation should use the same circular array technique as the queue.

   Write an implementation of `Deque` called `LnkDeque` that implements a deque using a linear linked structure. Unfortunately, since deletion must occur at both ends, a sequentially-linked structure cannot be used or else the deletion at the rear will be `O(n)`. Instead, a symmetrically-linked structure (Section 3.2 & Figure 3.25) can be used.

   Write a test harness to test your `Deque` implementations. Be sure to use the interface type `Deque` to declare the deques so that the same code, with the exception of the constructor call, will work for both the `ConDeque` and the `LnkDeque` implementation.

9 LISTS

CHAPTER OBJECTIVES

- Define the list abstract data type.
- Explain the behavior of a cursored list.
- Understand how the list ADT generalizes list-oriented collections.
- Implement a list using contiguous representation.
- Implement a list using linked representation.
- Apply the list ADT in a programming problem.

This chapter brings us to the last of the three list-oriented ADTs that we began studying in Chapter 5 with the stack. Like the stack and the queue a list is an ordered collection of items. Unlike the stack and the queue which had well-defined queuing disciplines—LIFO for a stack, FIFO for a queue—the list has no particular discipline. This makes it the most general of the list-oriented ADTs. With appropriate coding—handled by the client rather than the ADT—the list can represent a stack, or a queue, or any other queuing discipline.

While the operations and terminology for stacks are commonly accepted—and likewise at least the operations for queues—lists occur with many different definitions. The definition represented in this chapter is drawn from the work on the Eiffel library and represents a fairly concise, yet powerful ADT.

9.1 THE LIST ADT

A **list** is an ordered collection of items of some type to which items may be added and from which items may be removed. This definition is clearly not complete enough to allow us to define an ADT for a library. Various refinements are possible and we will consider one based on the Eiffel data structure libraries[10].

In processing the list we can consider the last item processed as the current item, and define operations relative to this item's position. We define a cursor (marker) that references the position of the current item, and all operations such as addition and removal are done relative to the cursor. With operations to move the cursor within the list, we have a functional ADT. Sometimes this definition is called a **list with cursor** or a **cursored list**.

Since a list is ordered, there must be a first item and a second, and so on. Thus each item except for the last must have a successor. The cursor can be positioned to the first item and then moved from one item to the next. If the item is the last one, advancing the cursor moves the cursor off of the list. The current item—the one at the cursor—can be accessed. However accessing the item at the cursor is not defined when the cursor is off the list and will result in an exception. Since any representation

---

[10] Meyer, B; *Reusable Software: The Base Object-Oriented Libraries*; Prentice Hall Object-Oriented Series; Prentice Hall(1994)

of the list will occupy storage and there is only finite memory available, it can happen that another item cannot be added to the list. This situation is called **list overflow** and results in an exception.

When an item is added (`add`) to the list the addition is relative to the cursor. Where should the insertion occur: before or after the current item? After the insertion, the inserted item is the current one so the cursor should reference it. Since the cursor can be advanced from the current item to its successor, if the insertion occurs before the cursor it is still possible to get to the prior current item. Insertion in front of the cursor also allows insertion at the front of the list. What about insertion at the end of the list? If the cursor is at the last item, insertion will be as second last. However, the cursor can be off the end of the list. We can define insertion in this state to insert at the end of the list. This is consistent with the situation when the list is empty where the cursor must be off the list and insertion would occur at the end—also being front—of the list.

The item referenced by the cursor can be accessed (`get`). If the cursor is off the end of the list, there is no current item and access is undefined causing an exception. Note that access in an empty list corresponds to access when the cursor is off the list and is thus an error.

Removal (`remove`) is of the item currently referenced by the cursor. If the cursor is off the end of the list, an exception occurs. What happens to the cursor after a removal? Clearly there is no current item so there are three possibilities: the predecessor of the removed item, the successor of the removed item or the cursor is undefined. Choosing to leave the cursor referencing the successor of the removed item makes the most sense. This way insertion and removal are inverses—an insert followed by a removal or a removal followed by an insert leaves the list unchanged. Deletion of the first item leaves the cursor at the new first item and deletion of the last item leaves the cursor off the list.

The cursor must be able to move through the list. If there is an operation that places the cursor at the front of the list (`toFront`) and an operation that advances the cursor to the next item in the list (`advance`), the list can be traversed. The list will be a sequentially accessed structure, like a sequential file. If the cursor is already off the end of the list, advancing has no effect. To allow the client to determine when the cursor has moved off the list, an appropriate inquiry operation should be available (`offList`).

Using this definition, Figure 9.1 shows the traversal of a list. It is simply a version of the generalized sequential traversal algorithm found in Figure 3.11.

```
list.toFront();
while ( ! list.offList() ) {
    process list.get();
    list.advance();
};
```

Figure 9.1 List Traversal Algorithm

The list ADT can be considered as a generalization of the list-oriented collections. The other list-oriented collections—the stack and the queue—are just special cases of the list. To use the list as a stack, the cursor is never advanced on its own. An insertion will occur in front of the front item and removal will be of the front item. Both of these operations will leave the cursor at the front item. To use the list as a sequentially ordered collection, an advance must be done after each insert. To

simulate a queue, the front of the list is the front of the queue. Insertions are preceded by traversal to the end of the list and removals are preceded by moving the cursor to the front of the list.

9.2 THE `List` INTERFACE

The generic `List` interface shown in Figure 9.2 defines a list ADT as part of the `Collections` package.

The `List` interface defines eight operations for lists. The method `add` adds the item in front of the cursor unless the cursor if off the list in which case it adds at the end of the list. The methods `get` and `remove` return the item at the cursor, with `remove` removing it. If the cursor is off the list a `NoItemException` is raised. After `remove` the cursor references the successor of the item removed, or is off the list.

The method `empty` returns `true` if the list contains no items. The method `length` returns the number of items in the list.

`toFront` moves the cursor to the front item in the list, unless the list is empty in which case the cursor is off the list. `advance` moves the cursor to the next item in the list, unless the cursor is off the list in which case it does nothing. `offEnd` returns `true` if the cursor is off the end of the list.

9.3 CONTIGUOUS IMPLEMENTATION OF `LIST`

The basis for a contiguous representation of a list is a variable-sized array. Unfortunately, there is no way to simultaneously achieve both contiguity of items—all items side by side without gaps—and `O(1)` execution for all operations. Insertion and removal can occur anywhere within the array. Even if contiguity is relaxed by marking removed items as being empty—for example, setting the element to `null`—insertion between two filled elements would still present a problem and eventually all elements will be either filled or marked as removed and the structure would appear to be full. Reusing the removed elements is not easy, as they are unlikely to be where they are needed for insertion. We will retain contiguity of the items at the expense of some operations being `O(n)`.

The list is represented as a traditional variable-sized array (`items`) with the list items occupying the lower `length` elements of the array. The `cursor` is represented as the index of the current item. If `cursor` is equal or greater than `length` of the list, it is off the list. Figure 9.3 shows this implementation with (a) showing a list with 4 items and the cursor marking the second item. Figure 9.3 (b) shows an empty list and Figure 9.3 (c) shows a list of 4 items with the cursor off the end. Note that the cursor is also off the end in Figure 9.3 (b).

A constructor with one `int` parameter—indicating maximum list size—would be the natural constructor and a default constructor should be defined.

The `add` method first checks that there is available space, throwing a `NoSpaceException` if not. It then creates an opening for the new item in front of the cursor by moving the items from the cursor to the end of the list up one element. This must be done with care. To avoid overwriting the items, the process is done from right to left—from the high end to the low end. Once space is created, the item is inserted and `length` updated. Note that `cursor` stays at the same element but now references the inserted item, which is in front of the prior item.

```
1  package Collections;
2
3  /** This interface defines the generic abstract data type list representing a
4    * sequence of items that can be accessed by their relative position in the list.
5    * Unless the cursor is off the end of the list, one list item (considered to be
6    * the current item) is referenced by the cursor and operations are relative to
7    * this item.
8    * @see NoSpaceException
9    * @see NoItemException
10   * @author D. Hughes
11   * @version 1.0 (Jan. 2014)                                                    */
12 public interface List < E > {
13
14     /** This method adds an item to the list in front of the cursor. If the cursor
15       * is off the end of the list, the insertion is at the end of the list. The
16       * cursor references the item just added.
17       * @param  item  the item to be added.
18       * @exception  NoSpaceException  no more room to add items.                 */
19     public void add ( E item );
20
21     /** This method removes the item at the cursor from the list. The cursor
22       * references the item following the one removed.
23       * @result  E  the item removed.
24       * @exception  NoItemException  no current item (cursor off list).          */
25     public E remove ( );
26
27     /** This method returns the item at the cursor.
28       * @result  E  the item at the cursor.
29       * @exception  NoItemException  no current item (cursor off list).          */
30     public E get ( );
31
32     /** This method returns true if the list has no items.
33       * @return  boolean  the list has no items.                                 */
34     public boolean empty ( );
35
36     /** This method returns the number of items in the list.
37       * @result  int  the number of items in the list.                           */
38     public int length ( );
39
40     /** This method moves the cursor to the first item on the list. If the list is
41       * empty, the cursor will be off the end of the list.                       */
42     public void toFront ( );
43
44     /** This method advances the cursor to the next item in the list. If the cursor
45       * was off the list, it remains off the list.                               */
46     public void advance( );
47
48     /** This method indicates if the cursor is off the end of the list.
49       * @return  boolean  the cursor is off the end of the list.                 */
50     public boolean offEnd ( );
51
52 } // List
```

Figure 9.2 Example—Generic List Interface

The `remove` method is the converse of the `add` method. After checking that there is an item to remove and throwing a `NoItemException` if not, it obtains the item at `cursor` and moves the following items down one element to fill the hole. This time the move is from left to right as the element at `cursor` is now "empty". After the move, there will be an extra reference to the last item which must be set to `null` to allow for garbage collection. `length` is then updated. Again `cursor` is still in the right place.

The remaining methods are trivial. The implementation is left as an exercise.

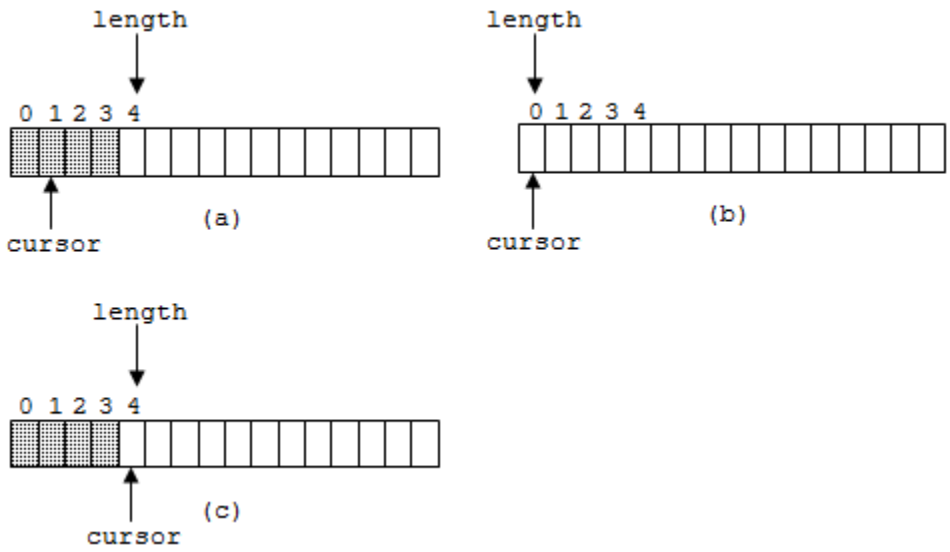


Figure 9.3 Contiguous List Representation

9.4 LINKED IMPLEMENTATION OF `List`

The representation of `List` as a linked structure is reasonably straightforward. As defined, list processing is sequential—`advance` moves through the list from beginning to end—and sequentially-linked structures are designed for such processing. The main issue is the cursor and insertion and deletion at the cursor.

The cursor references a particular item and can be moved through the list (`advance`). It makes sense that the cursor be represented as a `Node` reference, referencing the current item. `toFront` sets `cursor` to the first node in the list and `advance` moves cursor to the next node. The off list condition can be represented by `cursor` being `null`. This is the value of the `next` field in the last node in the list.

Insertion is to occur in front of the cursor. This presents a problem since it is necessary to have a reference to a node's predecessor if insertion is to occur in front of it. One solution would be to use a symmetrically-linked structure since each node would reference its predecessor, however this is overkill. The only node whose predecessor needs to be known is the one referenced by cursor. It would be sufficient to maintain a second `Node` reference (`precursor`) which always references the node in front of the node referenced by `cursor`. Insertion at the end of the list would work appropriately since `precursor` would reference the last node when `cursor` is `null`. This

arrangement is much like the pair of traveling pointers used in list manipulation algorithms such as insertion at the end of a linear-linked structure (Figure 3.13).

Removal is of the node referenced by `cursor`. This also requires a reference to the node's predecessor. `precursor` can assist here as well. Since insertion in front of the first node and deletion of the first node would otherwise require special cases, it is useful to include a header node.

Figure 9.4 shows the configuration of the sequentially-linked structure representing a `List`. Part a) shows the empty state. `precursor` references the header node and `cursor` is `null`. The cursor is off the list since `cursor` is `null`. Insertion before `cursor` inserts a node at the front of the list—after the header. Part b) shows the non-empty state with `cursor` referencing an arbitrary node. Insertion or deletion can be achieved with reference to the node's predecessor (`precursor`). Part c) shows a non-empty list in the off list state—when `cursor` has moved off the end of the list. This is the logical result of an `advance` when `cursor` is referencing the last node of the list. `cursor` is `null` indicating the off list condition and `precursor` references the last node of the list so insertion would occur at the end of the list. Note the empty state is just a special case of the off list state.

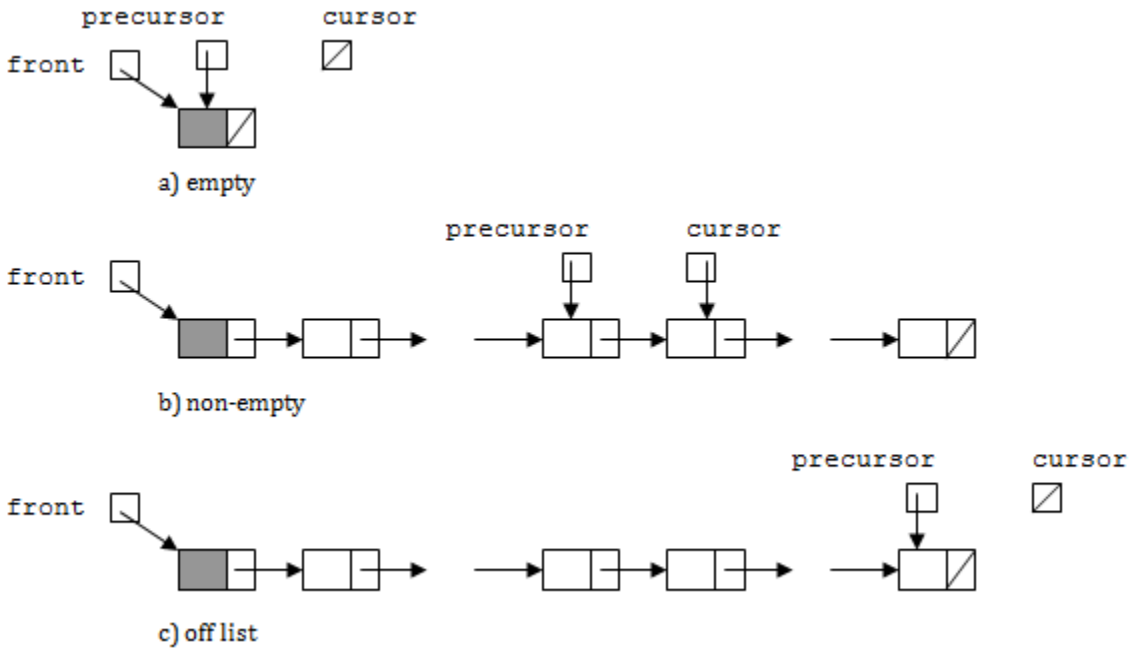


Figure 9.4 Linked List Representation

Each of the operations is now easily implemented. `add` is insertion between `precursor` and `cursor`. `remove` is deletion of the node at `cursor`. `get` simply accesses the item at `cursor`. `toFront` sets `precursor` to the header node and `cursor` to the first node. `advance` advances `precursor` and `cursor` in tandem to the next nodes. `offEnd` reports `true` when `cursor` is `null`. `length`, can be implemented as `O(1)` by maintaining a count of the number of items and updating it on insertion and removal. `empty` can simply check that the length is zero. The actual implementation is left as an exercise.

In comparing the two representations, the linked representation is usually preferable. All operations are `O(1)` for the linked representation while `add` and `remove` are `O(n)` for the contiguous representation since on average half the items will have to be moved to maintain contiguity. If the list size is known and insertion and removal are infrequent after the initial build of the list, the contiguous representation may be appropriate since it may occupy less space and be marginally more efficient. In most other cases, the linked representation would be used.

*9.5 JAVA `COLLECTION` FRAMEWORK

Data structures are fundamental to Computer Science. The standard Java release includes interfaces and classes supporting most of the standard data structures as part of the Java Collections Framework. Unless it is necessary to provide a specialized implementation of these data structures, or a specific implementation is not included, there is no reason for a programmer to write data structure implementations. Rather the classes in the Collections Framework should be used. These implementations have been well tested since they have been well used and are generally efficiently implemented.

The Java Collections Framework begins with the interface `Collection`. Figure 9.5 shows a partial list of the methods in the interface. The interface is generic in the type of element (`E`) in the collection. It describes the common operations of collections. Items may be added (`add`) and removed (`remove`). The collection can be queried to see if it contains an item (`contains`). The number of items in the collection can be determined (`size`) and it can be determined whether or not there are any items in the collection (`isEmpty`). All items can be removed from the collection (`clear`). A collection can be traversed using an `Iterator` and in a for each statement (`Iterable` and `iterator`, see Section 11.4). Finally, for interoperability between collections and array-based APIs, it is possible to produce an array of items from a collection (`toArray`).

```
1  package java.util;
2  public interface Collection <E> extends Iterable<E> {
3
4      public boolean add ( E o );
5      public void clear ( );
6      public boolean contains ( Object o );
7      public boolean isEmpty ( );
8      public Iterator<E> iterator ( );
9      public boolean remove ( Object o );
10     public int size ( );
11     public Object[] toArray ( );
12
13 }  // Collection
```

Figure 9.5 Java `Collection` Interface

The Collections Framework includes a class `Stack`. It might have been expected that there would have been an interface `Stack` and some number of implementations classes. However the `Stack` class predated the introduction of the Collections Framework. To maintain compatibility with legacy

code, the class `Stack` is included in the framework. The implementation of `Stack` is a contiguous representation based on the class `Vector`—an extensible array.

```
1  package java.util;
2  public class Stack <E> implements Collection<E>, Serializable {
3
4      /* From Collection */
5      public boolean add ( E o );
6      public void clear ( );
7      public boolean contains ( Object o );
8      public boolean isEmpty ( );
9      public Iterator<E> iterator ( );
10     public boolean remove ( Object o );
11     public int size ( );
12     public Object[] toArray ( );
13
14     /* Stack */
15     public boolean empty ( );
16     public E peek ( );
17     public E pop ( );
18     public E push ( E item );
19
20 }  // Stack
```

Figure 9.6 Java `Stack` Class

Figure 9.6 shows a partial list of the methods for `Stack`. It provides implementations for the methods defined for `Collection` as well as methods specifically defined for stacks. Items are added at the top of the stack (`push`) and removed from the top (`pop`). The top item can be examined (`peek`). The stack can be tested to see if it has any items (`empty`). `peek` and `pop` throw an `EmptyStackException` if the stack is empty. For convenience, `push` returns a reference to the item pushed. `add` and `isEmpty` are synonyms for `push` and `empty`. The other methods from `Collection` are not normally defined for stacks, but are present since `Stack` is a subtype of `Collection`. Normally they would not be used.

Figure 9.7 shows a partial list of the `Queue` interface. Unlike `Stack`, `Queue` is defined as an interface extending `Collection` and has a number of implementation classes. `ArrayDeque` implements a `Queue` as resizable array (like `Vector` for `Stack`). `LinkedList` implements `Queue` as a symmetrically-linked structure.

`offer` adds the item to the rear of the queue, returning `true` if the add was successful. `peek` returns a reference to the item at the front of the queue and `poll` removes the item at the front of the queue. `peek` and `poll` return `null` if the queue is empty. `isEmpty` from `Collection` returns `true` if the queue is empty and `size` from `Collection` returns the number of items in the queue. The other methods from `Collection` are not normally used for a `Queue`.

```
1  package java.util;
2  public interface Queue < E > extends Collection<E> {
3
4      /* From Collection */
5      public boolean add ( E o );
6      public void clear ( );
7      public boolean contains ( Object o );
8      public boolean isEmpty ( );
9      public Iterator<E> iterator ( );
10     public boolean remove ( Object o );
11     public int size ( );
12     public Object[] toArray ( );
13
14     /* Queue */
15     public boolean offer ( E o );
16     public E peek ( );
17     public E poll ( );
18
19 }  // Queue
```

Figure 9.7 Java `Queue` Interface

The `List` interface in the Java Collections Framework defines an indexed list where items are identified by their position in the list starting at `0`. `ArrayList` implements `List` as a resizable array and `LinkedList` implements `List` as a symmetrically-linked structure.

```
1  package java.util;
2  public interface List <E> extends Collection<E> {
3
4      /* From Collection */
5      public boolean add ( E o );
6      public void clear ( );
7      public boolean contains ( Object o );
8      public boolean isEmpty ( );
9      public Iterator<E> iterator ( );
10     public boolean remove ( Object o );
11     public int size ( );
12     public Object[] toArray ( );
13
14     /* List */
15     public void add ( int index, E element );
16     public E get ( int index );
17     public int indexOf ( Object o );
18     public E remove ( int index );
19     public E set ( int index, E element );
20
21 }  // List
```

Figure 9.8 Java `List` Interface

The method `add` with two parameters adds the item at the specified index position which must be between `0` and `size()` or an `IndexOutOfBoundsException` is thrown. Addition causes all items from the specified position to the end of the list to move up one position. Note that adding at position `size()` adds as the last item in the list. remove The method `remove` with an `int` parameter removes the item at the specified position. `remove` shifts all items to the right of the specified position down one position. `get` returns the item at the specified position without removing it and `set` replaces the item at the specified position with the argument, returning the replaced item. For `remove`, `get` and `set`, the index position must be between `0` and `size()-1` or an `IndexOutOfBoundsException` is thrown. `indexOf` searches for the first occurrence of the item in the list returning the index position if found and `-1` if not. `isEmpty` and `size` from `Collection` are used to determine if there are any items in the list and how many there are.

CASE STUDY: CAR RENTAL AGENCY REVISITED

In Chapter 3 the case study was an application for a car rental agency which involved maintaining lists of cars-available and rented. In that chapter, we discussed an implementation directly using a sequentially-linked structure. Conceptually any kind of list would work, it doesn't have to be a linked structure. We could abstract the implementation details by using our `List` ADT in the application instead. This would allow us to substitute any implementation of `List` in place of the linked structure used in Chapter 3. The other advantage is that, instead of writing the code for the linked structure manipulation ourselves, we and use a library that has likely been well used and tested, reducing the likelihood of programming errors.

The problem is repeated here.

A car rental agency owns a number of cars that they rent to customers. The agency maintains a record for each car including its licence plate number (string), its current mileage (integer) and its car class (integer: 0=Economy, 1=Full Size, 2=Van and 3=SUV). When a customer arrives at the agency, a car is selected for the customer. When a car is returned, the charge is computed based on the difference between the mileage at return and the mileage at rental and the mileage rate for the class of car rented.

The company needs a program to keep track of their fleet of cars. This can be done using two lists: one of cars available for rent and one for cars currently rented but not yet returned. When a car is rented, it is removed from the available list and added to the rented list. Similarly, on return a car is removed from the rented list and added to the available list. At any time, the agency should also be able to display the two lists to track their fleet.

VERSION 2. To minimize the wear and tear on the fleet, whenever a car is rented, the company wants to rent the one with the lowest mileage in the selected class. This can be accomplished by keeping the available list in order by mileage from smallest to largest. When a car is to be rented, the customer specifies a car class and the first car in available list that matches that class car has the lowest mileage and is removed, rented and added to the front of the rented list. When a car is returned, the rented list is searched for the car with the supplied licence plate. This item is removed from the rented list, the charge computed and mileage updated and the car is added to the available list in increasing order by mileage. The GUI and listing of the fleet information is the same as in version 1.

To keep the available list in order by mileage, the list must be traversed (see Figure 9.1) until a car with higher mileage is encountered. The insertion occurs before this entry. To remove the car with the lowest mileage in the desired class from the available list, the list must be traversed until a car in the desired class is encountered. This car is removed and rented. To remove a car from the rented list, the list must be traversed until the car with the desired licence plate is encountered. This car is then removed. Adding a car to the rented list can be done in any order, so we will add it at the front.

SUMMARY

A list is an ordered collection of items to which items can be added and from which items can be removed. It is the most general of the list-oriented collections and can be used to represent any of the others. To allow a concrete implementation, a more complete specification is needed. One such specification is a cursored list in which a cursor identifies a particular item on the list and operations are relative to the cursor position. Insertion occurs in front of the cursor, access is to the cursor item and deletion is of the cursor item.

The contiguous implementation of a cursored list uses a contiguous portion of an array and implements the cursor as an array index. Insertion and removal involve moving items within the array to maintain contiguity and are `O(n)`, all others are `O(1)`. The linked implementation of a cursored list involves a sequentially-linked structure with a header. The cursor is implemented as a `Node` reference. To achieve insertion in front of the cursor and deletion of the cursor item, a second `Node` reference is maintained to reference the node in front of the cursor. Insertion removal, and all other operations are `O(1)`.

REVIEW QUESTIONS

1. T F The List ADT is an ordered collection.

2. T F Insertion and removal in a List can be FIFO.

3. T F A cursor is an index into an array.

4. T F Deletion in a contiguous implementation of a cursored list is `O(1)`.

5. T F The `precursor` reference can never be `null` in the linked representation of a cursored list.

6. In a list, if deletion removes the item at the cursor leaving the cursor at the next item, insertion should occur:

   a) at the front of the list
   b) in front of the cursor
   c) after the cursor
   d) at the end of the list

7. To add items to a list in sequential order, which methods should be used?

   a) add
   b) advance
   c) advance then add
   d) add then advance

8. The following code:

```
for ( int j=cursor ; j<length ; j++ ) {
    items[j+1] = items[j];
};
```

   a) moves the items `cursor...length-1` up one position in the array
   b) moves the items `cursor...length-1` down one position in the array
   c) moves the items `cursor+1...length` up one position in the array
   d) none of the above

EXERCISES

1. As part of the `Collections` package, write the contiguous implementation of the list ADT. Write a test harness to test your implementation.

2. As part of the `Collections` package, write the linked implementation of the list ADT. Write a test harness to test your implementation.

3. Write a Java program to provide a personal information manager (PIM). A PIM is much like the old-fashioned "rolodex" where an entry (card) is maintained for each "contact" (person with whom you wish to communicate). Each card contains such information as name (a string), address (three strings: street, city, zip code), and telephone number (a string). In the PIM, you can add new contacts, remove outdated contacts, update contact information and, most importantly, search the entries for the one giving the information about a particular contact.

   The PIM program should first load the contact list with contact information that is stored in an `ASCIIDataFile`. It should then repeatedly prompt the user, using an `ASCIIPrompter`, for an operation to perform and display the result of the operation to an `ASCIIDisplayer`. When the user quits, it should write the updated contact information back to a new `ASCIIOutputFile`. The operations should include:

   `a` or `A` Add a new contact entry to the contact list. The contact information should be prompted for using the `ASCIIPrompter`.

   `r` or `R` Remove a contact entry from the contact list. The name should be prompted for and the entry with that name deleted.

   `d` or `D` Display a contact entry. The name should be prompted for and the entry with that name displayed, appropriately formatted.

`u` or `U` Update a contact entry. The name should be prompted for and that entry displayed. The user should then be prompted for the updated information. All information must be entered.

`q` or `Q` Quit and save the contact list.

Use the `List` type of the `Collections` package to represent the contact list. and choose the appropriate implementation class.

4. Write a Java program to implement a simple Christmas list application. The program maintains a list of the people for whom you buy Christmas presents. For each recipient, the following information is stored: name (string), gift purchased (description as a string, empty string if nothing purchased yet), purchase cost (double, 0.0 if nothing purchased yet). The program allows the user to perform four operations, indicated by the characters: `p` (purchase), `d` (display), `t` (total) or `q` (quit). For a purchase, it searches for the recipient by name and then records the gift description and cost. For a display, it searches for the recipient by name and displays the name, gift and cost. For a total, it computes the total amount spent on presents and displays this amount.

   A file of names of people that you intend to buy Christmas presents for has been prepared as an `ASCIIDataFile`. This file should be used to initialize the list with no gifts purchased.

   The program should use the `List` type of the `Collections` package to represent the recipient list and an appropriate implementation class chosen. The order of the recipients on the list is irrelevant, however, it makes sense to preserve the order in the data file. If a name is not located, an appropriate message should be produced. An invalid operation can just be ignored.

5. Write a Java program to implement a simple appointment schedule application. The program maintains a list of appointments—for example: classes, seminars, tutorials, lunch—that a student has during the day. For each appointment, the following information is stored: time (string, the beginning time of the appointment, `hh:mm`), description (string, the description of the appointment), duration (string, the duration of the appointment, `hh:mm`). The program should allow the user to perform four operations, indicated by the operation codes (char): `s`, schedule appointment; `d`, display; `r`, reschedule appointment to a different time slot; and `q`, quit. For scheduling an appointment, it adds the appointment prompting for time, description and duration, to the schedule. The program should verify that the appointment has not already been scheduled before scheduling it. That is, there must not be an existing appointment with the same description. It should generate a message if there is one. For display, it searches for the appointment by description, prompting for description, and displays the time, description and duration to an `ASCIIDisplayer`. If there is no such appointment, it should indicate this. For rescheduling an appointment, the program searches for the appointment by description, prompting for description, and updates the time and duration of the appointment, prompting for the new time and duration. If there is no such appointment, it should indicate this.

   A file giving the current schedule of appointments for the student provided as an `ASCIIDataFile`. The program should load these appointments before prompting the user for operations. After the user quits, the program should write the updated list to an `ASCIIOutputFile`.

The program should use the `List` type of the `Collections` package represent the appointment list and an appropriate implementation class should be chosen. The order of the appointments on the list is irrelevant, however it makes sense to preserve the order in the file. For the schedule operation, if the description already exists, an appropriate message should be produced. For the display and reschedule operations, if the description is not located, an appropriate message should be produced. You may assume that the operation code is correct.

6. Write a Java program to provide a To Do list manager. A To Do list is a list of tasks that an individual needs to accomplish such as assignments, reports, meetings etc.. Each task on the list has a projected due date (integer: `yymmdd`) and time (integer: `hhmm` in 24-hour form) and a description. Tasks are ordered on the list by due date and time. New tasks may be added, completed tasks can be removed and the to do list can be examined (i.e. listed). To enable easy identification of a task for deletion, a unique task id (integer) is associated with each task on the list.

   The To Do list program should first load the To Do list with task information that is stored in an `ASCIIDataFile`. This will consist of one line per task with date, time and description. It should then repeatedly prompt, using an `ASCIIPrompter`, the user for an operation to perform, displaying the result of the operation to an `ASCIIDisplayer` as necessary. When the user quits, it should write the updated task information back to a new `ASCIIOutputFile` in the same format as the original input file. The operations should include:

   `a` or `A` Add a new task to the To Do list. The due date and time and the description should be prompted for via the `ASCIIPrompter`.

   `d` or `D` Delete a task entry from the contact list. The task id should be prompted for and the entry with that id deleted.

   `l` or `L` List the To Do list. Each task entry should be listed in date and time order including task id, date, time and description.

   `q` or `Q` Quit and save the To Do list.

   Use the `List` type of the `Collections` package to represent the To Do list and choose an appropriate implementation class. The task id is just the relative position of the task in the list, with the first task, the one with the earliest due data and time, being task id 1. To keep the list in sorted order traversals of the list will have to be performed to locate the insertion point.

10 SEARCHING AND SORTING

CHAPTER OBJECTIVES

- Explain the need for searching in computer systems.
- Describe algorithms for searching namely: sequential search and binary search.
- Choose and apply an appropriate search algorithm in a problem.
- Explain the need for sorting in computer systems.
- Describe algorithms for sorting namely: selection sort, insertion sort, exchange sort and merge sort.
- Choose and apply an appropriate sorting algorithm in a problem.

Much of Computer Science is the organization of information for easy access or retrieval. In this chapter we will consider two problems: locating a particular piece of information within a collection—searching—and placing a collection of information into order by key—sorting. There are many different search and sort algorithms and we will only consider a few of the simplest. See Knuth[11] for a more complete discussion.

10.1 SEARCHING

If you go to the Registrar's Office to request a transcript, what has to happen in the background? Your academic information is contained in a record or records in the university's database system. To produce your transcript, the system must locate the record(s) that are pertinent. This process is called **searching**. You supply the key—your student number—and the system locates or searches for the record(s) within the file or database whose key field matches the supplied **search key**.

A search has two possible outcomes: success—where at least one record with the search key is located—or failure—where no record in the file matches the supplied key. Typically success leads to further processing such as preparing and printing the transcript, while failure leads to a message that the record cannot be found. If the key is unique only one record with that key will exist in the file and it is either found or not. If the key is not unique—for example if the search key is a course number—one of the records will be found. To retrieve all of the records—for example all students in a course—an **exhaustive search** is required.

Although we have described searching with respect to files, searching is often carried out within memory where the "records" are objects stored as elements of an array or entries in a linked structure. Searching in a file is called **external searching** since it occurs on external storage—not within main memory. Searching within memory is called **internal searching**. We will concentrate on internal searching however much of the discussion applies to external searching as well. There are other considerations on external media where access to a record is very slow compared to operations in memory. Discussion of these is beyond the scope of this book.

---

[11] Knuth, D. E; *Searching & Sorting: The Art of Computer Programming*; vol. 3; Addison-Wesley, Reading, MA; 1973.

There are many searching algorithms. To compare them we need to choose a significant step in the algorithms to count. Since all algorithms will have to compare the search key with the key in an item, this is what we will count. A key comparison is called a **probe** in a searching algorithm.

Simple searching algorithms are generally `O(n)`, however there are search algorithms which provide better performance (`O(lg n)`). It can be shown that no algorithm using a straight key comparison method can do better than `O(lg n)`. Hashing—which is a key transformation method—can get better results, however it is beyond the scope of this book.

All key comparison search algorithms have the same basic algorithm (see Figure 10.1). Since there are two possible outcomes, the loop must have two exit points, one for success and one for failure. In languages that allow only single exits from a loop, the two exit conditions can be combined and a test for successful completion made after the loop. Since the algorithm makes a number of probes at different locations, one step involves determining the point of the next probe. When there are no more locations to probe, the search has failed. If the probe is successful the search has succeeded, otherwise the process continues.

```
initialize for first probe
loop
if no more choices then not found exit
if search key matches key on item then found exit
      determine point of next probe
end
```

Figure 10.1 General Algorithm for Key Comparison Search

We will consider two search algorithms: sequential search and binary search. We will discuss the algorithms assuming the records (objects) being searched for are in an array. In Section 10.3 we will briefly discuss the situation for linked structures.

SEQUENTIAL SEARCH

The most obvious search strategy is to start at the beginning of the collection of items and look at them one at a time. This is a traversal with an early exit when found. We have described a related process in a linked structure when we deleted an element by key (Figure 3.18) and using a `List` when in the return a rental car (Case Study in Chapter 9). If the search is not exhaustive the algorithm terminates when it finds the first match. If it is exhaustive it continues to the end of the collection. In any event, if it reaches the end of the collection without finding a match it terminates as not found. The algorithm for the non-exhaustive sequential search is shown in Figure 10.2.

```
start at first item
loop
if no more items then not found exit
if search key matches key on item then found exit
      advance to next item
end
```

Figure 10.2 Sequential Search Algorithm

PERFORMANCE In determining the order of this algorithm we have a problem. The number of times through the loop is dependent on where the item we are searching for is located. If it is the first item the number of probes is `1` (`O(1)`). If it is the last item the number of probes is `n` (`O(n)`). To get a general representation of the order we use the average case. The average number of probes can be determined by assuming that we might search for any of the items with equal probability. If the item is first the number of probes is `1`. If it is second, the number is `2`, etc. The total number of probes is `1+2+...(n-1)+n` or `n(n+1)/2` and the average is `n(n+1)/2n` or `(n+1)/2`. This means the average number of probes is `O(n)`. To completely characterize the algorithm, we can indicate the minimum, maximum and average number of probes as shown in Table 10.1.

best case	O(1)
worst case	O(n)
average	O(n)

Table 10.1 Order of Sequential Search

IMPLEMENTATION Figure 10.3 is an excerpt from a simplified program to list the record(s) of selected students in a course. It assumes a `Student` class with accessor methods for student number, name and final grade. The user enters the student's student number—the search key—the program locates the student's record (`Student` object) and produces the listing of the record. The student records are read into the right-sized array `theClass`—an instance variable. Once the data has been loaded, the program repeatedly prompts for a student number and then uses the method `search` to locate the record within the array that matches the student number provided. If the search is successful, it lists the student's record. If the record is not found—indicated by a return value of `null`—it generates a message.

```
76      /** This method locates the student with a particular student number within
77        * the class array using sequential search.
78        * @param  stNum  the student number to search for
79        * @return Student  the student (null if not found).                    */
80      private Student search ( String stNum ) {
81          Student  result;
82          result = null;
83          for ( int i=0 ; i<theClass.length ; i++ ) {
84          if ( theClass[i].getStNum().equals(stNum) )
85                                                  { result = theClass[i]; break; };
86          };
87          return result;
88      };  // search
```

Figure 10.3 Example—Sequential Search in an Array

The method is passed the search key—the student number. `result` is set to not found (`null`) by default. Starting at the beginning of the array (`i=0`), if the index is beyond the last element (`i>=theClass.length`) there are no more elements and the search fails. Otherwise it compares the key on the record to the search key. If they are equal the search succeeds and `result` is set to the appropriate record. Finally, the search moves on to the next element (`i++`). When the loop terminates, either successfully or unsuccessfully, the method returns `result`.

Note that the search algorithm may be expressed as a method in a variety of ways. In fact, it need not even be a method but rather simply code within some other method. If a variable-sized array is used the number of items would be the bound on the loop. The method might return the index of the record rather than the record itself, or even simply return a boolean indicating whether the search was successful if it is only important to determine the presence of the item.

BINARY SEARCH

How would you search for a phone number in a telephone directory? Certainly you wouldn't start with the first name and read them in order until you found the one for which you were looking. This would be a sequential search and would take far too long! Rather you would open the telephone book at a point where you expected the name to be and then search from there. This works because you know the names are in alphabetical order in the phone book.

The binary search algorithm is essentially this method. If we know that the items are in order by key but have no idea how the keys are distributed—for example, we don't know that student number 283476 is likely about ¼ of the way through the collection—then the best place to look first is the middle. If the middle item isn't the one we are looking for we compare the search key with the key in the item. If the search key is greater we know the matching item must be in the upper half of the collection since the records are in order. If not it must be in the lower half. The next step is to restrict the search to the part of the collection that is likely to contain the item, determine the middle point again and probe this location. The binary search algorithm is shown in Figure 10.4.

```
start with bounds being first and last items;
loop
if bounds empty then not found exit
      select middle item between bounds
if search key matches key on item then found exit
      if search key greater than key on item then
            set lower bound to next item
      else
            set upper bound to prior item
end
```

Figure 10.4 Binary Search Algorithm

Figure 10.5 shows the situation at a particular step in the algorithm. The current upper and lower bounds bound the item matching the search key, if it exists. The probe is at the middle location between the bounds. In the diagram it is assumed that the search key is less than the key at the point of the probe, so the upper bound is reset to the point prior to the probe. Since the search key was less than the key at the probe, the new bounds will bound the matching record if the old bounds did.

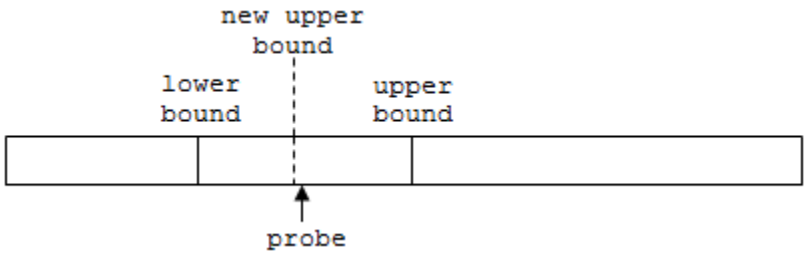


Figure 10.5 Probe in Binary Search

At some point either the item with the matching key will be located by being the item probed, or the bounds will continue to be reduced until they are empty and the algorithm will terminate unsuccessfully.

PERFORMANCE We have assumed that this method is more efficient than a sequential search, but what is its order? What we need to do is count the number of probes. As in the sequential search the number of probes depends on when the matching item is found. If the item is in the middle the number of probes is `1` (`O(1)`). What are the worst and average cases? To determine the worst case consider that the search must continually fail. The question is then how often can it fail? Each time a probe is done about half the collection is eliminated. Say there were 31 items in the collection. After the first probe there would be 15 left in consideration. After the second probe there would be 7. After the third, 3. After the fourth, 1. The fifth probe would definitely be the last. It takes at most 5 probes to search a table of 31 entries! The number of times a number can be divided in half is essentially $\log_2 n$ ($\log_2 31 = 4.95$). Thus the worst case number of probes for binary search is `O(lg n)`. Analysis of the average case is a bit more complex and we won't do it here. However the average case number of probes is also `O(lg n)`. This is summarized in Table 10.2.

minimum	O(1)
maximum	O(lg n)
average	O(lg n)

Table 10.2 Order of Binary Search

IMPLEMENTATION The method `search` in Figure 10.6 shows an implementation of binary search in the same problem as for sequential search above. `result` is initialized to not found (`null`) as default. The bounds of the search are represented by `lo` and `hi` and are initialized to the first and last items, respectively. As long as `lo<=hi` there are more items to be considered. Otherwise the loop terminates and the search is unsuccessful. The probe is made at the mid-point (`pos`) between `lo` and `hi`. It the keys match the item is found, `result` is set to the item and the loop terminates. If the item is not found `lo` or `hi` is adjusted appropriately. As for the sequential search, the algorithm

may be expressed in a variety of ways depending on whether it is a method, whether a variable-sized array is used and depending on the desired method result.

```
91      /** This method locates the student with a particular student number within
92        * the class array using binary search.
93        * @param  stNum  the student number to search for
94        * @return Student  the student (null if not found).                    */
95      private Student search ( String stNum ) {
96          Student  result;  // Widget found
97          int      lo;      // low end of bound
98          int      hi;      // high end of bound
99          int      pos;     // probe position
100         result = null;
101         lo = 0;
102         hi = theClass.length - 1;
103         while ( lo <= hi ) {
104             pos = (lo + hi) / 2;
105         if ( stNum.compareTo(theClass[pos].getStNum()) == 0 )
106                                                 { result = theClass[pos]; break; };
107             if ( stNum.compareTo(theClass[pos].getStNum()) > 0 ) {
108                 lo = pos + 1;
109             }
110             else {
111                 hi = pos - 1;
112             };
113         };
114         return result;
115     }; // search
```

Figure 10.6 Example—Binary Search in an Array

COMPARISON OF SEARCH ALGORITHMS

Our analysis has shown that the order of the sequential search is on average `O(n)` and that the binary search is on average `O(lg n)`. The binary search is thus generally more efficient. However, there are situations where sequential search would still be used. Binary search requires the items to be in key order. If they are not a sorting algorithm (see Section 10.2) would be required to put them into key order. Since sorting is much more expensive than searching, unless we were doing a very large number of searches it wouldn't be economical to sort first. Secondly, binary search requires random access to the items since the probes jump around. This is fine in an array however in a sequential file or a linear linked structure (Chapter 3), only sequential access is possible. Finally, the order is only relevant for large `n` (see Section 2.1). In the case of searching, with `n<=32` sequential search is usually more efficient even though it involves a few more probes.

To give credence to the analysis of the two search algorithms, a program was written to time their performance. An ordered table of 50,000 items was created and then 150,000 searches for random values within the table were preformed. Table 10.3 summarizes the results.

	Order	Time (ms)
sequential search	O(n)	9,735
binary search	O(lg n)	31

Table 10.3 Comparison of Search Algorithms

10.2 SORTING

Report generation is a very common operation in computer systems. Reports are summaries of information about a group of entities intended to be read by people. When the report consists of a large number of entries, it must be organized to make its use more convenient. This usually means sorting the entries into some order. For example when an instructor receives a class list from the Registrar's Office, he expects it in either student number or name order. When a CEO of a multinational company receives a quarterly earnings report for the company, she probably expects it ordered by department within division within country. Often the information in the report has been collected from a number of files and doesn't naturally occur in order by the desired key—or keys as in the earnings report—so the entries must be sorted.

Sorting is the process of taking a collection of items in arbitrary order and placing them into either ascending or descending order by some field called the **sort key**. This doesn't have to be the primary key of the item, it could be any field on which sorting is desired. The sort key doesn't even have to be unique. After sorting, items with duplicate keys will occur together. Like searching, sorting may be performed within memory—**internal sorting**—or on auxiliary storage where the items are records in a file—**external sorting**. Sorting involves considerable movement of information—after all it is intended to reorder the items. Algorithms for external sorting must try to minimize the very expensive data movement, while this is less true of internal sorting. As for searching, we will consider only internal sorting.

When the items are in an array there are two possible approaches to sorting. In the first approach, a new collection can be produced from the old one by moving the items from the old into the new in some order. In the second approach, the items are reorganized within the same collection. The first requires two separate collections—the old and the new—and hence has a requirement of $O(n)$ additional storage. The latter approach is called ***in situ*** (in place) sorting and involves only a fixed amount ($O(1)$) of additional storage. Most sorting algorithms can be written as *in situ* and we will concentrate on these. As we will see, if the items are stored in a linked structure rearrangement of items only requires changing links rather than actually copying the element so the issue of a sort being *in situ* is moot.

Often a report must be sorted on more than one key—for example sorted by department within division within country. When a sorting algorithm rearranges items so that items with the same key retain their original relative ordering, the sort is said to be **stable**. In a stable sort if item I_l and item I_r have the same value for the sort key and item I_l occurs prior to item I_r in the original ordering,

then item I_l will occur prior to item I_r in the sorted ordering. With a stable sort a collection sorted on one key—for example department—can be sorted on a second key—for example division—to yield a collection sorted on the first key (department) within the second (division). In general, it is preferable to use a stable version of a sorting algorithm.

When comparing internal sorting algorithms the step usually counted is a **key comparison**—that is the comparison of the key of one item with the key of another item. By this measure the simple sort algorithms exhibit $O(n^2)$ performance. It can be shown that key comparison sorting cannot perform any better than $O(n \lg n)$. We will examine three $O(n^2)$ sorts and discuss one $O(n \lg n)$ sort. Again more complete coverage is available elsewhere (such as Knuth[12]).

The simple sort algorithms all use the same basic approach. The algorithm makes a number of passes over the collection. Each pass involves comparison of the keys of an average of $O(n)$ items. Between the passes the collection can be viewed as having two parts: sorted and unsorted (Figure 10.7). The result of the pass is to move one item from the unsorted collection into the sorted collection. Thus it takes $O(n)$ passes to complete the sort and $O(n^2)$ key comparisons in total.

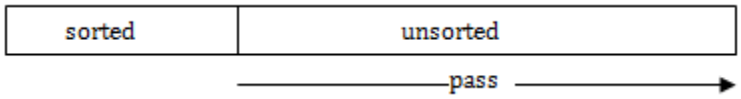


Figure 10.7 Pass in Sorting

The faster sorts use a different approach called divide-and-conquer. They divide the problem into two parts, sort each part and combine the results. This gives $O(n \lg n)$ performance since they can repeatedly divide the problem in half $O(\lg n)$ times and the reorganization for each division takes $O(n)$. Note that the binary search discussed in Section 10.1 also used a divide-and-conquer approach.

SELECTION SORT

The principle of selection sort is to, on each pass, select the minimum (or maximum) value from the unsorted portion and transfer it to the sorted portion. This is much the same as you might sort playing cards in a hand of bridge or euchre. Figure 10.8 shows a selection sort into ascending order in action. The figure has been simplified by representing each item solely by its key—here an integer. The bold line indicates the division between the sorted (left) and unsorted (right) parts and the highlighted item indicates the minimum key in the unsorted part. The item with the minimum key is exchanged with the item at the front of the unsorted part, essentially adding it to the end of the sorted part.

Note that the last pass is unnecessary—there is only one item left in the unsorted part. This item is greater than all others; otherwise it would have been selected earlier. It already follows the sorted part and therefore it need not be moved.

[12] Knuth, D. E; *Searching & Sorting: The Art of Computer Programming*; vol. 3; Addison-Wesley, Reading, MA; 1973.

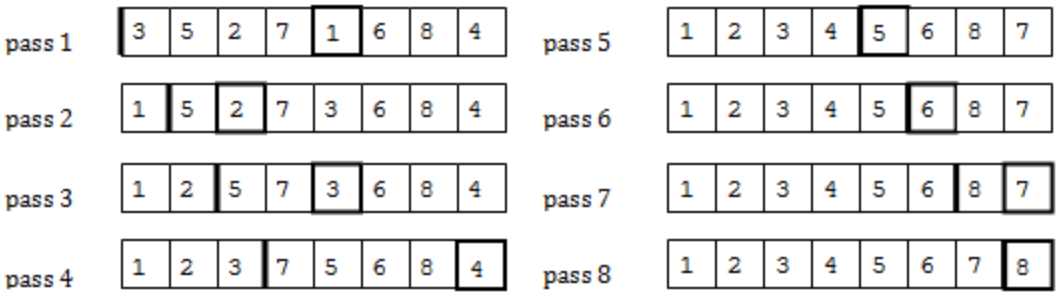


Figure 10.8 Selection Sort

This is stable as long as we always select the leftmost minimum value—the first one encountered if going left-to-right. The minimum value is moved to the left bypassing only values that are greater. Thus it never moves from the right of an equal value to the left of it and the relative orders are preserved.

PERFORMANCE The algorithm is clearly $O(n^2)$. Each pass places one item in order so it takes $O(n)$ (actually $n-1$) passes. On the first pass there are $n-1$ comparisons, on the second, $n-2$ and so on. There are on average $O(n)$ comparisons per pass.

IMPLEMENTATION The example in Figure 10.9 shows the selection sort in an actual application. The program produces a list of students in a course sorted in descending order by final grade. The items are the same `Student` objects described in Section 10.1. After the objects are loaded into a right-sized array (`theClass`, an instance variable) the array is sorted by final grade and the result displayed.

```
68      /** This method sorts the class in descending order by final mark using
69        * selection sort.                                                    */
70      private void sort ( ) {
71          int      largePos;  // position of largest item
72          Student  temp;
73          for ( int i=0 ; i<theClass.length-1 ; i++ ) {
74              largePos = i;
75              for ( int j=i+1 ; j<theClass.length ; j++ ) {
76                  if ( theClass[j].getFinalGrade()
77                              > (theClass[largePos].getFinalGrade())) {
78                      largePos = j;
79                  };
80              };
81              temp = theClass[i];
82              theClass[i] = theClass[largePos];
83              theClass[largePos] = temp;
84          };
85      };  // sort
```

Figure 10.9 Example—Selection Sort in an Array

The method `sort` implements a selection sort into descending order. On each pass (outer loop) the location (`largePos`) of the student with the highest grade in the unsorted portion (positions `i...n`) is

determined. This is done (inner loop) by comparing the largest so far (`theClass[largePos]`) with each of the others (`list[j]`) in turn (`j=i+1…n`), and modifying `largePos` when necessary. When the largest has been located it is exchanged with the item at the front (`list[i]`) of the unsorted portion. This increases the size of the sorted portion (`i++`). This continues `n-1` times until the list is sorted.

As in searching the algorithm need not be implemented as a method and the items could be stored in a linked structure. The algorithm is generalized as in Figure 10.10. For ascending order the test in the fourth line would be less than and likely we'd call the position `smallPos` rather than `largePos`.

```
for n-1 passes
    set largePos to reference first item in unsorted portion
    for remaining items in unsorted portion
        if key of item is greater than key of largePos then
            set largePos to reference this item
    end
    exchange item at largePos with item at front of unsorted
        portion
end
```

Figure 10.10 Selection Sort Algorithm

INSERTION SORT

Insertion sort is similar to selection sort in that it places one item into position at a time. It takes items one at a time from the unsorted part and places them into their correct position relative to the other items in the sorted part. Figure 10.11 shows the insertion sort in action. The bold line indicates the division between the unsorted (left) and sorted (right) portions. The item to the immediate left of the line is the item to be inserted and the arrow shows the insertion point. The item is inserted into the desired position by moving the items in front of the point to the left and placing it in the hole created. After pass eight, when `3` is moved, the list is sorted.

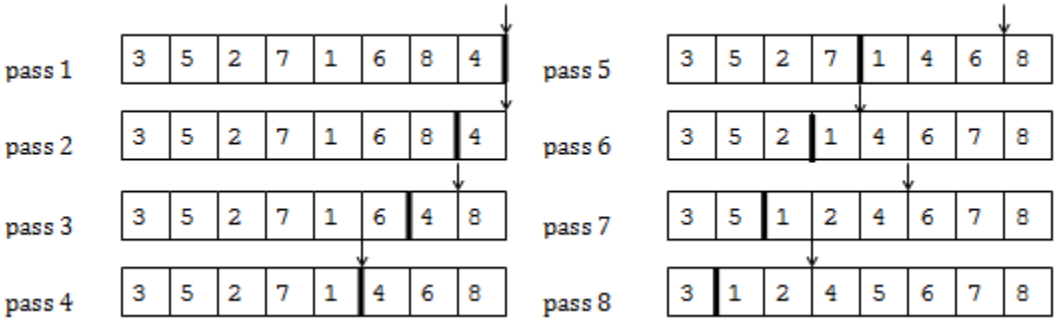


Figure 10.11 Insertion Sort

Note that the first pass is unnecessary. A single item is always sorted so the sorted part can start with one item. When the items are chosen for insertion from right-to-left, if the inserted item is placed to

the left (in front) of an item with equal key the sort is stable. Items with equal keys will still be in the same relative order.

PERFORMANCE The algorithm is $O(n^2)$. Since each pass puts one item into place, $O(n)$ (actually $n-1$) passes are required. Each pass takes on average $O(n)$ comparisons to find the point for the insertion. The insertion point could be at the front of the sorted part, the end, or anywhere in between. The average would be half way.

IMPLEMENTATION Figure 10.12 shows an implementation of the insertion sort in the same application as discussed for Figure 10.9. The outer loop controls the `n-1` passes decrementing `i` from `theClass.length-2` to `0`. `i` represents the position of the rightmost unsorted item—the one to be inserted. The insertion point is found and the item at `i` is inserted into its correct position. To do this, the items to the left of the insertion point will have to be moved to the left. It is most efficient to move the items at the same time as locating the insertion point. This is done in the inner loop. First the item at `i` is saved (`temp`). Then as long as the end of the list has not been reached and the next item (`theClass[j]`) has a key greater than item `theClass[i]` (in `temp`), the item at `j` is moved one position to the left. Note that an exchange is not necessary since the item at `j-1` was either the original one which was saved in `temp`, or it was moved on the previous iteration. When the insertion point or the end of the list is reached, insertion occurs in front of position `j`. That is, the original item (in `temp`) is inserted into `theClass[j-1]`. The item at `j-1` was either moved on the previous iteration or was already `temp`.

```
88      /** This method sorts the class in descending order by final mark using
89        * insertion sort.                                                    */
90      private void sort ( ) {
91          Student  temp;
92          int      j;
93          for ( int i=theClass.length-2 ; i>=0 ; i-- ) {
94              temp = theClass[i];
95              j = i + 1;
96              while ( j < theClass.length
97                      && theClass[j].getFinalGrade() > temp.getFinalGrade() ) {
98                  theClass[j-1] = theClass[j];
99                  j = j + 1;
100             };
101             theClass[j-1] = temp;
102         };
103     }; // sort
```

Figure 10.12 Example—Insertion Sort in an Array

As before, the algorithm may be expressed in a variety of ways. The general algorithm is shown in Figure 10.13.

```
for n-1 passes
    copy of the rightmost item in the unsorted portion to temp
    sequencing through the sorted portion from left-to-right
        while the key of the item is less than the key of temp
            shift the item to the left
    end
    copy temp into the last vacated position
end
```

Figure 20.13 Insertion Sort Algorithm

EXCHANGE SORT

The principle of exchange sort is a bit less obvious. Instead of selecting a specific item it simply looks at pairs of consecutive items in the collection—the 1st and 2nd, 2nd and 3rd, and so on. It reorders the items in the pair if they are out of order with respect to each other. In a sorted collection, no pair of items can be out of order, but how long does the exchange of consecutive items have to proceed to ensure the collection is sorted?

Figure 10.14 shows the exchange sort in action. All the steps of each pass are enumerated. In each step, the pair of items compared is highlighted. Notice what happens. Item `5` moves right until it encounters an equal or larger item (`7`) and then stops. Now item `7` moves right until it encounters an equal or larger item (`8`) and then it stops. Finally item `8` moves right until it reaches the end of the collection. After one pass the largest item (`8`) has been moved into position. This must be so since, in any comparison, it would have been bigger than the item to its right and would be exchanged.

On the second pass we don't have to go any further than the second last item since we know the largest is in the final position. After the second pass the second largest item (`7`) is in position. This behavior of the larger items moving towards the right until they encounter a still larger item is similar to bubbles rising in a carbonated beverage. This gives the exchange sort its other name: **bubble sort**.

PERFORMANCE Clearly we can guarantee that the collection is sorted in `n-1` passes. A last pass over one item is unnecessary since there is nothing to exchange with. So the number of passes is `O(n)`. On each pass, the consecutive items are compared. With `n` items there are `n-1` pairs to be compared. On the first pass this is `n-1`, on the second, `n-2`, etc. Thus the average number of comparisons per pass is `O(n)` and the algorithm is $O(n^2)$. Since the exchanges occur between pairs of items, as long as equal items in the pair are not exchanged, the algorithm is stable.

In the example in Figure 10.14 the collection is actually sorted after pass 4, although we cannot guarantee that it is sorted until all passes are complete. We did note earlier that if the collection is sorted there cannot be any pair-wise exchanges. Conversely, if there are no exchanges the collection must be sorted. If on any pass we note that there are no exchanges, the algorithm can terminate. This version of the algorithm still has $O(n^2)$ behavior on average However, if the collection is already in sorted order it is `O(n)` since it will find that the collection is sorted after one pass with no exchanges. The other sort algorithms do not have this property.

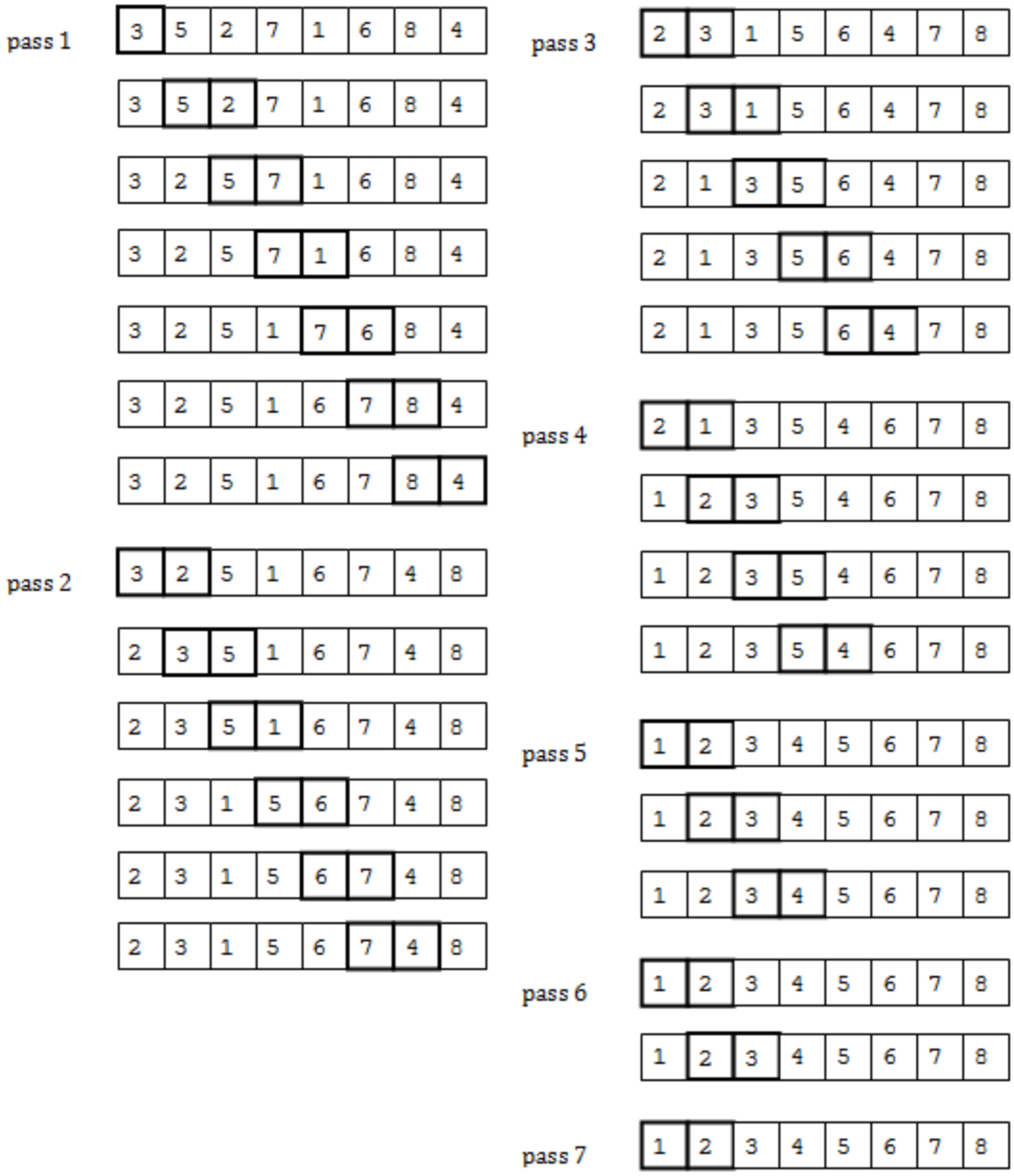


Figure 10.14 Exchange Sort

IMPLEMENTATION Figure 10.15 shows the exchange sort as in the previous sections. The outer loop controls the `n-1` passes decrementing `i` from `nStd-1` to `1`. `i` represents the position of the rightmost item to be compared. The inner loop compares consecutive pairs (`list[j]` and `list[j+1]`) up to `i`. When a pair is out of order, the items are exchanged. Code for premature exit if the list is sorted would involve a boolean variable indicating if an exchange occurs, which would be reset before each pass and set on an exchange. The variable could be tested before each pass, to determine if another pass is necessary.

```
106     /** This method sorts the class in descending order by final mark using
107       * exchange sort.                                                      */
108     private void sort ( ) {
109         Student  temp;
110         for ( int i=theClass.length-1 ; i>=1 ; i-- ) {
111             for ( int j=0 ; j<i ; j++ ) {
112                 if ( theClass[j+1].getFinalGrade() > theClass[j].getFinalGrade() ) {
113                     temp = theClass[j];
114                     theClass[j] = theClass[j+1];
115                     theClass[j+1] = temp;
116                 };
117             };
118         };
119     }; // sort
```

Figure 10.15 Example—Exchange Sort in an Array

Once again, the algorithm can be expressed in a variety of ways, as a method or in-line code. The exchange sort algorithm is expressed as a programming pattern in Figure 10.16.

```
for n-1 passes
    for each consecutive pair in unsorted part
        if key of first item is less than key of second item then
            exchange items
    end
end
```

Figure 10.16 Exchange Sort Algorithm

MERGE SORT

Merge sort applies the divide-and-conquer approach to sorting. The principle is that it is easier to sort a small collection than a big one. If the collection can be partitioned into two (or more) smaller collections that are each then sorted, they can be recombined producing a sorted result. Of course, we still have a sorting step which was our original problem. We can use the same technique to sort the (smaller) partitions. This leads to a recursive solution. Each recursive execution partitions the items into smaller partitions. This can only happen $O(\lg n)$ times—the number of times you can divide a set of n items into m parts is proportional to $\lg_m n$. If both the partitioning step and the recombination step are no worse than $O(n)$, the result will be $O(n \lg n)$. Figure 10.17 shows the general algorithm for a partitioning sort.

```
sort (items):
    partition items into m partitions p1…pm
    for each partition pi
        sort(pi)
    end
    recombine p1…pm into one collection
```

Figure 10.17 General Algorithm for Partitioning Sort

In merge sort the partitioning simply takes the first half of the items as partition one and the second half as partition 2. It then sorts the two partitions. Finally it recombines the two partitions by merging them similar to the algorithm for file merge.

Figure 10.18 shows the merge sort in operation. On each pass, the first half of the items becomes the first partition and the second half, the second. The partitioning is shown in the top half of the diagram by the lines connecting the collections. Each of the partitions is then sorted recursively resulting in smaller and smaller partitions until the partitions are trivial—the trivial case is a partition of 0 or 1 item, which is sorted by definition. The sorted partitions are them recombined via merging as shown in the bottom half of the diagram with the lines connecting merged collections.

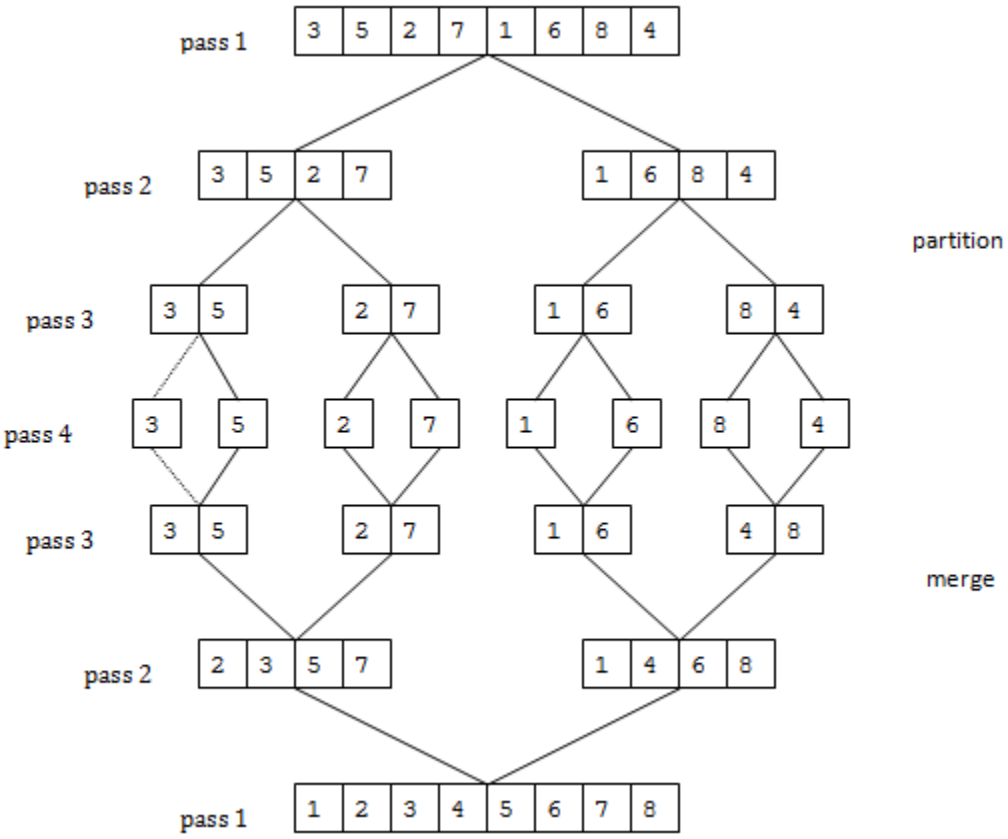


Figure 10.18 Merge Sort

PERFORMANCE Since we are dividing the collection in half at the partitioning step for each pass, there are `O(lg n)` passes. Each pass consists of a partition step of `O(n)` if we simply copy the items into the two partitions. The merge step is also `O(n)` since we would repeatedly select the first (smallest) item from one of the two partitions to copy to the sorted collection and there are `n` items in total. Since steps of `O(n)` are repeated `O(lg n)` times, the result is `O(n lg n)`.

```
122     /** This method sorts the class in descending order by final mark using merge
123       * sort. It is actually a wrapper method for the recursive mergeSort method.*/
124     private void sort ( ) {
125         mergeSort(theClass);
126     }; // sort
127
128     /** This method sorts the students in the class between the specified positions
129       * into descending order by final mark using quicksort. It is usually used by
130       * calling the wrapper method sort.                                          */
131     private void mergeSort ( Student[] toSort ) {
132         Student[]  left;   // left partition
133         Student[]  right;  // right partition
134         int        lp;     // position in left on merge
135         int        rp;     // position in right on merge
136         if ( toSort.length > 1 ) {
137             /* partition */
138             left = new Student[toSort.length/2];
139             right = new Student[toSort.length-left.length];
140             for ( int i=0 ; i<left.length ; i++ ) {
141                 left[i] = toSort[i];
142             };
143             for ( int i=0 ; i<right.length ; i++ ) {
144                 right[i] = toSort[left.length+i];
145             };
146             /* sort */
147             mergeSort(left);
148             mergeSort(right);
149             /* merge */
150             lp = 0;
151             rp = 0;
152             for ( int i=0 ; i<toSort.length ; i++ ) {
153                 if ( rp >= right.length ) {
154                     toSort[i] = left[lp];
155                     lp = lp + 1;
156                 }
157                 else if ( lp >= left.length ) {
158                     toSort[i] = right[rp];
159                     rp = rp + 1;
160                 }
161                 else if ( left[lp].getFinalGrade()
162                             >= right[rp].getFinalGrade() ) {
163                     toSort[i] = left[lp];
164                     lp = lp + 1;
165                 }
166                 else {
167                     toSort[i] = right[rp];
168                     rp = rp + 1;
169                 };
170             };
171         };
172     }; // mergeSort
```

Figure 10.19 Example—Merge Sort in an Array

IMPLEMENTATION Figure 10.19 shows the merge sort algorithm for the same problem as above. The method `sort` is a wrapper method that wraps the recursive method `mergeSort`, providing the common method header that was used for the previous sorting methods (see, for example, Figure 10.15). This is necessary if we wish to be able to substitute merge sort for another sorting method within an existing program since the recursive method requires different parameters. The method `mergeSort` is the actual recursive sorting method. As implemented in an array, `mergeSort` is stable but not *in situ* since it copies the partitions both at the partitioning and the merging steps requiring `O(n)` extra storage.

COMPARISON OF SORT ALGORITHMS

Selection, insertion and exchange sorts all are $O(n^2)$ while merge sort is $O(n \lg n)$. For smaller n, it is better to choose one of the simpler $O(n^2)$ algorithms. For example, insertion sort is effective for $n \leq 25$. For large n however merge sort (or one of the other $O(n \lg n)$ algorithms) would be preferable. In an array, merge sort is not *in situ* while there are other $O(n \lg n)$ algorithms that are *in situ*.

In choosing between the $O(n^2)$ algorithms—especially in languages where the complete item must be moved about in memory as opposed to Java where only references are moved—the number of exchanges of items is a consideration. Selection sort requires only one exchange per pass ($O(1)$) while insertion sort and exchange sort each require $O(n)$ exchanges per pass. Finally, if the original collection might be sorted—or nearly so—exchange sort with the described improvement, would be the best choice as it is $O(n)$ for a sorted list.

A timing test was done for the four sort algorithms as described using a list of 100,000 random items. The results are summarized in Table 10.4.

	In-situ (as implemented)	Stable (as implemented)	Order	Time (ms)
selection sort	yes	yes	$O(n^2)$	11,997
insertion sort	yes	yes	$O(n^2)$	5,304
exchange sort	yes	yes	$O(n^2)$	25,272
merge sort	no	yes	$O(n \lg n)$	32

Table 10.4 Comparison of Sorting Algorithms

10.3 SEARCHING AND SORTING IN SEQUENTIALLY-LINKED STRUCTURES

In the previous sections searching and sorting have been demonstrated assuming the items being processed are in an array. In Chapter 3 we discussed the relative advantages of arrays and linked structures in manipulating collections of information. Arrays have the advantage of random access at the disadvantage of more expensive reordering of items. Let us briefly consider searching and sorting in sequentially linked structures.

Both arrays and sequentially linked structures can be traversed sequentially so a sequential search—which is essentially a traversal with an early exit—is easily implemented in either data structure. Figure 10.20 shows a sequential search in a sequentially-linked structure for the same problem as discussed in Section 10.1.

```
83      /** This method locates the student with a particular student number within
84        * the class list using sequential search.
85        * @param  stNum  the student number to search for
86        * @return Student  the student (null if not found).                    */
87      private Student search ( String stNum ) {
88          Student  result;
89          Node     p;
90          result = null;
91          p = theClass;
92          while ( p != null ) {
93          if ( p.item.getStNum().equals(stNum) ) { result = p.item; break; };
94              p = p.next;
95          };
96          return result;
97      };  // search
```

Figure 10.20 Example—Sequential Search in a Sequentially-linked Structure

A binary search requires access to an arbitrary item—the mid-point of the collection— which is not efficient in a sequential structure so binary search is not implemented for a sequentially-linked structure. There is a linked structure called a binary search tree that maintains the advantages of a linked structure while exhibiting `O(lg n)` search times, but this is beyond the scope of this text.

The sort algorithms discussed only require sequential access—we generally process the collection in order. One of the primary operations in sorting is reordering of the items. This can generally be more efficient in sequentially-linked structures than in arrays. However this does not change the number of key comparisons (the significant step) and so the orders of the algorithms remain the same.

Figure 10.21 shows an exchange sort in a sequentially-linked structure for use in the same problem as discussed in Section 10.2. A header node is used to simplify operations at the front of the structure. The method makes `n-1` passes over the collection (lines 150-162). On each pass it considers pairs of consecutive items referenced by `p` and `p.next`. It reorders the items if they are out of order by changing the links (lines 154-158).

Implementation of other sorts in a sequentially-linked structure is left as an exercise.

```
145     /** This method sorts the class in descending order by final mark using
146       * exchange sort.                                                      */
147     private void sort ( ) {
148         Node  p;
149         Node  q;
150         for ( int i=1 ; i<numStd ; i++ ) {
151             q = theClass;
152             p = theClass.next;
153             while ( p.next != null ) {
154                 if ( p.next.item.getFinalGrade() > p.item.getFinalGrade() ) {
155                     q.next = p.next;
156                     p.next = p.next.next;
157                     q.next.next = p;
158                 };
159                 q = q.next;
160                 p = q.next;
161             };
162         };
163     }; // sort
```

Figure 10.21 Example—Exchange Sort in a Sequentially-Linked Structure.

SUMMARY

Searching is locating an item(s) that matches a particular criterion—typically the value of some field of the item called the search key—from within a collection of items. A simple search locates one such item while an exhaustive search locates all such items. A search is said to succeed if it locates an item matching the criterion or fail if it determines there is no item matching the criterion.

The most straightforward search is the sequential search—which examines each item in turn. Its average performance is $O(n)$. Sequential search is possible whenever sequential access is possible, and is the primary search for sequential data structures such as linear linked structures and sequential files.

If the items are sorted by search key and are accessible in random order—such as in an array—a binary search can be used. The binary search examines the middle item, determines which half of the collection contains the desired item, and then searches that half. This gives average performance of $O(\lg n)$. Sequential search may still be preferred if n is small—32 or less. It can be shown that no key-comparison search can be better than $O(\lg n)$.

Sorting is the process of ordering items in a collection into ascending or descending order by some field value, called the sort key. Most sorts can be implemented as *in situ* sorts where the sorting occurs within the original collection and the amount of additional memory required is $O(1)$. A stable sort is one in which, given two items with equal sort keys, the item occurring to the left before the sort will still occur to the left after the sort. With a stable sort it is possible to produce an ordering sorted on multiple keys by sorting on each key in turn.

Three simple sorts are selection sort, insertion sort, and exchange sort. All of these sorts are $O(n^2)$ since they take $O(n)$ passes over the data each time moving one item into place using $O(n)$ steps. In selection sort the smallest (largest) item is found and placed into its appropriate position. In

insertion sort each item is considered in turn, moving it to its correct place relative to the items processed so far. In exchange sort pairs of items are exchanged if they are out of order.

Sorting in `O(n lg n)` time can be achieved by using a divide-and-conquer approach. For example, in merge sort the items in the collection are partitioned into two sets each consisting of half the items. The two partitions are then sorted separately and merged using a sequential merge algorithm. Each time the items are partitioned `O(n)` steps are required and the number of partitionings is `O(lg n)`. Thus merge sort is `O(n lg n)` on average. There are other `O(n lg n)` sorts such as quicksort and heapsort. However it can be shown that no key comparison sort can perform better than `O(n lg n)`.

REVIEW QUESTIONS

1. T F The search key is the primary key for the items being searched.

2. T F Sequential search can complete in `O(1)` time.

3. T F If the collection is large, it is always better to sort the collection first and then use binary search.

4. T F A stable sort is one where the minimum, maximum and average performance are all of the same order.

5. T F The primary operation in a selection sort is the moving of an item to its correct place within the sorted collection.

6. An exhaustive search

   a) takes $O(n^2)$ time
   b) finds all items matching the search key
   c) is generally too expensive
   d) none of the above

7. Binary search

   a) requires items in key order
   b) is good for very large collections
   c) is `O(lg n)` on average
   d) all of the above

8. An *in situ* sort

   a) rearranges the items within the collection
   b) requires `O(1)` additional storage
   c) is exemplified by the exchange sort
   d) all of the above

9. In exchange sort

   a) consecutive items are exchanged if out of place
   b) each pass moves the largest (smallest) key its desired place
   c) it is possible to get `O(n)` performance
   d) all of the above

10. In merge sort

    a) the partitions are sorted using selection sort
    b) the merging step is `O(n)`
    c) the trivial case for recursion is encountering a sorted collection
    d) all of the above

EXERCISES

1. Implement selection sort, insertion sort and merge sort in a sequentially-linked structure with a header similar to the implementation of exchange sort in Figure 10.21.

2. A binary search can be expressed recursively in the following way. To search a collection of items, probe in the middle position and, if the search key is less than the middle key value, search the lower half of the collection, otherwise search the upper half. In such a formulation, what are the trivial cases? What is the reduction? Does the reduction always lead to a trivial case?

   Write a method:

```
private Student binSearch ( Student[] a, int lb, int ub, int key )
```

   that searches the array `a` between indices `lb` and `ub` for a `Student` whose student number is equivalent to `key` and returns a reference to the `Student` object found, or `null` if `key` was not found. Write a wrapper method that wraps the `binSearch` method above to make it standard with other search methods such as in Figure 10.6.

3. Write a modified version of the exchange sort (Figure 10.15) that incorporates a flag that determines if any exchanges happened on a pass, and if not, terminates the algorithm. Does this really improve things. Write a timing program that compares the standard and modified versions of the exchange Does the modification appear to have an improvement in a random set? Now generate a partially sorted set of data. Generate a set of `n` random integers and sort them. Now partially randomize the resulting array by repeatedly choosing two random numbers between `0` and `n-1` and exchanging the values in the two chosen positions. Do this a total about `n/5` times. Run the timing tests on this array. Do the results for the modified exchange sort look more promising?

11 SOFTWARE DEVELOPMENT

CHAPTER OBJECTIVES

- Describe the phases of a software development project.
- Describe the roles of the members of a software development team.
- Apply analysis to identify the classes that make up a system.
- Use CRC cards to perform responsibility-based design.
- Develop class specifications.
- Code a class from its specification.
- Perform testing of a system with multiple classes.

Large software systems can involve hundreds or thousands of classes and are developed and maintained over many years by many people. Systems of this complexity cannot be built unless a careful and well-thought-out development plan exists. Smaller systems developed by a single developer also benefit from such a process, even if it is done informally.

In this chapter we will consider the software development process. Many methodologies for development have been proposed and used and new ones proposed and used for object-oriented development. Here we will consider the common features of these processes and go through a development exercise from start to finish. As you proceed in your career, you will study this process in more detail in a course on Software Engineering.

11.1 THE DEVELOPMENT PROCESS

Large-scale software development is a complicated exercise often involving a large staff and many person-years. For software development to succeed, it is imperative that there be some overlying structure or methodology to the process. There are many different **software development methodologies** in use; however, they all share a number of similar phases that are performed more-or-less in order. You will see much more of this in your future study of software engineering. The common phases of software development are shown in Figure 11.1.

1. analysis
2. design
3. coding
4. testing
5. debugging
6. production
7. maintenance

Figure 11.1 Phases of Software Development

Analysis is the process of determining what is actually required of the proposed software system. We say system since it may consist of a number of programs that work together. Usually when a

system is first proposed all that is available is a general statement of what is desired, sometimes called a **problem statement**. This may have been written by a customer or by someone in a non-computing division of the organization and is typically not complete, unambiguous, or necessarily even feasible! Analysis is just analysis of what is proposed, to ensure that what is to be done is well-defined and feasible. The result of analysis is a clear specification of what is to be done, often called a **requirements specification**. When the development is being done on contract for another organization, the requirements specification may be part of the legal contract.

Analysis involves interaction between the computer scientist and the expected user group. In a large software development company there are specialists—usually senior computer scientists called **software or systems analysts**—who perform this task. In smaller organizations the task may be done by people who also write program code—often called **programmer/analysts**.

Since programs are information processing systems, one of the tasks of analysis is to determine what data the system needs and what information the system is to produce—the inputs and outputs. It is necessary to determine where the input will come from and what form it is in as well as where the output will go and its format.

Another task is to develop a model of the system—called the **analysis model**. This can be based on the existing system, or could be a model of a hypothetical system. Since real-world manual systems involve cooperation between a number of people, the model should reflect this cooperation. Here is an advantage of the object-oriented approach since object-oriented programs involve interacting objects. The model will be a description of a number of entities (objects) that interact in particular ways.

Design is the step in which we come to some decisions about how the system will be implemented in a programming language. We consider the objects in the analysis model and develop descriptions of the classes that will represent them. We consider the way the system will be used and the operations the classes must perform to support these uses. We look at the collaboration between the objects and determine relationships between the classes. The result of the design stage is a **class specification** for each class of the system that completely defines its responsibilities.

Again in a large organization, design will be done by senior staff often called **system designers** or sometimes **analyst/designers**. They must know programming well as well as have design experience. Design makes or breaks a project. In smaller organizations, design is performed by **senior programmers** or **programmer/analysts**.

Analysis and design can be done in a language-independent way. In the **coding** phase, code for the system is written in a particular programming language. Each of the classes identified in the design phase is coded as a class in a target language such as Java. In a large project, there may be many programmers performing this task each working on a different class. It is important that the class specifications are clear so each can know what their class is responsible for and what they can rely on other classes for. In large systems, no single individual can comprehend the details of all parts of the system at one time. Clear specifications allow a programmer to concentrate only on the details of his/her class without having to understand the details of other classes—applying abstraction.

Once a class has been written it is necessary to determine if it lives up to its specification. This is called testing. **Testing** involves putting a class (object) through its paces to see that it works correctly in all cases. Usually the number of cases is large or even infinite, so it is not possible to do exhaustive testing. Rather representative sets of tests are used that cover the possible conditions that

may occur. **Test sets** are part of the design specification of a class and should include an indication of the expected results against which the actual results of the test are compared. Test sets and outputs should be saved for future use in the maintenance phase.

Once individual classes are tested—**unit testing**—and found working it is necessary to determine if they work with other classes—**integration testing**— ultimately testing to see if all the classes work together—**system testing**. Usually the programmer is responsible for unit testing and possibly integration testing. In a large organization, some (or all) of the testing might be done by dedicated **testers**.

Typically a class or set of classes doesn't work as required right from the start. This means that the class(es) must be debugged. **Debugging** involves determining which class or part of the class is not performing as required and changing the code to correct the problem. The classes are then tested again in unit and integration testing, until all classes and the system itself pass all tests. Sometimes the error is not in the coding of the class but in the design or in the analysis. In these cases it is necessary to return to these earlier phases and correct the problem. This can be very costly and this is why analysis and design are very important and done by the most experienced staff. Note that testing can never prove that a system works—it can only provide a high level of confidence.

Once the system has passed all tests it can be released to the actual users to use in a **production** environment. At this time, the programming staff is not involved, although **trainers** and **technical support** staff are often necessary to assist the users.

Software is seldom static. Since testing cannot prove the system works, errors are sometimes found during production. As users are using the system they see additional things that the system could do for them. The environment—operating system, hardware—in which the system is used often changes. Sometimes the task that the system is required to perform changes. All of these lead to the next phase—maintenance. **Maintenance** involves returning to earlier phases to fix bugs or enhance the system. Bug fixes usually involve returning to the coding phase and the ultimate release of a fixed version of the system. The release number is indicated by a number to the right of the decimal point—such as in PaySys v1.1. Significant enhancements or major modifications usually mean starting again at analysis. They lead to a new version of the software indicated by a new version number to the left of the decimal point—such as in PaySys v2.0.

There is one more (not insignificant) part of software development—documentation. **Documentation** consists of **technical documentation** and **user documentation**. Technical documentation includes: the requirements specification, class specifications, test specifications and so forth, as produced by the analysts and designers; class-level documentation produced by programmers; and test results documented by testers. This documentation is produced to track the project and to assist in subsequent maintenance. User documentation consists of user manuals, guides, tutorials, on-line help, and other support documentation for user groups. Often this material is prepared by **technical writers**.

In the rest of this chapter will examine the details of the software development phases by considering a system to manage student marks in a course.

11.2 ANALYSIS

Analysis begins with some statement of the problem to be solved. This statement might be some notes doodled on the back of a napkin or a large formal document prepared by an organization contracting out a software development project. Figure 11.2 shows a simple informal problem specification for a course records system.

A program is needed to manage the marks for students registered in a course. Throughout the term students receive marks in various pieces of work. At the end of the term students are awarded a final grade computed from the marks in the pieces of work according to the marking scheme. A report containing the students' student number and final grade and the course average is sent to the Registrar's Office.

Figure 11.2 Problem Statement

There are basically three parts to the analysis: refining the problem statement, determining inputs and outputs, and developing the model. Our problem statement is almost complete; however, it does not describe what a marking scheme is. The refined problem statement is shown in Figure 11.3.

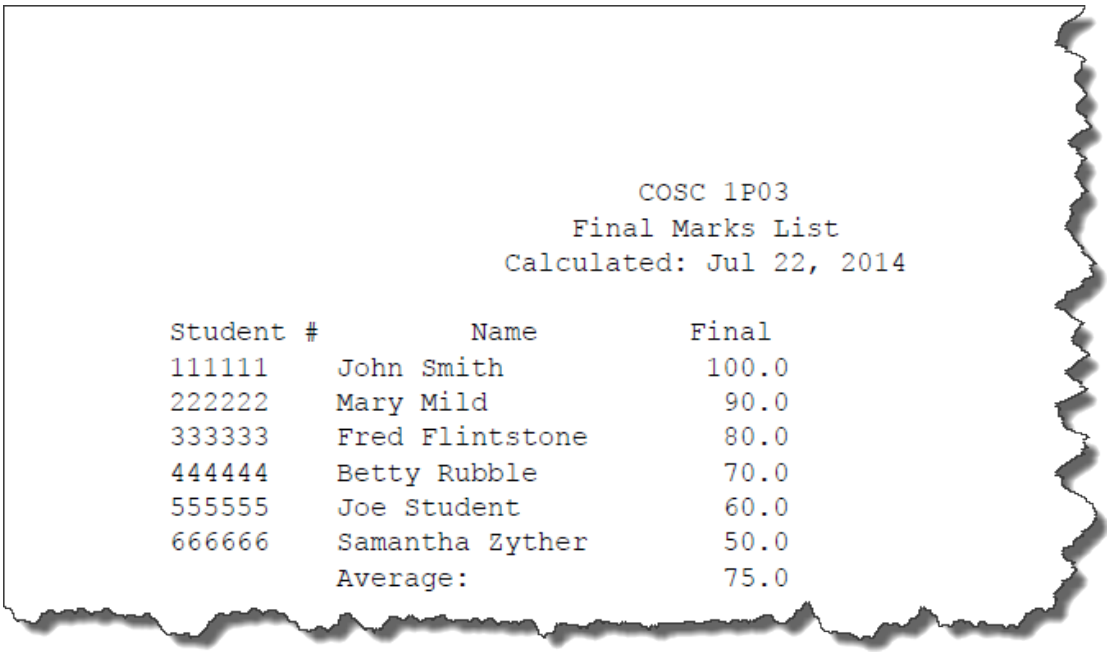
A program is needed to manage the marks for students registered in a course. Throughout the term students receive marks in various pieces of work. At the end of the term students are awarded a final grade computed from the marks in the pieces of work according to the marking scheme. A report containing the students' student number and final grade and the course average is sent to the Registrar's Office. *The marking scheme defines for each piece of work its base mark and weight towards the final mark.*

Figure 11.3 Refined Problem Statement

INPUTS & OUTPUTS

The inputs to the system must include the course name, the actual bases and weights for the pieces of work that make up the marking scheme and, for each student, the student number and marks in each piece of work. After meeting with the Registrar and instructors it is determined that the Registrar can supply a file containing the student numbers and names for each student in the course. The marking scheme must be determined at the beginning of the course and is known to the instructor and throughout the term, markers mark the pieces of work and can record the individual marks.

The outputs include the final mark report. According to the Registrar's requirements it must contain for each student, the student number and final grade as well as the course average. The format of the report is shown in Figure 11.4



```
                          COSC 1P03
                      Final Marks List
                  Calculated: Jul 22, 2014

Student #          Name          Final
111111     John Smith             100.0
222222     Mary Mild               90.0
333333     Fred Flintstone         80.0
444444     Betty Rubble            70.0
555555     Joe Student             60.0
666666     Samantha Zyther         50.0
           Average:                75.0
```

Figure 11.4 Grade Report Format

IDENTIFYING OBJECTS

To construct the model, it is necessary to determine what entities (objects) are present in the system. The easiest way to start this process is to underline all the nouns or noun phrases in the refined problem statement (Figure 11.5).

A program is needed to manage the marks for students registered in a course. Throughout the term students receive marks in various pieces of work. At the end of the term students are awarded a final grade computed from the marks in the pieces of work according to the marking scheme. A report containing the students' student number and final grade and the course average is sent to the Registrar's Office. The marking scheme defines for each piece of work its base mark and weight towards the final mark.

Figure 11.5 Selecting Candidate Objects

These nouns represent **candidate objects**. Figure 11.6 lists them with plurals and duplicates removed. The next step is to examine each candidate object to determine if it represents an actual entity. A candidate may be eliminated if it simply represents a value, if it is a synonym for another entity or if it is external to the system being developed.

program	mark	student	course
term	piece of work	final grade	marking scheme
report	student number	course average	Registrar's Office
base mark	weight		

Figure 11.6 Candidate Objects

We can eliminate mark, final grade, student number, course average, base mark and weight since they are simple values. Program, term, report and Registrar's Office are not part of the system. Program is what we are writing and produces the report as output. Term is a unit of time and the Registrar's Office is clearly outside the system. This leaves the objects listed in Figure 11.7.

student	course	piece of work	marking scheme

Figure 11.7 Identified Objects

ANALYSIS MODEL

There are a variety of notations for describing the model. We will use a simplified notation here. Essentially we want to show the relationships between the objects. In the diagram (Figure 11.8) the rounded rectangles identify the objects and the lines indicate relationships. The labels on the lines describe the relationships. The ranges on a line indicate the number of entities at that end that are associated with each entity at the other end. For example, there are zero or more students in each course. Where no ranges are given, it is a one-to-one association.

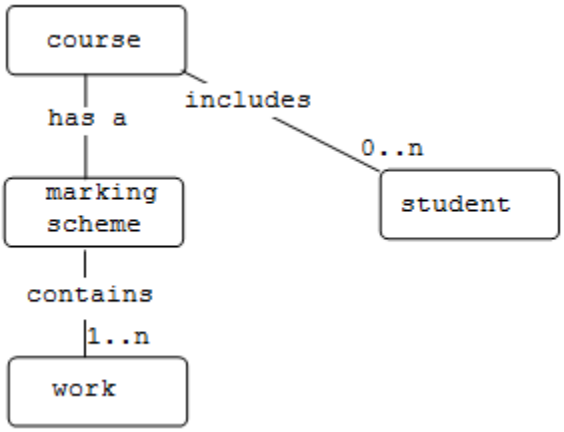


Figure 11.8 Analysis Model

Since this is not a software engineering course we won't write a formal requirements specification. However, this document would include the specification of inputs and outputs and the analysis model as well as a detailed version of the problem statement indicating all the relevant formulas for computing such values as the final marks.

11.3 DESIGN

What we are interested in is a detailed description of each class. This description must include the responsibilities of the class and how the classes cooperate. By responsibilities we mean a statement of what information it remembers and what operations it takes care of. One technique is to use CRC (Class Responsibilities Collaborators) cards, a simple design methodology described by Beck &

Cunningham[13] and refined by Wirfs-Brock and Wilkerson[14]. In this methodology a standard index card is used to describe each class. The card is divided into three areas: class, responsibilities, and collaborators, as shown in Figure 11.9. The responsibilities area is subdivided into responsibility for knowing and responsibility for doing. In the class area of the card we fill in the class name, in the responsibilities area we fill in the things that the class is responsible for and finally, in the collaborators area, we fill in the other classes with which the class collaborates.

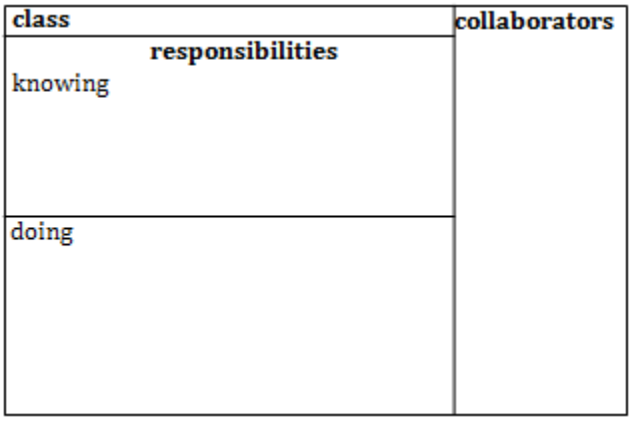


Figure 11.9 CRC card

We start with cards for each identified class: `Course`, `Student`, `MarkingScheme`, and `Work` and fill those names in the class entry of each card.

RESPONSIBILITY FOR KNOWING

The first step is to decide which class has responsibility for knowing what information. First we have to consider the information we might have to store. The list of nouns from the problem statement (Figure 11.6) is a good place to start. Those that were rejected as being values probably represent information we need to store. Eliminating the classes already selected, the duplicates and items not really part of the model, we get the fields shown in Figure 11.10. We may discover others as we continue with design; however this is a start.

mark	final grade	student number	course average
base mark	weight		

Figure 11.10 Field Values

[13] Beck, K. & Cunningham, W.; "A Laboratory for Teaching Object-Oriented Thinking"; Proc. OOPSLA '89 (New Orleans, LA Oct. 1989); SIGPLAN Notices v24, n10 (Oct. 1989), pp 1–6; ACM Press (1989)

[14] Wirfs-Brock, R & Wilkerson, B; "Object-Oriented Design: A Responsibility Approach"; Proc. OOPSLA '89 (New Orleans, LA Oct. 1989); SIGPLAN Notices v24, n10 (Oct. 1989), pp 71–76; ACM Press (1989)

Now we want to allocate these values to the classes. When we decide we write the field on the CRC card under knowing. We might want to do this in pencil in case we change our mind. This process is a bit of an art; however we can often consider the situation in the real world and ask who would know the piece of information. We can also develop an argument as to why a particular class should know the information. Normally we do not wish to duplicate values in different classes, since this uses extra storage and we have an update problem—we must update the value in every place it is stored when the value changes.

The marks are products of a particular student's work in the course. Although in the real world we would probably record these somewhere they logically belong to the student. Thus it makes sense for the `Student` class to take responsibility. Remember, we are in control of the model and defining how the objects behave so we don't have to worry about cheating!

Similarly the final grade and student number belong to `Student`. The base mark and weight define the contribution of the piece of work within the marking scheme so they belong with the `Work` class. Finally the course average is a property of the course—the result of students taking the course—so `Course` should take responsibility. That takes care of the fields we know about so far.

RESPONSIBILITY FOR DOING.

Now let's look at distributing the tasks the system is to perform amongst the classes. Again, the problem statement is a starting point. We select the verb phrases and examine each to see if it is part of the model and represents a task we must achieve. The candidate tasks are shown in Figure 11.11.

program is needed to manage *the marks*
students are awarded *a final grade*
report ... is sent
students receive *marks*
final grade computed
marking scheme defines

Figure 11.11 Candidate Tasks

The first is a statement of what the system does, the fifth defines an output and the sixth just implies that there is a way of computing a final grade. The second really means that a mark is recorded for a student. The third and fourth are really the same: compute a final grade. This leaves record a mark and compute a final grade as identified tasks.

Surely there is more going on than this! If we consider how such a course records system would be used in practice, we might identify more tasks. A common way of identifying prototypical uses of the system is via a use case[15]. A **use case** is a description of one way in which a system may be used. Figure 11.12 shows the use case that is obvious in the problem statement.

An Actor is the user who interacts with the system. The Goal is a short statement of the activity the use case describes. The Steps are the normal (usual) way that the use evolves. Computing the final grades is straightforward. Since the system would be used by a number of instructors for a number of courses, the instructor must select the course and then the system calculates the final grades and generates the report.

[15] Jacobson Ivar, Christerson M., Jonsson P., Övergaard G., *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, 1992

Use Case: Compute Final Grades

Actor: Instructor

Goal: At end of term Instructor computes final grades for submission to Registrar's Office

Steps:

1. Instructor selects course file
2. System calculates final grades
3. System produces final grade report

Figure 11.12 Calculate Final Grades Use Case

Where does the data come from? How is the course file created? These issues aren't addressed in the problem statement. Figure 11.13 is a use case for setting things up at the beginning of term. Remember the Registrar's Office provides a file containing a list of students registered in the course. The instructor must decide on the marking scheme for the course, so she is able to enter it at that time. The system creates the course file for the course that is used in other use cases.

Use Case: Initiate Course

Actor: Instructor

Goal: Prepare system for management of a course.

Steps:

1. Instructor selects class list file
2. Instructor enters marking scheme information
3. System creates course file

Figure 11.13 Initiate Course Use Case

Typically markers evaluate pieces of work submitted by the students and these marks must be provided to the system. Hidden in the problem statement is the identification of the use case for mark entry shown in Figure 11.14. The marker typically marks all of the submissions for one piece of work and then enters all of the marks at once. This is the usage described by this use case. The system presents each student in turn to the marker so the marker can enter the mark for the piece of work.

Sometimes there are alternative ways the use may evolve. A use case may include a section called Extensions that lists, for each step for which there are alternatives, the situation and alternative steps. Here a student may not have submitted a piece of work. This means the Marker cannot enter a mark in Step 3. The extension for Step 3—labeled 3a since there could be more than one—describes this alternative. The Marker might skip entry of a mark. Similarly, a marker may have only marked some of the students' submissions. Instead of skipping each of the remaining students (Extension 3a), Extension 4a indicates that the Marker could inform the system that he was finished entering data (Step 4a.1).

Use Case: Mark Entry

Actor: Marker

Goal: Enter marks for a piece of work for students.

Steps:

1. Marker selects course file
2. System presents student mark information for next student
3. Marker enters mark in piece of work for that student
4. if more students repeat steps 2-3

Extensions:

3a. No mark to record for student
.1 Marker skips entry of mark
4a. Marker decides to quit
.1 System doesn't repeat steps 2-3

Figure 11.14 Mark Entry Use Case

While brainstorming regarding use cases, other situations may be discovered. For example, what if the instructor decides to modify a mark for a student or accept an assignment late due to medical reasons. The instructor will have to change the mark that is currently recorded. Figure 11.15 shows this use case.

Use Case: Update Marks

Actor: Instructor

Goal: Instructor changes mark(s) for piece(s) of work for student(s).

Steps:

1. Instructor selects course file
2. Instructor enters student number
3. System presents mark information for student
4. Instructor updates mark in piece of work for student
5. repeat steps 2-4

Extensions:

2a. Instructor decides to quit
.1 steps 2-4 not repeated
3a. Student not in course
.1 continue at step 2

Figure 11.15 Update Marks Use Case

While this could be a special scenario for the Enter Marks use case, the interaction is probably different. The Instructor will not want to cycle through all of the students just to enter one mark. Rather, she probably would prefer to enter the student number (Step 2) and the System would present the appropriate student (Step 3). Since the Instructor may wish to enter more than one student's mark, the process can be repeated (Step 5). If the Instructor enters a student number that does not match any student in the course, the System cannot present the student. This leads to Extension 3a. When the Instructor has no more marks to enter, we have Extension 2a.

One final use case (Figure 11.16) provides the option of producing a report of the current set of marks within the system as opposed to the final grade report generated in the Calculate Final Grades use case.

Use Case: Produce Mark Report

Actor: Instructor

Goal: At any time the Instructor wants a report of the marks awarded to the students in the course.

Steps:

1. Instructor selects course file
2. System produces mark report

Figure 11.16 Produce Marks Report Use Case

We can discover additional tasks by examining the use cases, underlining verb phrases and determining if they represent tasks for the system just as we did with the problem statement. This results in the combined list of Identified tasks found in Figure 11.17. The last task was not explicit in the problem specification or use cases, but is implicit since it is part of the final grade report.

record/update *marks*	compute *final grade*	enter *marking scheme*
get *next student*	get *student by id*	compute *course average*

Figure 11.17 Identified Tasks

Now that we have some tasks we can make decisions about which class should perform each task. In general, it is best for the object that knows the required information to perform a task involving that information. We can often use the responsibilities for knowing as a guide. Since a `Student` is responsible for knowing her own marks, the operation of recording a mark belongs there. `MarkingScheme` knows about the pieces of work so it should enter the marking scheme. `Course` knows about the students so only it can sequence through the students or locate a student by id.

Computing the final grade for a student requires information from both `Student` and `MarkingScheme`. Clearly they will have to collaborate. `Student` should record the final mark. However it seems clear that the algorithm for computing the mark is part of the marking scheme. (What if the instructor decided to take the better of the two assignments? This would be a change in the marking scheme, not the student.) Thus the `MarkingScheme` should do the actual computation with information from the `Student`. Computing the course average requires a total of the students' marks and a count of the number of students. `Student` cannot do this since it represents only one student, so it is most reasonable for `Course` to do it in collaboration with `Student`.

COLLABORATION

When the information required for a task is not completely within a class, collaboration must occur to achieve the task. Sometimes one task performs a task that is necessary for completing a task assigned to another class. Again collaboration will result. In each case of collaboration, we have the issue of which class will drive the process and which will simply provide services or data.

class Course	collaborators
responsibilities	
knowing course name number of students course average	Student
doing get student by id get next student compute course average	

class Student	collaborators
responsibilities	
knowing student number name marks in pieces of work final grade	MarkScheme
doing record mark calculate final grade	

class MarkingScheme	collaborators
responsibilities	
knowing number of pieces of work	
doing enter marking scheme apply marking scheme	

class Work	collaborators
responsibilities	
knowing name base mark weight	
doing	

Figure 11.18 Course Records System Class Design

For example, since `Student` must record the final grade—the result of the final mark calculation—it seems reasonable that `Student` must drive the final mark calculation process. `MarkingScheme` will provide a method to apply the marking scheme to mark information supplied by `Student`. `Student` will be requested to perform the calculation when `Course` needs to compute the course average.

In the collaborators section we list the classes with which the controlling class collaborates, not the inverse. The completed CRC cards for the four classes are found in Figure 11.18.

CLASS SPECIFICATIONS

The class specifications can now be drawn from the CRC cards. This is often called **detailed design**. The responsibilities for knowing will become instance variables and the responsibilities for doing become methods. The detailed design involves determining the types for each instance variable and the result types and parameters for each method. We will express the specifications as Java interfaces since an interface is a specification for a class. Other notations could be used if the implementation language is not Java.

When doing a class specification we specify the resources of a class that other classes may wish to use. In Java interfaces, only public methods are defined. Instance variables and local methods are not be specified—they are left to the programmer in the Coding phase. However accessor and updater methods are defined to provide access to information known by the class as appropriate. These imply the existence and type of instance variables in the implementation class. The specification is not solely from the CRC cards. Experience with design will likely suggest additional operations. We should also spend some time considering possible uses of the class in future systems and perhaps add features now to make it easier to reuse the class later. Finally, the specification should include comments describing the class and each method, as we have done so far in our code. These comments help define what the classes and methods do for both the user of the class and the programmer writing the class, and so are very important. Writing them as JavaDoc comments allows us to generate on-line documentation via the JavaDoc processor.

Figure 11.19 shows the interface for `Course`. The course name, marking scheme, number of students and course average are made available via accessor methods. The method `getStudent` provides access to a student by id. `calcFinalGrades` computes the final grades for all students and then calculates the course average since the final grades for the students may not have previously been calculated.

The responsibility “get next student” implies that sequencing through the students in the course is necessary (see the Mark Entry use case). In Java, the interface `Iterable` declares a type for which iteration through components is defined. The `Iterable` interface defines a method called `iterator` that returns an object that is used to iterate through the sequence via the methods `hasNext` and `next`. The `Iterable` interface is the basis for the Java for-each statement. By declaring that `Course` extends `Iterable`, we are declaring that any `Course` implementation will implement the `Iterable` interface and thus can be used in a for-each statement. `Iterable` is a generic type. The type parameter is the type of the component in the `Iterable` type, in this case `Student` since we are sequencing through the students of the course. Note the inclusion of the description of the method `iterator` as a comment. This isn’t necessary since it is implied by the extends clause, however it makes it explicit that `iterator` is available and must be implemented.

```
1  package Course_Records;
2
3  /** This interface represents a course offering with the associated students. A
4    * course has a marking scheme and a class average.
5    * @see Student
6    * @see MarkingScheme
7    * @author D. Hughes
8    * @version 1.0 (Jan. 2014)                                                  */
9  public interface Course extends Iterable<Student> {
10
11     /** This method returns an iterator over the students in the course.
12       * @return  Iterator  an iterator over the students in the course.       */
13 //  public Iterator<Student> iterator ( ) ;  // from Iterable
14
15     /** This method returns the name of the course.
16       * @return  String  the course name.                                     */
17     public String getCourseName ( );
18
19     /** This method returns the marking scheme for the course.
20       * @return  MarkingScheme  the course marking scheme.                    */
21     public MarkingScheme getScheme ( );
22
23     /** This method returns the number of students in the course.
24       * @return  int  the number of students.                                 */
25     public int getNumStd ( );
26
27     /** This method returns the average final mark in the course if it has been
28       * computed, otehrwise it returns -1.
29       * @return  double  the course average.                                  */
30     public double getCourseAve ( );
31
32     /** This method returns a student in the class by student number.
33       * @param  stNum  the student number to search.
34       * @return  Student  the student or null if not found.                   */
35     public Student getStudent ( String stNum ) ;
36
37     /** This method calculates the final mark for the students in the course and
38       * computes the course average.                                         */
39     public void calcFinalGrades ( ) ;
40
41 } // Course
```

Figure 11.19 Example—`Course` Interface

The declaration that `Course` is (extends) `Iterable` means that, if we have a `Course` called `aCourse`, we can sequence through the students in the class via a for-each statement such as:

```
for ( aStudent : aCourse ) {
    process aStudent
};
```

The interface for `Student` is found in Figure 11.20. Accessor methods are provided for the student number, name, and final grade. The marks in the individual pieces of work are accessible (`getMark`)

by supplying the piece of work number (from 0). If no mark has yet been recorded, it returns a recognizable value (-1) to signal this. Calculation of the final grade is provided by the method `calcFinalGrade` which will access the `MarkingScheme` for the course. Again the final grade is only meaningful after it has been calculated by `calcFinalGrade` so `getFinalGrade` returns a recognizable `-1` to signal this. Updating the student's marks is provided by the method `update`. The Mark Entry and Update Marks use cases imply that a form would be presented by the system to allow the user to enter/update the marks for the student. It is anticipated that the update method would present such a form.

```
1  package Course_Records;
2
3  /** This interface represents a student in a course. A student has a student
4    * number, a name and marks in a number of pieces of work. Student marks can be
5    * updated and a final mark can be computed according to a marking scheme.
6    * @see Course
7    * @author D. Hughes
8    * @version 1.0 (Jan. 2014)                                                  */
9  public interface Student {
10
11     /** This method returns the student number of the student.
12       * @return  String  the student number                                   */
13     public String getStNum ( );
14
15     /** This method returns the student's name.
16       * @return  String  the student's name                                   */
17     public String getName ( );
18
19     /** This method returns the student's mark in a piece of work. If a mark
20       * hasn't been entered for the piece of work, it returns -1.
21       * @param  num  piece of work number (from 0)
22       * @return  double  the student's mark                                   */
23     public double getMark ( int num );
24
25     /** This method returns the final grade for the student. If the final grade
26       * has not yet been calculated it returns -1.
27       * @return  double  the student's final grade                           */
28     public double getFinalGrade ( );
29
30     /** This method calculates the final grade for the student by applying the
31       * course marking scheme to the student's marks.                        */
32     public void calcFinalGrade( );
33
34     /** This method updates the student's marks for the pieces of work in the
35       * marking scheme for the course.                                       */
36     public void update ( );
37
38 } // Student
```

Figure 11.20 Example—`Student` Interface

The `MarkingScheme` interface is found in Figure 11.21. Accessor methods are supplied for the number of pieces of work and the name, base mark and weight for each piece of work. Similar to `getMark` in `Student`, the piece of work methods take a piece of work number from `0`. It is implied

that the numbering should be consistent between `Student` and `MarkingScheme`. The method `apply` is used to apply the marking scheme to a particular student's work. The `Student` in question is provided as the parameter. The `MarkingScheme` can access the individual marks via the Student accessor method `getMark`. The `apply` method returns the final grade as computed according to the scheme so that the `Student` object can update its final grade.

```
1  package Course_Records;
2
3  /** This interface represents the marking scheme for a course. A marking scheme
4    * consists of a number of pieces of work each with a name, base mark and weight.
5    * The marking scheme can be applied to the marks for the pieces of work for a
6    * student computing a final grade.
7    * @see Work
8    * @author D. Hughes
9    * @version 1.0 (Jan. 2014)                                                    */
10 public interface MarkingScheme {
11
12     /** This method returns the number of pieces of work contributing to the final
13       * grade.
14       * @return  int  number of pieces of work.                                  */
15     public int getNumWork ( );
16
17     /** This method returns the name for a piece of work.
18       * @param  num  piece of work number (from 0)
19       * @return  String  name of piece of work.                                  */
20     public String getName ( int num );
21
22     /** This method returns the base mark for a piece of work.
23       * @param  num  piece of work number (from 0)
24       * @return  double  base mark.                                              */
25     public double getBase ( int num );
26
27     /** This method returns the weight for a piece of work.
28       * @param  num  piece of work number (from 0)
29       * @return  double  weight.                                                 */
30     public double getWeight ( int num );
31
32     /** This method applies the marking scheme to marks for the student producing
33       * a final grade.
34       * @param  theStudent  the student whose marks are to be evaluated
35       * @return  double  the final grade                                         */
36     public double apply ( Student theStudent );
37
38 } // MarkingScheme
```

Figure 11.21 Example—`MarkingScheme` Interface

Figure 11.22 is the interface for `Work` (a piece of work). It provides accessor methods for the name (e.g. "Assignment 1"), base mark and weight for the piece of work. The current description of the problem implies that the marking scheme is set at the beginning of the course and not changed. However, this may not always be the case. Maybe an instructor will want to change the base mark of a piece of work. With this future consideration in mind, updater methods for base mark and weight have been provided.

```
1  package Course_Records;
2
3  /** This interface represents a piece of work in the marking scheme for a course.
4    * A piece of work has a name, a base mark and a weight.
5    * @author D. Hughes
6    * @version 1.0 (Jan. 2014)                                                    */
7  public interface Work {
8
9      /** This method returns the name of the piece of work.
10       * @return  String  piece of work name                                     */
11     public String getName ( );
12
13     /** This method returns the base mark of the piece of work.
14       * @return  double  base mark                                              */
15     public double getBase ( );
16
17     /** This method returns the weight of the piece of work.
18       * @return  double  weight                                                 */
19     public double getWeight ( );
20
21     /** This method sets the base mark of the piece of work.
22       * @param  b  base mark                                                    */
23     public void setBase ( double b );
24
25     /** This method sets the weight of the piece of work.
26       * @param  w  weight                                                       */
27     public void setWeight ( double w );
28
29 } // Work
```

Figure 11.22 Example—`Work` Interface

Between the analysis model, CRC cards, use cases and class specifications we now have enough information to allow the classes to be written. One programmer can write a specific class such as `Student`. At the same time, another programmer may write a client class such as `MarkingScheme` that uses the `Student` class. This second class can be written since the `Student` interface details all that must be provided and all that can be expected from the `Student` class. Once the `Student` interface is compiled, the compiler can verify that the client class is making appropriate use of the resources of the `Student` class.

11.4 CODING

In the coding phase one or more programmers go about writing the implementation classes for the interfaces defined in the detailed design. Additional implementation classes are typically also written in support of the classes defined in the design phase. Basically the programmer has a contract to fulfill—the class specification—but is free to implement it in any reasonable manner. The programmer must think of this class as the ultimate goal and not be concerned about the system as a whole. The advantage of object-oriented programming is that components can be developed separately and can be assembled later. The system as a whole is generally far too large to be comprehended at any single instant and a programmer would easily get lost in the details.

In addition to the methods defined in the interface. the programmer defines the instance variables, defines and writes the constructors and writes any support or helper methods and classes required. The instance variables are derived in part from the "Responsibility for Knowing" on the CRC card for the class. Constructors are in part dependent on the instance variables and where the information to initialize these comes from. Helper methods and classes are at the discretion of the programmer. Of course, the programmer must also look to libraries for support. With good design, programming should be straightforward but not necessarily simple. The implementation is left as an exercise.

ITERATORS

Some discussion of the `Iterable` and `Iterator` interfaces is necessary. Figure 11.23 shows the `Iterable` interface.

```
1  package java.lang;
2
3  import java.util.Iterator;
4
5  /** Implementing this interface allows an object to be the target of the "foreach"
6    * statement.
7    *
8    * @param <T>  the type of elements returned by the iterator.                    */
9  public interface Iterable<T> {
10
11     /** Returns an iterator over a set of elements of type T.
12       * @return  Iterator<T>  an iterator.                                        */
13
14     public Iterator<T> iterator();
15
16 }  // Iterable
```

Figure 11.23 `Iterable` Interface

The `Iterable` interface defines what allows objects of a class to be the target in a for-each statement. The implementing class must provide a method that returns an `Iterator` over the components of the class. The components are of type `T`, the generic parameter. For example, `Picture` in the `Media` library implements `Iterable <Pixel>`. This allows the use of a for-each statement to iterate through all pixels in the picture such as:

```
for ( Pixel p : aPic ) {
    if ( p.getDistance(RED) < TOLERANCE ) {
      p.setColor(BLACK);
    };
}
```

The for-each statement creates an iterator over the `Picture` by calling `aPic.iterator()` and then uses the methods of the `Iterator` interface to provide the iteration. Figure 11.24 shows the `Iterator` interface.

```
1  package java.util;
2
3  /** An iterator over a collection. Iterator takes the place of Enumeration in the
4    * Java Collections Framework. Iterators differ from enumerations in two ways:
5    *      * Iterators allow the caller to remove elements from the underlying
6    *        collection during the iteration with well-defined semantics.
7    *      * Method names have been improved.
8    * This interface is a member of the Java Collections Framework.
9    * @param <E>  the type of elements returned by this iterator.                */
10 public interface Iterator<E> {
11
12     /** Returns true if the iteration has more elements. (In other words, returns
13       * true if next() would return an element rather than throwing an exception.)
14       * @return  true  if the iteration has more elements                        */
15     public boolean hasNext();
16
17     /** Returns the next element in the iteration.
18       * @return  the next element in the iteration
19       * @throws  NoSuchElementException  if the iteration has no more elements   */
20     public E next();
21
22     /** Removes from the underlying collection the last element returned by this
23       * iterator (optional operation). This method can be called only once per call
24       * to next(). The behavior of an iterator is unspecified if the underlying
25       * collection is modified while the iteration is in progress in any way other
26       * than by calling this method.
27       * @throw  UnsupportedOperationException  if the remove operation is not
28       *                                        supported by this iterator
29       * @throw  IllegalStateException  if the next method has not yet been called,
30       *                                or the remove method has already been called
31       *                                after the last call to the next method.   */
32     public void remove();
33
34 }  // Iterator
```

Figure 11.24 `Iterator` Interface

An `Iterator` over a collection of objects of type `E` (`Pixel` in the above example) provides a method `hasNext` which returns `true` if there are additional objects in the collection that have not yet been processed. When `hasNext` returns `true`, the method `next` returns a reference to the next unprocessed object in the collection. The method `remove` is optional, throwing an `UnsupportedOperationException` if it is not implemented. The for-each statement above is implemented as:

```
Iterator<Pixel> it = aPic.iterator();
while ( it.hasNext() ) {
    p = it.next();
    if ( p.getDistance(RED) < TOLERANCE ) {
      p.setColor(BLACK);
    };
}
```

When a class implements the `Iterable` interface—either because it implements `Iterable` directly or it implements an interface that extends `Iterable` as is our case— it must implement the method `iterator`. The first step is to define a class that implements `Iterator` for this specific collection class. For example we would implement a class `CourseIterator` that implements `Iterator<Student>`. The `Course iterator` method would create an instance of the `CourseIterator` and return it as its result such as:

```
public Iterator<Student> iterator ( ) {  // from Iterable

    return new CourseIterator(theClass);

};  // iterator
```

assuming the representation of the collection of students in the `Course` is an array of `Student` called `theClass`.

The `CourseIterator` class is shown in Figure 11.25. It is a non-public class within the same package as the `Course` implementation class. The only way for a client class in another package to create a `CourseIterator` is via the `iterator` method of the `Course` implementation class. Since it is in the same package, it is reasonable that it "knows" the representation of the collection of students is an array of `Student`. As instance variables (lines 12 & 13) it has a reference to the `Student` array in the `Course` implementation class (`theClass`) and its own index variable (`index`) into that array. It can thus independently sequence through the elements of the student array without affecting the `Course` implementation. The constructor (lines 19–22) receives the reference to the student array and initializes the index to the first element. `hasNext` returns `true` as long as `index` hasn't reached the number of elements in the array. As long as there are more elements, `next` returns the element at `index` and increments `index`. `remove` is not implemented.

Note that there is a high degree of coupling between the `Course` implementation and the `CourseIterator`. The `CourseIterator` must know the implementation of the `Course` to be able to work. They are defined in the same package so a higher level of coupling is acceptable than if they were in different packages. It would be expected that the author of the `Course` implementation class would also write the `CourseIterator` class.

THE MAIN CLASS(ES)

There is one additional decision that is necessary in developing a system—the issue of the main class. There are two aspects to the decision. The first is whether the main class should be one of the classes identified during design or a separate class identified during coding. The second is whether there should be one main class or more than one.

Sometimes one of the identified classes is the obvious main class, sometimes it is an open question. In our case `Course` is the only candidate as a main class since it is the only class that "knows" directly or indirectly the others. If `Course` takes on the responsibility of being a main class then it would have to support all use cases and a newly identified use case could require modification of the `Course` class. This is probably not desirable.

```
1  package Course_Records;
2  import java.util.*;
3
4  /** This class defines an iterator over the students in a course as required by
5    * the Iterator interface.
6    * @see Course
7    * @see Student
8    * @author D. Hughes
9    * @version 1.0 (Jan. 2014)                                                    */
10 class CourseIterator implements Iterator<Student> {
11
12     private Student[]  theClass;  // the students of the ccourse
13     private int        index;     // next student to access
14
15     /** This constructor creates an iterator over the specified array of students
16       * being the students in the course. Iteration begins at the first student in
17       * the list.
18       * @param  c  the array of students in the course.                          */
19     CourseIterator ( Student[] c ) {
20         theClass = c;
21         index = 0;
22     };  // constructor
23
24     /** This method returns true if there is at least one more student in the
25       * course.
26       * @return  boolean  true if at least one more student in coruse.          */
27     public boolean hasNext ( ) {  // from Iterator
28         return index < theClass.length;
29     };  // hasNext
30
31     /** This method returns the next student in the course.
32       * @return  Student  the next student in the course
33       * @throws  NoSuchElementException  no students left in the course.        */
34     public Student next ( ) {     // from Iterator
35         if ( ! hasNext() ) {
36             throw new NoSuchElementException();
37         };
38         index = index + 1;
39         return theClass[index-1];
40     };  // next
41
42     /** This method is unsupported.
43       * @throws  UnsupportedOpreationException  always                          */
44     public void remove ( ) {      // from Iterator
45         throw new UnsupportedOperationException();
46     };  // remove
47
48 }  // CourseIterator
```

Figure 11.25 Example—`CourseIterator` Class

The more common case is to identify a new class at coding time that will serve as the main class. The class would provide what is necessary to load up and shut down the system and then would use resources of the identified classes to provide the uses of the system. As we have seen there are often

a number of use cases that involve quite different interactions with the system. This means that a single main class would have to accommodate all of these use cases. It would likely present a GUI that allows the user to select the use they wish to make. The advantage of this approach is that a user often wants to make multiple sequential uses of the system, such as to enter marks and then produce a mark report.

Another alternative is to have main classes for each (or many) use cases. Remember, all that is required for a class to be a main class is for it to implement the method `main` and any number of classes in the same package or project may be a main class. Having multiple main classes has the advantage of keeping the main classes quite simple but may give the user the impression that he is using different systems. It may also lead to reduced consistency in the GUIs presented by the different uses.

11.5 TESTING

Once the code has been written, it must be tested. Tests are designed during the analysis and design phases and are performed during the testing phase. First each class must be tested on its own—unit testing. Later a class is tested with classes with which it collaborates—integration testing—until the complete system is assembled and system tests are performed. We will not describe the complete testing here, but rather look at an example.

CLASS STUB

Let's consider unit level testing of the `Student` class. We have a bit of a problem. There is no main class and the class `MarkingScheme` class isn't available for testing—at least not until integration testing.

We solve the problem of the missing `MarkingScheme` class by writing what is called a class stub. A **class stub** is an implementation class that provides all of the methods defined by the interface, but does not include the actual code. Rather it has stubs for the methods—simple versions of the methods that receive the parameters and indicate (usually by doing I/O) what is going on. If it is a function method it returns some well-defined value. An example class stub for the `MarkingScheme` class is found in Figure 11.26.

In the class stub there are typically no instance variables, and the constructor is often the default constructor that does nothing. The accessor methods print that they were called (including the parameter they were called with) and return constant values. Similarly the `apply` method simply displays the parameters it was passed and returns an arbitrary but known value. We are not supposed to be writing the actual `MarkingScheme` class.

Stubs are also written for each other class with which `Student` collaborates.

```
1  package Course_Records;
2
3  public class MarkingSchemeStub implements MarkingScheme {
4
5      public int getNumWork ( ) {
6          System.out.println("MarkingScheme.getNumWork returned 4");
7          return 4;
8      };
9
10     public String getName ( int num ) {
11         System.out.println("MarkingScheme.getName("+num+") returned Work");
12         return "Work";
13     }
14
15     public double getBase ( int num ) {
16         System.out.println("MarkingScheme.getBase("+num+") returned 10");
17         return 10;
18     }
19
20     public double getWeight ( int num ) {
21         System.out.println("MarkinhScheme.getWeight("+num+") returned 25");
22         return 25;
23     }
24
25     public double apply ( Student theStudent ) {
26         System.out.println("MarkingScheme.apply("+theStudent+") returned 75");
27         return 75;
28     };
29
30 } // MarkingScheme
```

Figure 11.26 Example—`MarkingScheme` Class Stub

TEST HARNESS

To test the `Student` class, we also need a main class. A specialized main class for testing a class is called a **test harness**. The test harness should perform all the desired tests of the class by calling the appropriate methods and displaying the results. What the test harness does will depend on the test specifications developed in the analysis and design phases. Tests should be repeatable. If the test harness is doing I/O, it should be from a file so the file can be saved along with the test harness for future use.

A test harness for the `Student` class is found in Figure 11.27. It must test the constructor and all methods of `Student` in all appropriate cases. We test the constructor and all accessor methods by creating an object and displaying its state as evidenced by the accessor methods (`display`). We test `calcFinalGrade` once with no marks recorded and once after marks have been recorded and check that it calls `apply` appropriately. `update` is tested to verify that it loads new data and changes the object state. The console output from a sample test run is found in Figure 11.28.

```
1  package Course_Records;
2  import BasicIO.*;
3
4  public class StudentTest {
5
6      private Course         aCourse;
7      private ASCIIDataFile  from;
8      private Student        aStudent;
9      private int            numWork;
10
11     public StudentTest ( ) {
12         aCourse = new CourseStub();
13         numWork = aCourse.getScheme().getNumWork();
14         from = new ASCIIDataFile();
15         aStudent = new StudentImpl(from,aCourse);
16         from.close();
17         System.out.println();
18         System.out.println("New Student created: "+aStudent);
19         display();
20         System.out.println();
21         System.out.println("Calculate Final Grades");
22         aStudent.calcFinalGrade();
23         display();
24         System.out.println();
25         System.out.println("Update");
26         aStudent.update();
27         display();
28         System.out.println();
29         System.out.println("Calculate Final Grades");
30         aStudent.calcFinalGrade();
31         display();
32     };  // constructor
33
34     private void display ( ) {
35         System.out.println("Student: "+aStudent);
36         System.out.println("    stNum:    "+aStudent.getStNum());
37         System.out.println("    Name:     "+aStudent.getName());
38         for ( int i=0; i<numWork ; i++ ) {
39             System.out.println("    Mark["+i+"]: "+aStudent.getMark(i));
40         };
41         System.out.println("    Final:    "+aStudent.getFinalGrade());
42     };  // display
43
44     public static void main ( String[] args ) { StudentTest s = new StudentTest(); };
45
46 }  // StudentTest
```

Figure 11.27 Example—`Student` Class Test Harness

Note how the student object is printed on lines 18 and 35. Since the object is concatenated to a string, the `toString` method of `StudentImpl` is called. If we do not write `toString`, the default version is used. The default implementation displays the class name (`Course_Records.StudentImpl`) and an unique identifier (`@17cd3d6`)—actually the address of the object in memory. This is seen in the first line in Figure 11.28. We can use this information to verify that the correct `Student` object is being referenced.

```
New Student created: Course_Records.StudentImpl@17cd3d6
Student: Course_Records.StudentImpl@17cd3d6
    stNum:   111111
    Name:    John Smith
    Mark[0]: -1.0
    Mark[1]: -1.0
    Mark[2]: -1.0
    Mark[3]: -1.0
    Final:   -1.0

Calculate Final Grades
Course.getScheme() returned Course_Records.MarkingSchemeStub@1f40067
MarkingScheme.apply(Course_Records.StudentImpl@17cd3d6) returned 75
Student: Course_Records.StudentImpl@17cd3d6
    stNum:   111111
    Name:    John Smith
    Mark[0]: -1.0
    Mark[1]: -1.0
    Mark[2]: -1.0
    Mark[3]: -1.0
    Final:   75.0

Update
Course.getScheme() returned Course_Records.MarkingSchemeStub@1f40067
StudentForm(Course_Records.StudentFormStub@1e2771e) created for: Course_Records.StudentImpl@17cd3d6
    with scheme: Course_Records.MarkingSchemeStub@1f40067
StudentForm.update() returned true
StudentForm.readMark(0) returned 0
StudentForm.readMark(1) returned 10
StudentForm.readMark(2) returned 20
StudentForm.readMark(3) returned 30
StudentForm.close() called
Student: Course_Records.StudentImpl@17cd3d6
    stNum:   111111
    Name:    John Smith
    Mark[0]: 0.0
    Mark[1]: 10.0
    Mark[2]: 20.0
    Mark[3]: 30.0
    Final:   75.0

Calculate Final Grades
Course.getScheme() returned Course_Records.MarkingSchemeStub@1f40067
MarkingScheme.apply(Course_Records.StudentImpl@17cd3d6) returned 75
Student: Course_Records.StudentImpl@17cd3d6
    stNum:   111111
    Name:    John Smith
    Mark[0]: 0.0
    Mark[1]: 10.0
    Mark[2]: 20.0
    Mark[3]: 30.0
    Final:   75.0
```

Figure 11.28 `TestStudent` Console Output

The test output in Figure 11.28 shows that a student object was created correctly with marks in all pieces of work as N/A (`-1`) and no calculated final grade (`-1`). Calculating the final grade caused the `MarkingScheme.apply` method to be called with this student object as argument and the result value (`75` returned by `apply`) was recorded in the `Student` object. The `update` method updated

the mark values of the `Student` object using `StudentForm` (also a class stub). Finally the second calculation of final grade also called `MarkingScheme.apply` appropriately.

Each of the other classes would be tested with appropriate harnesses and stubs until each is determined to be working. Then integration testing would be done with complete classes tested together (possibly with stubs for other classes) and a harness. Finally, all classes would be grouped for the system test this time using the real main class. In all the testing care should be taken to test all cases. This means that a number of different values should be tested for each piece of data including values at the beginning and end of any data ranges. For example, both 0 and full marks should be tested as well as at least one value in between. In addition, any error situations that the program should be able to handle should be tested. For files, at least a case with an empty file and one containing a number of records should be tested.

11.6 DEBUGGING, PRODUCTION, AND MAINTENANCE

During any phase of testing the program may crash or run but produce unexpected results. This means that debugging must be done. Often the test harness and class stubs give enough information to pinpoint the source of the error and it can be corrected. Sometimes the source of the error is a bit harder to detect. In these cases it may be useful to place calls to `System.out.println` at appropriate points in the code to trace what is happening to the values of variables. In difficult cases it might be helpful to use a **debugger**—a tool in an IDE that allows breakpoints to be set in the code where inspection of the objects in the system is possible.

When the error is corrected, the class-level testing must be reapplied continuing to integration testing and finally system testing. This continues until no further bugs are detected and the system is considered complete.

The system is then released to the users—entering the production phase. Maintenance now begins. Records are kept of any problems (bugs) encountered by users, and the bugs are fixed on a not-yet released copy of the system. At some point it is determined that it is time to issue a new release so this copy is moved into production. Over time, new features requested by users are analyzed. The development cycle for the next version is started, and we're back to the start.

SUMMARY

Large-scale software development is often done by teams of developers over an extended period of time. To ensure that such projects successfully come to completion, a well-defined process must be followed. This process is usually defined to have seven phases: analysis, design, coding, testing, debugging, production, and maintenance. Even if the project is relatively small and involves only one developer, the development can benefit from the use of the process even if only done informally.

The analysis phase involves determination of what the proposed system is supposed to do through the development of a requirements specification and a model of the system. During design the primary classes of the system are identified and the responsibilities are divided amongst the classes. A common tool for this part of design is the CRC card. During detailed design complete specifications of these classes are produced. Coding involves the realization of the classes in a programming language. Testing involves running the classes individually and in groups to determine if the classes and ultimately the system perform as required. Debugging is recoding, redesigning, or possibly

reanalysis of the system to address errors detected during testing. Finally, when the system is felt to be free of problems, it is released to the user community—production phase. At this point maintenance begins, where further errors are corrected and new features added to the existing system.

REVIEW QUESTIONS

1. T F Candidate objects are identified by underlining nouns in the problem statement.

2. T F Responsibility for doing should be determined before responsibility for knowing in designing classes.

3. T F A class stub is used as the mainline when testing a class.

4. T F The main purpose of analysis is to write the problem statement.

5. T F We do not wish to duplicate values in different classes because it causes an update problem.

6. T F "Responsibilities for knowing" become methods.

7. T F A client class is a class written for a client of the software development company.

8. Which of the following is not a phase in software development?

   a) debugging
   b) compiling
   c) designing
   d) testing

9. What is the product of the analysis phase of software development?

   a) requirements specification
   b) inputs and outputs
   c) system model
   d) all of the above

10. Trainers and technical support personnel are involved in which phase of software development?

    a) analysis
    b) testing
    c) production
    d) maintenance

11. Which of the following is part of the design phase?

    a) determining the inputs and outputs of the system
    b) writing a class specification for each class
    c) developing a model of the system
    d) writing a requirements specification

12. Which of the following is not a valid reason for eliminating a candidate object?

    a) it represents a value
    b) it is just another name for an existing entity
    c) there could be more than one object of that type
    d) it is not part of the system being developed

13. A specialized main class for testing a class is called a:

    a) class stub
    b) method stub
    c) test harness
    d) test stub

14. System testing:

    a) is usually done by testers
    b) is done on a class-by-class basis
    c) is done by programmers
    d) should be avoided

15. A CRC card

    a) is a Cyclic Redundancy Check card
    b) should always be used in analysis
    c) helps in distributing responsibilities to objects
    d) none of the above

EXERCISES

1. Implement the Course Records system as described in this chapter. Provide implementation classes for each of the identified classes and design and implement any other implementation classes necessary to support the identified classes. Write individual main classes for each of the use cases identified.

2. ACME Widgets Inc. requires a program to process its payroll. Employees in the company are paid weekly, and their salary is based on their hours worked and rate of pay. The federal and state governments require that the company withhold tax based on a formula provided by the governments and subject to change annually.

   Employees are paid straight-time for the first 40 hours worked and time-and-a-half for overtime hours (hours in excess of 40). Federal tax is based on a three-tier system. Zero tax is paid on the amount less than or equal to the first tier amount, a lower tax rate on the amount greater than the first tier amount and less than or equal to the second tier amount, and finally a higher tax rate on the amount exceeding the second tier amount. For example, the system might be that $0 is paid on the first $13,000 (first tier), 30% (low rate) on the amount between $13,000 and $52,000 (second tier), and 50% on the remaining. If the employee earned $62,000, the tax would be $16,700 ($0 + (39,000*0.3) + (10,000*0.5)). Provincial tax is computed as a percentage of federal tax.

A file of timesheet information is created each week as an `ASCIIDataFile`, containing information the employees. The file contains, for each employee: (1) the employee number (int), (2) pay rate (`double`), and (3) hours worked (`double`). Another file (a second `ASCIIDataFile`) of taxation information is also available containing: (1) first tier amount (`double`), (2) low rate (`double`), (3) second tier amount (`double`), (4) high rate (`double`), and (5) provincial rate (`double`). The limits are provided based on weekly pay, which is annual rate/52.

The program is to input the employee information, compute pay and taxes, and generate a report (`ReportPrinter`) indicating the employees' gross pay, federal tax withheld, state tax withheld, and net pay. Since the company must remit the federal and provincial taxes withheld to the respective governments, the program must also display the total taxes withheld. In addition, so that the auditors may audit the payroll records, the total gross and total net pay paid out must be computed and displayed. Appropriate happiness messages should be generated to an `ASCIIDisplayer` stream.

The report generated by the program might look similar to the following,:

```
                    ACME WIDGETS

 Emp#    Gross Pay    Fed Tax    State Tax     Net Pay
1111       $500.00      $75.00      $33.75      $391.25
2222     $2,400.00     $925.00     $416.25    $1,058.75
Total    $2,900.00   $1,000.00     $450.00    $1,450.00
```

3. The Hydro-Electric Commission requires a program to do its monthly billing. For each customer, a record (line) is entered in an `ASCIIDataFile` recording: customer number (`int`), customer type (`char`, `c` for commercial and `r` for residential), previous reading (`double`), and current reading (`double`). The program should produce a report that gives, for each customer, the customer number, consumption, and amount billed, in a paginated report with appropriate headers. The report summary should indicate the total amount billed. The report should look something like:

```
   Hydro-Electric Commission
         Billing Report

Customer     Consumption    Amount
1111            1215.0      $92.90
  :               :            :
Total Billed            $23,259.70
```

There is a fee schedule that determines the amount to be billed based on customer type and consumption. For residential customers, there are two billing levels. Consumption up to the specified limit is billed at the first (higher) rate, and consumption in excess of the limit is billed at the second (lower) rate. Commercial customers are billed at a single rate for all consumption. A file (`ASCIIDataFile`) is prepared that contains the fee schedule amounts for the month. The information is recorded in order: first residential rate (`double`), limit (`double`), second

residential rate (`double`) and commercial rate (`double`). Readings are in kilowatt-hours and rates are in dollars per kilowatt-hour.

4. Sharkey's Loans loans money to individuals and each month collects a payment with (considerable) interest. Every month, Sharkey's Loans produces a report that specifies the details of each loan.

   Sharkey is a member of a business consortium that enforces the prompt payment of the minimum balance each month for each loan customer. Due to changing market conditions, the business people in the consortium frequently change the interest rates applied to the loans.

   The business people have noticed that loans with a high outstanding balance tend to require enforcement of payment. For this reason, the interest rate for the next month is calculated using a three-tier system based on the new balance from the current month. A low interest rate is used on the first tier amount, a middle interest rate on the second tier amount, and a high interest rate on the third tier amount.

   Interest is paid on the previous balance, plus debits, minus credits. For example, suppose that the consortium has decided to charge 10% monthly interest on the first $1,000 (first tier = $1,000), 20% interest on the amount between $1,000 and $6,000 (second tier = $6,000), and 30% interest on the remaining amount. Then the interest this month for a loan with a previous balance of $9,000, debits this month of $3,500 and credits this month of $2,500 would be $2,300, computed as:

```
0.10*1000 + 0.20*(6000-1000) + 0.30*(10000-6000)
```

giving a new balance of $12,300 ($10,000 + $2,300). The minimum payment each month can also vary, but is calculated as a straight percentage of the new balance.

For each loan, the information concerning each month's activities is stored in an `ASCIIDataFile`. Each line concerns a separate loan, and includes the following information: loan number (`int`), previous balance (`double`), amount borrowed by the customer this month ("debits": `double`), and amount paid by the customer this month ("credits": `double`). Another `ASCIIDataFile` of rate information is also available containing: low rate (`double`), first tier amount (`double`), middle rate (`double`), second tier amount (`double`), high rate (`double`), and minimum payment rate (`double`). Note that all rates are given as monthly percentages.

The monthly report might look similar to the following:

```
                    Sharkey's Loans
                     Monthly Report

Loan#    PrevBal    Debits   Credits   Interest   NewBal   MinPaymt
123    $1,000.00   $200.00   $400.00    $80.00   $880.00   $220.00
456    $2,000.00     $0.00   $500.00   $200.00 $1,700.00   $425.00
789    $5,000.00 $3,000.00 $2,000.00 $1,100.00 $7,100.00 $1,775.00
Totals $8,000.00 $3,200.00 $2,900.00 $1,380.00 $9,680.00 $2,420.00
```

In the report, the loan number, previous balance, debits and credits are the values from the monthly data file. The interest is calculated as described above and the new balance is calculated as the previous balance plus debits minus credits plus interest. The minimum balance is calculated as the specified percentage of the new balance. The summary totals are the totals of the previous balance, debits, credits, interest, new balance, and minimum payments, respectively.