Java with BlueJ Part 2

Ron McFadyen

March 27, 2016

Department of Applied Computer Science

University of Winnipeg

515 Portage Avenue

Winnipeg, Manitoba, Canada

R3B 2E9

r.mcfadyen@uwinnipeg.ca

ron.mcfadyen@gmail.com

The website for this book is

www.acs.uwinnipeg.ca/rmcfadyen/CreativeCommons/

To Callum

Contents

Chapter 1

One-Dimensional Arrays

There are many situations where we deal with a collection of information. Some examples are:

1. names of students in a class
2. courses offered by a department
3. temperatures for the last month
4. employees in a company

The above cases all have one thing in common: in each case there can be more than one value. For instance, there would be several students in a class and for each student there is a name, for example: "John", "Mary", "Lee", etc. In Java, one way of handling a collection like this is to use a *data structure* called an *array.* The array is declared similar to other variables and then an integer (called an index) is used to refer to its elements individually. So, `studentName` can be the name of the collection and `studentName[0]`, `studentName[1]`, `studentName[2]`, etc. is the way we refer to elements of the collection. To declare an array of names where each element of the array can be a `String` value we use:

```
String[ ] studentName;
```

The square braces `[ ]` are used to indicate a one-dimensional array. Its called one-dimensional because one index value is used to refer to an individual element of the array. In Java index values begin at 0 and go up to the length of the array -1. We can declare arrays of any type, for example:

declaration	sample purpose
String[] studentName;	an array of names
int[] mark;	an array of marks
double[] temperature;	an array of temperatures
boolean[] answer;	an array of true/false answers
char[] letter;	an array of multiple choice answers

The above are examples of how to declare an array. Before the array can be used the programmer must also declare its size. Once the programmer declares the size it cannot be made larger - this is one of the drawbacks to using arrays and why sometimes another technique will be chosen. To declare an array that can hold, say, 100 names we use:

```
String[] studentName;
studentName = new String[100];
```

or, we can combine the above into one line:

```
String[] studentName = new String[100];
```

or, if an `int` variable holds the length we can write:

```
int arraylength = 100;
String[] studentName = new String[arraylength];
```

Every array has an int field named `length` that is a part of it; the value stored is the length of the array. So, for `studentName` above the value stored in `studentName.length` is 100. This field is very useful; for instance if we need to display all the names in `studentName` we can use the code:

```
for (int i=0; i<studentName.length; i++}
    System.out.println(studentName[i]);
```

The length field is *immutable* which means it cannot be altered once it is set. This means that once you have declared an array to be a certain length you cannot change its length.

1.1 Initializing arrays

Because arrays can have multiple values there is a different syntax used when its necessary to set initial values. For instance, suppose we need an array to hold the number of days in each month. We can declare and initialize as:

```java
int[] daysInMonth =
   {31,28,31,30,31,30,31,31,30,31,30,31};
```

The Java syntax for initializing an array is to enclose a comma-separated list of values between the pair `{ }`. Initializing arrays this way also sets the length of the array. The value of `daysInMonth.length` is 12.

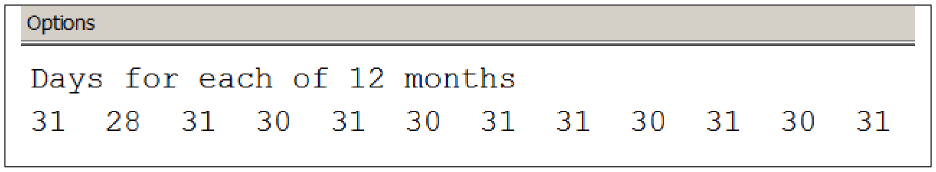
Example 1 Array initialization and displaying each element

Consider the following program where `daysInMonth` is initialized and displayed.

Listing 1.1: Initializing and displaying an array.

```java
/**
 * Display number of days in each month
 */
public class MonthLengths
{
    public static void main(String[] args){
        int[] daysInMonth =
            {31,28,31,30,31,30,31,31,30,31,30,31};
        System.out.println("Days for each of "
            +daysInMonth.length+" months ");
        for (int i = 0; i< daysInMonth.length; i++)
            System.out.print(daysInMonth[i]+"  ");
    }
}
```

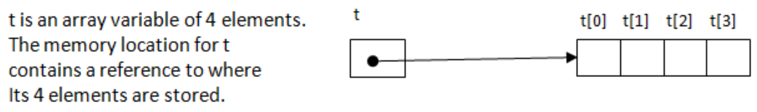
The output:



```
Days for each of 12 months
31  28  31  30  31  30  31  31  30  31  30  31
```

1.2 Storage of arrays and copying arrays

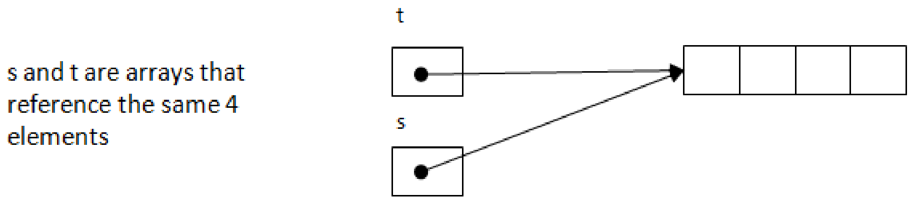
Arrays are objects in Java and so the memory location for the array variable contains a reference to the actual storage locations holding the array's values. For instance the memory allocations for an array can be visualized as:


t is an array variable of 4 elements.
The memory location for t
contains a reference to where
Its 4 elements are stored.
t
t[0]
t[1]
t[2]
t[3]

Now suppose we need to make a copy of the array. If we just use:

```
s = t; //s and t are arrays of same type
```

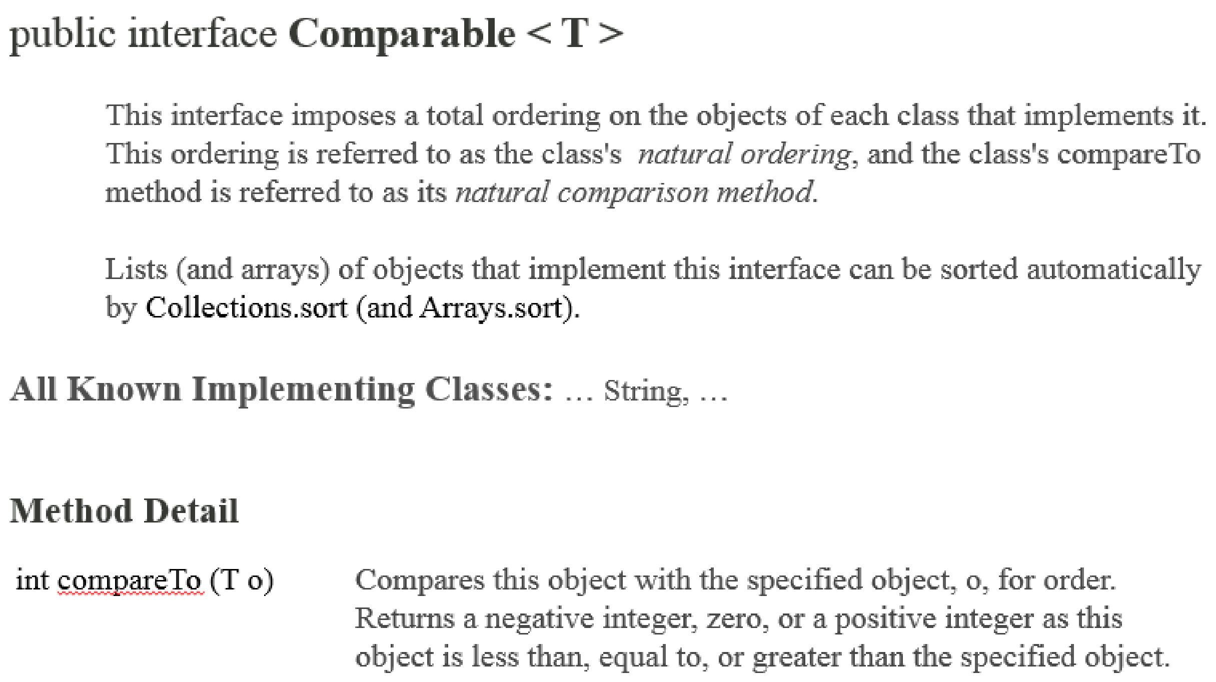
what we end up with is two storage locations for `s` and `t` that reference the same 4 elements. We haven't created a copy, rather we have two array variables that reference the same 4 elements:



If we need a real copy of the array `t` then we require a loop to accomplish this:

```
// s and t are of the same type
for (int i=0; i<t.length; i++) s[i] = t[i];
```

You can re-instantiate an array variable. New locations are assigned to the array (see below) and the old ones are reclaimed for reuse according to an internal Java garbage collection procedure.



public interface **Comparable < T >**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).

All Known Implementing Classes: … String, …

Method Detail

int compareTo (T o)	Compares this object with the specified object, o, for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

1.3 The *enhanced for*

There is a variation on the for called the *enhanced* for that can be used when a program iterates from the first element through to the last element of an array and does not change any values. The syntax is

```
for ( type variable : array )
         statement
```

The `for` statement in the previous example can be rewritten:

```
for (int days : daysInMonth)
        System.out.print(days+"   ");
```

Example 2 Calculating an average

Consider the following program where `temperature` is assigned values obtained from a user and then the average temperature is displayed. The assignments must be done using a `for` whereas the calculation of the sum can be done with a *enhanced* `for`.

Listing 1.2: Initializing an array from input.

```java
import java.util.Scanner;
/**
 * Display average of 7 values
 */
public class AverageTemperature
{
    public static void main(String[] args){
        Scanner kb = new Scanner(System.in);
        double[] temperature = new double[7];
        System.out.println("Enter 7 temperatures:");
        for (int i=0; i<7; i++)
            temperature[i] = kb.nextDouble();
        double sum = 0.0;
        for (double t:temperature) sum +=t;
        System.out.println("average= "+sum/7.0);
    }
}
```

When to use the ***enhanced for***

The *enhanced* `for` helps to express a programming idiom succinctly as no loop counter is required. However, there are many cases where the *enhanced* `for` cannot be used:

1. iterate backwards through the array elements
2. access elements of more than one array
3. partially filled arrays (discussed later)
4. assigning new values to array elements.

1.4 Passing string values into main()

In all of our main methods we have specified a String array named `args`:

```
    public static void main(String[] args)
```

In the above line `args` is declared to be an array of `String`. The variable `args` is used to pass values (that are strings) into a method. When you have used BlueJ to execute the main() method you have the opportunity to pass an array of strings to the program.

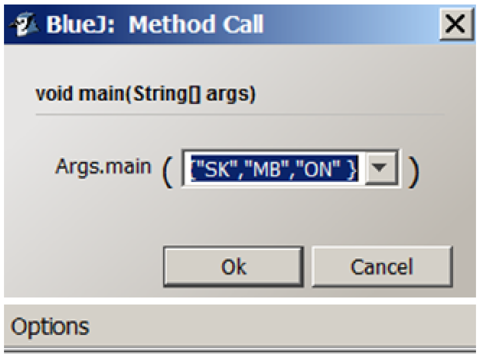
Example 3 Passing arguments to main()

The following program just lists the strings passed into the program.

Listing 1.3: String values passed into `main()`.

```java
/**
 * Print the values passed into the program
 */
public class Args
{
   public static void main(String[] args){
      System.out.println("The elements of args:");
      for (String s: args) System.out.print(" "+s);
   }
}
```

The following shows a user executing main() and passing in 3 strings with the resulting output from the program:



```
The elements of args:
 SK MB ON
```

1.5 Parallel arrays

There are times when two or more arrays have exactly the same number of elements and where array elements at the same index relate to one another in some meaningful way. For example suppose we have one array of student names and another of student numbers. If the arrays represent information for the same set of students then we would want to arrange that the i^{th} element of the name array and the i^{th} element of the number array are for the same student.

Example 4 Parallel arrays: student names and numbers

Consider the following where two arrays hold information for 5 students: one array of names and the other an array of student numbers. For simplicity we initialize the arrays inline. The program prompts the user for a student number and displays the student's name. In order to get the name of the student the program goes through all the elements of `number` and when it finds a number matching the input, it displays the corresponding name in the other array.

Listing 1.4: Finding information in parallel arrays.

```java
import java.util.Scanner;
/**
 * Student information is in two arrays.
 * Find student number and report name.
 */
public class StudentInfo
{
    public static void main(String[] args){
        String[] name =
           {"Joe","Linda","Mary","Peter","Lee"};
        int[] number = {123, 222, 345, 567, 890};
        Scanner kb = new Scanner(System.in);
        System.out.println("Enter student number:
           ");
        int toFind = kb.nextInt();
        for (int i=0; i<number.length; i++)
            if (toFind==number[i])
                System.out.println(name[i]);
    }
}
```

This program performs what is usually called a *search* operation: scanning an array looking for a specific element. The program as it was written always iterates through the whole `number` array; normally a programmer would stop the iteration once the element has been found - that is left as an exercise.

1.6 Partially filled arrays

In our examples so far the arrays are completely full - every element has a value. In general we do not expect this to always be the case and so, for some applications, we keep track of how many locations are actually filled.

Example 5 Partially filled: calculate average

Suppose we need to calculate the average monthly sales. Since there are 12 months we use an array of length 12. We want a user to use the program at any time of year and so there may be fewer than 12 values. The program prompts the user for the monthly sales values, and requests the last value entered to be -1 (a *stopper* value). The program keeps track of how many elements are filled. Consider the following program and the points discussed after the listing:

Listing 1.5: Average sales for up to 12 months.

```java
import java.util.Scanner;
/**
 * From monthly sales calculate monthly average.
 */
public class MonthlySales
{
    public static void main(String[] args){
        double[] sales = new double[12];
        Scanner kb = new Scanner(System.in);
        System.out.println("Enter monthly sales"
            +" enter -1 after last value");
        int numberMonths=0;
        double aSale = kb.nextDouble(); //1st month
        while(aSale != -1) {
            sales[numberMonths++] = aSale;
            aSale = kb.nextDouble();
        }
        double sum = 0;
        for (int i=0; i<numberMonths; i++)
            sum+=sales[i];
        if (numberMonths>0) System.out.println(
            "average = "+sum/numberMonths);
23        }
24  }
```

The program exhibits some important features:

1. The `sales` array is of length 12 and the variable `numberMonths` keeps track of how many months of data the user provides.
2. Prior to the `while`, in line 13, the first sales amount is obtained
3. The `while` tests the value of the last sales amount obtained.
4. In the body of the `while` the previously obtained sales amount is placed into the array, and the next value is obtained.
5. Lines 19 and 20 accumulate the total sales
6. Testing for no months of data in line 21 prevents the program from crashing if the user entered -1 as the first value (division by zero).

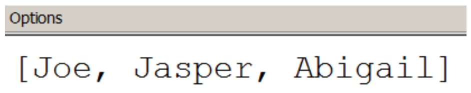
Arrays and ArrayLists

In some cases you may want to use the functionality of the `ArrayList` class but for whatever reason the data you are working with is in an array. It is easy to create an `ArrayList` from an array as shown in the program below.

Listing 1.6: Initializing an `ArrayList` from an array.

```java
 1 import java.util.ArrayList;
 2 /**
 3  * Create an ArrayList from an array of strings
 4  */
 5 public class ArrayListFromArray
 6 {
 7     public static void main(String[] args){
 8         // An array that will be used to provide
 9         //   initial data for an ArrayList
10         String[] name={"Joe","Jasper","Abigail"};
11         ArrayList<String> nameAL =
12                       new ArrayList(name.length);
13         // The add() method is used to append
14         //     an element to the ArrayList
15         for (String n: name) nameAL.add(n);
16         // Printing an ArrayList results in each
17         //   of its elements being displayed as
18         //   in a comma-separated list.
19         System.out.println(nameAL);
20     }
21 }
```

Line 15 is an enhanced `for` where each element of the array is added to the `ArrayList`. Line 19 prints the `ArrayList`. Note the output below and how the `ArrayList` is displayed as a comma-separated-values list embedded in square brackets `[ ]` - this is the default display for an `ArrayList`.



```
[Joe, Jasper, Abigail]
```

1.7 Array utilities in Java class libraries

Arrays are often used in programming and there are many important array algorithms. For instance, copying an array was discussed previously. The `System` class contains a method `arraycopy()` that can be used to copy a portion of one array to another. The method takes 5 arguments (in this order): name of the source array, starting element position in the source, the destination array, the starting element position in the destination, and the total number of elements to copy. For instance to copy all elements of the array t to the array s we could use:

```
    System.arraycopy(t, 0, s, 0, t.length);
```

There is a Java library class named `java.util.Arrays` that has additional methods which include:

1. `equals()`: Returns true if two arrays are equal to one another. The arrays are equal if they have the same number of elements and if corresponding elements are equal.
2. `sort()`: Rearranges the elements of an array so they are in ascending sequence.
3. `binarySearch()`: Returns the index of an element if it was found in a sorted array. Binary search is a type of search technique that takes advantage of the fact that an array is sorted. The general idea is to continually bisect the array looking for the required element. The process examines the middle element and determines if the required element is above or below the middle element; then the process continues on that subset of the array where the required element may be present. The process continues until the required value is found or there is nothing left to examine.
4. `fill()`: Assigns a specified value to every element of an array.

Example 6 Sorting and searching an array

The interested student is referred to the Java Class Library documentation for complete information regarding Arrays. Here, we demonstrate how one can sort an array and then search the array for a specific entry. Consider that we have an array of names. To simplify we shall initialize the array in the code. The program prompts the user for a name, performs a search, and then responds accordingly. Following the listing there are some remarks.

Listing 1.7: Initializing and displaying an array.

```
 1 import java.util.Arrays;
 2 import java.util.Scanner;
 3 /**
 4  * An array of names is sorted and then
 5  * searched for a specific name.
 6  */
 7 public class SortAndSearch
 8 {
 9     public static void main(String[] args){
10         String[] name =
11           {"Joe","Linda","Mary","Peter","Lee","Patricia"};
12         Arrays.sort(name);
13         Scanner kb = new Scanner(System.in);
14         System.out.println("Enter a name: ");
15         String toFind = kb.next();
16         int foundAt =
17             Arrays.binarySearch(name, toFind);
18         if (foundAt >= 0)
19             System.out.println(
20                 "Found in position "+foundAt);
21         else System.out.println("Not Found ");
22     }
23 }
```

Note the following points regarding SortAndSearch above:

1. The Arrays class is imported in line 1.
2. The `sort()` method is invoked in line 12. As a result the entries of name have been rearranged are are now sorted alphabetically.

3. In line 17 `binarySearch()` is used to search for the name entered by the user. If the value is not negative then that is the index where the name was found.

Exercises

1. Modify Example 1 to include a parallel array for the names of months. On 12 lines, one per month, display each month and its number of days.

2. Modify Example 2 to determine the minimum and the maximum of the 7 temperatures. Note that this is similar to Exercise 1 in the Section on the `for` statement, but in this case the elements are stored in an array.

3. Modify Example 3 so that it sorts the strings before they are displayed.

4. Modify lines 14-16 in Example 4 so that the loop stops if the number is found.

5. Modify Example 5 so that it displays the name of the month when sales were their largest.

6. Write a program to determine someone's score on a multiple-choice test having 12 questions. The program has two char arrays: correctAnswers[] and studentAnswers[]. The array correctAnswers holds the correct answers to the test. Use the following for correct answers:
   a b c d a b c d a b c d

   The student's answers will be provided by the user of the program. These must be stored in the array `studentAnswers[]`. After the student answers have been obtained the program must determine the student's score: the number of questions the student answered correctly.

   For example if the student answers are:
   a a a b b b c c c d d d
   then the score for this student is 4.

7. Write a program to analyze text such that each word (token) found is stored in an array. Use the file `Readme.txt`. Sort the array and display its contents.

Chapter 2

Arrays of Arrays

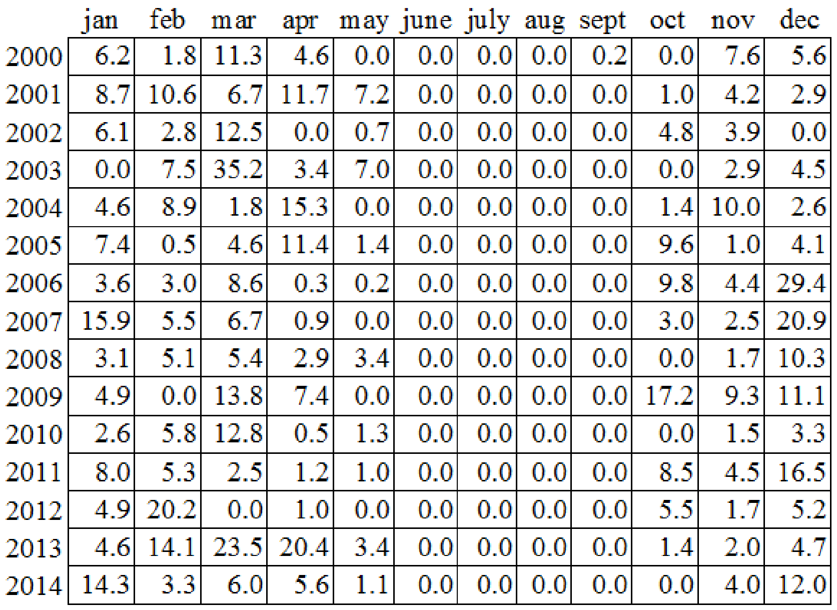
Java programmers frequently use an array (also called a one-dimensional array) to deal with a linear collection of elements (where the elements are of the same type). However there are times when a more complicated array structure is useful. For instance suppose we are keeping track of snowfall by month for Denver, Colorado, for the years 2000 through to 2014. We can represent this information readily in a tabular format - see Figure 2.1.

It is easy for someone to get information from such a table as that above. To do so you need to know the meanings of three things:

- What do the values in the cells of the table represent?
  - In this example *snowfall in inches.*
- What do the rows represent ?
  - In this example *years* from 2000 to 2014.
- what do the columns represent?
  - In this example *months* January, ... December.

One can use the table to find out the snowfall during some month and some year. If we want to obtain the snowfall for the month of February in 2005 we need to go the sixth row from the top and then to the second element from the left. There we see the value 0.5 ... that is, in February of 2005 Denver received a half inch of snow.

Figure 2.1: Monthly snowfall from 2000 to 2014 for Denver, Colorado.



	jan	feb	mar	apr	may	june	july	aug	sept	oct	nov	dec
2000	6.2	1.8	11.3	4.6	0.0	0.0	0.0	0.0	0.2	0.0	7.6	5.6
2001	8.7	10.6	6.7	11.7	7.2	0.0	0.0	0.0	0.0	1.0	4.2	2.9
2002	6.1	2.8	12.5	0.0	0.7	0.0	0.0	0.0	0.0	4.8	3.9	0.0
2003	0.0	7.5	35.2	3.4	7.0	0.0	0.0	0.0	0.0	0.0	2.9	4.5
2004	4.6	8.9	1.8	15.3	0.0	0.0	0.0	0.0	0.0	1.4	10.0	2.6
2005	7.4	0.5	4.6	11.4	1.4	0.0	0.0	0.0	0.0	9.6	1.0	4.1
2006	3.6	3.0	8.6	0.3	0.2	0.0	0.0	0.0	0.0	9.8	4.4	29.4
2007	15.9	5.5	6.7	0.9	0.0	0.0	0.0	0.0	0.0	3.0	2.5	20.9
2008	3.1	5.1	5.4	2.9	3.4	0.0	0.0	0.0	0.0	0.0	1.7	10.3
2009	4.9	0.0	13.8	7.4	0.0	0.0	0.0	0.0	0.0	17.2	9.3	11.1
2010	2.6	5.8	12.8	0.5	1.3	0.0	0.0	0.0	0.0	0.0	1.5	3.3
2011	8.0	5.3	2.5	1.2	1.0	0.0	0.0	0.0	0.0	8.5	4.5	16.5
2012	4.9	20.2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	5.5	1.7	5.2
2013	4.6	14.1	23.5	20.4	3.4	0.0	0.0	0.0	0.0	1.4	2.0	4.7
2014	14.3	3.3	6.0	5.6	1.1	0.0	0.0	0.0	0.0	0.0	4.0	12.0

2.1 Two-dimensional arrays in Java

The snowfall table can be stored in a Java array, but in this case we would use a two-dimensional array and it could be defined as:

```
double [][] snowfall;
```

Each element of the array must be of the same type . . . in this case they are `double`s. Notice the two pairs of square braces, [ ] and [ ], in the declaration statement. There are two pairs because we will use two subscripts to reference an element of the table. To obtain the element for February 2005 we use `snowfall[5][1]`. Recall with Java that subscript values begin at 0 and so the sixth row is row 5 and the second column is column 1. Its a lengthy statement but to initialize `snowfall` we could use:

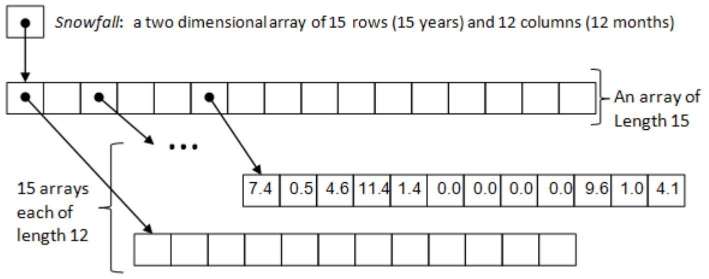
```
private double[][] snowfallInInches ={
{6.2,1.8,11.3,4.6,0,0,0,0,0.2,0,7.6,5.6}
{8.7,10.6,6.7,11.7,7.2,0,0,0,0,1,4.2,2.9},
{6.1,2.8,12.5,0,0.7,0,0,0,0,4.8,3.9,0.0},
{0,7.5,35.2,3.4,7,0,0,0,0,0,2.9,4.5},
{4.6,8.9,1.8,15.3,0,0,0,0,0,1.4,10,2.6},
{7.4,0.5,4.6,11.4,1.4,0,0,0,0,9.6,1,4.1},
{3.6,3,8.6,0.3,0.2,0,0,0,0,9.8,4.4,29.4},
{15.9,5.5,6.7,0.9,0,0,0,0,0,3,2.5,20.9},
{3.1,5.1,5.4,2.9,3.4,0,0,0,0,0,1.7,10.3},
{4.9,0,13.8,7.4,0,0,0,0,0,17.2,9.3,11.1},
{2.6,5.8,12.8,0.5,1.3,0,0,0,0,0,1.5,3.3},
{8,5.3,2.5,1.2,1,0,0,0,0,8.5,4.5,16.5},
{4.9,20.2,0,1,0,0,0,0,0,5.5,1.7,5.2},
{4.6,14.1,23.5,20.4,3.4,0,0,0,0,1.4,2,4.7},
{14.3,3.3,6,5.6,1.1,0,0,0,0,0,4,12},
};
```

Some important points about `snowfall`:

1. The data is presented in row order. And for each row the data is presented in column order. There are 16 rows of data in the table.
2. Note how values are separated by commas and the data for each row is enclosed in a pair of curly braces, { }. Each row contains 12 values, one per month.
3. Only the snowfall amounts are in the array. A program must *know* the year that a row represents and *know* the month that a column represents.

When the JVM creates the array, it actually creates an array where each element is an array. The diagram in Figure 2.2 shows how the JVM organizes a two dimensional array in memory as an array of arrays.

Figure 2.2: Two dimensional array - an array of arrays.


Snowfall: a two dimensional array of 15 rows (15 years) and 12 columns (12 months)
An array of Length 15
...
15 arrays each of length 12
7.4
0.5
4.6
11.4
1.4
0.0
0.0
0.0
0.0
9.6
1.0
4.1

When a program references an array element the JVM accesses the storage structure using the subscripts in sequence. So for `snowfall[5][1]` the JVM uses the first subscript, `5`, to access the sixth element of the 15 element array; then the JVM uses the second subscript, `1`, to access the second element of a 12-element array where the value 0.5 is stored in the above figure.

In the foregoing we initialized the array in a declaration statement. That is not always appropriate. If we were obtaining the values from input we would first of all declare the array of the appropriate size and then proceed to read data into the array. The declaration statement for our Denver snowfall example would be:

```
    private double [][] snowfall = new double[15][12];
```

The program in Listing 2.1 reads the data from a file named SnowfallInInches.txt from the same folder as where the program is located. In this program you should note the following:

- line 10 declares the two dimensional array with 15 rows and 12 columns.
- lines 19 to 24 read the data from the file.
  - The outer loop controls the row subscript. Notice the use of the `length` field ... the number of rows is `snowfall.length()`.
  - The inner loop controls the column subscript. Notice again the use of the `length` field. Each row is in fact an array, and the ith row is referenced by `snowfall[i]` ... the number of elements in the ith row is `snowfall[i].length()`.
  - In line 22 the value read is stored in the ith row and jth column of `snowfall`. Recall that the two dimensional storage structure the JVM creates is an array of arrays. And so in terms of the storage structure its true to say that the value is stored as the jth element of the ith array of `snowfall`.
- Lines 26 to 32 display the values in a tabular format.

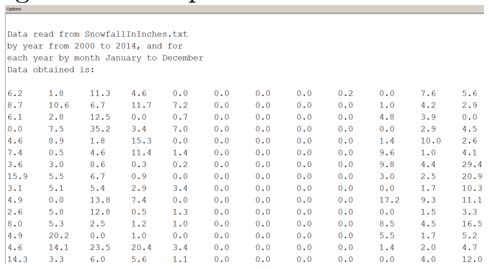
The output from the program is shown following the listing

Listing 2.1: Reading array data

```
1  import java.util.Scanner;
2  import java.io.File;
3  import java.io.FileNotFoundException;
4
5  public class Snowfall
6  {
7      public static void main(String[] args)
```

```java
    throws FileNotFoundException {
        // an array of 15 rows and 12 columns
        double [][] snowfall = new double[15][12];
        // get data from file
        Scanner f = new Scanner(
            new File("SnowfallInInches.txt"));
        System.out.println("\nData read from "
            +"SnowfallInInches.txt "+
            "\nby year from 2000 to 2014, and for "+
            "\neach year from January to December");
        // outer loop controls the row subscript
        for (int i=0; i<snowfall.length; i++){
            // inner loop controls the column
            for (int j=0;j<snowfall[i].length;j++){
                snowfall[i][j] = f.nextDouble();
            }
        }
        // display the contents of the table by year
        System.out.println("Data obtained is:");
        for (int i=0; i<snowfall.length; i++){
            System.out.println();
            for (int j=0;j<snowfall[i].length;j++){
                System.out.print(snowfall[i][j]+"\t");
            }
        }
    }
}
```

Figure 2.3: Output: Snowfall in Denver.

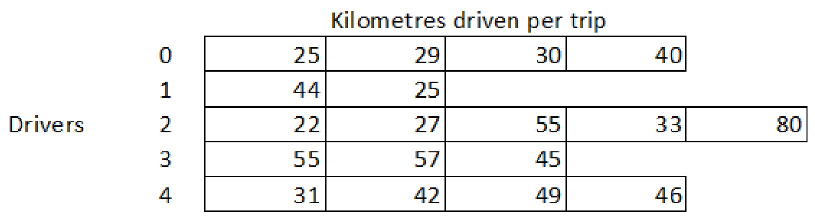


```
Data read from SnowfallInInches.txt
by year from 2000 to 2014, and for
each year by month January to December
Data obtained is:

6.2     1.8     11.3    4.6     0.0     0.0     0.0     0.0     0.2     0.0     7.6     5.6
8.7     10.6    6.7     11.7    7.2     0.0     0.0     0.0     0.0     1.0     4.2     2.9
6.1     2.8     12.5    0.0     0.7     0.0     0.0     0.0     0.0     4.8     3.9     0.0
0.0     7.5     35.2    3.4     7.0     0.0     0.0     0.0     0.0     0.0     2.9     4.5
4.6     8.9     1.8     15.3    0.0     0.0     0.0     0.0     0.0     1.4     10.0    2.6
7.4     0.5     4.6     11.4    1.4     0.0     0.0     0.0     0.0     9.6     1.0     4.1
3.6     3.0     8.6     0.3     0.2     0.0     0.0     0.0     0.0     9.8     4.4     29.4
15.9    5.5     6.7     0.9     0.0     0.0     0.0     0.0     0.0     3.0     2.5     20.9
3.1     5.1     5.4     2.9     3.4     0.0     0.0     0.0     0.0     0.0     1.7     10.3
4.9     0.0     13.8    7.4     0.0     0.0     0.0     0.0     0.0     17.2    9.3     11.1
2.6     5.8     12.8    0.5     1.3     0.0     0.0     0.0     0.0     0.0     1.5     3.3
8.0     5.3     2.5     1.2     1.0     0.0     0.0     0.0     0.0     8.5     4.5     16.5
4.9     20.2    0.0     1.0     0.0     0.0     0.0     0.0     0.0     5.5     1.7     5.2
4.6     14.1    23.5    20.4    3.4     0.0     0.0     0.0     0.0     1.4     2.0     4.7
14.3    3.3     6.0     5.6     1.1     0.0     0.0     0.0     0.0     0.0     4.0     12.0
```

2.2 Ragged Arrays

Recall how two dimensional arrays are actually arrays of arrays. Its possible then that rows can have different numbers of elements. For example suppose we have five drivers who drive trucks delivering goods, and for each driver and delivery we keep track of the kilometres they drive. If it is the case that the number of deliveries varies for these drivers we can use a two dimensional array; consider the following sample data:

Figure 2.4: Five drivers with varying numbers of trips.



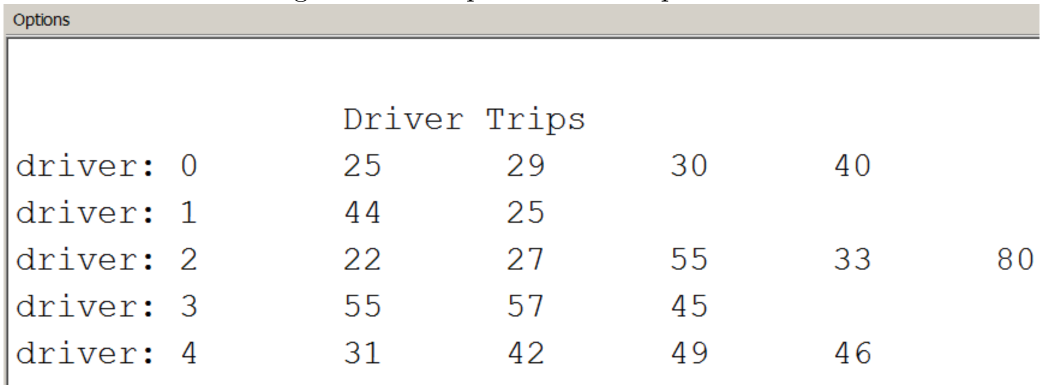
		Kilometres driven per trip				
	0	25	29	30	40	
	1	44	25			
Drivers	2	22	27	55	33	80
	3	55	57	45		
	4	31	42	49	46	

The program in Listing 2.2 initializes the array with 5 rows, one per driver, and varying elements for each array that makes up a row. Following the program listing is the output from running the program.

Listing 2.2: Reading array data

```java
import java.util.Scanner;
public class DriversTrips
{
    public static void main(String[] args ){
        // 2D array with varying number
        //    of elements per row
        int[][] trips ={
            {25, 29, 30, 40},
            {44, 25},
            {22, 27, 55, 33, 80},
            {55, 57, 45},
            {31, 42, 49, 46}
        };
        System.out.println("\n\t\tDriver Trips");
        // number of drivers = number of rows
        //    is trips.length
        for (int i=0; i<trips.length; i++){
            System.out.print("driver: "+i+"\t");
            // number of trips for ith driver
            //    is trips[i].length
            for (int j=0;j<trips[i].length;j++){
                System.out.print(trips[i][j]+"\t");
            }
            System.out.println();
        }
    }

}
```

Figure 2.5: Output: DriversTrips.



```
                Driver Trips
driver: 0       25      29      30      40
driver: 1       44      25
driver: 2       22      27      55      33      80
driver: 3       55      57      45
driver: 4       31      42      49      46
```

2.3 Examples

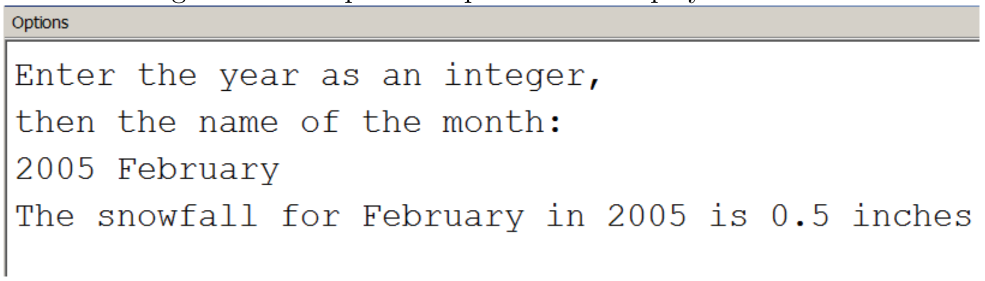
Example 1 Accessing a specific array element

Consider the following program that displays the snowfall for a specific year and a month obtained from the user. The program *converts* the year and month into appropriate subscript (`int`) values. The output for a sample run follows.

Listing 2.3: Display a specific cell in a 2D array

```java
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class DisplaySnowfall
{
    public static void main(String[] args)
    throws FileNotFoundException {
        // initialize the snowfall array
        double [][] snowfall = new double[15][12];
        Scanner f = new Scanner(
                new File("SnowfallInInches.txt"));
        for (int i=0; i<snowfall.length; i++){
            // inner loop controls the column ac
            for (int j=0;j<snowfall[i].length;j++){
                snowfall[i][j] = f.nextDouble();
            }
        }
        // prompt user ... display snowfall
        System.out.println("Enter the year as "+
            "an integer, "+
            "\nthen the name of the month:");
        Scanner kb = new Scanner(System.in);
        // convert year to proper subscript
        int year = kb.nextInt()-2000;
        String month = kb.next();
        // convert month to proper subscript
        int monthInt = convertMonth(month);
        System.out.print("The snowfall for "+
            month+" in "+ (2000+year)+" is "+
            snowfall[year][monthInt]+" inches");
    }

    public static int convertMonth(String month){
        int monthInt;
        switch (month.toLowerCase()){
            case "january": monthInt = 0; break;
            case "february": monthInt = 1; break;
            case "march": monthInt = 2; break;
            case "april": monthInt = 3; break;
            case "may": monthInt = 4; break;
            case "june": monthInt = 5; break;
            case "july": monthInt = 6; break;
            case "august": monthInt = 7; break;
            case "september": monthInt = 8; break;
            case "october": monthInt = 9; break;
            case "november": monthInt =  10; break;
            case "december": monthInt = 11; break;
            default: monthInt = -1; // bad month
               name
        }
        return monthInt;
    }
}
```

Figure 2.6: Output: Sample run for DisplaySnowfall.



```
Enter the year as an integer,
then the name of the month:
2005 February
The snowfall for February in 2005 is 0.5 inches
```

Example 2 Accessing all elements in a row

Consider the following program that displays the total snowfall for 2005. This program accesses all elements in a specific row. Of particular importance to this program is the use of the enhanced `for` in lines 24-25 to access the elements in the row for 2005:

```
    for (double s : snowfall[year])
        total+=s;
```

Listing 2.4: Display a specific cell in a 2D array

```
 1 import java.util.Scanner;
 2 import java.io.File;
 3 import java.io.FileNotFoundException;
 4 
 5 public class TotalSnowfall2005
 6 {
 7     public static void main(String[] args)
 8     throws FileNotFoundException {
 9         // initialize the snowfall array
10         double [][] snowfall = new double[15][12];
11         Scanner f = new Scanner(
12                 new File("SnowfallInInches.txt"));
13         for (int i=0; i<snowfall.length; i++){
14             // inner loop controls the column ac
15             for (int j=0;j<snowfall[i].length;j++){
16                 snowfall[i][j] = f.nextDouble();
17             }
18         }
19         // display snowfall for 2005
20         // convert year to proper subscript
21         int year = 2005-2000;
22         // get total of values in the row for 2005
23         double total = 0;
24         for (double s : snowfall[year])
25             total+=s;
26         System.out.print("The snowfall for 2005 "+
27             " is "+total+" inches");
28     }
29 }
```

Example 3 Accessing all elements in a column

Consider the following program that displays the average snowfall for the month of February. In lines 22-23 this program accesses elements in the second column of each row.

Listing 2.5: Display a specific cell in a 2D array

```
 1 import java.util.Scanner;
 2 import java.io.File;
 3 import java.io.FileNotFoundException;
 4 
 5 public class AverageFebruarySnowfall
 6 {
 7     public static void main(String[] args)
 8     throws FileNotFoundException {
 9         // initialize the snowfall array
10         double [][] snowfall = new double[15][12];
11         Scanner f = new Scanner(
12                 new File("SnowfallInInches.txt"));
13         for (int i=0; i<snowfall.length; i++){
14             // inner loop controls the column ac
15             for (int j=0;j<snowfall[i].length;j++){
16                 snowfall[i][j] = f.nextDouble();
17             }
18         }
19         // get total of values in for Februar
20         // by accessing second element of each row
21         double total = 0;
22         for (int i=0; i<snowfall.length; i++)
23             total+=snowfall[i][1];
24         System.out.print("The average February "+
25             "snowfall  is "+(total/snowfall.length)+
26             " inches");
27     }
28 }
```

Example 4 Ragged arrays:using row length

Consider the following program that displays the number of trips per driver. In lines 14-15 this program determines the number of trips for driver `i` by just using the `length` field for the array comprising row i.

Listing 2.6: Display total number of trips for each driver.

```
 1 import java.util.Scanner;
 2 public class TripsPerDriver
 3 {
 4     public static void main(String[] args ){
 5         int[][] trips ={
 6             {25, 29, 30, 40},
 7             {44, 25},
 8             {22, 27, 55, 33, 80},
 9             {55, 57, 45},
10             {31, 42, 49, 46}
11         };
12         // one line for each driver
13         for (int i=0; i<trips.length; i++){
14             System.out.println("driver "+i+
15                 " made  "+trips[i].length+
16                 " deliveries");
17         }
18     }
19
20 }
```

Example 5 Representing a matrix

In mathematics there is a structure called a matrix that, in Java terms, is just a two dimensional array. Operations such as addition and multiplication are defined for matrices where certain properties of the matrices involved must be true. For example, two matrices with the same number of rows and columns can be added together to produce a third matrix. The following program initializes two matrices A and B, and then adds them producing a third matrix, C.

In this program examine the loops in lines 25-27 where corresponding elements are added. The program uses a method `displayMatrix`, lines 33-41, that accepts 2 parameters: a heading to display, and a matrix to display.

Listing 2.7: Display a specific cell in a 2D array

```
 1 import java.util.Scanner;
 2 public class MatrixAddition
 3 {
 4     public static void main(String[] args) {
 5         int[][] a ={
 6             {1, 2, 3, 4},
 7             {1, 2, 3, 4},
 8             {1, 2, 3, 4}
 9         };
10         int[][] b ={
11             {1, 2, 3, 4},
12             {5, 6, 7, 8},
13             {9, 10, 11, 12}
14         };
15         int [][] c ={
16             {0, 0, 0, 0},
17             {0, 0, 0, 0},
18             {0, 0, 0, 0}
19         };;
20         // C = A + B
21         // For each c[i][j] in C
22         //    c[i][j] = a[i][j]+b[i][j]
23         // A and B must have the same
24         //    number of rows and columns
25         for (int i=0; i< a.length; i++)
```

```
            for (int j=0; j<a[i].length; j++)
                c[i][j] =a [i][j] + b[i][j];
        // display the 3 matrices
        displayMatrix("A = ",a);
        displayMatrix("B = ",b);
        displayMatrix("C = ",c);
    }
    public static void displayMatrix(
        String heading,int[][] m){
        System.out.println(heading);
        for (int i=0; i<m.length; i++){
            for (int j=0; j<m[i].length; j++)
                System.out.print(m[i][j]+"\t");
            System.out.println();
        }
    }
}
```

2.4 Higher Dimensions

You can define and use arrays with any number of dimensions. For instance suppose we are recording values for each second of each minute of each hour in a day, we could use a 3-dimensional array such as:

```
    double [][][] obs = new double[24][60][60];
```

Of course the storage structure used in this case would involve an array of 24 elements, where each of those is an array of 60 elements and where each of those is an array of 60 elements.

2.5 Exercises

1. Write a program that displays the names of the months for the year 2005 when the snowfall in Denver exceeded 1 inch.

2. Write a program that displays the name of the month in the year 2005 when Denver received the greatest amount of snow.

3. Write a program that calculates and displays the total number of kilometres driven by each driver.

4. Write a program that calculates and displays the total number of kilometres driven (totalled over all drivers).

5. Write a program that displays each driver's name and the total number of kilometres driven. As well as the two dimensional array `trips`, your program must include a one dimensional array containing driver names.

6. Modify the program in Example 5 so that it forms the product of A and B. If A has n rows and m columns and B has m rows and p columns, then the product $A \times B$ yields a third matrix of n rows and p columns. Each element of C is a sum of products involving the i$^{\text{th}}$ row of A and the j$^{\text{th}}$ column of B:
   $c_{i,j} \ = \ \sum_{k=1}^{n} \ a_{i,k} \ b_{k,j}$

7. Suppose A and B are two matrices as described in the previous question. However now let A be an $m \times m$ identity matrix. An identity matrix is one that has 1s on the diagonal and 0s everywhere else. That is,
   $A_{i,j} \ = \ 1$ where $i = j$ and
   $A_{i,j} \ = \ 0$ where $i \neq j$
   For example, if $m = 4$ we have $A =$

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

   The program then multiplies matrix A by the matrix B ... the result should be B.

8. Write a method `initSequentialValues(...}` that sets the values of the elements of a matrix to the values 1, 2, 3, ... For example suppose

A is a matrix of 4 rows and 5 columns. Then the result of calling
`initSequentialValues(A)`
we have $A =$

$$\begin{vmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{vmatrix}$$

9. Similar to the previous question, but place the sequential values in column order. Write a method `initSequentialValues(...}` that sets the values of the elements of a matrix to the values 1, 2, 3, ... For example suppose A is a matrix of 4 rows and 5 columns. Then the result of calling `initSequentialValues(A)`
we have $A =$

$$\begin{vmatrix} 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{vmatrix}$$

Chapter 3

Validation and Text Manipulation

In this chapter we look at issues related to verifying the correctness of your program, verifying the validity of argument values passed into a method, verifying the validity of field values, and the general concern of manipulating textual data.

3.1 Testing and the assert statement

When we develop a program there is the important task of verifying the program works as required. Often a programmer will insert multiple `println` statements to display program data, or to track the progression of some algorithm. In addition to debugging a program this way a programmer can also use the Java `assert` statement. This statement has two forms:

- `assert` $expression_1$;
- `assert` $expression_1$: $expression_2$;

where $expression_1$ is a boolean expression and $expression_2$ is a value. If $expression_1$ evaluates to false then your Java progam will terminate immediately with an `Assertion` error. If the $expression_1$ evaluates to true then nothing happens. In the second form of the assert statement $expression_2$ provides a message to be displayed when the program is terminated.

Example 1. Calculate age

Suppose we need to calculate a person's age in years. Given a person's birthdate we can easily make this calculation. In Java 1.8 two classes introduced were `LocalDate` and `Period`. These classes model dates according to the ISO-8601 calendar system. Using `LocalDate` we can instantiate an instance representing a particular date such as *today*, or, someone's birthday. `Period` can be used to create an interval that has a start date and an end date, such as the period between someone's birthday and today. `Period` has several utility methods; for instance `getYears()` returns the number of years in a period. The following method, given someone's date of birth in the format `YYYY-MM-DD` calculates age in years.

```java
/**
     * getAge - determines age in years
     *
     * @param  yyyymmdd   birthdate YYYY-MM-DD
     * @return age in years
     */
    public static int getAge(String yyyymmdd){
        int yyyy =
           Integer.parseInt(yyyymmdd.substring(0,4));
        int mm   =
           Integer.parseInt(yyyymmdd.substring(5,7));
        int dd   =
           Integer.parseInt(yyyymmdd.substring(8,10));
        // age is difference between today and the
           birthday
        LocalDate birthday = LocalDate.of(yyyy, mm,
           dd);
        LocalDate today    = LocalDate.now();
        Period p = Period.between(birthday, today);
        return p.getYears(); // age in years
    }
```

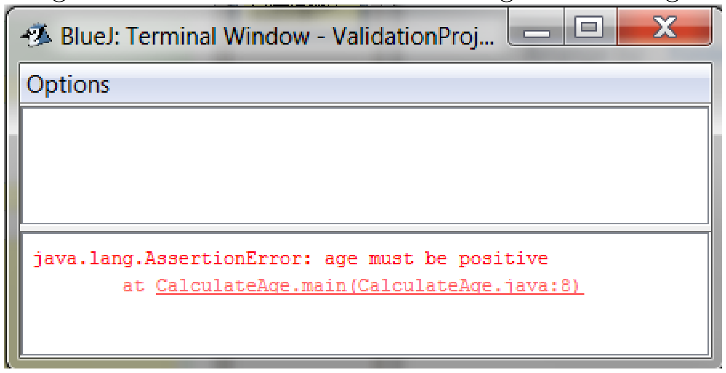
Please study the code above to know how it works; it does perform the calculation properly. However there are cases where we can get a result we consider to be inappropriate. For example, suppose we use this method in a situation where we are calculating ages of employees. In this situation we expect someone's age to be positive - a non-positive age indicates something is wrong with our logic or with our data.

In the program `CalculateAge` below we include an `assert` statement to trap the case where the calculated age is not positive (we have used the birth date `2090-01-01`). The result of executing this code is shown after the program listing. Note that the program has been terminated with an Assertion error.

Listing 3.1: Use assert to recognize a problem

```java
import java.util.Scanner;
import java.time.LocalDate;
import java.time.Period;
public class CalculateAge
{
    public static void main(String[] args){
        int age = getAge("2090-01-01");
        assert age > 0 : "age must be positive";
        System.out.println(age);
    }

    /**
     * getAge - determines age in years
     *
     * @param  yyyymmdd   birthdate YYYY-MM-DD
     * @return age in years
     */
    public static int getAge(String yyyymmdd){
        int yyyy =
           Integer.parseInt(yyyymmdd.substring(0,4));
        int mm   =
           Integer.parseInt(yyyymmdd.substring(5,7));
        int dd   =
           Integer.parseInt(yyyymmdd.substring(8,10));
        // age is difference between today and the
           birthday
        LocalDate birthday = LocalDate.of(yyyy, mm,
           dd);
        LocalDate today    = LocalDate.now();
        Period p = Period.between(birthday, today);
        return p.getYears(); // age in years
        }
}
```

Figure 3.1: Assertion error on running CalculateAge.



```
java.lang.AssertionError: age must be positive
        at CalculateAge.main(CalculateAge.java:8)
```

Assert statements are used by programmers when they are developing and testing code. They should not be used in *production* code. BlueJ, by default, enables assertions. This default behaviour is reasonable for an environment that is a learning environment, but assertions can be turned off. At the time of writing assertions can be disabled by changing a flag in the BlueJ `defs` file from `ea` to `da`.

3.2 Validating parameters

When a method has been coded there is an expectation that the argument values passed in are appropriate. For instance, the `getAge` method in the previous section expects a string argument to contain a valid date. What happens if it does not? Examine that code and you should agree that the method will fail and the program will terminate. The programmer would then need to investigate to determine why the program failed. Instead of leaving things that way we can redevelop `getAge` so it checks its parameters and when they are invalid it could fail, but indicate to us the nature of the problem, and the programmer who is debugging the code will know where to begin her/his investigation.

Whenever you develop a method that has parameters you should know what constitutes valid values. You should develop code to check for improper values and, if that is the case, terminate execution in a planned way instead of an unplanned way.

Example 2. Throwing an illegal argument exception

`getAge` requires the value of `yyyymmdd` to be a string that has 4 digits, a dash, 2 digits (00 to 12), a dash, followed by 2 more digits (00 to 31, depending on the month). We will base our validation on some character processing techniques we are aware of. We consider the following rules for a valid `yyyymmdd` value:

- yyyy, mm, dd are separated by a single dash
- yyyymmdd is a string of length 10
- yyyy can be any 4 digits
- mm can be any 2 digits (really these should be in the range 01 ... 12)
- dd can be any 2 digits (really these should be appropriate for the month)
- we ignore leap years

Some sample values that are invalid according to the above are:

- `"1990-1-1"` is not 10 characters long
- `"abcd-01-01"` has non-numeric digits
- `"2010/01/01"` has / as a separator instead of a dash

We have seen before that certain errors result in exceptions that cause a program to terminate abruptly. For instance, if an array subscript is *out of bounds* the program is terminated with an ArrayOutOfBoundsException exception. The `getAge` method in the program below introduces the Java exception `IllegalArgumentException`; this exception is thrown if the argument passed in does not pass the checks performed. If this exception is thrown then a program terminates immediately.

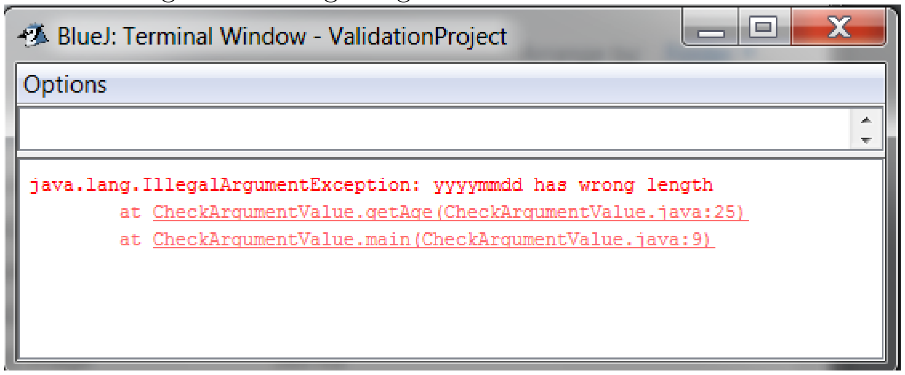
Listing 3.2: Throwing an exception

```java
import java.util.Scanner;
import java.time.LocalDate;
import java.time.Period;
import java.lang.IllegalArgumentException;
public class CheckArgumentValue
{
    public static void main(String[] args){
        // bad value passed to getAge
        int age = getAge("2090-01-1");
        assert age > 0 : "age must be positive";
        System.out.println(age);
    }

    /**
     * getAge - determines age in years
     *
     * @param  yyyymmdd   birthdate YYYY-MM-DD
     * @return age in years
     */
    public static int getAge(String yyyymmdd){
        String arg = yyyymmdd;
        // check length
        boolean valid = arg.length()==10;
        if (!valid)
           throw new
              IllegalArgumentException("yyyymmdd
              has wrong length");
        // check for dashes
        valid = arg.charAt(4)=='-' &&
           arg.charAt(7)=='-' ;
        if (!valid)
           throw new
```

```
                 IllegalArgumentException("yyyymmdd
                 does not have dashes in correct
                 places");
30          // check digit positions are numeric
31          for (int i=0; i<arg.length() && valid; i++)
32             valid = ((i==4 || i==7) ||
                 Character.isDigit(arg.charAt(i)));
33          if (!valid)
34             throw new
                 IllegalArgumentException("yyyymmdd is
                 not numeric");
35          // the following executes only if no
               exception was thrown above
36          arg = arg.replaceAll("-",""); // remove
               dashes
37          int yyyy =
               Integer.parseInt(yyyymmdd.substring(0,4));
38          int mm   =
               Integer.parseInt(yyyymmdd.substring(5,7));
39          int dd   =
               Integer.parseInt(yyyymmdd.substring(8,10));
40          // age is difference between today and the
               birthday
41          LocalDate birthday = LocalDate.of(yyyy, mm,
               dd);
42          LocalDate today    = LocalDate.now();
43          Period p = Period.between(birthday, today);
44          return p.getYears(); // age in years
45      }
46  }
```

Below is the output produced when the program runs. Note the exception includes the informative message and the line number where the exception was thrown.

Figure 3.2: Illegal argument detected at run time.



```
java.lang.IllegalArgumentException: yyyymmdd has wrong length
        at CheckArgumentValue.getAge(CheckArgumentValue.java:25)
        at CheckArgumentValue.main(CheckArgumentValue.java:9)
```

3.3 Manipulating Text

In Part 1 of this text several methods of the `String` and `Character` classes were presented and used. Here we introduce two more methods of the `String` class which are quite useful in practical coding. Following this we discuss the `StringBuilder` class that provides capabilities similar to String.

3.3.1 split and toCharArray

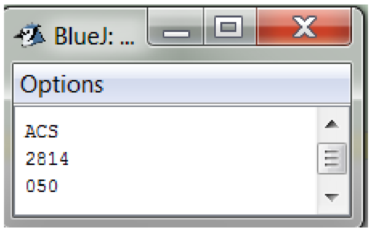
`toCharArray` is used to convert a string into an array of `char`. For instance the code snippet below counts the number of numeric characters in a string `s`.

```
        // count the number of digits in s
        char[] x = s.toCharArray();
        int count = 0;
        for (char c: x)
           if (Character.isDigit(c)) count++;
```

The `split` method is used to split a string into an array of strings based on a splitting expression. For example the following code snippet creates 3 strings based on `"-"` as a delimiter.

```
        // split a string based on the delimiter "-"
        String course = "ACS-2814-050";
        String[] parts = course.split("-");
        for (String p: parts) System.out.println(p);
```

The output of the code snippet follows - the string `"ACS-2814-050"` was split into 3 strings, each stored as an element of the array `parts`:



```
ACS
2814
050
```

The argument provided to the `split` method is formally called a *regular expression*. Regular expressions are string patterns that can be used for matching purposes. Regular expressions range from simple to very complex

- at some time you may be interested in pursuing some more advanced expressions, but for the time being we list a few simpler patterns below. Note that `[` and `]` are used to specify any character within the brackets, `"+"` is used to specify one or more of a preceding character, and `-` is used to specify a range of characters.

required purpose	pattern
to match a comma	`","`
to match a space	`" "`
to match multiple spaces	`" +"`
to match a dash	`"-"`
to match a dash followed by a comma	`"-,"`
to match a dash or a comma	`"[-,]"`
to match a digit	`"[0-9]"`
to match a letter	`"[a-zA-Z]"`

Example 3. Capitalize and compress words

Suppose we have a line of text where words are separated by spaces. We want to form a new word which is the catenation of the original words and where only the first letter of each word is is capitalized. For instance if the line is

`retail sales tax`

then the program creates the word

`RetailSalesTax`

The following program uses the simple regular expression `" "` (a pattern for a single space) and the `split` method to split the line of text into an array of strings. Each word is then catenated in turn to the result (with its first letter forced to be a capital letter).

Listing 3.3: splitting text on a space

```java
public class CatenateWords
{
    public static void main(String[] args){
        String course = "retail    sales      tax";
        String result = formNewWord(course);
        System.out.println(result);
    }

    public static String formNewWord(String line){
        String newWord = "";
        // for each word
        for(String s: line.split(" +"))
            // catenate to previous result
            newWord +=
               Character.toUpperCase(s.charAt(0)) +
               s.substring(1).toLowerCase();
        return newWord;
    }
}
```

Example 4. Validating the format of a SIN

Suppose we want to verify that a social insurance number is properly composed as 3 digits, a dash, 3 digits, a dash, 3 digits. For example, `"659-123-999"` is a properly formatted SIN. The following program splits the SIN into parts based on a dash (`"-"`), and then checks to see that each part consists of 3 digits.

Listing 3.4: splitting text on a dash and validating elements are numeric

```
public class ValidateFormat
{
    public static void main(String[] args){
        String sin = "659-123-999";
        if (validateSIN(sin))
            System.out.println(sin+" is correctly
               formed");
        else
            System.out.println(sin+" is not
               correctly formed");
    }

    public static boolean validateSIN(String sin){
        // ensure a value was passed in
        boolean valid = !(sin == null);
        if (valid) {
            String[] sinParts;
            sinParts = sin.split("-");
            // must have 3 parts
            valid &= sin.length()==11;
            valid &= sinParts.length==3;
            if (valid) {
                // remove the dashes
                String temp = sin.replace("-","");
                for (char c: temp.toCharArray())
                    valid &= Character.isDigit(c);
            }
        }
        return valid;
    }
}
```

3.3.2 StringBuilder

Java strings are said to be immutable; this means that a string value, once initialized, cannot be changed. Of course we have seen that a variable of type `String` can have one value and then through, say an assignment statement, have its value changed. Consider the statements

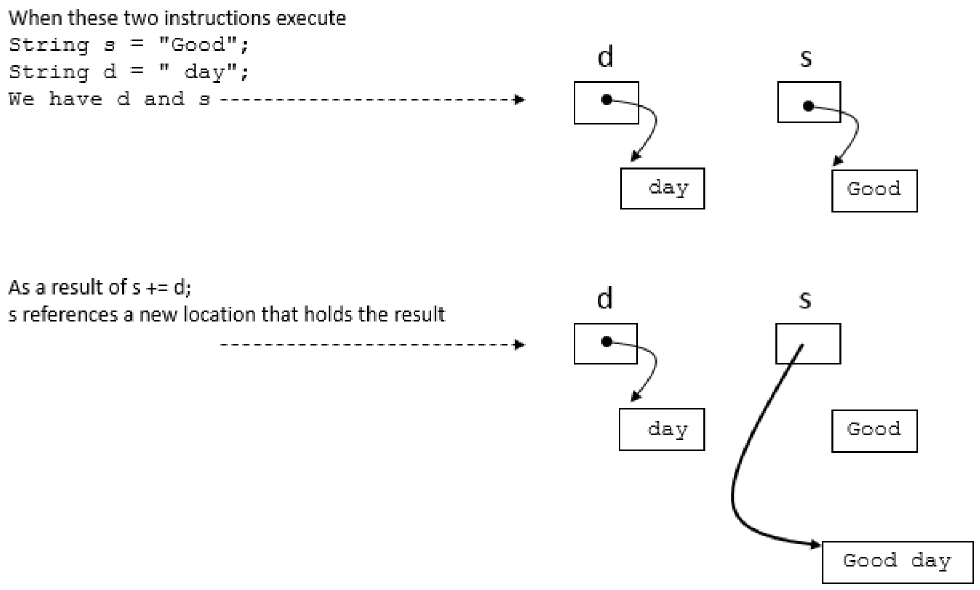
```
String s = "Good";
String d = " day";
s += d;
```

After the first statement is executed `s` references a location containing the word `"Good"`. As a result of the second statement the variable d references a location containing `" day"`. From the programmer's perspective the third statement clearly causes the value associated with `s` to change. The JVM performs the following steps when the third statement executes:

- another area of memory is allocated for `s` - large enough to hold the result of the catenation
- the contents of the string that `s` currently refers to is copied into the new area (`"Good"` is copied)
- the contents of the string that `d` refers to then copied into the new area (`" day"` is copied)
- the variable `s` is changed so it references the new area.

We can visualize the above steps with the following diagram.

Figure 3.3: When a string variable gets a new value, a new area of memory is allocated.



For many programming situations immutable strings work very well; for instance when a line of output is being established through catenation. However, if a program may do a great deal of string manipulation there is another class, `StringBuilder`, which for many operations does not have the same overhead as String.

An instance of `StringBuilder` is created using constructors. We show three ways to instantiate an instance of `StringBuilder`:

```
// a StringBuilder instance has an
// initial space allotment.
// that space can grow larger as necessary.
// If no arguments
// ... space for 16 characters
StringBuilder sb1 = new StringBuilder();
// If a string is provided
// ... space for that string plus 16
StringBuilder sb2 = new
   StringBuilder("Good");
// If a number is provided
// ... space for the size specified
StringBuilder sb3 = new StringBuilder(8);
```

Some `StringBuilder` methods are listed in the next table. Note many of these are the same or similar to those of the `String` class; the last five methods listed starting with `append` are not defined for `String`.

Following the table we give an example using `StringBuilder` to build up a string through catenation. With `StringBuilder` we cannot use the catenation operator ("+"), instead we use the `append` method.

Useful StringBuilder methods		
method name	type	description
charAt(. . .)	char	returns the character at a specified position
length()	int	returns the length of a string
indexOf(. . .)	int	determines where a string starts
substring(. . .)	String	returns a substring
append(. . .)	String	appends a string to the current string
insert(. . .)	String	inserts a string
delete()	String	removes a string
replace()	String	replaces a substring
reverse()	String	the instance is replaced by its reverse

Table 3.1: Some of the useful StringBuilder methods

Example 5. Catenating strings

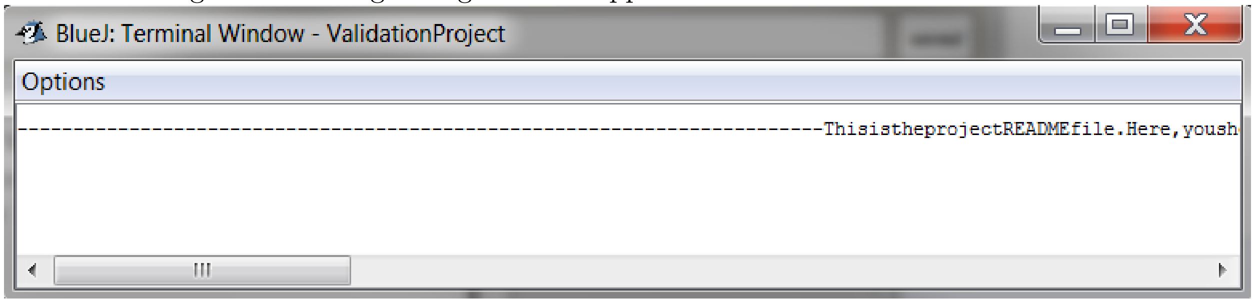
This example reads the file Readme.txt token-by-token catenating each token to the next. The `append` method is used to append strings to the current contents of the `StringBuilder` instance.

Listing 3.5: Catenate strings with StringBuilder.

```java
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
public class CatenateStringsWithStringBuilder
{
    public static void main(String[] args)
                                    throws IOException {
        Scanner f = new Scanner(
                        new File("ReadMe.txt"));
        StringBuilder result =
                        new StringBuilder(1000);
        while (f.hasNext()){
            result.append(f.next());
        }
        System.out.println(result);
    }
}
```

The output from CatenateStringsWithStringBuilder is one long line:

Figure 3.4: Using StringBuilder's append method.



```
------------------------------------------------------------------------ThisistheprojectREADMEfile.Here,yoush
```

Example 6. Checking for a palindrome.

You may have noted the `reverse` method of `StringBuilder`. If you have not programmed reversing a string, you should - that is a good exercise in programming. However, now that you know there is a method to perform the reversing of a string we expect you would, if needed, use `reverse` in the future. Consider the following program that gets a string from the user and determines if it is a palindrome.

Listing 3.6: Check for a palindrome with StringBuilder.

```java
import java.util.Scanner;
public class ReverseString
{
    public static void main(String[] args) {
        Scanner kb = new Scanner(System.in);
        System.out.println("Enter a string");
        // two instances of StringBuilder
        StringBuilder original =
                new StringBuilder(kb.next());
        StringBuilder reversed =
                new StringBuilder(original);
        // reverse one of these
        reversed.reverse();
        System.out.println("original :"
                         +original.toString());
        System.out.println("reversed :"
                         +reversed.toString());
        // test for equality
        // need to compare strings
        // because StringBuilder does not
        // override equals in Object
        if
          (original.toString().equals(reversed.toString()))
           System.out.println("a palindrome");
        else
           System.out.println("not a palindrome");
    }
}
```

3.4 Exercises

1. Write a program that counts the number of digits in a string. The program must use a method that begins
   `public static int countDigits(String line)`
   It is possible for the method to be called with a null argument. For example, `countDigits(null)`.

   (a) Write the method `countDigits` so that it tests the argument to verify that it is not null. Use an `assert` statement.

   (b) Write the method `countDigits` so that it tests the argument to verify that it is not null. If it is null your code must throw an `IllegalArgumentException`.

2. At some universities courses are coded in three parts separated by a punctuation character such as a dash. For instance "ACS-2814-050" specifies a course 2814, section 050, offered by the ACS department. Write a program that obtains a string from the end user and then determines if the string properly designates a course where there must be 3 letters, a dash, 4 digits, a dash, and 3 more digits.

3. Modify the program `CheckArgumentValue` so that it does more error checking. That is, verify the value of `mm` is between 01 and 12, and that the value of `dd` is appropriate for the month.

4. Write a program that accepts some words from the end user and creates an acronym from those words. Allow for multiple spaces to appear between words. The acronym comprises the first letter of each word and is all uppercase. For instance if the user entered
   ```
   retail    sales  tax
   ```
   then the acronym is
   ```
   RST
   ```

5. Consider the program `CatenateStringsWithStringBuilder`. Modify the program to display the amount of time it takes to run. Do this by adding this code at the beginning:
   ```
   long tStart = System.currentTimeMillis();
   ```
   and this code at the end:
   ```
   long tEnd = System.currentTimeMillis();
   long tDelta = tEnd - tStart;
   double elapsedSeconds = tDelta / 1000.0;
   System.out.println("elapsedSeconds "+elapsedSeconds);
   ```

6. Again, consider `CatenateStringsWithStringBuilder` and rewrite the program to use `String` instead of `StringBuilder`. Replace the contents of `ReadMe.txt` with something quite long, say one of Shakespeare's plays. Incorporate the timing code from the above question and compare the results of the two programs. This was done by the author at the time of writing with the text of *Romeo and Juliet*; the elapsed time using `StringBuilder` was about 100 times faster than with `String`. The text *Effective Java* by Joshua Bloch (ISBN: 978-0-321-35668-0; Publisher: Addison-Wesley) discusses the difference in performance between `String` and `StringBuilder` in *Item 51*.

Chapter 4

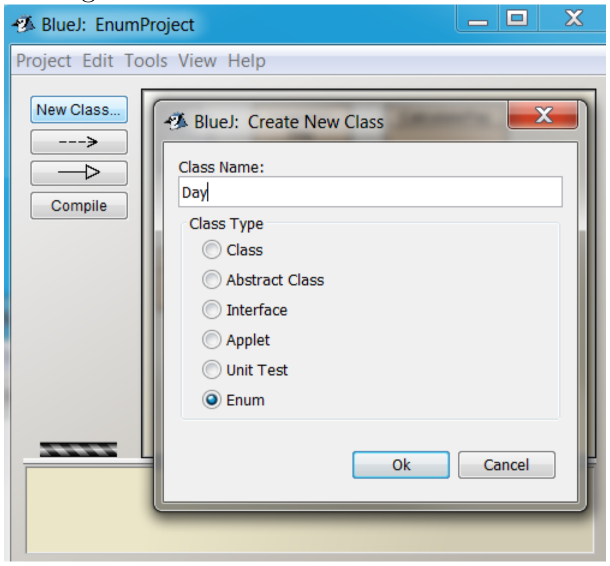
Enumeration Classes (Enums)

To enumerate is to list all values. There are many situations where a Java application works with a well-defined set of values. Consider these examples:

1. days of the week: Sunday, Monday,. . .
2. months of the year: January, February, . . .
3. planets of the solar system: Mercury, Venus, . . .
4. suits in a deck of cards: Spades, Hearts, . . .
5. card faces in a deck of cards: Ace, Deuce, Three, . . .
6. states of a door: open, closed
7. four directions: north, south, east, west
8. grades: A, B, . . .

Enums were added to the Java language in release 1.5. To create an enum using BlueJ: you click the New Class button and then, in the pop-up window you select Enum instead of Class, give a name to the enum, and click OK. See Figure 4.1 where a new enum named Day is to be created.

Figure 4.1: Create an enum with BlueJ



Class Name: Day

Class Type
Class
Abstract Class
Interface
Applet
Unit Test
[✓ Enum]

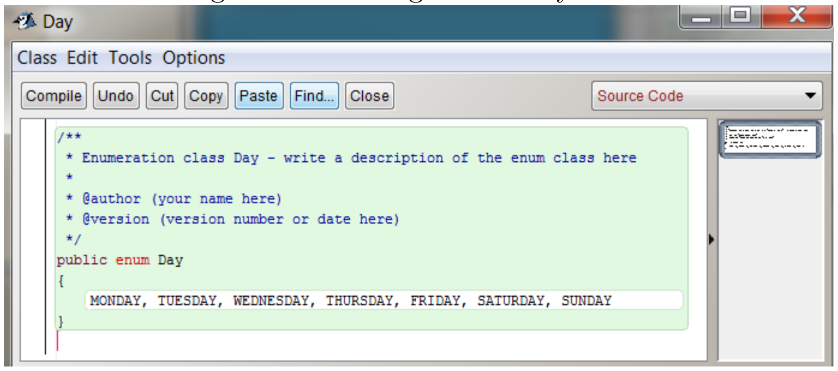
In the following sections we introduce programming with enumerations:

- Defining the simplest of enums: A usable enum requires very few lines of code.
- The `values()` method: this pre-defined method for enums returns a list of values that is convenient for iteration.
- Comparing enum values: Enum instances can be compared using the == operator.
- The `valueOf(...)` method: given a string value, this pre-defined method returns the enum constant with that name.
- Enums are objects: Enum is a special type of class in the Java language, but an enum can have data and methods.

4.1 The basic Enum

Figure 4.2 shows the default code placed in an enum by the BlueJ editor. This code is generated for any enum and happens to be exactly what someone would use (or start with) for an enum to represent the 7 days of the week. If we were creating a different enum we would need to edit the code appropriately.

Figure 4.2: Code generated by BlueJ



```
/**
 * Enumeration class Day - write a description of the enum class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum Day
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Note how the declaration of an enum is similar to that of a class with the word *class* replaced by *enum*:

```
    public enum Day {
```

Figure 4.2 shows a complete enum in its simplest form . . . just an enumeration of values. Note how the values are in upper case; this is a convention of coding - not necessary, but it is a common practice to name constants using upper case characters.

```
    public enum Day
    {
        MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY, SUNDAY
    }
```

In a program it is usual for these constants to be referenced with a prefix "`Day.`" For example: `Day.MONDAY`. Given that an enum `Day` has been defined one can now make other declarations such as

```
    Day today; \\ declare today to be of type Day
    today = Day.MONDAY;\\ assign today a value
```

4.2 `values()` method

Java provides several methods that are pre-defined for enums. Here we consider the `values()` method which returns an array of values for an enum. For instance if we coded

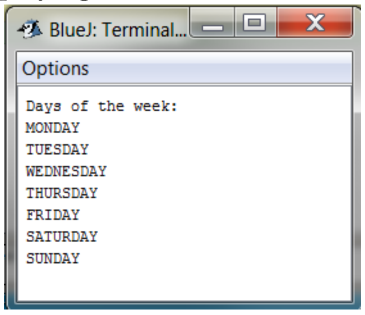
```
Day[] mydays = Day.values();
```

we would have the set of days as elements of an array named `mydays`. Consider Listing 4.1 where we iterate through an enum's values. Figure 4.3 shows the output.

Listing 4.1: Use values() to get all of the enum constants.

```java
/**
 * Use values() to get the enum values
 */
public class EnumValues
{
    public static void main(String[] args){
        Day[] daysOfWeek = Day.values();
        System.out.println("Days of the week:");
        for (Day d: daysOfWeek)
            System.out.println(d);
    }
}
```

Figure 4.3: Displaying the elements returned from values()



```
Days of the week:
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
SUNDAY
```

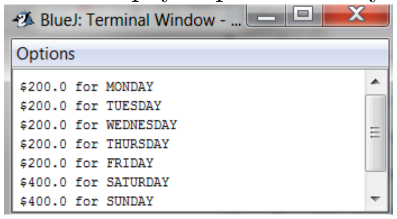
4.3 Comparing enum values

In this example we use enum `Day` and a method `calcPay()` to determine gross pay. If an employee works on the weekend their rate of pay is doubled. In Listing 4.2 the `for` statement iterates through enum values, and the `if` statement tests enum values with `==`. output follows in Figure 4.4

Listing 4.2: Testing for the day of the week.

```java
/**
 * Show gross pay for different days
 *
 */public class CalculatePay
{
    public static void main(String[] args){
        for (Day d: Day.values()){
            System.out.println("$"
             +calcPay(10.0, 20, d)+" for "+d);
        }
    }
    private static double calcPay(
    double rate, int hours, Day day){
        double gross;
        if (day==Day.SATURDAY || day==Day.SUNDAY)
            gross = 2*rate*hours;
        else
            gross = rate * hours;
        return gross;
    }
}
```

Figure 4.4: Gross pay depends on day of week.



```
$200.0 for MONDAY
$200.0 for TUESDAY
$200.0 for WEDNESDAY
$200.0 for THURSDAY
$200.0 for FRIDAY
$400.0 for SATURDAY
$400.0 for SUNDAY
```

4.4 valueOf(. . .) method

Consider an application where the user is prompted for the rate of pay, the hours and the day for calculating gross pay. The value entered by the user for the day is a text value that indirectly identifies an enum. The method `valueOf(...)` takes an argument, a string, and returns the enum constant of that name. For instance the result of `Day.valueOf("MONDAY")` is `Day.MONDAY`. Consider Listing 4.3 where the user supplies the arguments for gross pay calculation.

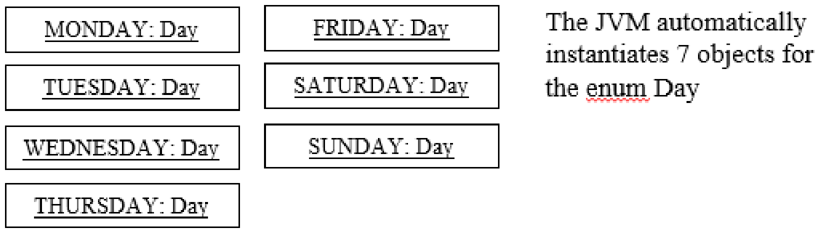
Listing 4.3: Testing for the day of the week.

```java
import java.util.Scanner;
/**
 * Calculate gross for a specific rate, hours, day
 *
 */public class CalculatePayForOneDay
{
    public static void main(String[] args){
        Scanner kb = new Scanner(System.in);
        System.out.println("Enter rate hours day:");
        double rate = kb.nextDouble();
        int hours   = kb.nextInt();
        // the day as a string
        String dayStr = kb.next().toUpperCase();
        // the day as an enum constant
        Day day = Day.valueOf(dayStr);
        System.out.println("gross is $"
                   +calcPay(rate, hours, day));
    }
    private static double calcPay(
    double rate, int hours, Day day){
        double gross;
        if (day==Day.SATURDAY || day==Day.SUNDAY)
            gross = 2*rate*hours;
        else
            gross = rate * hours;
        return gross;
    }
}
```

4.5 Enum constants are objects

Enums are different from other classes we have used up to this point as there are a fixed number of objects automatically created - one object for each enum constant. In our example involving the days of the week there are 7 constants and so there are exactly 7 objects. Using a UML diagram we illustrate the objects in Figure 4.5

Figure 4.5: Enum objects



Because enums are represented by objects they can have data and methods. In this section we introduce an enum with data and fields. The enum is EnhancedDay . . . our previous Day enum modified to have a field to indicate if a day is a weekend. Consider the code for EnhancedDay shown in Listing 4.4, and the following points:

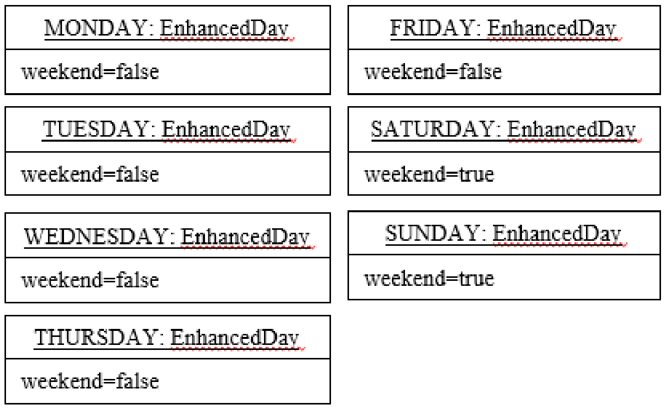
1. Line 9 is a list of enum constants with boolean values in parentheses and the list is ended with a semicolon. The values will be passed to the constructor as the JVM creates the enum objects.
2. Line 10 declares a field `weekend` which is assigned a value when the constructor executes.
3. Lines 11-13 define a constructor that sets the value of weekend. The constructor is automatically called; the programmer does not use `new`.
4. Lines 14-16 define a method to return the value of weekend. `isWeekend()` can be used by an application.

Listing 4.4: An enum with a data and methods.

```java
/**
 * EnhancedDay is a day with
 *    a field: weekend
 *    a constructor: sets value of weekend
 *    a method: isWeekend()
 */
public enum EnhancedDay
{
    MONDAY(false), TUESDAY(false),
       WEDNESDAY(false), THURSDAY(false),
       FRIDAY(false), SATURDAY(true), SUNDAY(true);
    private boolean weekend;
    private EnhancedDay (boolean indicator){
        weekend = indicator;
    }
    public boolean isWeekend(){
        return weekend;
    }
}
```

Now in Figure 4.6 we show objects for the EnhancedDay.

Figure 4.6: Enum objects with fields



The program is Listing 4.5 has a method to calculate the gross pay and uses the enum `EnhancedDay`. Note line 21 where `isWeekend()` is used to determine if the day is a weekend day or not.

Listing 4.5: An enum with a data and methods.

```java
1  import java.util.Scanner;
2  /**
3   * Calculate gross for a specific rate, hours, day
4   *
5   */public class CalculatePayForEnhancedDay
6  {
7      public static void main(String[] args){
8          Scanner kb = new Scanner(System.in);
9          System.out.println("Enter rate hours day:");
10         double rate = kb.nextDouble();
11         int hours   = kb.nextInt();
12         EnhancedDay day=
13                EnhancedDay.valueOf(kb.next().toUpperCase());
14         System.out.println("gross is $"
15                    +calcPay(rate, hours, day));
16     }
17     private static double calcPay(
18     double rate, int hours, EnhancedDay day){
19         double gross;
20         //Double time on weekend
21         if (day.isWeekend())
22             gross = 2*rate*hours;
23         else
24             gross = rate * hours;
25         return gross;
26     }
27 }
```

4.6 Summary

In the table below we list common methods used with enums.

Useful methods for enums		
method	example	comment
`values()`	`Day.values()`	Returns the list of enum constants for Day.
`valueOf(...)`	`Day.valueOf("TUESDAY")`	Returns the constant `Day.TUESDAY` which has the name `"TUESDAY"`.
`toString()`	`day.toString()`	Returns the name of the enum. Automatically invoked by a print statement. Of course, `toString()` can be overridden by the programmer.

In a discussion on enums many texts will also include a method named `ordinal()`. On the surface, `ordinal()` is simple in that it returns the position of an enum constant within the list of enum values; for instance, the ordinal values for `Day` and `EnhancedDay` are `0, 1, ...6 for MONDAY, TUESDAY, ...SUNDAY`.
However, the official Java documentation states:

> Most programmers will have no use for this method. It is designed for use by sophisticated enum-based data structures, such as EnumSet and EnumMap.

and so we do not include it. In a later chapter we do discuss Java map structures and we will revisit enums at that time. A problem associated with using `ordinal()` is that the values are dependent on the ordering of the enum constants in the list which defines them . . . if maintenance modifies the order then it can be that existing code may break. A further reference on enums is the book, mentioned in the last chapter, *Effective Java* by Joshua Bloch.

Java enums are said to be typesafe because the programmer has listed all possible values and given the set a name. It then becomes impossible to assign a value to an enum that is not part of the set. Suppose instead of an enum a programmer used a set of, say, `int` values. For instance, instead of declaring an enum for the days of the week a programmer could define a set of constants, for example:

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
public static final int WEDNESDAY = 2;
```

Note the `final` qualifier which means that once set to the value shown, the value cannot be changed. So, once `MONDAY` has the value `0`, that value cannot be altered. That is a good thing, but problems can arise. Consider the following program where `int`s are used in `calcPay()` instead of an enum. It is easy to pass in an improper value (in this case the value 99).

Listing 4.6: No enum is used ... integers used instead.

```
/**
 * calculate gross pay
 *   calcPay uses ints for days
 *
 */public class CalculatePayWithoutEnum
{
    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;
    public static void main(String[] args){
        // bad call to calcPay
        // day is out of range
        System.out.println(calcPay(10.0, 20, 99));
    }
    private static double calcPay(
    double rate, int hours, int day){
        double gross;
        if (day==SATURDAY || day==SUNDAY)
            gross = 2*rate*hours;
        else
            gross = rate * hours;
        return gross;
    }
}
```

4.7 Exercises

1. Develop an enum named `Month` that includes the months of the year: January, . . . , December. Test your enum with a program similar to `EnumValues`.

2. Develop a program to print the number of days in each month. This program uses a `switch` statement to determine the number of days in a month. Of course, ignore the concept of leap year . . . February will have 28 days.

3. Now develop an enum `EnhancedMonth` that incorporates a field, `days`, which is initialized to the number of days in the month. Test `EnhancedMonth` with a program that calls `getDays()` to obtain the number of days in the month.

4. Develop an enum to represent Grades A, B, C, D, and F. Each grade has a data field, the grade point value: 4, 3, 2, 1, and 0. Test your enum with a program that displays each grade and its corresponding grade point value.

5. Develop two enums: Suit and Face that represent the suits in a card deck (Hearts, Spades, Diamonds, Clubs) and the faces of the cards (Ace, Deuce, Three, . . . King).

6. Develop a class named Card that has two fields: suit and face. The definition of Card begins:
   ```
   public class Card {
       private Suit suit;
       private Face face;
       public Card(Suit s, Face f){
           suit = s;
           face = f;
       }
   ```
   Develop a program that creates and displays a deck of cards. This program declares the card deck as:
   ```
       ArrayList<Card> deck = new ArrayList();
   ```
   and then adds cards to the deck using:
   ```
   for (Suit s: Suit.values())
       for (Face f: Face.values())
           deck.add(new Card(s, f));
   ```

7. Extend the program of the previous exercise to deal a hand of 5 cards to the user. The program must shuffle the deck first. The `Collections` class has a method, `shuffle(...)`, that randomly rearranges the ArrayList that is passed as an argument. For instance, to shuffle an `ArrayList` named `deck` one uses the statement:
   ```
         Collections.shuffle(deck);
   ```
   After executing the statement above, the contents of deck are rearranged to be some random order. Now, the program just has to *deal* the first five cards of the deck.

8. In some card games a hand has a value ... the cards in the hand are counted in some way according to the rules of the game. Suppose we must determine the value of a hand by summing the individual values of each card in the hand. The value of a card is determined by its *face*; for example, an Ace is worth 1, a Deuce is worth 2, ... a Nine is worth 9, and the others are worth 10 each. The enum Face could be modified to account for this:
   ```
     public enum Face {
       DEUCE(2), THREE(3), FOUR(4), FIVE(5), SIX(6),
       SEVEN(7), EIGHT(8), NINE(9), TEN(10), JACK(10),
       QUEEN(10), KING(10), ACE(1);
   ```
   Modify the enum `Face` to include a field, `faceValue`, and pertinent methods. Extend the program of the previous exercise to deal five cards from a shuffled deck, display the cards that are dealt, and the value of the hand.

Chapter 5

Hierarchies

When systems are being designed and there are several classes it is typical to see similarities amongst some classes. The type of similarity we consider here is *generalization / specialization*.

For instance, suppose a medical system involves doctors and pharmacists. When requirements are being gathered we could determine that for a pharmacist we would have a field for *place of employment*, and for a doctor a field for *specialty* (i.e. Dermatology, Pediatrics, ...). But there would be other fields such as *first name*, *last name*, and *gender* that can be seen as common to doctors and to pharmacists. Noting such similarities we could generalize and propose three classes: Practitioner, Doctor, and Pharmacist. In Java we can define Practitioner as a generalization of Doctor and Pharmacist (this also means that Doctor and Pharmacist are specializations of Practitioner ...the two terms, generalization and specialization, go hand-in-hand).

Generalization and specialization are important topics and there are several related concepts that we will introduce in this chapter, including:

- IS-A associations,
- superclasses
- subclasses
- inheritance
- overriding
- polymorphism
- `abstract` classes
- the `Object` class

5.1 IS-A

If we generalize and make statements such as:

- a doctor is a practitioner
- a pharmacist is a practitioner

IS-A

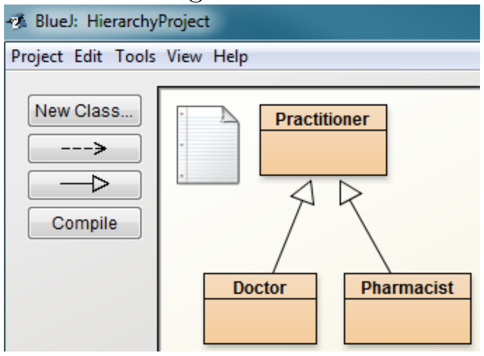
superclass

subclass

class hierarchy

then we say there is an IS-A relationship between Practitioner and Doctor and between Practitioner and Pharmacist. Figure 5.1 is a BlueJ class diagram showing Practitioner as a *superclass* and Doctor and Pharmacist as its *subclasses*. The lines joining the superclass to its subclasses have a triangle shape below Practitioner to indicate Practitioner is the superclass. A structure like this is called a *class hierarchy*. The example is a simple one . . . in general a hierarchy can have many levels and involve a substantial number of classes.

Figure 5.1: BlueJ Class Diagram for the Practitioner Hierarchy.



Distinguishing characteristics of a hierarchy are:

root

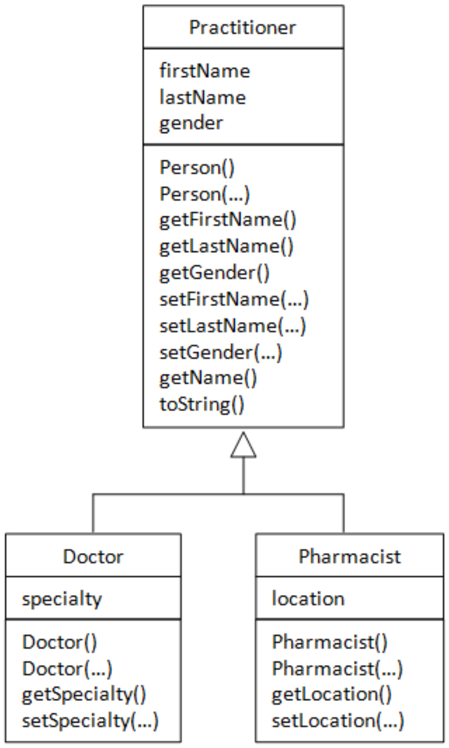
- there is only one class that has no superclass and that class is referred to as the *root* of the hierarchy;
- each class (except the root) has one other class that is its superclass.

base and derived classes

Synonyms for superclass and subclass are *base class* and *derived class* respectively.

Now let us consider the Practitioner hierarchy in more detail. The class diagram in Figure 5.2 shows 3 compartments (rectangles) for each class: class name, fields, and methods.

Figure 5.2: UML Class Diagram for the Practitioner hierarchy.



In the field and method compartments for Practitioner we show fields and methods that are common to doctors and pharmacists. The Doctor class contains only what distinguishes a doctor from other types, and similarly for Pharmacist - only what is necessary to distinguish pharmacists from other types.

In the next section we discuss Practitioner in more detail, and include a program that instantiates practitioners. Following that we consider Doctor and Pharmacist and present a program that instantiates all three types.

5.2 The Practitioner Hierarchy

In this section we discuss each of the classes in the hierarchy.

Practitioner

The `Practitioner` class (next page) is straightforward. There is nothing here to indicate it is part of a class hierarchy.

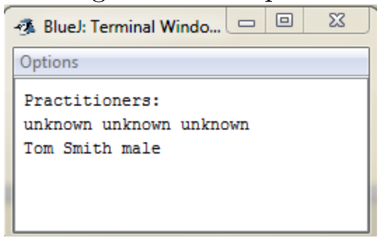
- There are three fields that all practitioners have.
- There are two constructors.
- There are getter and setter methods for each field, and we have included `toString()` and `getName()` methods.

Below is a Java class and its output - the program creates two practitioners.

Listing 5.1: Create practitioners.

```java
/**
 * Demonstration class to create practitioners
 */
public class CreatePractitioners
{
    public static void main(String[] args){
        Practitioner john = new Practitioner();
        Practitioner tom  = new
           Practitioner("Tom","Smith","male");
        System.out.println("Practitioners:\n"+john+"\n"+tom);
    }
}
```

Figure 5.3: Output.



```
Practitioners:
unknown unknown unknown
Tom Smith male
```

Listing 5.2: The Practitioner class - root of the hierarchy.

```java
public class Practitioner {
    private String firstName;
    private String lastName;
    private String gender;
    public Practitioner()     {
        firstName = lastName = gender  = "unknown";
    }
    public Practitioner(String firstName, String
       lastName, String gender)     {
        this.firstName = firstName;
        this.lastName  = lastName;
        this.gender    = gender;
    }
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public String getGender(){
        return gender;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
    public void setGender(String gender){
        this.gender = gender;
    }
    public String toString(){
        return getName()+" "+gender;
    }
    public String getName(){
        return firstName+" "+lastName;
    }
}
```

Pharmacist

The first thing to notice about the Pharmacist class is the `extends` clause (line 6); this is how we specify that Pharmacist is a subclass of Practitioner.

inheritance

Line 8 One field is defined for pharmacists: `location`. However, since `Pharmacist` is a subclass of `Practitioner`, `Pharmacist` *inherits* all fields defined in `Practitioner`. Therefore, a pharmacist object has four fields: first name, last name, gender, location.

no-arg constructors

lines 9-16 The no-arg constructor simply sets location to "unknown". What we do not see is the default action performed by Java for a no-arg constructor: prior to executing this constructor the no-arg constructor of the superclass, `Practitioner`, is automatically called and the other 3 fields will be set.

lines 18-26 This constructor begins by calling a `Practitioner` constructor, as specified by:

super

```
super(firstName, lastName, gender);
```

To invoke a specific constructor in the superclass you must use the keyword `super` with the proper arguments. Such a call must be the first statement of the constructor. If you review the `Practitioner` constructors you see there is one that accepts those 3 fields.

Lines 27-34 The other methods are the standard getter and setter for the location field. However, as `Pharmacist` inherits all the fields and methods of `Practitioner` a program can use any of those fields and methods too with reference to an `Pharmacist` object. For instance if `p` is a reference to a pharmacist object then we can code `p.getFirstName()`.

inheritance

Listing 5.3: The Pharmacist class - subclass of Practititioner.

```java
/**
 * The Pharmacist class
 * - a subclass of Practitioner
 * - a pharmacist "is a" practitioner
 */
public class Pharmacist extends Practitioner
{
    private String location;
    /**
     * by default, the no-arg constructor calls
     * the no-arg constructor in Practitioner
     */
    public Pharmacist()
    {
        location = "unknown";
    }

    /**
     * constructor for when information is available
     */
    public Pharmacist(String firstName, String
       lastName, String gender, String location)
    {
        // note the explicit call to a Practitioner
           constructor
        super(firstName, lastName, gender);
        this.location = location;
    }
    // getters
    public String getLocation(){
        return location;
    }
    // setters
    public void setLocation(String location){
        this.location = location;
    }  }
```

Doctor

The Doctor class has an extends clause (line 6) and so it is a subclass of Practitioner.

inheritance

Line 7 One additional field is defined for doctors: their area of specialty. However, since Doctor is a subclass of `Practitioner`, doctors inherit all fields defined in `Practitioner`. Therefore, a doctor object has four fields: first name, last name, gender, and specialty. Doctors have four fields as do pharmacists but the last field is different.

no-arg

lines 8-15 The no-arg constructor for the superclass is called first which simply sets `specialty` to "unknown"; What we do not see is the default action performed by Java for a no-arg constructor: prior to executing this constructor the no-arg constructor of the superclass, `Practitioner`, is automatically called causing the other 3 fields to be set.

super

lines 16-23 The other constructor begins by calling a `Practitioner` constructor, as specified by:

```
super(firstName, lastName, gender);
```

To invoke a specific constructor in the superclass you use the keyword `super` with the proper arguments. Such a call must be the first statement of the constructor. If you review the Practitioner constructors you see one that sets the values for those 3 fields.

inheritance
inheritance

Lines 24-29 The standard getter and setter methods are defined for the specialty field. However, `Doctor` inherits all the fields and methods of `Practitioner` and so a program can use any of the `Practitioner` fields and methods too with reference to a `Doctor` object. For instance if `d` is a doctor object then we can code `d.getFirstName()`.

method override

Lines 30-32 `Doctor` has a method `getName()` which is the same method name and parameter list as `getName()` in Practitioner. We say the subclass method overrides the method in the superclass. When `getName()` is executed for a Doctor instance, it is this method that will execute, and not the method defined in `Practitioner`.

Listing 5.4: The Doctor class - subclass of Practititioner.

```java
/**
 * The Doctor class
 * - a subclass of Practitioner
 * - an instructor "is a" practitioner
 */
public class Doctor extends Practitioner {
    private String specialty;
    /**
     * no-arg constructor, recall default call
     * to Practitioner no-arg constructor
     */
    public Doctor()
    {
        specialty = "unknown";
    }
    /**
     * constructor with firstname etc
     */
    public Doctor(String firstName, String
       lastName, String gender, String specialty)
           {
        // note call to superclass constructor
        super(firstName, lastName, gender);
        this.specialty = specialty;
    }
    public String getSpecialty(){
        return specialty;
    }
    public void setSpecialty(String specialty){
        this.specialty = specialty;
    }
    public String getName(){
        return "Dr. "+getFirstName()+"
           "+getLastName()+", "+getSpecialty();
    }
}
```

5.2.1 Creating and using objects of the hierarchy

Given the hierarchy, when an instance of Doctor is created that instance is (at the same time) an instance of Practitioner, and so a Doctor instance includes all the properties of Practitioner. Similarly when an instance of Pharmacist is created, that instance is (at the same time) an instance of Practitioner and includes all the properties of Practitioner. This means, for our example, that a doctor (or pharmacist) includes first name, last name, and gender, and includes all the methods defined for Practitioner. However if we explicitly instantiate a Practitioner object that object is an instance of Practitioner, and not an instance of Doctor (or Pharmacist). To sum this up:

- an instance of Doctor is also (at the same time) an instance of Practitioner
- an instance of Pharmacist is also (at the same time) an instance of Practitioner
- an instance of Doctor is not an instance of Pharmacist
- an instance of Pharmacist is not an instance of Doctor,
- a Practitioner instance may be an instance of Doctor, or may be instance of Pharmacist, but cannot be an instance of both.
- an instance of Practitioner may be an instance of neither Doctor nor Pharmacist

Next, we consider a program that instantiates instances from each part of the hierarchy.

The Java class in Listing 5.5 creates practitioner, doctor, and pharmacist objects, then adds them to an `ArrayList` of `Practitioners`, and then displays their first names. Some important points about the code:

Lines 7-9. Objects of the three types are created.

Lines 10-12. Objects are added to an `ArrayList` of `Practitioners`. The objects are of different types, but they are (at the same time) instances of a base class `Practitioner`, and so they are valid objects to be added to an `ArrayList` of `Practitioners`.

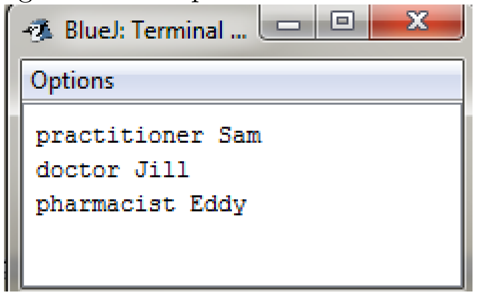
Lines 13-18. The loop iterates through the collection and displays each object's type and the instance's first name. The method `getFirstName()` is defined in `Practitioner` and nowhere else . . . so it is this method that executes for each object (each object is an instance of `Practitioner`).

instanceof

The Java operator `instanceof` is used to determine an object's type. `instanceof` is a binary operator that tests an object (the first argument) and returns `true` if the object is an instance of the second argument (a Java class), and returns `false` otherwise.

The Java class, `Practitioners`, that creates practitioner, doctor, and pharmacist objects appears on the next page in Listing 5.5; its output is below in Figure 5.4.

Figure 5.4: Output from Practitioners.



```
practitioner Sam
doctor Jill
pharmacist Eddy
```

Listing 5.5: Programming example with a class hierarchy.

```
import java.util.ArrayList;
public class Practitioners {
   public static void main(String[] args){
      // List of practitioners
      ArrayList<Practitioner> practitioners = new
         ArrayList();
      // Create some practitioners
      Practitioner pr = new
         Practitioner("Sam","Smith","female");
      Doctor       dr = new
         Doctor("Jill","Jones","female","Dermatology");
      Pharmacist   ph = new
         Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
      for (Practitioner p: practitioners) {
         String type="practitioner";
         if (p instanceof Doctor) type="doctor";
         if (p instanceof Pharmacist)
            type="pharmacist";
         System.out.println(type+"
            "+p.getFirstName());
      }
   }
}
```

5.3 Overriding methods

The `Practitioner` hierarchy has two methods with the same header:

```
    public String getName()
```

One of these is defined in `Practitioner` and the other is defined in `Doctor`. That is, one is in the superclass and the other is in a subclass.

- In the Practitioner class the method getName() returns the first name follwed by the last name:

  ```
  public String getName(){
      return firstName+""+lastName;
  }
  ```

  If `p` refers to a `Practitioner` instance (not a `Doctor` instance) then `p.getName()`, causes this method to execute.

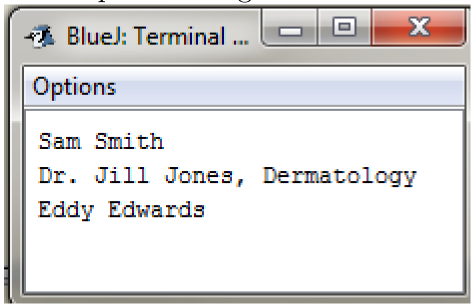
- In the `Doctor` class the method `getName()` returns the doctor's name with a prefix `"Dr."` and a suffix, the doctor's specialty:

  ```
  public String getName(){
      return "Dr. "+getFirstName()+""+getLastName()
             +", "+getSpecialty();
  }
  ```

  If `d` refers to a `Doctor` instance then `d.getName()` causes this method to execute. In this situation we say that `getName()` in `Doctor` *overrides* `getName()` in `Practitioner`. *overrides*

Overriding methods is one way to specialize a subclass. Listing 5.6 creates the same objects as Listing 5.5 and displays the names using `getName()`. The output is shown in Figure 5.5 - you will see that `getName()` in `Doctor` has overridden the method defined in `Practitioner`.

Figure 5.5: Output showing use of overriden method.



```
Sam Smith
Dr. Jill Jones, Dermatology
Eddy Edwards
```

Listing 5.6: getName() in Doctor overrides getName() in Practitioner.

```java
import java.util.ArrayList;
public class Practitioners1 {
   public static void main(String[] args){
      // List of practitioners
      ArrayList<Practitioner> practitioners = new
         ArrayList();
      // Create some practitioners
      Practitioner pr = new
         Practitioner("Sam","Smith","female");
      Doctor       dr = new
         Doctor("Jill","Jones","female","Dermatology");
      Pharmacist   ph = new
         Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
      for (Practitioner p: practitioners) {
         // display name
         // getName() in Doctor overrides
         // getName() in Practitioner
         System.out.println(p.getName());
      }
   }
}
```

5.3.1 Calling an overridden superclass method

The `Doctor` class includes a `getName()` method that overrides `getName()` in `Practitioner`. We saw that this method was automatically invoked by the JVM when referenced by an object of type `Doctor`. Now suppose you want to invoke the overridden method instead. In order to do this you prefix the method with `super`. For instance we can rewrite the `getName()` method as follows:

```
    public String getName(){
        return "Dr. "+ super.getName()+", "+getSpecialty();
        }
```

Note that the prefix `super` is required in order to explicitly call the `getName()` method in the superclass. If you mistakenly coded:

```
    public String getName(){
        return "Dr. "+ getName()+", "+getSpecialty();
        }
```

then the call to getName() in the return statement would be a *recursive* call to the method being defined in `Doctor` - recursion is covered in a later chapter.

5.4 Exercises

1. Develop another subclass of the `Practitioner` hierarchy (Dentist). Demonstrate your new subclass with a class that instantiates Dentist objects.
2. Override the `toString` method in the subclasses of `Practitioner` and demonstrate the effect of your overriding methods.
3. Develop a set of classes that form a hierarchy for persons where a person can be a student and where a person can be an instructor. Include fields that are common to students and instructor in a `Person` class. Include fields that distinguish students and instructors in `Student` and `Instructor` subclasses. All fields must have getters and setters. Demonstrate your hierarchy with a class that instantiates an instance of each type.

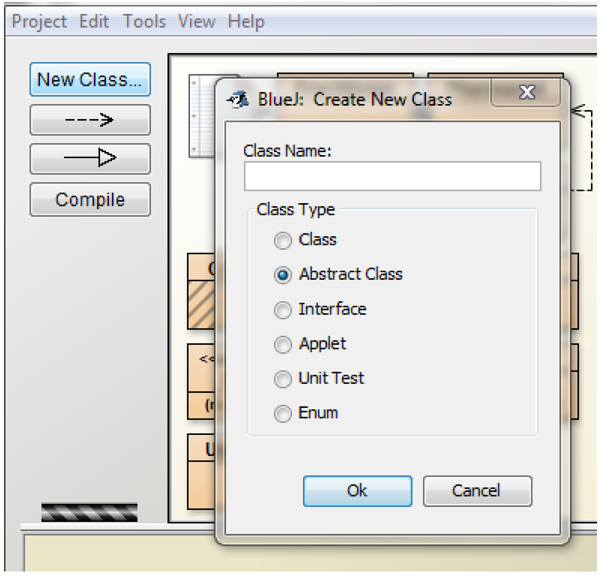
5.5 Abstract Classes and Methods

In the practitioner hierarchy example we instantiated objects of all three classes. If our system only deals with doctors and pharmacists and if there is no need to ever explicitly instantiate a practitioner then we can ensure this can never happen if we make Practitioner an *abstract* class. An `abstract` class cannot be instantiated - only a non-abstract subclass can be instantiated.

abstract class

If you are using the BlueJ IDE, then when you create a class, you can choose to make the class `abstract` - see Figure 5.6.

Figure 5.6: BlueJ Create New Class window - Abstract Class selected.



Class Name:

Class Type

Class

[✓ Abstract Class]

Interface

Applet

Unit Test

Enum

If you compare the code generated by BlueJ for a regular class and an abstract class the only difference you will see is the keyword `abstract` appearing in the class header. If `Practitioner` was defined as an `abstract` class the class header is:

```
    public abstract class Practitioner
```

instead of

```
    public class Practitioner
```

An abstract class can have one or more methods that are abstract too. An abstract method declared as abstract has no body - only a semicolon, for example:

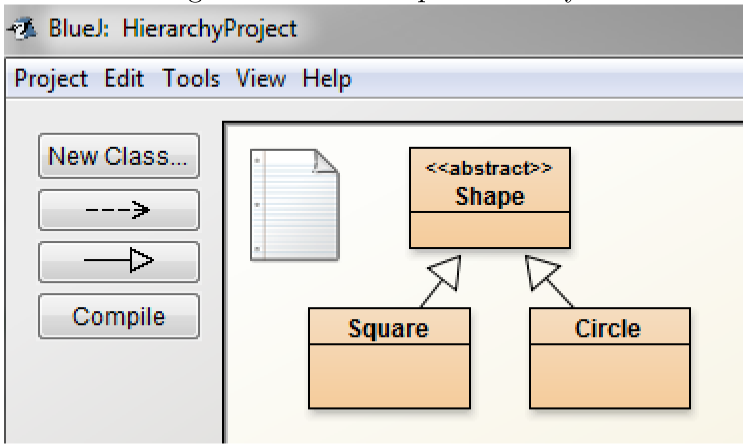
```
        public abstract String getName();
```

The reason for defining a method as abstract is to force each subclass to define the method since there is no default code in the superclass. In the next section we consider a shape hierarchy where the superclass is abstract with an abstract method.

Example: A shape hierarchy

Consider a class hierarchy of shapes. Figure 5.7 shows the Shape hierarchy as depicted in BlueJ. In the diagram it is clear that Shape is abstract and the triangular symbols make it clear that Square and Circle are subclasses.

Figure 5.7: The Shape Hierarchy.



Listing 5.7 lists the classes that define this hierarchy with an abstract superclass Shape (lines 1-5), a subclass `Circle` (lines 6-15) and a subclass `Square` (lines 16-25). The class `UseShapes` (lines 26-36) creates a circle and a square but treats both as shapes.

Shape is `abstract` with an `abstract` method `area`. Each shape must have a different formula and so the pertinent formula is defined only in subclasses. The advantage of having a superclass is that all shapes can be included in the same `ArrayList` and be treated alike.

Some points to observe:

Lines 3-5. `Shape` is abstract and has an abstract method named `area`.

Lines 6-15. `Circle` is a subclass of `Shape` - it extends the `Shape` class. `Circle` contains an implementation of `area()`.

Lines 16-25. `Square` is a subclass of `Shape` - it extends the `Shape` class. `Square` contains an implementation of `area()`.

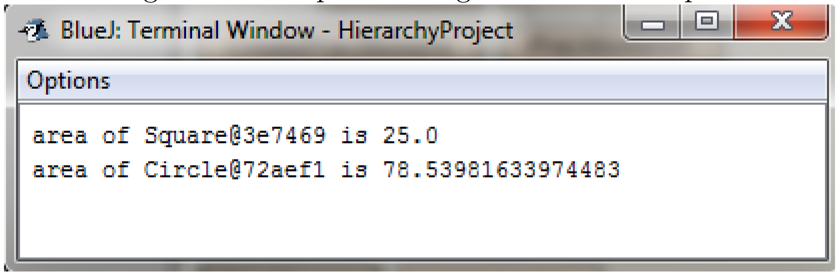
Lines 27-34. `UseShapes` creates a square and a circle, and both are added to an `ArrayList` that holds shapes. For each shape in the `ArrayList` the shape and its area are printed. The proper `area` method is chosen at runtime according to the type of object.

`UseShapes` displays each shape using the following `for` statement (lines 32-33):

```
for (Shape s: shapes)
   System.out.println("area of "+s+"is "+s.area());
```

The println displays the shape, `s`, and its area. Shape, Circle and Square do not include a `toString()` method and so the default `toString()` method executes ... the output for a shape object is just the name of the class it belongs to and its memory location. Thus the output does show the type of the object and its area - shown in Figure 5.8 below.

Figure 5.8: Output showing area of each shape.



```
area of Square@3e7469 is 25.0
area of Circle@72aef1 is 78.53981633974483
```

Listing 5.7: Shapes hierarchy and use.

```java
// Abstract superclass

public abstract class Shape {
    public abstract double area();
}
// subclass Circle
public class Circle extends Shape {
    private int radius;
    public Circle (int radius) {
        this.radius = radius;
    }
    public double area() {
        return Math.PI*radius*radius;
    }
}
// subclass Square
public class Square extends Shape {
    private int length;
    public Square(int length) {
        this.length = length;
    }
    public double area() {
        return length*length;
    }
}
// Create shapesimport java.util.ArrayList;
public class UseShapes {
public static void main(String[] args) {
    ArrayList<Shape> shapes = new ArrayList();
    shapes.add( new Square(5) );
    shapes.add( new Circle(5) );
    for (Shape s: shapes)
       System.out.println("area of "+s+" is
          "+s.area());
    }
}
```

5.6 Exercises

1. Consider the `Shape` hierarchy. Include a new subclass (Triangle) and include its area method. To demonstrate your new subclass, write a class that creates triangles and displays their areas.

2. Modify the `Practitioner` class making it an `abstract` class. Write a class to demonstrate your `Practitioner` class. What happens if you include a statement such as the following?

   ```
   Practitioner p = new Practitioner();
   ```

5.7 Summary

- We have seen that Java classes can be ogranized into hierarchies. In fact all classes in Java do extend one class either directly or indirectly; this is the `Object` class. The `Object` class is a special class that provides methods that all classes inherit; some of these we have encountered, such as `equals` and `toString`.

Object class

- A UML class diagram has a line from a superclass to a subclass. Where the line connects to the superclass one uses a triangle symbol to designate the superclass.
- An object instantiated from a class hierarchy is automatically an instance of that class and at the same time it is an instance of its superclass (and its superclass' superclass, etc. ...all the way up to the root of the hierarchy).
- Any method defined in a superclass can be used by an object of a subclass.
- For any object and class you can test for the object being an instance of that class. `instanceof` is a binary operator that produces a `boolean` result. The syntax is:

instanceof

  object `instanceof` *class name*

  If an object say, `x`, is a `Doctor` then "`x instanceof Practitioner`" and "`x instanceof Doctor`" are both `true`, and "`x instanceof Pharmacist`" is `false`.

- When a subclass defines a method of the same name and parameters as the superclass we say the subclass is overriding the method in the superclass. In Doctor we included a `getName()` method that overrides the `getName()` method in `Practitioner`. That is, if `getName()` must execute for an object that is an instance of `Doctor` then `getName()` in `Doctor` will be executed instead of `getName()` in the superclass, `Practitioner`.

overriding

- A typical dictionary definition of the word *polymorphism* is "the quality or state of existing in or assuming different forms". The Java language implements forms of polymorphism. Consider the `for` loop where the type of `p` is `Practitioner`:

  polymorphism

  ```
  for (Practitioner p: practitioners){
      System.out.println(p.getName());
  ```

  The `for` loop iterates over the objects and for each object `getName()` is executed. We know that these objects are of types `Practitioner`, `Doctor` and `Pharmacist`; `p` takes many forms - `p` is polymorphic. The JVM selects the `getName()` that is appropriate to the object - in particular the JVM selects the overriding method in the case of an `Doctor` object. We say at runtime the JVM *polymorphically* selects the behaviour appropriate to the object.

- We have seen that some classes cannot be instantiated; these are called `abstract` classes. The keyword `abstract` appears in the class header.

- It is possible to specify some methods as `abstract`. Such methods have no body, and somewhere lower in the class hierarchy `abstract` methods must have a definition. At the lowest levels of a hierarchy there must be classes that can be instantiated - these cannot be abstract.

- A Java class can have an `extends` clause where a *single* class can be named as the superclass. This is called *single inheritance*. The `extends` clause is used then to define a class hierarchy. A class hierarchy is where each class, except the root class, has one superclass. In the next chapter we look at `interface`s; it is possible for a class to *implement* several `interface`s.

- It is usually the case that some methods are fully defined in an abstract class, but it is possible for an abstract class to have only abstract methods. Such abstract classes are similar to `interface`s, but more limited due to single inheritance.

- `Enum`s cannot be included as part of a class hierarchy.

5.8 Exercises

1. Develop a hierarchy for vehicles. Include subclasses `Mortorized` and `NonMortorized`. Subclass each of these so you can include `Mortocycle`, `Car` and `Truck` as subclasses of `Mortorized`, and `Bicycle` as the only subclass of `NonMortorized`. All classes except `Mortorcyle`, `Car`, `Truck` and `Bicycle` are `abstract`.

2. Develop a hierarchy for certain products that are sold in a store. Include classes for CDs (music) and DVDs (movies). Common fields include title and cost. CDs have a field artist, and DVDs have a field director. Make the superclass abstract. Demonstrate your hierarchy with a class that instantiates some CDs and DVDs. The class then calculates the total cost of those objects.

Chapter 6

Interfaces

An interface is a class-like construct considered to be a reference type. This means that an object may be of the interface type and referenced as such. A Java interface can include any of:

Constants. Fields and their values can be specified. The values are treated as constants - their values cannot be changed. The Java term used to describe something that cannot be altered is `final`.

Method signatures. Methods are named, their return type and parameter lists are specified. Prior to Java 8 these methods could only be `abstract`.

Default methods. As of Java 8 it is possible to specify an implementation for a method. These methods are declared as `default` methods. An implementing class can choose to override - and so they are called *optional* methods. Optional methods in an interface provide a mechanism for interfaces to be altered in the future without breaking existing implementations.

If you were to examine the Java class libraries you would notice there is extensive use of interfaces. One interface that we have used (without drawing this to your attention previously) is named `Comparable`. `Comparable` is an interface that is used to provide what is termed the natural ordering for objects. We discuss this interface and demonstrate in the first two examples how we can compare and sort strings. Examples 1 and 2 could be examined by the reader now, and they can make perfect sense based on previous study in Java. In Example 1 a `String` method named `compareTo` is used. This method compares two strings and returns a negative value, a positive value, or 0 according to whether one string is less than, greater than, or equal to another. Example 2 creates an array of strings and sorts the strings using the `sort` method of the `Arrays` class. The two examples, on their own, do not explicitly state anything about interfaces. But as we will explain later the method `compareTo` is part of the `Comparable` interface, the `String` class implements `Comparable`, and it this feature that the `sort` method relies on for sorting strings.

Then we consider two further examples where we present a `Person` class. When you develop a class, it is up to you to decide if the objects have an ordering. What characteristic (or characteristics) would you choose to determine if one object is less than another object? As a person has a name we will implement a natural ordering on persons based on names. To achieve this, `Person` will implement `Comparable` and then we will demonstrate how simple it is to sort an array of persons.

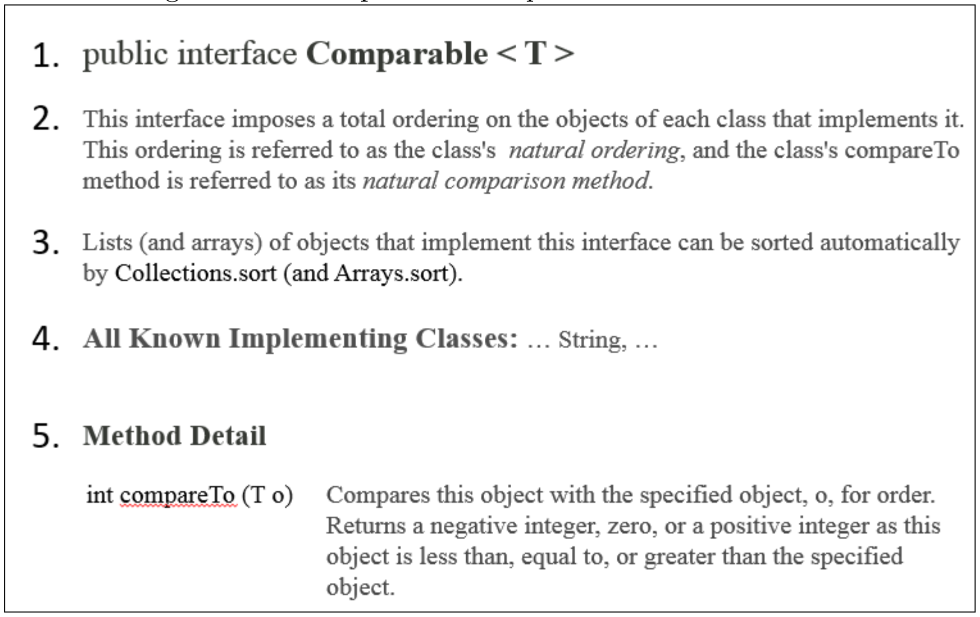
In the second section we consider a situation involving teams and players. We will present a `Team` class that depends on an interface named `Player` ... a team is made up of several players. Then we will reuse the `Person` class (mentioned above), modifying it to implement `Player`. The `Player` interface is a connection of sorts between `Team` and `Person`.

Lastly, we discuss another interface from the Java class libraries, `Comparator`. This interface has similarities to `Comparable`. `Comparable` is used to provide a natural ordering for objects, but `Comparator` can be used to provide one or more alternate orderings for objects.

6.1 Comparable interface

Consider Figure 6.1 that is taken from the Comparable documentation (we have excluded some content to keep our discussion focussed). The numbered points of the figure are disussed below.

Figure 6.1: Excerpt from Comparable documentation.



1. public interface **Comparable < T >**

2. This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method.*

3. Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort).

4. **All Known Implementing Classes:** … String, …

5. **Method Detail**

int compareTo (T o)	Compares this object with the specified object, o, for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

1. The documentation begins by naming the interface `Comparable`:
   ```
   public interface Comparable < T >
   ```
   The syntax element `<T>` indicates that we can name a specific class to which `Comparable` applies.
2. The general description states `Comparable` imposes a total ordering on the objects of the class implementing it.
3. Further, and we will use this information later, it is mentioned that `Collections.sort` and `Arrays.sort` can be used to sort lists and arrays of objects where the objects belong to a class that implements `Comparable`.
4. Next there is a list of *all known implementing classes*. There are very, very, many such classes; we have omitted all but one: `String`. We are very familiar with the `String` class - it was introduced in Part 1.
5. In the detail section we see that `compareTo` returns an `int`.

String implements Comparable

The String class has a `compareTo` method that implements the required behaviour for `Comparable`:

```
int compareTo(String anotherString)
```

> The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string, `anotherString`. The result is a negative integer if this `String` object lexicographically precedes `anotherString`. The result is a positive integer if this `String` object lexicographically follows `anotherString`. The result is zero if the strings are equal; compareTo returns 0 exactly when the `equals` method would return true.

Lexicographic ordering:

> A language dictionary is also called a lexicon. So, lexicographical ordering is an ordering of strings in dictionary order. The String implementation of this ordering is based on the numeric encoding of characters. For example suppose there are two strings, s1 and s2, that are different but of equal length. At some index (let k be the smallest such index) the characters are different:
> If `s1.charAt(k)< s2.charAt(k)` then `s1` lexicographically precedes `s2`.
> If `s1.charAt(k)> s2.charAt(k)` then `s2` lexicographically precedes `s1`.
> To learn more try reading the official documentation on `String` and its `compareTo` method.

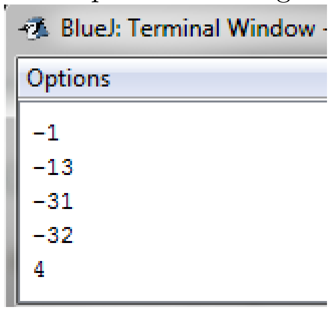
Example 1, `compareTo` values

For a first example we present a simple program that displays the value of `compareTo` when one string is compared to another. Examine the program and output and you will get a sense of the ordering defined by `String`. The output indicates `"123"` is less than `"124"`, `"124"` is less than `"12A"`, `"12A"` is less than `"PROGRAM"`, `"PROGRAM"` is less than `"program"`, and `"programming"` is greater than `"program"`. This should not be surprising given the underlying integer values the system uses to represent characters: in general digits are less than capital letters, and capital letters are less than lowercase letters.

Listing 6.1: Comparing strings lexicographically.

```java
/**
 * Specific strings are compared using the
 * compareto method of the String class
 */
public class StringCompareto
{
    public static void main(String[] args){
        System.out.println("123".compareTo("124"));
        System.out.println("124".compareTo("12A"));
        System.out.println("124".compareTo("PROGRAM"));
        System.out.println(
                      "PROGRAM".compareTo("program"));
        System.out.println(
                  "programming".compareTo("program"));
    }
}
```

Figure 6.2: Output from StringCompareto.



```
-1
-13
-31
-32
4
```

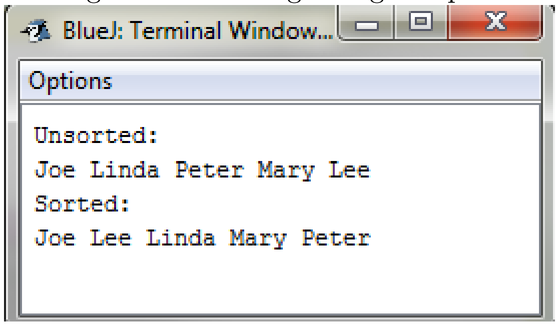
Example 2, Sorting an array of strings

Recall in Section 1.7 on *array utilities* we discussed the sort method in the `Arrays` class. `Arrays.sort()` can work with an array of strings since `String` implements `Comparable`. Consider Listing 6.2 where the array `names` is displayed, sorted, and displayed again; output is shown in Figure 6.3.

Listing 6.2: Sorting strings is easy since they are naturally ordered.

```
import java.util.Arrays;
/**
 * Sort an array of strings using Arrays.sort
 */
public class SortingStrings
{
   public static void main(String[] args){
       String [] names =
       {"Joe", "Linda", "Peter", "Mary", "Lee"};
       System.out.println("Unsorted:");
       for (String n: names)
          System.out.print(n+" ");
       Arrays.sort(names);
       System.out.println("\nSorted:");
       for (String n: names)
           System.out.print(n+" ");
    }
}
```

Figure 6.3: Sorting strings output.



```
Unsorted:
Joe Linda Peter Mary Lee
Sorted:
Joe Lee Linda Mary Peter
```

Example 3, A natural ordering for persons

In this example we use a class to represent persons (Listing 6.3). To impose a natural ordering on persons we have chosen to order persons based on their names (examples: ”Peter” is less than ”Susan”, ”Susan” is less than ”Susanna”). This natural ordering will be implemented as part of the Comparable interface (line 1) and the compareTo method (lines 8-12). Note how in line 11, because the name field is of type `String`, we can utilize `String`'s implememtation of `compareTo`.

Listing 6.3: Person class implements Comparable.

```
 1 public class Person implements Comparable<Person> {
 2     private String name;
 3     private int year; // year of birth
 4     public Person(String n, int y) {
 5         name = n;
 6         year = y;
 7     }
 8     public int compareTo (Person p) {
 9         // compare the name of this object
10         // to the name of object p
11         return name.compareTo(p.name);
12     }
13     public String getName(){
14         return name;
15     }
16     public void setName(String n){
17         name = n;
18     }
19     public int getYear(){
20         return year;
21     }
22     public void setYear(int y){
23         year = y;
24     }
25     public String toString(){
26         return name;
27     }
28 }
```

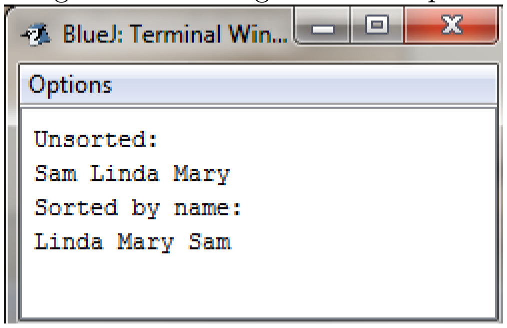
Example 4, Sorting using `Comparable`

Here we show a class (Listing 6.4) that utilizes the fact that we can easily sort an array of persons - lines 5-8 initiallize the array `people` so it holds references to `Person` objects. The `main` method displays persons, calls Arrays.sort in line 12, and displays the persons again (output shown in Figure 6.4).

Listing 6.4: Person implements Comparable and can be sorted.

```java
1  import java.util.Arrays;
2  public class SortingPersons
3  {
4     public static void main(String[] args){
5         Person [] people = {
6             new Person("Sam",1972),
7             new Person("Linda", 1974),
8             new Person ("Mary", 1957)};
9         System.out.println("Unsorted:");
10        for (Person p: people)
11           System.out.print(p+" ");
12        Arrays.sort(people);
13        System.out.println("\nSorted by name:");
14        for (Person p: people)
15           System.out.print(p+" ");
16     }
17 }
```

Figure 6.4: Sorting Persons output.



```
Unsorted:
Sam Linda Mary
Sorted by name:
Linda Mary Sam
```

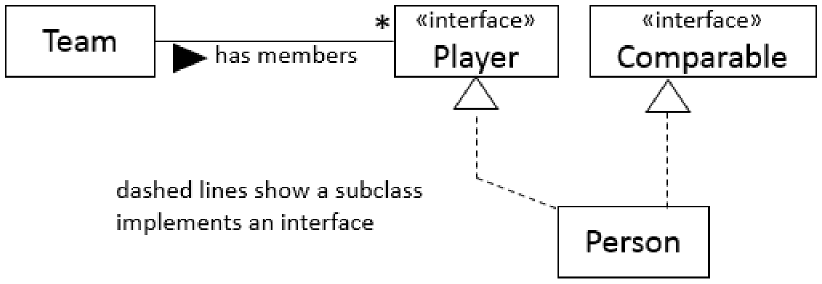
6.2 Defining an interface

Complex systems of course involve several classes. To help reduce the overall complexity a development strategy is to *design to an interface* rather than an implementation.

In this section we consider a sports application where we have two basic concepts of team and player. It is common for a large system to be created by groups of developers. For instance we could have one group tasked with developing software relating to teams and another group tasked with developing software pertinent to players. To assist these groups an interface can be specified for player-related functionality.

Our examples will use a `Team` class, a `Player` interface, and we will reuse the `Person` class modifying it to implement `Player`. As you will see the `Team` class will be coded so it depends on the `Player` interface and not the `Person` class. By doing this it would be possible to easily replace, if the need arises, the `Person` class by some other implementation. Figure 6.5 is a UML class diagram showing that a team has members who are players. The Person class is shown to implement two interfaces: Player and Comparable.

Figure 6.5: UML diagram for Team, Player, Person, Comparable.

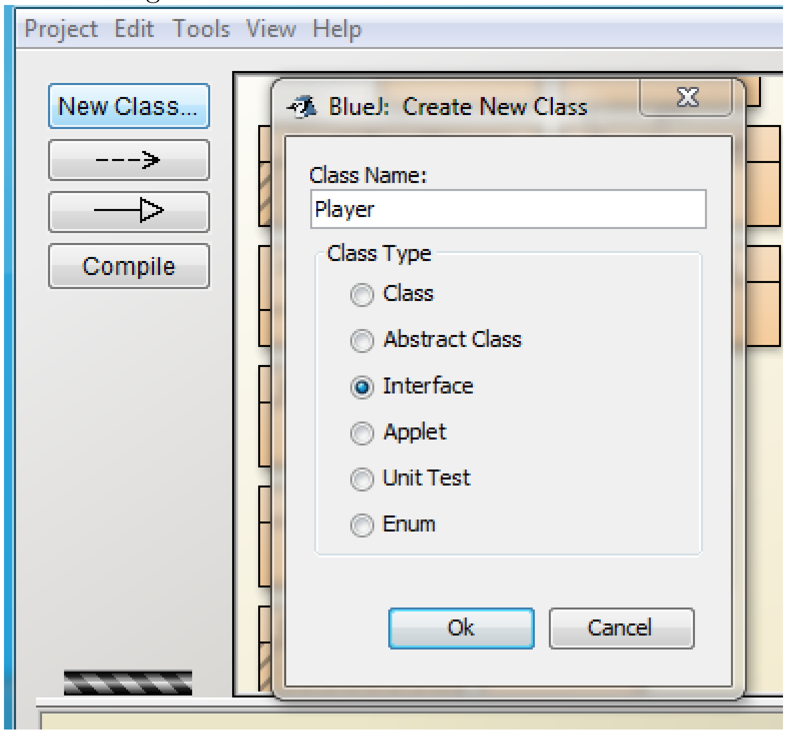


Before presenting the code examples we briefly discuss how to use BlueJ to create an interface.

Creating an interface with BlueJ

To create an interface using BlueJ click on the New Class button and choose Interface as shown in Figure 6.6.

Figure 6.6: Create an interface with BlueJ.

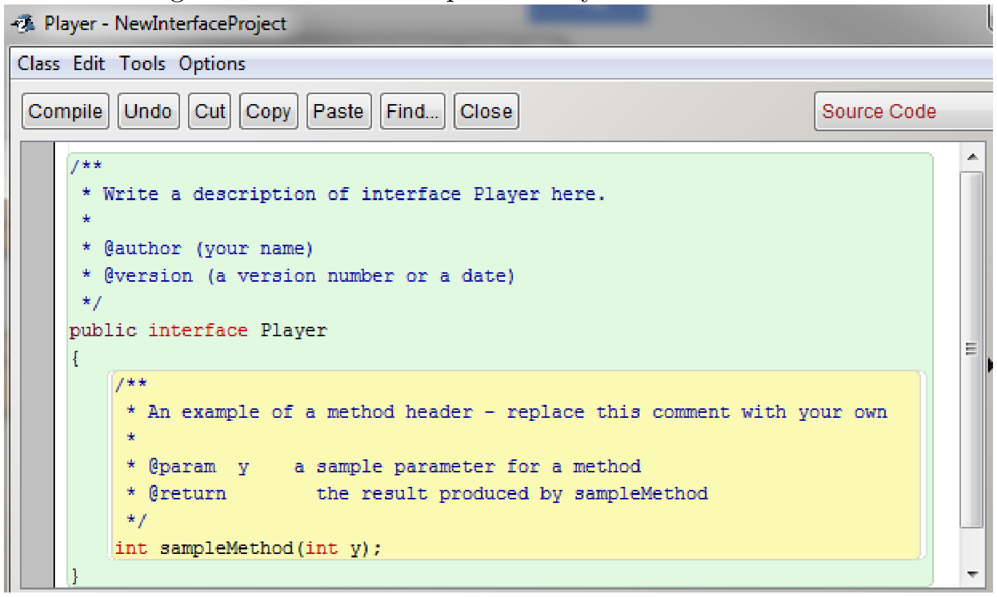


Class Name: Player

Class Type
( ) Class
( ) Abstract Class
[✓ Interface]
( ) Applet
( ) Unit Test
( ) Enum

When you create an interface using BlueJ there is some initial code provided, see Figure 6.7. The initial code can be edited in the BlueJ code editor to contain the necessary definition of, say, the Player interface shown later in Listing 6.5.

Figure 6.7: Initial code provided by the BlueJ editor.


Player - NewInterfaceProject
Class Edit Tools Options
Compile Undo Cut Copy Paste Find... Close
Source Code

```
/**
 * Write a description of interface Player here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public interface Player
{
    /**
     * An example of a method header - replace this comment with your own
     *
     * @param  y    a sample parameter for a method
     * @return      the result produced by sampleMethod
     */
    int sampleMethod(int y);
}
```

Example 5, `Player` interface

To keep our example simple we have the `Player` interface as defined in Listing 6.5:

Listing 6.5: Player interface.

```
 1 /**
 2  * The Player interface
 3  */
 4 public interface Player
 5 {
 6     String getName();
 7     String getPosition();
 8     int getJerseyNumber();
 9     void setPosition(String p);
10     void setJerseyNumber(int j);
11 }
```

Line 4 gives the interface the name Player.

Lines 6-10 specify the names, return type, and parameters of five methods: `getName`, `getPosition`, `getJerseyNumber`, `setPosition`, and `setJerseyNumber`.

A developer of, say, the Team class knows there can be references to objects of type Player and that those objects can respond to requests for their name, position, etc.

Example 6, `Team` class

Given the `Player` interface in Listing 6.5 a development group could develop a `Team` class as shown in Listing 6.6. Things to note in this listing are:

- A team comprises players - in line 8 a list `members` is defined as a list of `Player`. Any object can be treated as an instance of `Player` as long as the object's class implements the `Player` interface.
- The `addPlayer` method in lines 15-18 is used to add players to a team.
- The `display` method in lines 20-26 iterates through the list of players using interface methods to get information from each player.

Listing 6.6: Team consists of players.

```
 1 import java.util.ArrayList;
 2 /**
 3  * A team comprises several players
 4  */
 5 public class Team
 6 {
 7     private String teamName;
 8     private ArrayList<Player> members;
 9     public Team(String name)
10     {
11         teamName = name;
12         members = new ArrayList();
13     }
14 
15     public void addPlayer(Player p)
16     {
17         members.add(p);
18     }
19 
20     public void display(){
21         System.out.println(teamName);
22         for (Player p: members)
23             System.out.println(p.getName()
24                 +" \t"+ p.getJerseyNumber()
25                 +" \t"+ p.getPosition());
26     }
27 }
```

Example 7, Persons can be players

Suppose the developer group tasked with developing Player-related software realizes the Person class has some of the capabilites of a player and can be modified for the rest (see Listing 6.7). The group just has to modify the code adding more fields (position and jersey number) and any missing methods of the Player interface. Some points to note in the listing are:

- Line 2 states that Person implements two interfaces: `Player` and `Comparable`.
- Lines 12-26 implementation of `Player` interface.
- Lines 28-30 implementation of `Comparable` interface.

Listing 6.7: Person modified to be a player.

```
 1  public class Person
 2       implements Comparable<Person>, Player {
 3      private String name;
 4      private int year; // year of birth
 5      private String position;
 6      private int jerseyNumber;
 7      public Person(String n, int y) {
 8          name = n;
 9          year = y;
10      }
11      // methods of the Player interface
12      public String getName(){
13          return name;
14      }
15      public String getPosition(){
16          return position;
17      }
18      public int getJerseyNumber(){
19          return jerseyNumber;
20      }
21      public void setPosition(String p){
22          position = p;
23      }
24      public void setJerseyNumber(int j){
25          jerseyNumber = j;
26      }
```

```
    // methods of the Comparable interface
    public int compareTo (Person p) {
        return name.compareTo(p.name);
    }
    // original methods of Person
    public void setName(String n){
        name = n;
    }
    public int getYear(){
        return year;
    }
    public void setYear(int y){
        year = y;
    }
    public String toString(){
        return name;
    }
}
```

Example 8, Creating a team of players

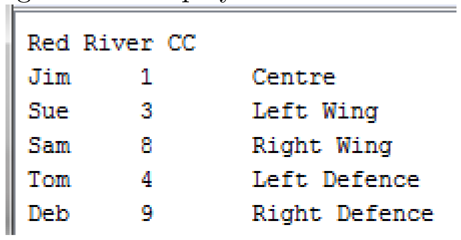
To complete our sports example we present the class below in Listing 6.8 and its output in Figure 6.8. The `main` method instantiates Person objects and adds those Person objects as Players to a team. In lines 5-19 there are five `Person` objects instantiated but the variables `p0`, `p1`, `p2`, `p3`, `p4` are of type `Player`. Because `Person` implements `Player` those objects can be referenced using `Player` variables. The `Team display` method is called in line 28.

Listing 6.8: Create persons that are added as players to a team.

```
 1 public class CreateTeam {
 2     private Team team;
 3     public static void main(String[] args) {
 4         // create 5 persons but know them as Players
 5         Player p0 = new Person("Jim",1978);
 6         p0.setPosition("Centre");
 7         p0.setJerseyNumber(1);
 8         Player p1 = new Person("Sue",1962);
 9         p1.setPosition("Left Wing");
10         p1.setJerseyNumber(3);
11         Player p2 = new Person("Sam",1975);
12         p2.setPosition("Right Wing");
13         p2.setJerseyNumber(8);
14         Player p3 = new Person("Tom",1975);
15         p3.setPosition("Left Defence");
16         p3.setJerseyNumber(4);
17         Player p4 = new Person("Deb",1966);
18         p4.setPosition("Right Defence");
19         p4.setJerseyNumber(9);
20         // create a team
21         Team rr = new Team("Red River CC");
22         // add players to the team and display
23         rr.addPlayer(p0);
24         rr.addPlayer(p1);
25         rr.addPlayer(p2);
26         rr.addPlayer(p3);
27         rr.addPlayer(p4);
28         rr.display();
29     }
}
```

The output of `CreateTeam` is the list of players on the team:

Figure 6.8: Display of team members.



```
Red River CC
Jim     1       Centre
Sue     3       Left Wing
Sam     8       Right Wing
Tom     4       Left Defence
Deb     9       Right Defence
```

6.3 `Comparator` Interface

Recall the `Comparable` interface is a way of implementing a natural ordering for a collection of objects. However, there are times when one needs to compare objects based on some other criteria. Perhaps we usually sort our persons by name, but there could be times when it is required to sort based on height, weight, age, etc. This is when the `Comparator` interface becomes useful. We can use the `Comparator` interface and implement an alternate ordering for a class of objects. The `Comparator` interface, like `Comparable`, specifies one method that must be implemented. This method is named `compare`. The `compare` method returns an `int`, just as `compareTo`, to indicate for two objects, say `o1` and `o2`, which of the following is true:

```
o1 less than o2
o1 equals o2
o1 greater than o2
```

Suppose we want to order our persons based on their year of birth instead of name. To arrange for such an ordering we can define the `Comparator` class as shown in Listing 6.9:

Listing 6.9: A comparator for year of birth.

```
1 import java.util.Comparator;
2 public class OrderByYear
3                     implements Comparator<Person>
4 {
5     public int compare(Person o1, Person o2) {
6         return o1.getYear()-o2.getYear();
7     }
8 }
```

Some comments regarding the above comparator:

Line 1 For the code to compile we must specify an import for `Comparator`.
Line 2 We have named our Comparator class OrderByYear.
Line 3 We indicated this is a `Comparator` for `Person` objects.
Line 5 This begins the definition of a compare method for persons. Note there are two arguments . . . two objects of type `Person`.
Line 6 We must return an `int`. The expression `o1.getYear()-o2.getYear()` will yield an `int` that is

- Negative . . . o1's year of birth $<$ o2's year of birth
- Positive . . . o1's year of birth $>$ o2's year of birth
- Zero . . . o1's year of birth $==$ o2's year of birth

Example 9, Using a `Comparator`

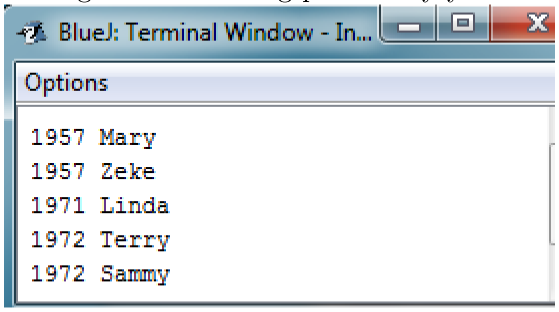
Now we will show how to use a comparator for sorting purposes. There are overloaded versions for `Arrays.sort` and `Collections.sort` that provide for a second argument, an object of type `Comparator`. Consider Listing 6.10 that creates, sorts (according to year of birth), and then displays the sorted objects. Note the call to sort in line 12 with the additional parameter, the comparator instance.

Listing 6.10: Sort persons by year of birth.

```
1  import java.util.Arrays;
2  public class SortingPersonsByYearWithComparator
3  {
4     public static void main(String[] args){
5         Person [] people = {
6             new Person("Mary", 1957),
7             new Person("Terry",1972),
8             new Person ("Zeke", 1957),
9             new Person("Sammy",1972),
10            new Person("Linda", 1971)};
11        // sort with a comparator
12        Arrays.sort(people, new OrderByYear());
13        for (Person p: people)
14           System.out.println(p.getYear()+" "+p);
15    }
16 }
```

The output lists the persons in sequence from oldest to youngest:

Figure 6.9: Sorting persons by year.

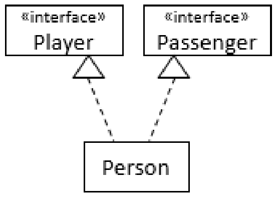


```
1957 Mary
1957 Zeke
1971 Linda
1972 Terry
1972 Sammy
```

6.4 Summary

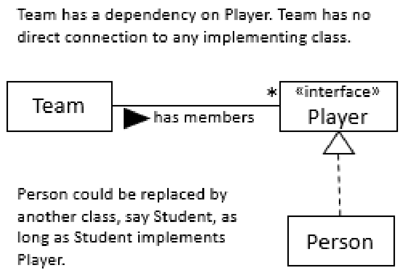
One of the things that is special about interfaces is that a class can *implement* any number of interfaces. If we have a class that implements several interfaces then we can treat objects of that class as multiple types. If we develop an interface, say `Passenger`, and we have that `Person` implements both `Passenger` and `Player`, then we can say that some `Person` object can be referenced as a passenger, and/or, referenced as a player. We illustrate this with a UML class diagram:

Figure 6.10: UML diagram showing Person implementing 2 interfaces.



When complex software is being developed there may be several groups involved developing different parts. If different programming groups are working to implement a system, then if they agree on a set of interfaces then the groups can work independently to develop their own code. Interfaces can be considered a sort of contract which defines how to interact with software. In this chapter we saw a `Team` class that depends only on something called `Player`. The `Team` class knows nothing about the implementation of `Player` - `Team` is not aware of the `Person` class. In the future `Person` could be replaced by some other class and `Team` would continue to function as long as the replacing class implements the methods of `Player`.

Figure 6.11: UML diagram, Team depends only only on Player.


Team has a dependency on Player. Team has no direct connection to any implementing class.
Team
has members
*
«interface»
Player
Person could be replaced by another class, say Student, as long as Student implements Player.
Person

There are many examples of interfaces in the Java class libraries. For instance if you examine the documentation for the String class you would see it implements 3 interfaces, `Serializable` and `CharSequence` in addition to `Comparable`.

Comparator is another useful interface as it provides a modular approach a programmer can adopt to provide alternate orderings for objects. For any class we can define several comparators.

We have not given any examples of constants or default methods. The interested reader is referred to other texts such as *Java in a Nutshell* and *Effective Java*.

6.5 Exercises

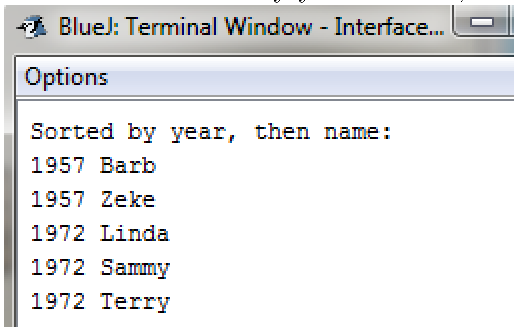
1. Example 3 uses a natural ordering based on names. Modify the `Person` class in Listing 6.3 so the natural ordering is based on year of birth, but where two or more persons have the same year of birth they are ordered by name. Verify your implementation using Listing 6.11. In the listing note the import for `Arrays` in line 1, the definition of `people` in lines 5-10, and the use of `sort` in line 11.

   Listing 6.11: Create, display, sort, display practitioners.

```
 1 import java.util.Arrays;
 2 public class SortingPersonsByYearAndName
 3 {
 4    public static void main(String[] args){
 5        Person [] people = {
 6            new Person("Terry",1972),
 7            new Person("Linda", 1972),
 8            new Person ("Zeke", 1957),
 9            new Person("Sammy",1972),
10            new Person ("Barb", 1957)};
11        Arrays.sort(people);
12        System.out.println("Sorted by year, then
             name:");
13        for (Person p: people)
14           System.out.println(p.getYear()+" "+p);
15     }
16 }
```

   The output should be:

   Figure 6.12: Persons sorted by year of birth, then name.



```
Sorted by year, then name:
1957 Barb
1957 Zeke
1972 Linda
1972 Sammy
1972 Terry
```

2. Consider the `Practitioner` hierarchy in Chapter 5.
   Modify `Practitioner` so it implements `Comparable`. The ordering of practitioners should be based on last names and first names: Where two or more practitioners have the same last name then those should be in sequence by first name. You can use Listing 6.12 to verify your implementation. In the listing note the `import` statement for `Collections` in line 1, and the use of the `sort` method in line 16. Note also that the list `practitioners` holds objects of type `Practitioner`, `Doctor`, and `Pharmacist` (lines 6-13).

Listing 6.12: Create, display, sort, display practitioners.

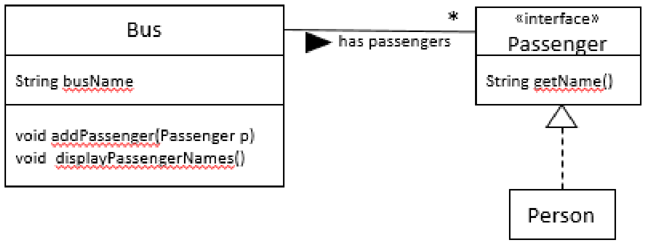
```
 1 import java.util.Collections;
 2 import java.util.ArrayList;
 3 public class SortPractitioners {
 4     public static void main(String[] args){
 5         // List of practitioners
 6         ArrayList<Practitioner> practitioners =
              new ArrayList();
 7         // Create some practitioners
 8         practitioners.add(new
              Practitioner("Sid","Smith","female"));
 9         practitioners.add(new
              Practitioner("Sam","Smith","male"));
10         practitioners.add(new
              Doctor("Jill","Jones","female",
11                                    "Dermatology"));
12         practitioners.add(new
              Pharmacist("Eddy","Jones","male",
13                                    "Drugco"));
14         // Collections class contains the sort
15         // method for array lists
16         Collections.sort(practitioners);
17         for (Practitioner p: practitioners)
18             System.out.println(p);
19     }
20 }
```

3. Consider the `Practitioner` hierarchy in Chapter 5 again.
   Create a `Comparator` class that defines an ordering for practitioners based on gender. Where practitioners are of the same gender, they must by ordered by last name, and where they have the same last name practitioners are ordered by first name. Develop this comparator and then test your code using a class similar to Listing 6.12 but where the call to `sort` is changed appropriately.

4. Suppose we need to display the members of a team in jersey number order. We could provide for this by creating a comparator that compares two players based on their jersey numbers. Develop this comparator and then modify the `display` method in the `Team` class so that sort is called prior to displaying the members. Test your code using the CreateTeam class in Listing 6.8.

5. Develop the `Bus` class and the `Passenger` interface indicated in the UML diagram below, and make any necessary changes to `Person` so that person objects can be passengers on a bus. Add necessary features suggested for buses such as a name for a bus, a method to add a passenger to a bus, and a method to display the names of a bus's passengers. Develop a class to test your code. You must add several persons to the passenger list of a bus and then display the names of those passengers.

Figure 6.13: Buses have several passengers.



6. Develop a comparator for Passenger that provides an ordering for passengers based on their names. Develop a class to test your code. You must add several persons to the passenger list of a bus and then display the names of those passengers in alphabetical order.

Chapter 7

Files

In this chapter we discuss storage and retrieval of Java data and objects in files. A useful way to categorize files is whether they are binary files or text files. We are quite familiar with the notion of a text file as these are generally considered to be human-readable. For instance when you edit the source code for a Java class we say that the source is stored in a text file. You could open a .java file in many editors and see exactly what you expect to see - readable Java statements. In contrast to text files there are binary files where information is stored as it exists in memory. If the data is say, an `int`, then the data is encoded in 32 bits as a binary number. Unless you are very familiar with binary numbers, you will have great difficulty understanding what is present. When Java source code is compiled and a .class file is generated we would say that the .class file is a binary file - understandable only by the JVM.

Previously in Part I of these notes (2.6.2 Redirecting System.out and 4.3 Scanner) we saw that we could use print and println to write to a text file, and that we can read information in from a text file. In this chapter we will expand our knowledge regarding files. We will consider binary files and XML files. XML is an encoding that yields self-describing data.

First we examine the reading and writing of primitive data (int, double, etc.) and strings and show how to accomplish this for binary and XML files. Then we examine how we can read and write objects.

7.1 Primitive data and strings

We are concerned in this section with strings and primitive data fields of type int, double, boolean, ... We begin with writing and reading for binary files and then we consider XML files.

7.1.1 Binary files

A binary file contains data encoded as it would be in memory. As such these files hold the data in a compact form. Being a binary format, the data is generally unreadable by humans, but well-suited to computers. In this section our examples work with a file named myData.ser.

Writing primitive data and strings to a binary file

If we can write data to a binary file then we are storing data in a most efficient way in terms of space and time. Essentially the data is just transferred from memory to disk.

The technique we present here is based on the use of two classes: `FileOutputStream` and `DataOutputStream`. The methods we use from `DataOutputStream` are specific to the type of data we are writing. For instance to write data of type `int` we use `writeInt`, and to write out data of type `String` we use `writeUTF` - see the table following the program listing. An example appears in Listing 7.1. Some points of interest for this program are:

Lines 1-3. There are imports for `DataOutputStream`, `FileOutputStream`, and `IOException`.

Lines 8-10 A `DataOutputStream` object named `os` references `myData.ser`.

Line 11. Five `int` values are written using `writeInt`.

Line 12. The file is closed to free resources and allow other programs to use the file.

Listing 7.1: Writing data out to a binary file.

```java
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteBinary {
   public static void main(String[] args)
                                 throws IOException {
      int[] myData = {5, 20, 30, 2, 7};
      DataOutputStream  os
         = new DataOutputStream(
               new FileOutputStream("myData.ser"));
      for (int i=0; i<5; i++)
         os.writeInt(myData[i]);
      os.close();
   }
}
```

The table below lists the methods available for writing primitive data types and strings. `writeUTF` encodes string data using UTF-8. This approach means that any character in the Unicode standard can be represented, and each character is represented in 1, 2, 3, or 4 bytes.

Methods of DataOutputStream
close()
writeBoolean(boolean b)
writeByte(byte b)
writeChar(int c)
writeDouble(double d)
writeFloat(float f)
writeInt(int i)
writeLong(long m)
writeShort(short s)
writeUTF(string s)

Reading primitive data and strings from a binary file

In the previous section we showed how to write data to a file and now we present how we can retrieve the data back in a separate program at a later time. The Java classes we need mirror the ones we used for output: DataInputStream and FileInputStream. The methods are similar too: for reading an int we use readInt() and so on. These are listed in the table following the program listing.

Listing 7.2 reads the data in myData.ser into an `int` array. The values are displayed and the output is shown in Figure 7.1. Points of interest in this program are:

Lines 1-3. There are imports for `DataInputStream`, `FileInputStream`, and `IOException`.

Lines 8-10 A data `DataInputStream` object references `myData.ser`.

Lines 12-13. Five `int` values are read using `readInt`.

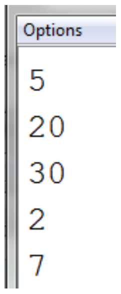
Line 15 The values are displayed

Line 16. The file is closed.

Listing 7.2: Reading data from a binary file.

```
 1 import java.io.DataInputStream;
 2 import java.io.FileInputStream;
 3 import java.io.IOException;
 4 public class ReadBinary {
 5    public static void main(String[] args)
 6                                 throws IOException {
 7       int[] myData = new int[5];
 8       DataInputStream  is
 9          = new DataInputStream(
10                new FileInputStream("myData.ser"));
11       // get values from file into array
12       for (int i=0; i<5; i++)
13          myData[i] = is.readInt();
14       // display values in array
15       for (int i: myData)System.out.println(i);
16       is.close();
17    }
18 }
```

Figure 7.1: The values previously written are obtained from the binary file.



```
5
20
30
2
7
```

In the table below note the method `readUTF` - it is used to read string data.

Methods of DataInputStream
close()
readBoolean()
readByte()
readChar()
readDouble()
readFloat()
readInt()
readLong()
readShort()
readUTF()

7.1.2 XML files

XML is a text format for self-describing and human-readable information. The XML format serves many purposes - for example as a medium for transferring data between applications. Here we are concerned with writing data of primitive types and strings out to an XML formatted file and being able to retrieve that data later on in a separate program. Our examples use a file named `myData.xml`

Writing primitive data and strings to XML

The two Java classes we will use are `XMLEncoder` and `FileOutputStream`. To write data we will use the `writeObject` method. The program in Listing 7.3 creates a file and writes the same array of `int` we used previously. Some points of interest for this program are:

Lines 1-3. There are imports for `XMLEncoder`, `FileOutputStream`, and `IOException`.

Line 7 An `XMLEncoder` object is created to reference `myData.xml`.

Lines 9-10. Five `int` values are written using `writeObject`.

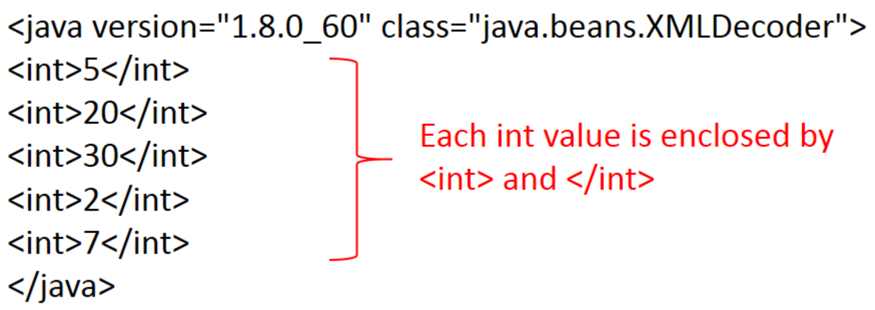
Line 11. The file is closed.

Listing 7.3: Writing data out to an XML file.

```
 1 import java.beans.XMLEncoder;
 2 import java.io.FileOutputStream;
 3 import java.io.IOException;
 4 public class WritePrimitiveDataToXML {
 5     public static void main(String[] args)
 6            throws IOException{
 7       XMLEncoder encoder = new XMLEncoder(new
            FileOutputStream("myData.xml"));
 8       int[] myData = {5, 20, 30, 2, 7};
 9       for (int i=0; i<5; i++)
10           encoder.writeObject(myData[i]);
11       encoder.close();
12     }
13 }
```

The content of the file myData.xml is shown in Figure 7.2. As you can see it is human-readable, and its easy to see where and how the `int` values are encoded. Next we will examine how to read such information back into a program.

Figure 7.2: The contents of an XML file.



```
<java version="1.8.0_60" class="java.beans.XMLDecoder">
<int>5</int>
<int>20</int>
<int>30</int>
<int>2</int>
<int>7</int>
</java>
```

Reading primitive data and strings from XML

In the previous section we showed how to write primitive data to an XML file and now we consider retrieving the same data back in a separate program at a later time. The Java classes we need mirror the ones we used for output: XMLDecoder and FileInputStream. To read data in we use `readObject`.

Listing 7.4 reads the data in myData.xml into an `int` array. The `readObject` method gets the next data encoded in the file, but to use such a value in a program we must *cast* the object to its type. The values are displayed and the output is shown in Figure 7.3. Points of interest in this program are:

Lines 1-3. Imports for `XMLDecoder`, `FileInputStream`, and `IOException`.

Line 8. An `XMLDecoder` object is created that references `myData.xml`.

Lines 11-12. Five `int` values are read using the `readObject` method and cast to `int`.

Lines 14-15. The values are displayed

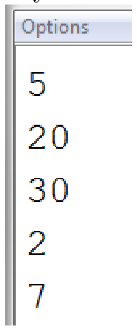
Line 17. The file is closed.

Listing 7.4: Reading primitive data from an XML file.

```
 1 import java.beans.XMLDecoder;
 2 import java.io.FileInputStream;
 3 import java.io.IOException;
 4 public class ReadPrimitiveDataFromXML {
 5     public static void main(String[] args)
 6             throws IOException{
 7       // decoder object references the XML file
 8       XMLDecoder decoder = new XMLDecoder(new
            FileInputStream("myData.xml"));
 9       // get the five int values
10       int[] myData = new int[5];
11       for (int i=0; i<5; i++)
12           myData[i] = (int) decoder.readObject();
13       // display the array and close the file
14       for (int i: myData)
15           System.out.println(i);
16       decoder.close();
17     }
18 }
```

Figure 7.3 is the output from the program - the values displayed are the same values written to the file previously in Listing 7.3

Figure 7.3: The values previously written are read from the XML file.



```
5
20
30
2
7
```

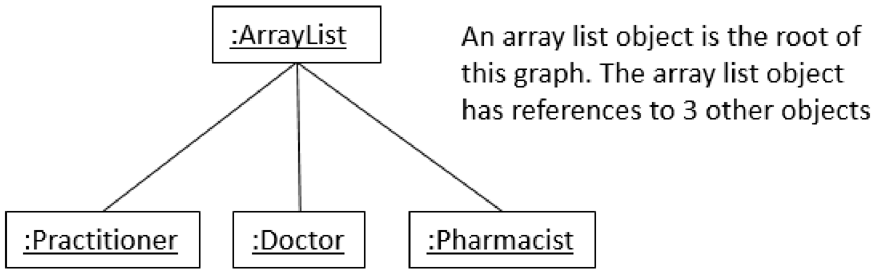
7.2 Objects

In this section we consider two techniques a programmer can use to save, and restore, the state of Java objects. When an object is converted into a sequence of bytes we say the object is *serialized*. Serializing an object can be to a binary form or to an XML form. The general idea is to write the state of objects out to a file, and then at some later time, restore the objects by reading the objects back in.

When an object is written/read, it is the entire object graph which starts with the object that is processed. An object graph has, as its root, one object and includes all the objects that are reachable. The object graph includes all objects that are referenced from the root object, plus all those objects referenced from those, and so on until there are no further referenced objects.

We will work with an example that involves practitioners (recall the Practitioner hierarchy from Section 5.2). The object graph we will process is shown in Figure 7.5.

Figure 7.4: An object graph.


:ArrayList
An array list object is the root of this graph. The array list object has references to 3 other objects
:Practitioner
:Doctor
:Pharmacist

The first technique we cover serializes the objects in a binary form and the second stores objects in an XML form. The XML form is human-readable - the binary form is not.

7.2.1 Binary files

To serialize an object to a binary file the objects must belong to a class that implements the interface `Serializable`. This interface has no methods - it is a so-called *marker* interface. As such, the compiler and JVM know a class can be serialized.

In our examples below we use a binary file named `practitioners.ser`. The first example will create a binary file and the second example shows that we can read that file and re-instantiate objects at a later time.

Writing objects to a binary file

To serialize objects to a binary format we use the Java classes `FileOutputStream` and `ObjectOutputStream`, and we use the `writeObject` method in `ObjectOutputStream`. Recall Listing 5.2 where Practitioner.java is defined; we need to alter the `Practitioner` class so it implements Serializable. So the class header becomes:

```
public class Practitioner implements Serializable
```

and now we are able to write the practitioners to a binary file.

Recall Listing 5.5; we can modify this to write out the whole list of practitioners - see Listing 7.5 Some important points about this listing are:

Lines 1-3. Import statements included for `ObjectOutputStream`, `FileOutputStream`, `IOException`.

Line 17. An `ObjectOutputStream` object references `practitioners.ser`.

Line 20. The `writeObject` method is used to write an object graph.

Line 21. The file is closed.

Listing 7.5: Writing practitioners to a binary file.

```java
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersToBinary {
   public static void main(String[] args) throws
      IOException {
      // List of practitioners
      ArrayList<Practitioner> practitioners = new
         ArrayList();
      // Create some practitioners
      Practitioner pr = new
         Practitioner("Sam","Smith","female");
      Doctor       dr = new
         Doctor("Jill","Jones","female","Dermatology");
      Pharmacist   ph = new
         Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
      // create an object to reference
         practitioners.ser
      ObjectOutputStream os = new
         ObjectOutputStream( new
         FileOutputStream("practitioners.ser"));
      // write out the object grapsh
      // that begins with the ArrayList
      os.writeObject(practitioners);
      os.close();
   }
}
```

Reading objects from a binary file

The previous program in Listing 7.5 creates a binary file that holds an `ArrayList` of three practitioners. Listing 7.6 shows that a binary file can be read to re-instantiate objects. Some important points about the program are:

Lines 1-3. Import statements included for `ObjectInputStream`, `FileInputStream`, `IOException`.

Line 6. As well as the chance of throwing an `IOException`, the class could possibly throw a `ClassNotFoundException`. This can arise if the binary file indicates an object should be instantiated from a class the JVM is not aware of.

Line 7. An `ObjectInputStream` object references `practitioners.ser`.

Line 12. The `readObject` method is used to obtain an object graph. This is of type Object and is cast into the appropriate type, `ArrayList`. When the ArrayList is instantiated all objects it holds are instantiated - that is, the object graph is instantiated. Hence, for our example a total of four objects are created.

Line 13. The file is closed.

Lines 14-19. The practitioner objects are displayed - the output is the same as that shown in Figure 5.4.

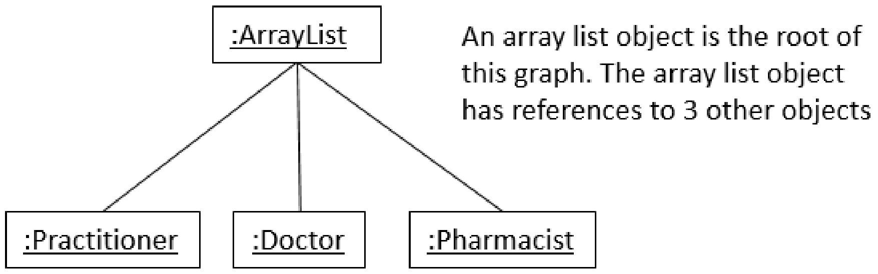
Listing 7.6: Obtaining practitioners from a binary file.

```java
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersFromBinary {
   public static void main(String[] args) throws IOException, ClassNotFoundException{
      ObjectInputStream is = new ObjectInputStream(new FileInputStream("practitioners.ser"));
      // The JVM only knows the object read as being of type Object.
      // Since we know the object being read is of type ArrayList
      //    we include a cast to type ArrayList to the right of
      //    the assignment operator.
      ArrayList<Practitioner> practitioners = (ArrayList) is.readObject();
      is.close();
      for (Practitioner p: practitioners) {
         String type="practitioner";
         if (p instanceof Doctor) type="doctor";
         if (p instanceof Pharmacist) type="pharmacist";
         System.out.println(type+" "+p.getFirstName());
      }
   }
}
```

7.2.2 XML files

In this section we consider serializing objects to XML. Our examples will use an XML file named `practitioners.xml`, and the same object graph as in the previous section, shown again below. The first example will create

Figure 7.5: An object graph.


:ArrayList
An array list object is the root of this graph. The array list object has references to 3 other objects
:Practitioner
:Doctor
:Pharmacist

the XML file and the second example shows that we can read that file and re-instantiate objects at a later time.

Writing objects to XML

We will use the XMLEncoder class again. In order to write objects using XMLEncoder we must have objects instantiated from clases that have:

- a no-arg constructor
- getters
- setters

Recall Listing 5.5; with a few small changes to Practitioners.java we can write the practitioners array list out to an XML file (see Listing 7.7 where `PractitionersToXML.java` is defined). Some important points about this listing are:

Lines 1-3. Import statements included for `XMLEncoder`, `FileOutputStream`, `IOException`.

Line 17-18. An `XMLEncoder` object references `practitioners.xml`.

Line 20. The `writeObject` method is used to write an object graph.

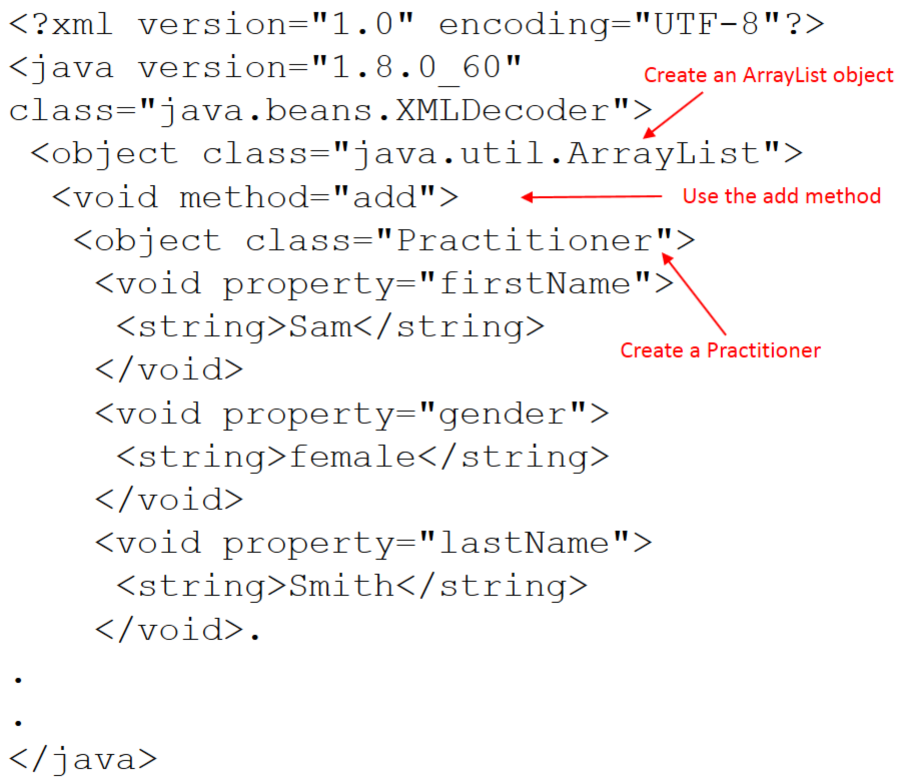
Line 22. The file is closed.

Listing 7.7: Writing practitioners to XML.

```
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersToXML {
   public static void main(String[] args) throws
      IOException{
      // List of practitioners
      ArrayList<Practitioner> practitioners = new
         ArrayList();
      // Create some practitioners
      Practitioner pr = new
         Practitioner("Sam","Smith","female");
      Doctor       dr = new
         Doctor("Jill","Jones","female","Dermatology");
      Pharmacist   ph = new
         Pharmacist("Eddy","Edwards","male","Drugco");
      practitioners.add(pr);
      practitioners.add(dr);
      practitioners.add(ph);
      // the encoder object references the file
      XMLEncoder  encoder = new XMLEncoder(
        new FileOutputStream("practitioners.xml"));
      // write out the practitioner object graph
      encoder.writeObject(practitioners);
      // close the xml file
      encoder.close();
   }
}
```

In Figure 7.6 we show the first few lines of the XML file created by PractitionersToXML.java. Typically you can double-click on the file name (using for example, Windows Explorer) and the XML file will be presented in your web browser. It is human-readable. The information contained is essentially the information required to re-create objects. For instance, the first object to create is an `ArrayList` and then the `add` method must be used to add an object.

Figure 7.6: The first few lines of the XML file.



```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_60"
class="java.beans.XMLDecoder">
 <object class="java.util.ArrayList">
  <void method="add">
   <object class="Practitioner">
    <void property="firstName">
     <string>Sam</string>
    </void>
    <void property="gender">
     <string>female</string>
    </void>
    <void property="lastName">
     <string>Smith</string>
    </void>.
.
.
</java>
```

Create an ArrayList object
Use the add method
Create a Practitioner

Reading objects from XML

To read objects from an XML file we use the XMLDecoder class. When an object is read it is an object graph that is read.

Recall Listing 5.5 (Practitioners.java) where there is a `for` loop which steps through the list displaying information about each practitioner. Listing 7.8 (PractitionersFromXML.java) obtains the ArrayList of practitioners from the XML file `practitioners.xml` and then displays those practitioners. Some important points about this program:

Lines 1-3. Import statements for `XMLDecoder`, `FileInputStream`, and `IOException`.

Line 8. The `XMLDecoder` object references `practitioners.xml`.

Lines 13-14. The `readObject` method obtains the object graph. In our example `readObject` executes once and four objects are obtained. When an object is read it is considered to be of type Object. So in the statement:

```
ArrayList<Practitioner> practitioners =
                      (ArrayList)decoder.readObject();
```

we use a cast `(ArrayList)` so the Java compiler accepts the assignment to the variable practitioners which is of type `ArrayList`.

Line 15. The file is closed.

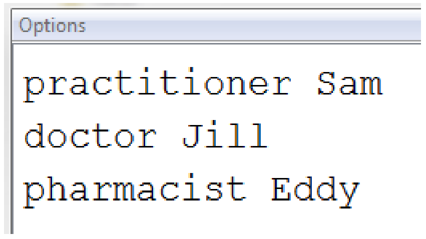
Lines 17-22. The practitioners are displayed generating the output shown in Figure 7.7

Listing 7.8: Retrieving practitioners from XML.

```java
import java.beans.XMLDecoder;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
public class PractitionersFromXML {
   public static void main(String[] args) throws
      IOException{
      // decoder object references the xml file
      XMLDecoder decoder = new XMLDecoder( new
         FileInputStream("practitioners.xml"));
      // The JVM only knows the object read as
         being of type Object.
      // Since we know the object being read is of
         type ArrayList
      //    we include a cast to type ArrayList to
         the right of
      //    the assignment operator.
      ArrayList<Practitioner> practitioners =
                   (ArrayList) decoder.readObject();
      decoder.close();
      // display the practitioners, doctors, etc.
      for (Practitioner p: practitioners) {
         String type="practitioner";
         if (p instanceof Doctor) type="doctor";
         if (p instanceof Pharmacist)
            type="pharmacist";
         System.out.println(type+"
            "+p.getFirstName());
      }
   }
}
```

The output from the program is shown in Figure 7.7 and is exactly the same as that of Practitioners.java in Chapter 5.

Figure 7.7: Practitioners retrieved are from the file.



```
practitioner Sam
doctor Jill
pharmacist Eddy
```

7.3 Summary

- In this chapter we have considered how we can write individual values (primitive data and strings) to a file and retrieve those values later on in a separate program.
- We have also handled objects where the writing and reading was done on an object graph basis. When an object is read in it is only known as being of type `Object` and so it is necessary to use casting to assign the object to a variable of a specific type.
- We have considered both binary and XML files.
- In a later chapter we cover `Exceptions`. We will see then how we can use `try` and `catch` blocks to process a complete file without knowing how much data or lines are present.
- We have only introduced file processing; there is much more to processing files than covered here. The following are some topics you might follow up on at a later time in your studies.
  - By default we have been processing files sequentially - from the beginning to the end. Java has facilities to process binary files randomly where you can, for instance, read a portion of a file without having to read through all the data preceding the data you are after. Random access to data is particularly important in database systems.
  - In our examples a new file has been created each time the program was writing data out to a file. If the file existed previously the file's previous content was deleted. In contrast there is a way to write data out to a file preserving its contents - in this case one is appending data to an existing file.

7.4 Exercises

1. Consider the Practitioner class. Each practitioner has a first name, last name, and gender. Create some practitioners and then for each one write these strings to a file. Then, read the values back in and display them. Do not write out `Practitioner` objects, rather write out strings.

   a) do this with a binary file

   b) do this with an XML file

2. Consider the `Person` class in Listing 6.3 from chapter 6 Interfaces. Each person has a name and a year of birth. Write programs to serialize `Person` objects as indicated below.

   a) Write a program to create some persons and serialize the objects to a binary file. You will need to specify that `Person` implements `Serializable`. Write another program to recreate the persons from the binary file. Display each person to verify you have successfully reinstantiated those objects.

   b) Write a program to create some persons and serialize the objects to an XML file. The `Person` class has the necessary getters and setters, but you will need to add a no-arg constructor. Write another program to recreate the persons from the XML file. Display each person to verify you have successfully reinstantiated those objects.

Chapter 8

Exception Handling

An *exception* is an event that occurs during program execution that disrupts the normal flow. When an exception occurs an exception object is created that contains information about the situation and the state of the program at that time. The creation of an exception object, and its handoff to the Java runtime system, is called *throwing* an exception.

When an exception is thrown, the runtime system searches for a method to handle the exception. If one is not found the program is terminated with a stack trace. Up to now, this is the way we have handled exceptions. This approach is fine for the category of exceptions called *unchecked* exceptions. Unchecked exceptions are generally considered to be something that should never occur, and so if one does occur the program should terminate. An example of this is the ArrayIndexOutOfBounds exception. If an index value goes beyond the bounds of an array there must be a logic error in the program.

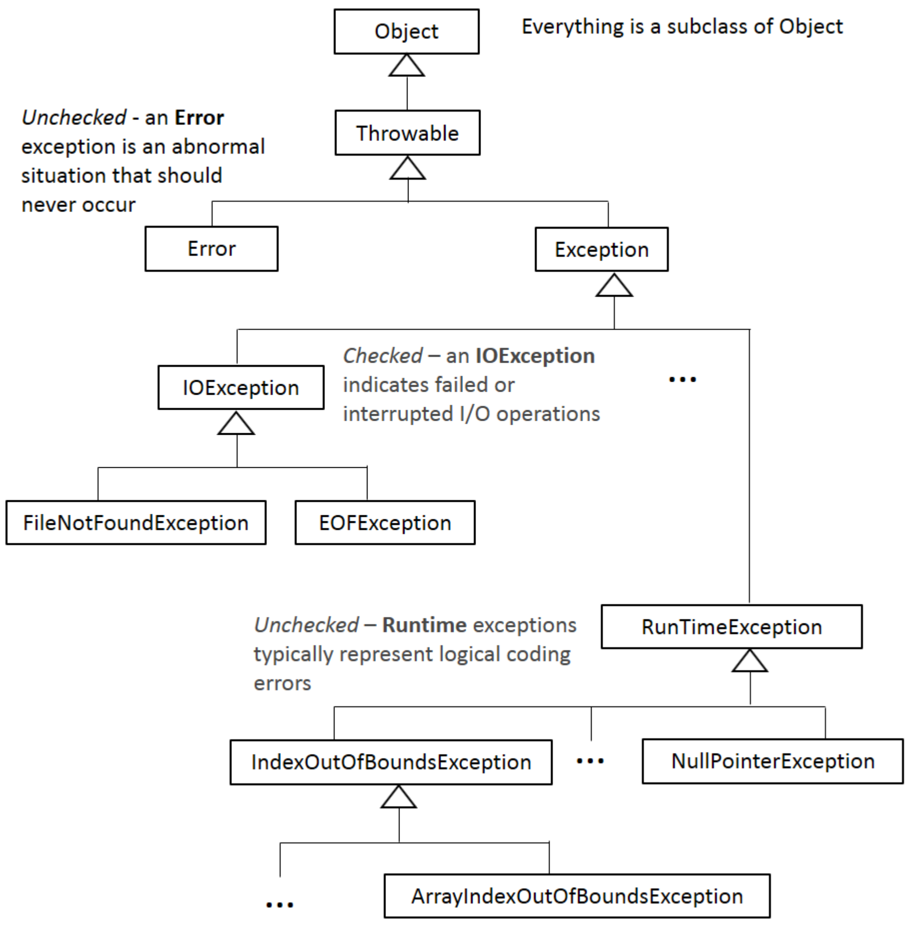
Another category of exception is called *checked*. These represent situations for which a program can be expected to recover from. An example of a checked exception is EOFException; another example is FileNotFoundException. When we encountered these cases before we saw that we needed to include a throws phrase to indicate we were aware of a possible error such as file not found. These events are situations where a program can easily recover. For example:

- An EOFException is a normal situation when a program is reading a file and the program cannot know how much data is available - the exception indicates to such a progam that all the information in the file has been accessed.

- The FileNotFound exception arises when a program tries to access a file that does not exist. Upon recognizing this situation a program might try looking elsewhere, or perhaps prompt the user for another name and/or location.

Figure 8.1 shows some classes that belong to the Exception Hierarchy. This chapter will focus on checked exceptions; we will cover how to catch an exception using a try statement, and how to create your own custom exception class.

Figure 8.1: Some classes in the Exception hierarchy.


Object
Everything is a subclass of Object
Throwable
Unchecked - an **Error** exception is an abnormal situation that should never occur
Error
Exception
IOException
Checked – an **IOException** indicates failed or interrupted I/O operations
...
FileNotFoundException
EOFException
Unchecked – **Runtime** exceptions typically represent logical coding errors
RunTimeException
IndexOutOfBoundsException
...
NullPointerException
...
ArrayIndexOutOfBoundsException

8.1 Catching an exception

If a checked exception could occur in a program then that program must either include a throws clause for it or include a try that catches the exception. Consider Listing 7.2 (ReadBinary) in the previous chapter on Files. In that program we used the DataInputStream class to read objects from a file. In that example we included a throws clause for IOException. In Figure 8.1 you can see IOException has two subclasses FileNotFoundException and EOFException. If a program has a throws for IOException then that also includes its subclasses FileNotFoundException and EOFException.

If the main method in Listing 7.2 executes and if the file myData.ser does not exist then a FileNotFoundException is thrown and the program terminates abruptly with a FileNotFoundException. Instead of letting that happen the program can include a try statment to catch and handle the exception. Handling FileNotFoundException is covered in Example 1 below.

Listing 7.2 reads exactly 5 integers. If there were fewer than 5 integers in the file then an EOFException would be thrown and the program would terminate abruptly with a EOFException. Instead of letting that happen we can include a try/catch to handle that type of error, and then we can have a program that is able to read a file with any number of integers in it. Reading a file until end-of-file is covered in Example 2.

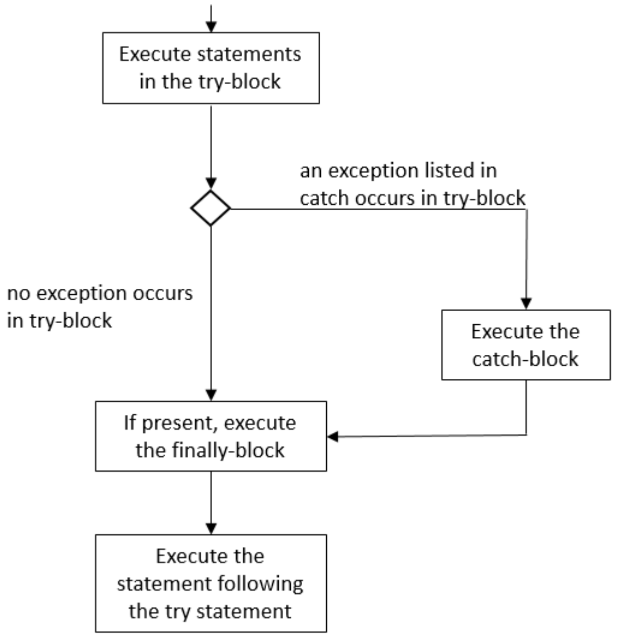
To catch checked exceptions and respond to them Java provides the try statement; its general form is:

```
try {
   some Java statements
}
catch ( one or more exceptions ){
   some Java statements
}
finally {
   some Java statements
}
```

As shown in Figure 8.2 the try-block executes first. If, during the execution of the try-block an exception occurs then the try-block is terminated immediately, and if that exception is listed in a catch-block, then control transfers to the catch-block. Regardless, the finally-block contains code to be executed before execution resumes at the statement following the try.

If an exception occurs and if the exception is not listed in the catch, then the method containing the `try` is terminated, the runtime environment searches for a handler in the calling routing and invokes that handler. If no handler is found then the program is terminated with a trace showing the called methods.

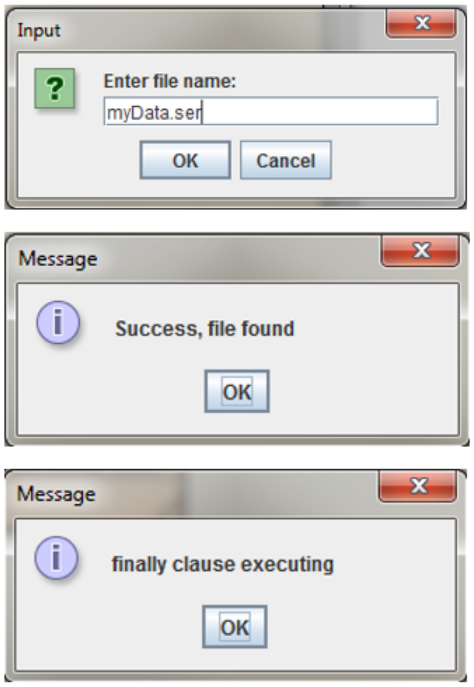
Figure 8.2: Execution of a try statement.


Execute statements in the try-block
an exception listed in catch occurs in try-block
no exception occurs in try-block
Execute the catch-block
If present, execute the finally-block
Execute the statement following the try statement

Example 1, File not found

Listing 8.1 prompts the user for a file name and attempts to create a reference to the file. The program uses message dialogs to get a file name, to display a message regarding the file's existence, and then displays a message dialog to show the finally-block executing. For example, if the file does exist the user sees message dialogs as shown in Figure 8.3.

Figure 8.3: Message dialogs.



Enter file name:
myData.ser

Success, file found

finally clause executing

Interesting points regarding this program are:

- There is an import for `FileNotFoundException` in line 3.
- The try statement in lines 10-22 has a try-block where a `DataInputStream` is created if the file exists. If the file does not exist then the system throws `FileNotFoundException`.
  - If creating a file reference works then line 12 executes - a *success* message appears.
  - If creating a file reference fails then `FileNotFoundException` is thrown immediately, line 12 does not execute - the catch-block

for the `FileNotFoundException` executes and an *error* message appears.

- Regardless, whether the file exists or not, the finally-block executes. The last dialog message will always appear. As indicated in the comments this is where many programs would close any open files - we have left this issue until the next example.

Listing 8.1: Handling file not found.

```java
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import javax.swing.JOptionPane;
public class HandleFileNotFound {
    public static void main(String[] args)
    {
        String fileName = JOptionPane.showInputDialog("Enter file name:");
        DataInputStream is = null;
        try {
            is = new DataInputStream( new FileInputStream(fileName));
            JOptionPane.showMessageDialog(null, "Success, file found");
        }
        catch (FileNotFoundException e){
            JOptionPane.showMessageDialog(null, "Error, file not found");
        }
        finally{
            JOptionPane.showMessageDialog(null, "finally clause executing");
            // Normally we would close open files here, but for DataInputStream
            // that involves another exception, and so we leave that to a later
            // example.
        }
    }
}
```

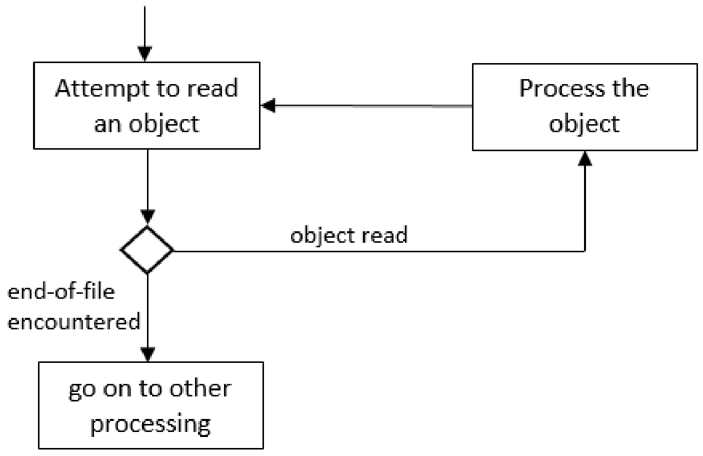
Example 2, Reading a binary file until end-of-file

In listing 7.1 an array of 5 integers was written to a binary file. In Listing 7.2 the array was read back in using a for statement that was executed exactly 5 times. That worked because we knew in advance how many integers had been written to the file.

If we do not know how many integers had been written then we must read integers until there are none left - that is, until end-of-file. Unlike the `Scanner` class, `DataInputStream` does not have a `hasNext` method to inform us if there are more objects. We must read objects using `readInt` until the `EOFException` occurs - which we can catch. If you read the documentation for `DataInputStream` you will see there is one more exception that can occur when a program executes `readInt`. This other exception is IOException that will cover any other unexpected problems when an integer is read.

In general, when reading until end-of-file we need a code structure illustrated as:

Figure 8.4: Basic structure to read until end-of-file.



The program in Listing 8.2 reads a binary file obtaining each integer and accumulating a total. There are imports for `EOFException`, `IOException`, and `FileNotFoundException`. The program uses `JOptionPane` to communicate with the user. To simplify we have used a main method that calls three methods:

1. Line 10, call `getFile`: to get the name of the file from the user.

   This method has a try statement that catches `FileNotFoundException`. A reference to a `DataInputStream` object is returned, or null if the file does not exist.

2. Line 13, call `getTotal`: to read the file accumulating a total until end-of-file.

   If you read the documentation for `DataInputStream` you will see there is a possibility of two exceptions being thrown when a program executes `readInt`. These two exceptions are `IOException` and `EOFException`. Recall from Figure 8.1 that IOException is the superclass of EOFException, and being the superclass `IOException` is more general and includes `EOFException`.

   This method has a try statement with two catch-blocks. The first catch is for `EOFException`, and the second is for `IOException`. Because EOFException is more specific that IOException, its catch-block is listed before the catch-block for `IOException`. When an exception occurs, the catch-blocks are examined in the order listed to find the first one that covers the exception.

   Regardless of which exception occurs the variable endOfFile is set to true so the while loop will terminate. If IOException occurs there is an addtional error message displayed to the user.

3. Line 14, call `closeFile`: to close the file.

   If you examine the documentation for `DataInputStream` you will see that when `close` is executed there is a possibility of an `IOException` occurring, and so the `try` statement includes a catch-block for `IOException`.

Listing 8.2: Retrieving objects until end-of-file.

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.FileNotFoundException;
import javax.swing.JOptionPane;
public class ReadFromBinaryUntilEndOfFile {
    public static void main(String[] args)
    {
        DataInputStream is = getFile();
        int total = 0;
        if ( is!=null ){
            total = getTotal(is);
            closeFile(is);
        }
        JOptionPane.showMessageDialog(null, "total
           = "+total);
    }

    public static DataInputStream getFile(){
        String fileName =
           JOptionPane.showInputDialog("Enter file
           name:");
        DataInputStream is = null;
        try {
            is = new DataInputStream( new
               FileInputStream(fileName));
            JOptionPane.showMessageDialog(null,
               "Success, file found");
        }
        catch (FileNotFoundException e){
            JOptionPane.showMessageDialog(null,
               "Error, file not found");
            is = null;
        }
        return is;
    }

```

```java
    public static int getTotal(DataInputStream is){
        int total = 0;
        // read integers until there are none left
        boolean endOfFile = false;
        int i;
        while (! endOfFile){
            try{
                // readInt can throw EOFException or an IOException
                i = is.readInt();
                total += i;
            }
            catch (EOFException e){
                // force loop to terminate
                endOfFile = true;
            }
            catch (IOException e){
                JOptionPane.showMessageDialog(null, "Error, an IOException occurred reading the file");
                endOfFile = true;
            }
        }
        return total;
    }
    public static void closeFile (DataInputStream is){
        // close may throw an IOException
        try{
            is.close();
            JOptionPane.showMessageDialog(null, "program terminated normally");
        }
        catch (IOException e){
            JOptionPane.showMessageDialog(null, "Error, an IOException occurred closing the file");
        }
    }
}
```

8.2 Designing a custom exception

Exceptions in Java are organized in a class structure. It is certainly possible for a programmer to define their own `Exception` subclass for the purposes of an application. In general this is not recommended. Before developing a separate exception class the extensive set of subclasses of `Throwable` should be examined for one that fits.

But if the need does arise it is fairly straightfoward to develop a custom Exception class. As an example, suppose the Human Resources (HR) department requires a system that involves employees. It has been decided that an employee will be described by 3 text fields : a social insurance number (SIN), a name, and a position. The HR department requires the SIN to be a valid SIN whenever an employee object is created.

We develop a custom exception for the case where there is an attempt to instantiate an employee with an invalid SIN. The constructor for an employee can recognize this case and include a statement such as

```
throw new InvalidSINException(sin);
```

The program that attempts to create an employee could include a `try`/`catch` such as:

```
try {
     e = new Employee("123456789","Joe Who","instructor");
}
catch (InvalidSINException e){
     System.out.println("Employee could not be created."+
     "The provided SIN, "+ e.getSin()+"is invalid ");
}
```

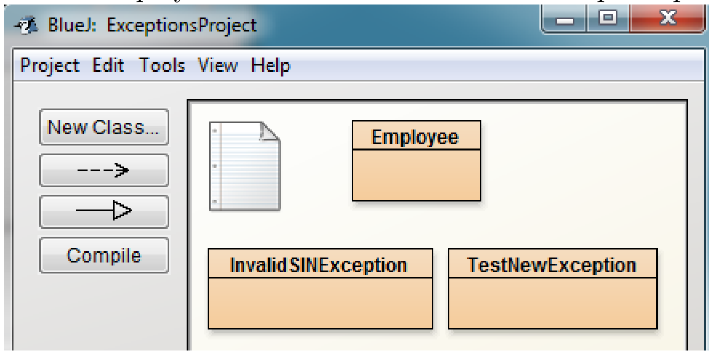
Normally a custom exception is defined as a Java class which extends either `Exception` or `RuntimeException`:

- You should subclass `Exception` if you require a checked exception
- You should subclass `Runtime` if you require an unchecked exception.

Example 3, Handling an invalid SIN

Our example involves three classes shown below in a BlueJ project:

Figure 8.5: Employee can throw an InvalidSINException project.



1. `InvalidSINException`, Listing 8.3.

   The custom exception class has a field for the social insurance number, a constructor, and a getter for `sin`.

2. `Employee`, Listing 8.4.

   The constructor validates the social insurance number field. If you wish to consider the details of the validiation you can consult the Wikipedia page on SIN - in brief a weighted sum is calculated that must be evenly divisible by 10. If the validation fails the constructor throws an exception. In line 6 note the constructor declares that is can throw an `InvalidSINException`, and in line 7 if `validate(sin)` returns false the exception is thrown - note the exception is created using the `new` operator.

3. `TestNewException`, Listing 8.5

   8.5 is a test program that creates two employees - the second create will fail. Note the catch-block uses the `printStackTrace` method defined in the superclass `Throwable`; this method prints the standard trace showing where the program failed.

Figure 8.6 shows the output when Listing 8.5 runs. Note the first instantiation of an employee works, but the second one fails with a call trace.

Listing 8.3: Custom exception for social insurance number.

```java
/**
 * Exception for invalid social insurance number
 * as a subclass of Exception, and so a calling
 * program must use try/catch or a throws clause.
 */
public class InvalidSINException extends Exception
{
    private String sin;
    /**
     * Constructor that accepts a SIN
     */
    public InvalidSINException(String s)
    {
        sin = s;
    }
    /**
     * getter for SIN that caused the exception
     */
    public String getSin(){
        return sin;
    }
}
```

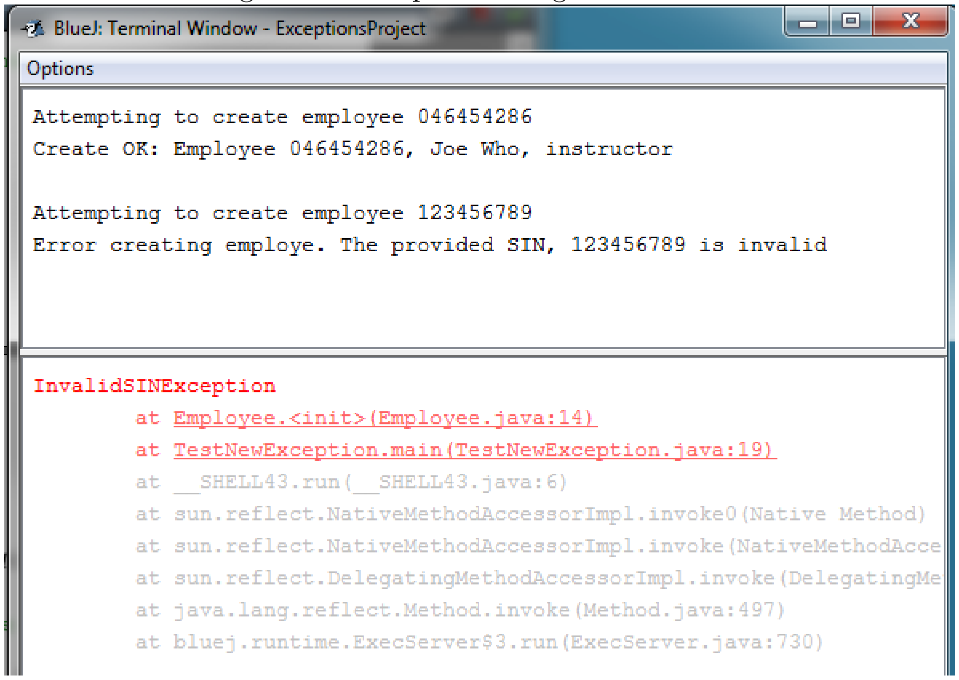
Listing 8.4: Employee class that throws InvalidSINException.

```java
public class Employee
{
    private String sin;
    private String name;
    private String position;
    public Employee (String sin, String name,
       String position) throws InvalidSINException{
        if (! validate(sin)) throw new
           InvalidSINException(sin);
        this.sin = sin;
        this.name = name;
        this.position = position;
    }

    public String getSin(){
        return sin;
    }

    public String getName(){
        return name;
    }

    public String getPosition(){
        return position;
    }

    public void setName(String name){
        this.name = name;
    }

    public void setPosition(String position){
        this.position = position;
    }

    public String toString(){
        return "Employee "+getSin()+",
           "+getName()+", "+getPosition();
    }

    public boolean validate(String sin){
        // validate SIN using rules in Wikipedia
        if (sin == null || sin.length()!=9) return
           false;
        char[] sinChars = sin.toCharArray();
        // validate the check sum for a SIN
        int total = 0;
        for (int i=0; i<9; i++){
            char c = sinChars[i];
            if (!Character.isDigit(c)) return false;
            int digit =
               Character.getNumericValue(c);
            // double every second digit
            if (i%2 == 1) digit *= 2;
            // correction if 10 or more
            if (digit > 9) digit -= 9;
            total += digit;
            // System.out.println(i+" "+c+"
               "+digit+" "+total);
        }
        return (total%10 == 0);
    }
}
```

Listing 8.5: Testing the custom exception.

```java
/**
 * Create two employees.
 * catch-block displays a stack trace if creating an employee fails.
 */
    public class TestNewException
{
    public static void main(String[] args)
    {
        try {
            // a valid SIN.
            System.out.println("Attempting to create employee 046454286");
            Employee e1 = new Employee("046454286", "Joe Who", "instructor");
            System.out.println("Create OK: "+e1);
            // not valid SIN.
            // An exception will be thrown
            System.out.println("\nAttempting to create employee 123456789");
            Employee e2 = new Employee("123456789", "Steve Stephens", "instructor");
            System.out.println("Create OK: "+e2);
        }

        catch (InvalidSINException e){
            System.out.println("Error creating employee. "+
              "The provided SIN, "+ e.getSin()+" is invalid ");
              e.printStackTrace();
        }
    }
}
```

Figure 8.6: Output showing a stack trace.



```
Attempting to create employee 046454286
Create OK: Employee 046454286, Joe Who, instructor

Attempting to create employee 123456789
Error creating employe. The provided SIN, 123456789 is invalid
```

```
InvalidSINException
        at Employee.<init>(Employee.java:14)
        at TestNewException.main(TestNewException.java:19)
        at __SHELL43.run(__SHELL43.java:6)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcce
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMe
        at java.lang.reflect.Method.invoke(Method.java:497)
        at bluej.runtime.ExecServer$3.run(ExecServer.java:730)
```

Exercises

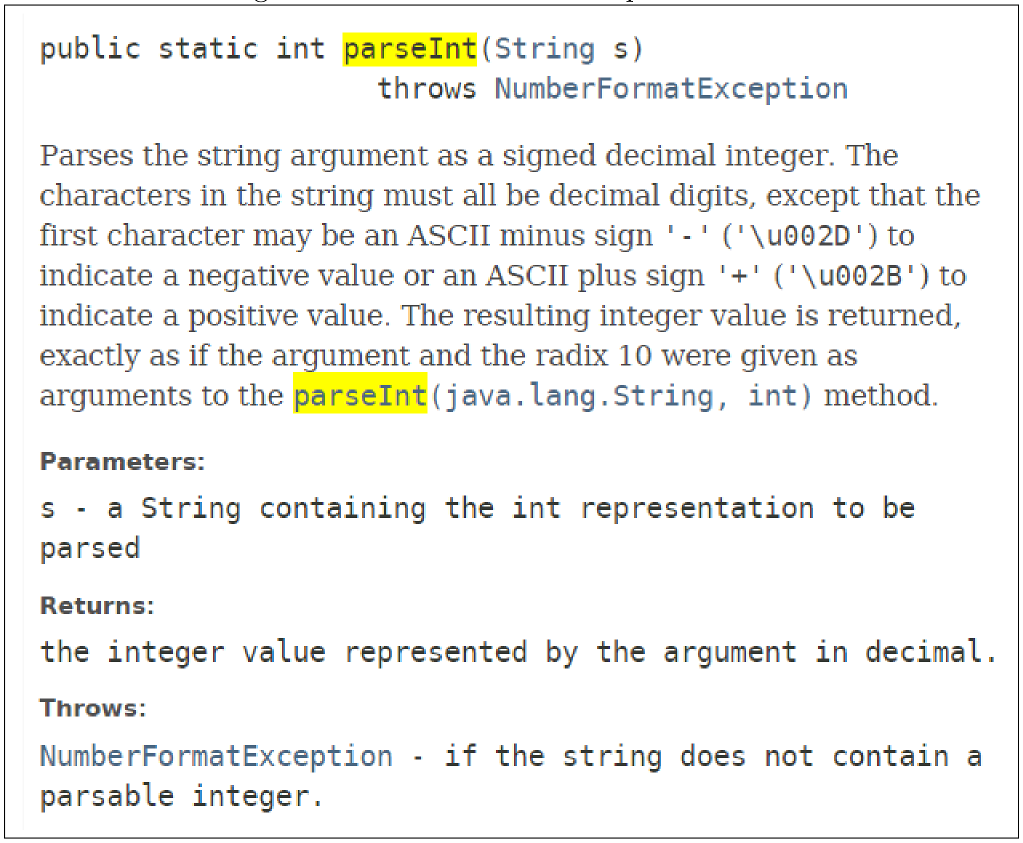
1. Consider the program `HandleFileNotFound` in Listing 8.1. The `main` method gets a file name from the user and attempts to create a reference to it in line 11, then displays a message indicating success or not, and then terminates. Write a program that keeps prompting the user for a file name if the file cannot be found. When a file reference can be made (line 11) successfully the program terminates.

2. Consider the program in Listing 8.6 below. This program computes the sum of the integers entered by the user. When this program runs the user signals end-of-input with ctrl-z on Windows (ctrl-d on unix/Mac), which causes the `Scanner` method `hasNext` to return `false`. A user can easily mistype an integer (say, pressing a non-numeric key) and the program would fail abruptly at line 10 with a `NumberFormatException`. This exception is documented for `parseInt` on the page for class `Integer` which appears below in Figure 8.7 - you can find this by using the BlueJ Help menu, selecting Java Class Libraries, and opening the page for `Integer`. Modify the program to handle such an error: inform the user that the input was invalid and prompt the user to re-enter the number.

Listing 8.6: Sum integers until end of input.

```
1  import java.util.Scanner;
2  public class Q2Source
3  {
4      public static void main(String[] args){
5          int total = 0;
6          Scanner kb = new Scanner(System.in);
7          System.out.println("enter a number: ");
8          while (kb.hasNext()){
9              String number = kb.next();
10             int n = Integer.parseInt(number);
11             total += n;
12             System.out.println("enter a number:
                   ");
13         }
14         System.out.println("total of numbers is
                 "+total);
15     }
16 }
```

Figure 8.7: Documentation for parseInt.



```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign `'-'` (`'\u002D'`) to indicate a negative value or an ASCII plus sign `'+'` (`'\u002B'`) to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the `parseInt(java.lang.String, int)` method.

Parameters:

`s - a String containing the int representation to be parsed`

Returns:

`the integer value represented by the argument in decimal.`

Throws:

`NumberFormatException - if the string does not contain a parsable integer.`

3. In Listing 8.4 the `Employee` class appears. Note there is no setter method for social insurance number. Modify `Employee` to include the `setSin` method. However, this method must throw the `InvalidSINException` if `setSin` is called with an invalid social insurance number.

4. In Listing 7.6 in *Java with BlueJ Part I* the `Student` class is defined. The no-arg constructor sets the gender field to '?'; and the other constructor sets the gender to whatever value has been passed in. Develop an `InvalidGenderException` exception that is to be thrown if there is an attempt to instantiate a student with a gender that is not one of '?', 'm', 'f'. Code a main method to verify the exception is thrown.

5. Consider the program in Listing 7.3. This program reads the file `myData.xml` and displays the integers in the file. On the documentation page for `XMLDecoder` you will find that exceptions can arise when this program runs:

   - `FileNotFoundException`: when the `XMLDecoder` object is created.
   - `ArrayIndexOutOfBoundsException`: when the input stream has no more objects. For this exception you need the import:
     `java.lang.ArrayIndexOutOfBoundsException`

   Modify this program in two ways:

   (a) it should terminate with a message ”file not found” if a FileNotFoundException occurs.

   (b) assume you do not know how many integers are in the file - it should read as many integers as the file holds.

Chapter 9

Recursion

Recursion as an approach that applies to methods and to data structures:

- A recursive method is a method that, directly or indirectly, calls itself.
- A recursive data structure is a structuring of classes so an instance can have a reference to another instance of the same class.

9.1 Recursive methods

We design methods to solve problems. A recursive method is a method that breaks a problem into smaller problems of the same problem type that can be combined to form a solution. The decomposition into smaller problems must be constructed is such a way as to terminate in a finite number of steps, and to end as a basic form that has a known solution.

Example 1, Factorials

Let us consider a classic example, the computation of n! Consider the definition:

```
0! = 1
n! = n * (n-1)!  for n>0
```

These two expressions are exactly what we need to code a recursive method:

- The first expression is called a base case, one where the value is known: `0!=1`.
- The second expression defines the problem, calculation of `n!`, in terms of a smaller problem, the calculation of `(n-1)!`.

The method to calculate n! is straightforward to express in Java - see Listing 9.1:

Listing 9.1: factorial method.

```java
public class Factorial {
    public int factorial (int n){
        if (n==0)
            return 1;// base case is solved
        return n*factorial(n-1);// smaller problem
    }
}
```

When the method `factorial` above is called there are two outcomes. Either the method returns 1, or the method calls itself to calculate (n-1)!.

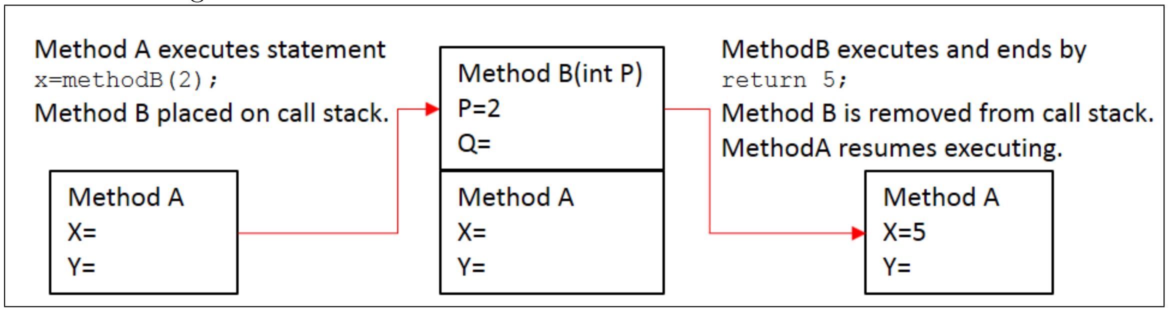
The Call Stack

To control the execution of methods the JVM uses a memory structure called the Call Stack. When method A calls method B the JVM

- ensures the field values of method A are in A's area of the stack,
- notes the location in method A to which control will return,
- allocates a new area on top of what has been used so far for the newly called method B.

When method B finishes and returns control to method A the space method B has occupied on the stack is released and control resumes in method A from the point where A called B including, if B is value-returning, the value method B returns. Consider the scenario for methods A and B in Figure 9.1

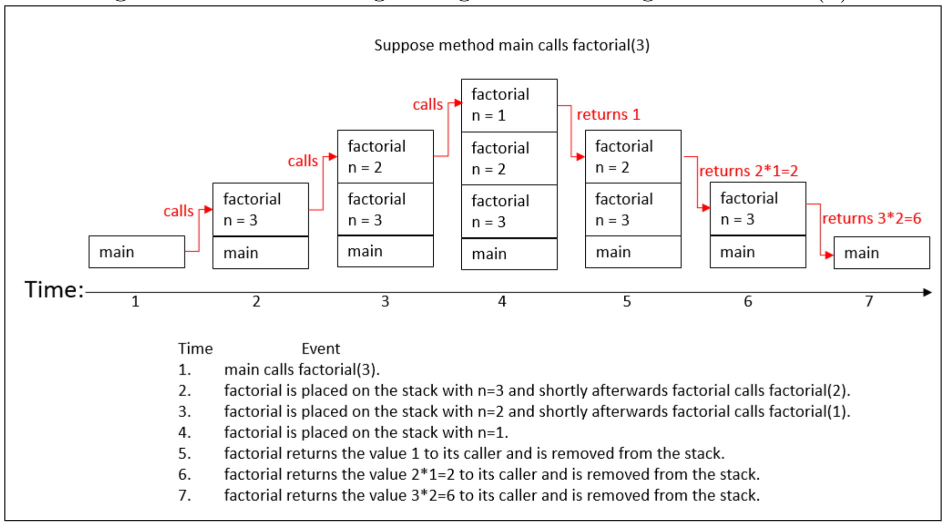
Figure 9.1: Call Stack: method A calls method B.


Method A executes statement
x=methodB(2);
Method B placed on call stack.
Method A
X=
Y=
Method B(int P)
P=2
Q=
Method A
X=
Y=
MethodB executes and ends by
return 5;
Method B is removed from call stack.
MethodA resumes executing.
Method A
X=5
Y=

Call Stack: factorials

Consider the factorial method shown above. Suppose factorial is called with `n=4`, then we can visualize the call stack growing and shrinking as in Figure 9.2. One of the concerns with recursive calls is the stack space that can be used up. If your program has a bug in it, and never can reach a base case, your program will run out of stack space eventually and be terminated by the JVM.

Figure 9.2: Call stack growing and shrinking for factorial(3).



Time	Event
1.	main calls factorial(3).
2.	factorial is placed on the stack with n=3 and shortly afterwards factorial calls factorial(2).
3.	factorial is placed on the stack with n=2 and shortly afterwards factorial calls factorial(1).
4.	factorial is placed on the stack with n=1.
5.	factorial returns the value 1 to its caller and is removed from the stack.
6.	factorial returns the value 2*1=2 to its caller and is removed from the stack.
7.	factorial returns the value 3*2=6 to its caller and is removed from the stack.

Example 2, Fibonacci Numbers

The Fibonacci sequence can be defined as:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

This is the infinite sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ... where successive terms are just the sum of the previous 2 terms. The definition gives us two base cases where recursion will terminate. Consider Listing 9.2 and these points:

- Line 4: The `if` tests for the first base case and simply returns 0.
- Line 5: The `if` tests for the second base case and simply returns 1.
- Line 6: This statement call `fibonacci` recursively two times, once with `n-1` and then with `n-2`.

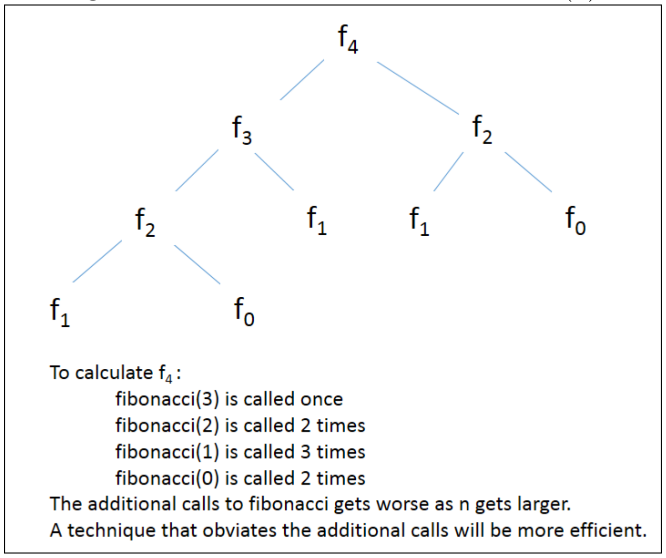
Listing 9.2: Fibonacci method.

```
1 public class Fibonacci
2 {
3     public int fibonacci (int n){
4         if (n==0) return 0;     // first base case
5         if (n==1) return 1;     // second base case
6         return fibonacci(n-1) + fibonacci(n-2); //
             two simpler problems
7     }
8 }
```

The definition of the Fibonacci sequence is simple enough and easy to implement but it does not lead an efficient computation. Consider Figure 9.3 that shows the recursion involved for f_4:

The amount of time required to calculate f_n will vary according to the specifications of a system you are using. However, run the program a few times, say for f_{10}, f_{20}, f_{30}, f_{40}, ... and see if you notice a degradation in performance.

Figure 9.3: Recursive calls for fibonacci(4).



To calculate f_4 :

    fibonacci(3) is called once
    fibonacci(2) is called 2 times
    fibonacci(1) is called 3 times
    fibonacci(0) is called 2 times

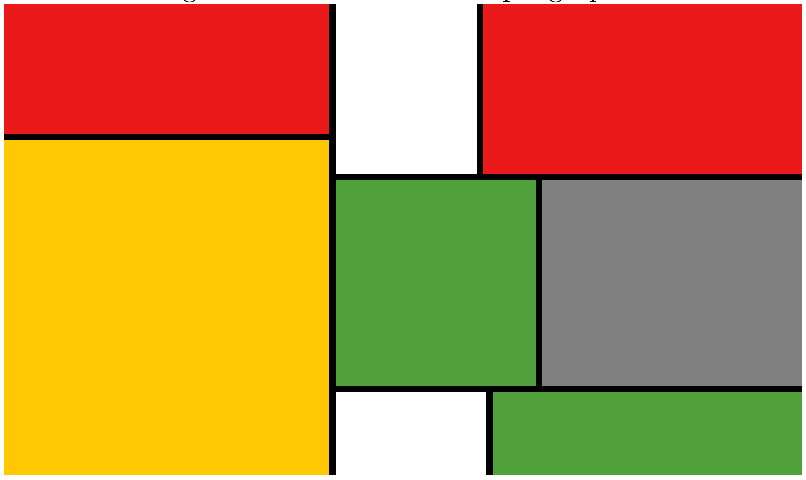
The additional calls to fibonacci gets worse as n gets larger.
A technique that obviates the additional calls will be more efficient.

Example 3, Drawing Mondrian-esque graphics

The following image was produced by a recursive program that continuously and randomly subdivides a canvas vertically or horizontally. Piet Mondrian was an artist (1872 - 1944) whose works are easily discovered using internet browser searches. Some of the artworks he produced have inspired a number of programmers to attempt similar works.

The example in this section uses an approach where the canvas (a Java CanvasPane) is subdivided (vertically or horizontally) in a continuous manner. Each subdivision creates a new rectangle on the canvas - with a colour chosen by the program.

Figure 9.4: A Mondrian-esque graphic.



Our interest here is to illustrate a recursive program, but one that is not just an *interesting* mathematical definition. Listings 9.3 (`Mondrianesque`) and 9.4 (`MRectangle`) are shown later. We refer you to the comments in `Mondrianesque.java` for the details of its specific design choices. To run the example you must also download and compile in the same project, Canvas.java, which is the central class of the BlueJ Shapes example.

`Mondrianesque` starts with a canvas (aka rectangle) of random colour. Then, method `subdivide` is called with this first rectangle.

`subdivide` : input - one rectangle, output - two rectangles.
This method replaces a rectangle R with two rectangles R1 and R2. Together R1+R2 are equal to R.

1. Choose D - the axis for subdivision (vertical or horizontal). R will be subdivided either vertically or horizontally.
2. Choose a point along D for subdivision - this determines the lengths and widths of R1 and R2.
3. Choose colours for R1 and R2.
4. Create a black border to separate R1 and R2.

Listing 9.3: `Mondrianesque` contains the recursive method `subdivide`.

```
import java.util.Random;
/**
 * The main method creates an intial rectangle,
 * and then calls subdivide to have it recursively
 * subdivided.
 */
public class Mondrianesque
{
    // a maximum depth of 4 recursive calls
    private static final int MAX_DEPTH = 4;
    private static int depth = 1;
    private static Random r = new Random();
    public static void main(String [] args){
        // the initial canvas (rectangle)
        MRectangle m = new MRectangle(0,0,500,300,chooseColor());
        m.makeVisible();
        subdivide (m);
    }

    /**
     * subdivide: replaces one rectangle by two rectangles,
     * and also a black border line to separate them.
     * The black border is a narrow black rectangle.
     *
     * @param  r2  a rectangle
     */
    public static void subdivide(MRectangle r2){
        // The original rectangle R is replaced by two rectangles
        // R1 and R2.
        // r2 is the incoming rectangle that will be resized
        // r1 is generated below.
        //
        // choose the x or y axis for subdivision
```

```java
        Axes axis = Axes.xAxis;
        boolean toSubdivide = (r2.xlen>250 || r2.ylen>150);
        MRectangle r1 = null;
        MRectangle border = null;
        if (toSubdivide){
            // if both sides of the rectangle are reasonably long, choose one randomly
            if (r2.xlen>250 && r2.ylen>150)
               if (0 == r.nextInt(2)) axis = Axes.xAxis;
               else axis = Axes.yAxis;
            // if only the x side is long enough choose X axis
            else if (r2.xlen>250) axis = Axes.xAxis;
            // if only the y side is long enough choose Y axis
            else axis = Axes.yAxis;

            // handle split on the X axis
            if (axis == Axes.xAxis){
                // choose where to split along the X axis
                int deltaX = r.nextInt(250);
                // choose a colour for this rectangle ... the other rectangle will have the original colour
                String color = chooseColor();
                while (color.equals(r2.color)) color = chooseColor();
                // R1 is a new rectangle
                r1= new MRectangle(r2.xPosition+deltaX+4,r2.yPosition,r2
                r1.makeVisible();
                // R2 obtained by just changing size of original rectangle
                r2.changeX(deltaX);
                // create the black border along the axis
                border=new MRectangle(r2.xPosition+deltaX,r2.yPosition,4,r2.ylen,"
                border.makeVisible();
            }
            // handle the subdivision along the Y axis
            else {
                // choose point along Y axis for splitting
                int deltaY = r.nextInt(150);
                // pick a colour
                String color = chooseColor();
                while (color.equals(r2.color)) color = chooseColor();
                // R1 is the new rectangle
                r1= new MRectangle(r2.xPosition,r2.yPosition+deltaY+4,r2.xlen,r
                r1.makeVisible();
                // R2 obtained by just changing size of original rectangle
                r2.changeY(deltaY);
                // create the black border along the axis
                border = new MRectangle(r2.xPosition,r2.yPosition+deltaY,r2.xlen,4,"
                border.makeVisible();
            }
            depth++;
            // do not recurse any more than maximum set
            if (depth<MAX_DEPTH){
                subdivide(r1);
                subdivide(r2);
            }
        }
        //          c
    }

    public static String chooseColor(){
        String choice = "white";
        switch (r.nextInt(13)) {
            case 0: choice = "red";
            break;
            case 1: choice = "green";
            break;
            case 2: choice = "blue";
            break;
            case 3: choice = "black";
            break;
            case 4: choice = "yellow";
            break;
            case 5: choice = "magenta";
            break;
            case 6: choice = "white";
            break;
            case 7: choice = "red";
            break;
            case 8: choice = "orange";
            break;
            case 9: choice = "pink";
            break;
            case 12: choice = "gray";
            break;
        }
        return choice;
    }

public enum Axes {xAxis, yAxis};
}
```

Listing 9.4: The basic rectangle.

```
import java.awt.*;
/**
 * This class is a simple rectangle that is
     designed to work with
 * the Shapes example from BlueJ - in particular
     the Canvas class.
 */
public class MRectangle
{
    public int xlen;
    public int ylen;
    public int xPosition;
    public int yPosition;
    public String color;
    private boolean isVisible;

    /**
     * Create a new rectangle.
     */
    public MRectangle(int x, int y, int xlen, int
       ylen, String color)
    {
        this.xlen = xlen;
        this.ylen = ylen;
        xPosition = x;
        yPosition = y;
        this.color = color;
        isVisible = true;
    }

    /**
     * Make this square visible. If it was already
         visible, do nothing.
     */
    public void makeVisible()
    {
        isVisible = true;
        draw();
    }

    /**
     * Change the size to the new size (in pixels).
         Size must be >= 0.
     */
    public void changeX(int newXlen)
    {
        erase();
        xlen = newXlen;
        draw();
    }
    public void changeY(int newYlen)
    {
        erase();
        ylen = newYlen;
        draw();
    }

    /**
     * Draw the square with current specifications
         on screen.
     */
    private void draw()
    {
        if(isVisible) {
            Canvas canvas = Canvas.getCanvas();
            canvas.draw(this, color,
                        new Rectangle(xPosition,
                           yPosition, xlen, ylen));
            canvas.wait(10);
        }
    }

    /**
     * Erase the square on screen.
     */
    private void erase()
    {
        if(isVisible) {
                Canvas canvas = Canvas.getCanvas();
                canvas.erase(this);
            }
        }
}
```

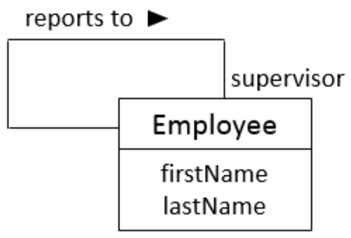
9.2 Recursive data structures

In Part 1 Chapter 7, Designing Java Classes, we learned how to implement associations between classes. If we have an association that involves the same class at both ends of the association then we say the association is recursive, or reflexive. Recursive associations arise in many situations, for example:

- a team plays against another team
- a person is a parent of another person
- an employee reports to another employee

Consider the class diagram in Figure 9.5 for the case of an employee reporting to another employee. A role name is shown to indicate that one employee is known as the supervisor.

Figure 9.5: An employee reports to another employee.



A binary association such as this can be viewed in two ways:

1. an employee *reports to* another employee, and
2. an employee *may supervise many* other employees.

We are ignoring the second one above for simplicity reasons. To implement the first view above we just need a field in Employee such as:

```
    Employee supervisor;
```

Consider the code in Listing 9.5. The class is fairly simple: there are 3 fields and so 3 getters and 3 setters. The interesting aspect of the class is that the `supervisor` field can have a value that references another `Employee`.

Listing 9.5: An Employee class that references an employee.

```java
/**
 * The Employee class
 * that implements the recursive association
 * "an employee reports to another employee"
 */
public class Employee
{
    private Employee supervisor;
    private String firstName;
    private String lastName;
    public Employee()
    {
        firstName="unknown";
        lastName="unknown";
        supervisor = null;
    }
    // getters
    public String getFirstName(){
        return firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public Employee getSupervisor(){
        return supervisor;
    }
    // setters
    public void setFirstName(String first){
        firstName = first;
    }
    public void setLastName(String last){
        lastName = last;
    }
    public void setSupervisor(Employee s){
        supervisor = s;
        }
}
```

In a large company there are many employees and a reporting structure could be very long. For example, suppose we have:

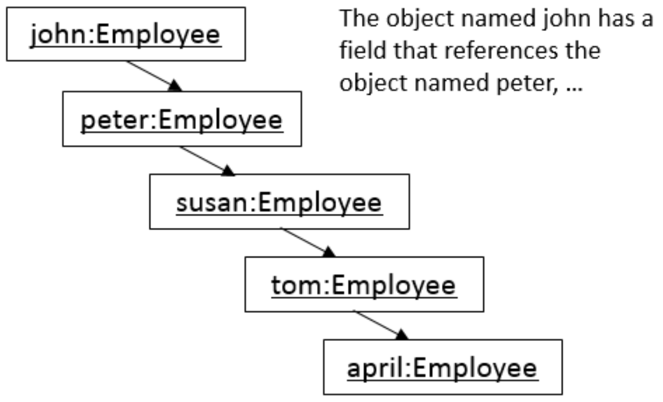
1. John Smith reports to Peter Jones
2. Peter Jones reports to Susan Darwin
3. Susan Darwin reports to Tom Evans
4. Tom Evans reports to April Barnes

The above lists 5 employees where one might say John Smith is at the bottom of the reporting structure and April Barnes is at the top (she does not have a supervisor). We can create this reporting structure quite easily. Assuming that the five objects exists (say, john, peter, susan, tom, april) we just need to execute the following four statements to set their supervisor fields:

```
    john.setSupervisor(peter);
    peter.setSupervisor(susan);
    susan.setSupervisor(tom);
    tom.setSupervisor(april);
```

Figure 9.6 illustrates this chain of objects using a UML object diagram.

Figure 9.6: A chain of objects where each references the next.


john:Employee
The object named john has a field that references the object named peter, ...
peter:Employee
susan:Employee
tom:Employee
april:Employee

Now, if we need to display the name of someone's supervisor, say the supervisor for employee `john`, we just need to code:

```
Employee aSuper = peter.getSupervisor();
System.out.println(aSuper.getFirstName()
                  +" "+aSuper.getLastName());
```

A program that creates these five employees, sets up their reporting structure, and displays someone's supervisor is shown in Listing 9.6.

Recursive data structures are a topic in advanced programming courses where structures such as *linked lists* and *trees* are discussed. The structure we have implemented here is a linked list. This section is just an introduction.

Listing 9.6: Creating a reporting structure.

```

/**
 *
 * Create 5 employees and set up a
 * reporting structure
 *
 */
public class Create5Employees
{
    public static void main(String[] args){
        //
        // create 5 employees
        Employee john = new Employee();
        john.setFirstName("John");
        john.setLastName("Smith");
        Employee peter = new Employee();
        peter.setFirstName("Peter");
        peter.setLastName("Jones");
        Employee susan = new Employee();
        susan.setFirstName("Susan");
        susan.setLastName("Darwin");
        Employee tom = new Employee();
        tom.setFirstName("Tom");
        tom.setLastName("Evans");
        Employee april = new Employee();
        april.setFirstName("April");
        april.setLastName("Barnes");
        //
        // set the supervisors for
        // john, peter, susan, and tom
        john.setSupervisor(peter);
        peter.setSupervisor(susan);
        susan.setSupervisor(tom);
        tom.setSupervisor(april);
        //
        // display Peter's supervisor
        Employee aSuper = peter.getSupervisor();
        System.out.println(aSuper.getFirstName()+"
           "+aSuper.getLastName());
    }
}
```

9.3 Exercises

1. The tribonacci numbers are like the Fibonacci numbers, but instead of starting with two predetermined terms, the sequence starts with three predetermined terms and each term afterwards is the sum of the preceding three terms. The first few tribonacci numbers are:
   ```
   0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81
   ```
   Write a recursive method that calculates the n^{th} tribonacci number.

2. Consider Listing 9.2. What is the sequence of calls to `fibonacci` for `fibonacci(4)`? That is, what is the order in which Fibonacci numbers are calculated?

3. A drawback to the recursive fibonacci method in Listing 9.2 is that it recalculates so many Fibonacci numbers, and with that comes the overhead of recursive calls. Figure 9.3 shows in the calculation of f_4, f_2 was calculated twice and f_1 was calculated 3 times. In situations like this there is a technique that can be used called *memoization*. Memoization is an optimization approach where values are stored so they may be used again (simply retrieved and not recalculated). To apply this to the Fibonacci numbers we can use an array of `int`s to store Fibonacci numbers. We represent this new approach in pseudocode:

   ```
   Step 1. Initialize an array of Fibonacci numbers to zeros. The array
           is of some fixed, but large enough size.
   Step 2. fibonacci(n):
       2.1 if n is 0 then return 0
       2.2 if n is 1 then return 1
       2.3 if fib[n] is not 0 then return fib[n]
       2.4 fib[n] = fibonacci(n-1) + fibonacci(n-2)
       2.5 return fib[n]
   ```

   The method outlined above calculates and saves a Fibonacci number the first time that number is required (step 2.4), but after that it will simply return the value that was pre-calculated (step 2.3). Develop this approach in a Java program.

4. The ratio of successive Fibonacci numbers is the golden ratio which is 1.6180339... Write a non-recursive program that displays this ratio for the first 20 Fibonacci numbers.

5. Write a recursive method `reverse` that reverses a string. If we execute

   ```
   System.out.println(reverse("Fibonacci"));
   ```

   the output will be:

   ```
   iccanobiF
   ```

6. Write a recursive method `count` that counts the number of `'x'`s in a string. If we execute

   ```
   System.out.println(count("xerox"));
   ```

   the output will be:

   ```
   2
   ```

7. Modify the `subdivide` method of `Mondrianesque` to use some different rule for stopping the recursive subdivision process. For instance consider not subdividing if the current rectangle has an area that is less than some fixed value.

8. Suppose we have a class `Person`; objects of `Person` represent people. Suppose we must implement the association `Person` *married to* `Person`. Develop a class Person with attributes first name, last name, and gender. Also, include an attribute:

   ```
   Person spouse;
   ```

   which implements the *married to* association. Implement the following people and relationships:

   - Peter Smith is married to Susan Darwin
   - Tom Lee is married to Barbara Eck
   - Linda Lee is married to Sammy Barnes
   - David Chan, but he is not married.

   Test your implementation by displaying each person and their spouse (or dashes if they have no spouse). The output for the sample data would be:

Person	Spouse
Peter Smith	Susan Darwin
Susan Darwin	Peter Smith
Tom Lee	Barbara Eck
Barbara Eck	Tom Lee
Linda Lee	Sammy Barnes
Sammy Barnes	Linda Lee
David Chan	————

Chapter 10

Sorting and Searching

For lots of reasons lists need to be placed into order. Email is listed by date, phone contacts listed alphabetically, class rosters sequenced by student number or by last name, etc.

It may not be important how we go about organizing a small list, but when the lists are lengthy the approach (i.e. algorithm) can be critical. In this chapter we consider a number of sorting algorithms: selection, insertion, bubble, and quicksort. To simplify we consider sorting lists of integers into ascending order.

We also consider algorithms for searching a list. If we know a list is sequenced we can use that knowledge when we want to find a specific element.

Next we discuss each sorting algorithm from a conceptual perspective. Then we present implementations of each technique. Next we show how to implement sorting for objects. Lastly, we consider algorithms for searching a list to locate a specific entry or to determine that the entry is not present.

The implementations we give use arrays. A convenience method we have used is `toString` of `Arrays` that displays all elements of the array. However, if you are working with `ArrayLists` you can use methods in the `Collections`, `Arrays`, and `ArrayList` classes to convert from one type to the other. For instance, to add all elements of an array to a list you can use

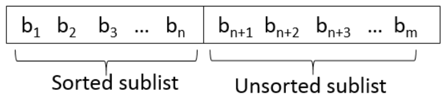
```
    Collections.addAll(toList, fromArray);
```

Also, there are methods `asList` in the `Arrays` class and a method `toArray` in the `ArrayList` class.

10.1 Sorting Algorithms

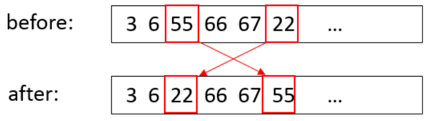
For many algorithms a list is conceptually partitioned into a sorted sublist and an unsorted sublist. At an intermediate stage a list is partially sorted: for example, a sorted sublist on the left and an unsorted sublist on the right. The nature of the algorithm eventually reduces the unsorted sublist to nothing and the sorted sublist comprises all elements. For simplicity and consistency in our presentation we consider sorting lists of integers in ascending sequence.

Figure 10.1: A list comprising a sorted sublist and an unsorted sublist.



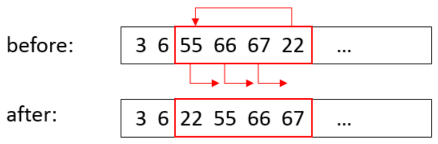
These algorithms use some basic operations such as swapping two elements and shifting of elements. Consider the diagram below where swapping (interchanging) two elements is illustrated. In this case the 3rd and the 6th elements (22 and 55) are swapped.

Figure 10.2: 3^{rd} and 6^{th} elements swap positions.



In some algorithms elements are shifted to the left or right to make room for an element. Consider the diagram below where the 6^{th} element is moved to the 3^{rd} position and so the 3^{rd} through 5^{th} elements are shifted right.

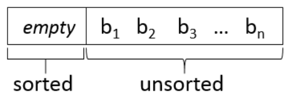
Figure 10.3: 6^{th} element moves to 3^{rd} positions; others shift right.



10.1.1 Selection Sort

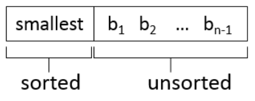
This algorithm is very straightforward. The sorted sublist begins as the empty list, and the unsorted sublist comprises all n elements.

Figure 10.4: Sorted sublist is empty; Unsorted sublist has all elements.



The algorithm begins by finding the smallest entry in the unsorted list and then swaps that entry with the first entry of the unsorted list. At this point we say the sorted sublist comprises that first element and the unsorted sublist consists of the remaining $n - 1$ elements.

Figure 10.5: Sorted sublist has smallest; Unsorted sublist of $n-1$ elements.



As a next step, the algorithm finds the smallest entry of the unsorted sublist and then swaps that entry with the first element of the unsorted sublist. At this point a list of n elements has a sorted sublist of 2 elements and an unsorted sublist of $n - 2$ elements.

This process continues until the unsorted sublist has one remaining element. That one entry is the largest, and so the whole list is now sorted. If there are n elements in the list, then finding the smallest and swapping is done $n - 1$ times. We outline the basic algorithm in pseudocode below.

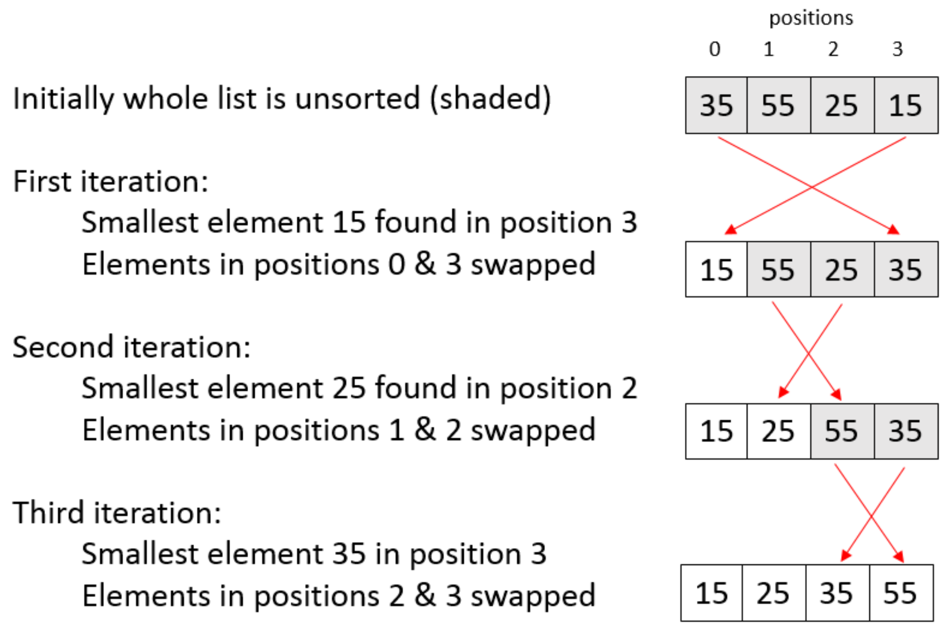
Selection Sort (list):

```
1. Sorted sublist ← empty.
2. Unsorted sublist ← list to be sorted.
3. Iteration: Repeat until the unsorted sublist comprises a single element:
    a) find the smallest entry in the unsorted sublist, call this $X$.
    b) swap the first entry of the unsorted sublist with $X$.
    c) $X$ is now part of the sorted sublist and the unsorted sublist has 1
       fewer elements.
```

Example

Consider the list below as it slowly changes from unsorted to sorted through the application of the Selection sort algorithm. The list is 4 elements long and 3 iterations of Step 3 are required to complete the sort. The unsorted elements appear shaded, and the sorted elements are not shaded.

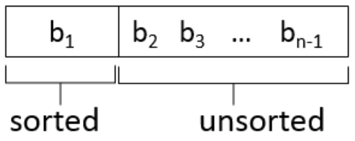
Figure 10.6: Finding smallest and swapping with first element of unsorted sublist.


positions
0 1 2 3
Initially whole list is unsorted (shaded)
35 55 25 15
First iteration:
Smallest element 15 found in position 3
Elements in positions 0 & 3 swapped
15 55 25 35
Second iteration:
Smallest element 25 found in position 2
Elements in positions 1 & 2 swapped
15 25 55 35
Third iteration:
Smallest element 35 in position 3
Elements in positions 2 & 3 swapped
15 25 35 55

10.1.2 Insertion Sort

This algorithm partitions a list into a sorted sublist followed by the unsorted sublist. Initially the sorted sublist consists of the first element (a list of one element is a sorted sublist). The unsorted sublist is the following $n-1$ elements.

Figure 10.7: Sorted sublist is the first element; Unsorted sublist is the next $n-1$ elements.



The Insertion Sort involves iterations where, in each iteration, the first unsorted element is moved into its proper position amongst the sorted sublist and the unsorted sublist shrinks by 1 element. After $n-1$ iterations the list is totally sorted. We outline the basic algorithm in pseudocode below.

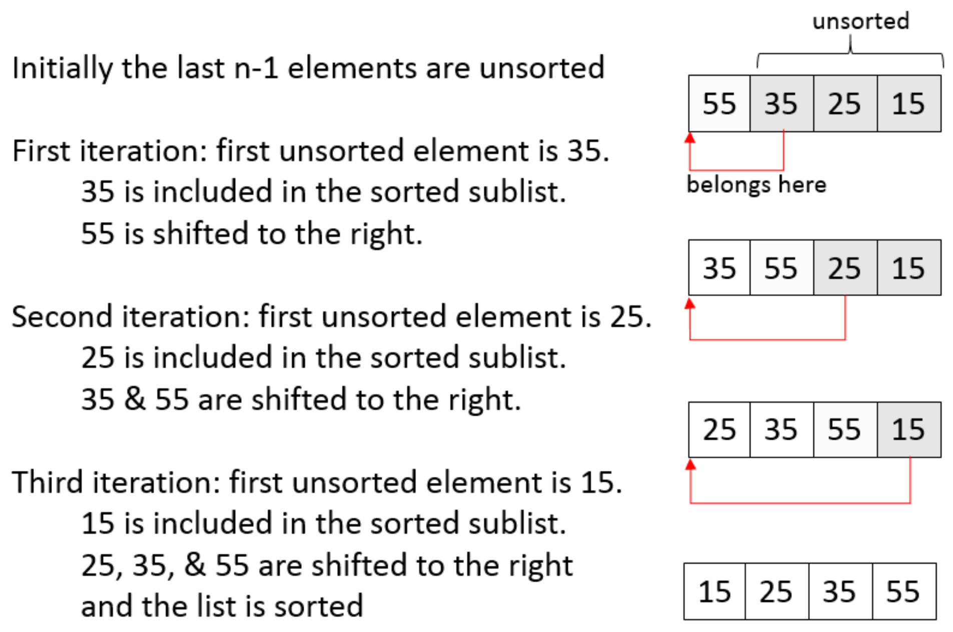
Insertion Sort (list):

1. Sorted sublist $\leftarrow$ first element of list.
2. Unsorted sublist $\leftarrow$ remaining $n-1$ elements of list.
3. Iteration: Repeat until the unsorted sublist is empty.
   (a) Let X be the first element in the unsorted list
   (b) Shift entries that are greater than X to the right making room for X in its proper position in the sorted sublist.
   (c) X is now part of the sorted sublist and the unsorted sublist is 1 element shorter

Example

Consider the list below as it slowly changes from unsorted to sorted through the application of the Insertion Sort algorithm. The list is 4 elements long and 3 iterations of Step 3 to complete the sorting. The unsorted elements appear shaded, and the unsorted elements are not shaded.

Figure 10.8: Inserting the first unsorted element into the sorted sublist involves shifting already sorted elements.



Initially the last n-1 elements are unsorted

First iteration: first unsorted element is 35.
35 is included in the sorted sublist.
55 is shifted to the right.

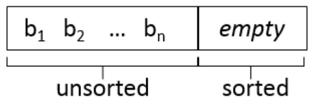
Second iteration: first unsorted element is 25.
25 is included in the sorted sublist.
35 & 55 are shifted to the right.

Third iteration: first unsorted element is 15.
15 is included in the sorted sublist.
25, 35, & 55 are shifted to the right
and the list is sorted

10.1.3 Bubble Sort

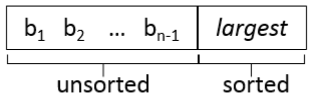
This algorithm partitions a list into an unsorted sublist followed by a sorted sublist. Initially the unsorted sublist is the whole list, and the sorted sublist is empty.

Figure 10.9: Unsorted sublist has all n elements; Unsorted sublist is empty.



The Bubble Sort involves iterations where the entire unsorted sublist is scanned pairwise, pair-by-pair: each pair of elements are compared, and if the pair is out of order they are swapped. This has the effect of moving larger elements towards the end of the unsorted sublist, and it moves the largest element to the end of the unsorted sublist.

Figure 10.10: Larger elements move right; largest is included in the sorted sublist.



The notion that larger elements move via swapping of two adjacent elements gives rise to *bubble* in the name of this algorithm. After a pass of the unsorted sublist, the unsorted sublist shrinks by 1 element and the sorted sublist grows by 1 element. After $n - 1$ iterations the list is totally sorted. We outline the basic algorithm in pseudocode below.

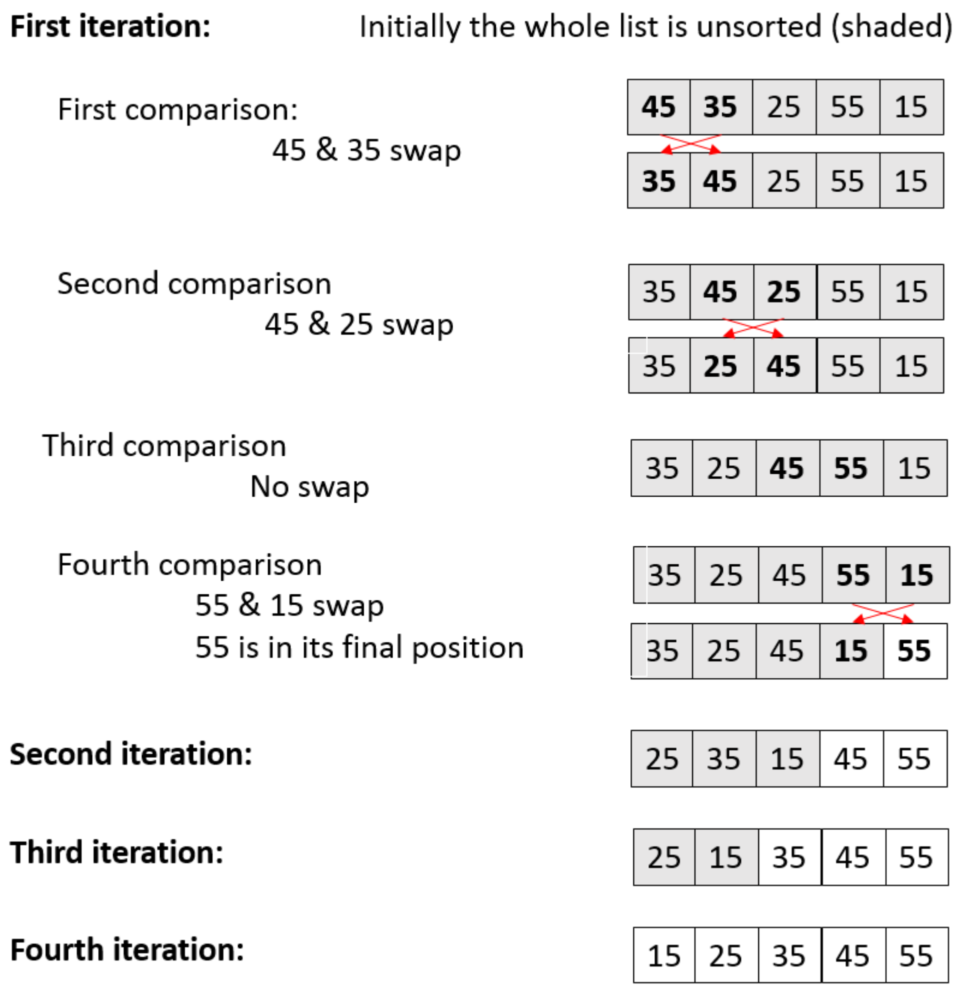
Bubble Sort (list):

```
1. Unsorted sublist ← the original list.
2. Sorted sublist ← an empty list.
3. Iteration: Repeat until the unsorted sublist is empty.
    (a) Scanning forward from the beginning of the unsorted list, consider
        the elements pairwise ($a_1$ and $a_2$, then $a_2$ and $a_3$, etc.). For each
        pair where $a_i > a_{i+1}$, swap $a_i$ and $a_{i+1}$. After a scan the largest
        element has moved to the end of the unsorted list.
    (b) The unsorted sublist shrinks by 1 element; sorted sublist increases
        by 1 element.
```

Example

Consider the list below as it slowly changes from unsorted to sorted through the application of the Bubble sort algorithm. The unsorted elements appear shaded, and the unsorted elements are not shaded.

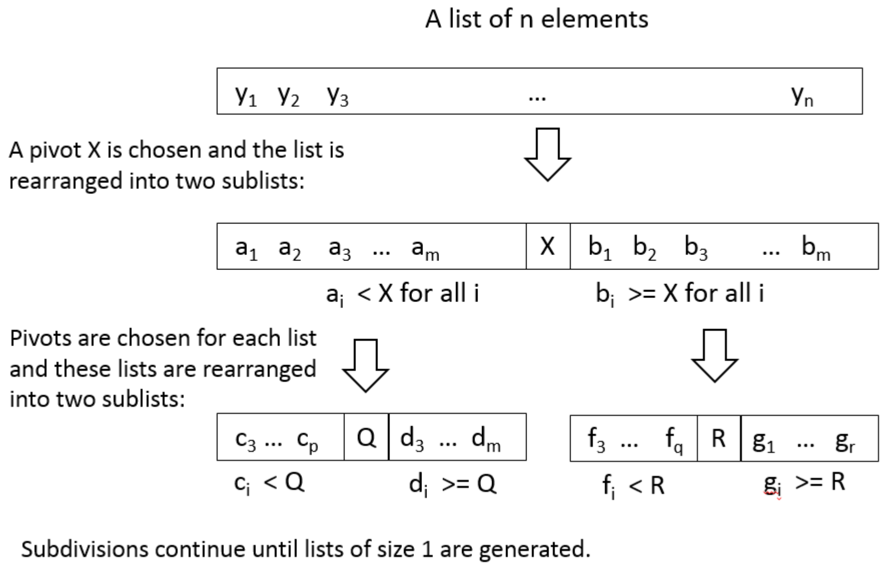
Figure 10.11: With each pass of the unsorted sublist larger elements move towards the end; the largest moves to the end.



10.1.4 Quicksort

The Quicksort algorithm is different from the previous algorithms in that it continuously subdivides a list into 2 sublists separated by a pivot value. The essential idea is that a pivot value, say X, is chosen from the current list and then the remaining elements are rearranged into lists: one where all elements are $< X$, and another where all elements are $>= X$. This process continues recursively on each new sublist until all lists consist of one, or zero, elements - at this point the list is sorted. The general process is shown below.

Figure 10.12: A list is subdivided into two sublists based on a pivot. The subdivision recurs until all sublists have $<= 1$ element.


A list of n elements
y1 y2 y3 ... yn
A pivot X is chosen and the list is rearranged into two sublists:
a1 a2 a3 ... am | X | b1 b2 b3 ... bm
ai < X for all i
bi >= X for all i
Pivots are chosen for each list and these lists are rearranged into two sublists:
c3 ... cp | Q | d3 ... dm
ci < Q
di >= Q
f3 ... fq | R | g1 ... gr
fi < R
gi >= R
Subdivisions continue until lists of size 1 are generated.

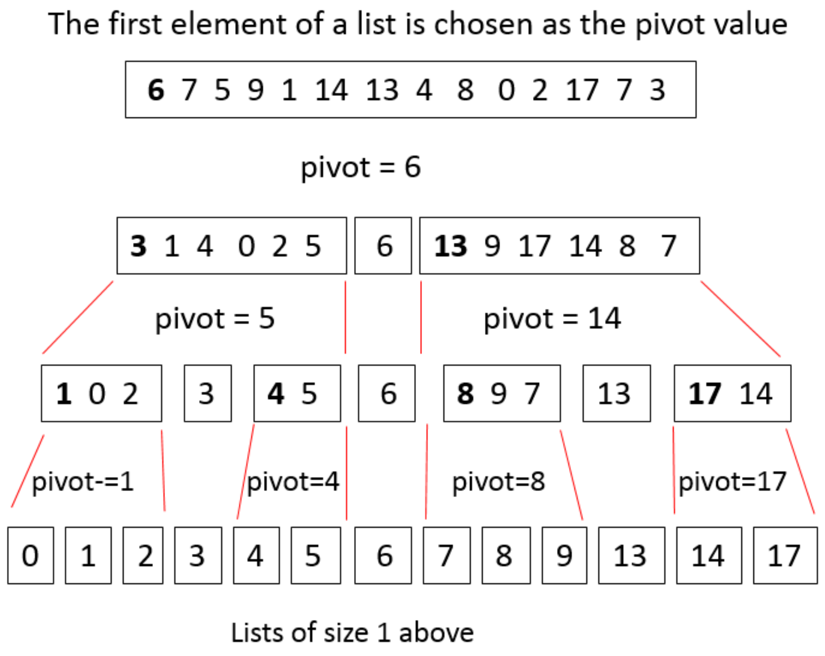
We outline the basic algorithm in pseudocode below.
Quicksort (list):

```
1. If list of length $<= 1$ return.
2. Choose a pivot value $X$.
3. Rearrange the elements of list: left sublist L with elements $< X$; right
   sublist R with elements $>= X$.
4. Quicksort(L)
   Quicksort(R)
```

Example

Below we show a list continuously subdivided producing a sorted list. In this example the first element of a list is always chosen as the pivot value.

Figure 10.13: Subdivisions of lists until the sublists are of size 1 (0 length sublists not shown).



10.1.5 Implementation

In the previous section we introduced each algorithm at a conceptual level. Now we consider an implementation of each. The code is commented to guide your understanding.

In practice and further study you will encounter variations and improvements on all of these algorithms. Sorting has been studied by researchers and practitioners for decades. If you view the implementation code for the Java class libraries you will see the sort procedures for Arrays and Collections use variations of these techniques and other algorithms as well (such as Timsort and Merge Sort).

The analysis of algorithms is an important study in computer science. You may encounter further courses or study that discusses the expected performance of these algorithms in terms of the worst case, average case, and best case scenarios. For instance, how long does it take to sort an already sorted list? At this time in your study it should not be too difficult to set up a simulation study with small and large lists to see how much time it takes to sort lists that are randomly organized, or already sorted.

As is usual for sorting algorithms, we consider the implementation in terms of one-dimensional arrays. The following listings comprise Java code for each algorithm.

10.1.6 Listings

- Selection sort - Listing 10.1
- Insertion sort - Listing 10.2
- Bubble sort - Listing 10.3
- Quicksort - Listing 10.4

Listing 10.1: Selection Sort.

```
import java.util.Arrays;
public class SelectionSort
{
    public static void main(String[] args){
        int [] toSort = {35, 55, 25, 15};
        System.out.println(Arrays.toString(toSort));
        selectionSort(toSort);
        System.out.println(Arrays.toString(toSort));
    }
    public static void selectionSort(int[] a){
        int n = a.length;
        // n-1 passes of the unsorted sublist
        for (int i=0; i<n-1; i++) {
            // find the index of the smallest
               element in the unsorted sublist
            int iSmallest = i;
            // test against successive elements
            for (int j=i+1; j<n; j++) {
                // update the index of the smallest
                if (a[j] < a[iSmallest]) {
                    // found new smallest
                    iSmallest = j;
                }
            }
            // swap the smallest element with the
               start of the unsorted sublist
            if(iSmallest != i) {
                int temp = a[i];
                a[i] = a[iSmallest];
                a[iSmallest] = temp;
            }
        }
    }
}
```

Listing 10.2: Insertion Sort.

```java
import java.util.Arrays;
public class InsertionSort
{
    public static void main(String[] args){
        int [] toSort = {55, 35, 25, 15};
        System.out.println(Arrays.toString(toSort));
        insertionSort(toSort);
        System.out.println(Arrays.toString(toSort));
    }
    public static void insertionSort(int[] a){
        int n = a.length;
        // initially sorted list is a[0]
        // initially unsorted list is a[1], a[2], ...
        // i is index of first element in the unsorted list
        for (int i=1; i<n; i++) {
            // x is the element to be placed in its proper position within the sorted list
            int x = a[i];
            // shift elements to the right to find the position for x
            int j=i-1;
            while (j>=0 && a[j]>x){
                a[j+1] = a[j];
                j--;
            }
            a[++j]=x;
        }
    }
}
```

Listing 10.3: Bubble Sort.

```java
import java.util.Arrays;
public class BubbleSort
{
    public static void main(String[] args){
        int [] toSort = {45, 35, 25, 55, 15};
        System.out.println(Arrays.toString(toSort));
        bubbleSort(toSort);
        System.out.println(Arrays.toString(toSort));
    }
    public static void bubbleSort(int[] a){
        int n = a.length;
        // initially sorted list is empty
        // initially unsorted list is a[0], a[1],
           a[2], ...
        // i is index of first element in the
           unsorted list
        for (int i=0; i<n-1; i++) {
            // scan the unsorted sublist to bubble
               the largest to the end
            for (int j=0; j<n-i-1; j++){
                if (a[j] > a[j+1]) {
                    // pair is out of order so swap
                       them
                    int temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
                }
            }
        }
    }
}
```

Listing 10.4: Quicksort.

```
import java.util.Arrays;
public class Quicksort
{
    public static void main(String[] args){
        int [] toSort = {6, 5, 9, 0, 11, 8, 4, 10,
           2, 1, 12, 7, 3};
        System.out.println("before:
           "+Arrays.toString(toSort));
        quicksort(toSort, 0, toSort.length-1);
        System.out.println("after:
           "+Arrays.toString(toSort));
    }

    public static void quicksort(int[] a, int
       start, int end){
        // check for list of 0 or 1 elements
        if (start >= end) return;
        // the pivot is chosen as the first element
        int pivot = a[start];
        // the list is partitioned into a
        // left sublist and a right sublist.
        //   left  sublist has values <  pivot.
        //   right sublist has values >= pivot.
        int left = start;
        int right = end;
        // move larger elements to a right sublist
        // move smaller elements to a left sublist
        while (left < right){
            // search from the right to find
            //  a value that is less than pivot
            //  and move it to the left.
            while (right > left && a[right]>=pivot){
                right--;
            }
            // if a lesser value is not found
            //  then exit.
            if(left==right) break;
            // move a value smaller than pivot
            // to left of pivot.
            a[left]=a[right];
            left++;
            // search from the left for a value
            //   larger than pivot.
            while (right > left && a[left] <= pivot){
                left++;
            }
            // if a larger value is not found
            //   then exit.
            if(left == right) break;
            // move the value larger than pivot
            //   to the right of the pivot.
            a[right]=a[left];
            right--;
        }
        // the pivot is moved to a position
        //   separating left and right.
        a[right]=pivot;
        // apply quicksort to the sublists.
        quicksort(a, start,   right-1);
        quicksort(a, right+1, end    );
    }
}
```

10.2 Sorting Objects

The examples and implementations above considered sorting integers. It is fairly simple to modify the algorithms for other primitive types or for objects.

Below we present code for a Duck class and a Selection Sort for objects. To sort ducks we only need to ensure the Duck class follows the convention:

- Duck must implement the `Comparable` interface - it must have a `compareTo` method for comparing one duck to another duck.

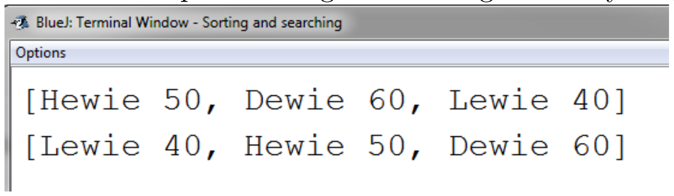
In our example we have used the weight of ducks for comparisons, and so after sorting they appear in order from lightest to heaviest. We have included a `toString` method so we could display the ducks before and after sorting.

The listings that follow include:

- Duck class, Listing 10.5
- Selection Sort adapted for objects, Listing 10.6
  - Note the type of the array passed in is `Comparable`.
  - Note when two ducks are swapped that the variable `temp` is declared as `Comparable`.

The output from running 10.6 is shown below; the first line is before sorting and the second line shows the ducks in sequence by weight.

Figure 10.14: Output showing ducks being sorted by weight.



```
[Hewie 50, Dewie 60, Lewie 40]
[Lewie 40, Hewie 50, Dewie 60]
```

Listing 10.5: Duck.

```java
/**
 * Ducks that can be sorted:
 *      - the class must implement Comparable
 *      - there must be a method compareTo
 *      - for convenience a toString method
 */
public class Duck implements Comparable
{

    private String name;
    private int weight;

    public Duck(String n, int w)
    {
        name = n;
        weight = w;
    }
    // comparing ducks
    public int compareTo(Object d)
    {
        Duck aDuck = (Duck) d;
        return this.weight-aDuck.weight;
    }
    // printing of ducks
    public String toString(){
        return name+" "+weight;
    }
}
```

Listing 10.6: Selection Sort for Objects.

```java
import java.util.Arrays;
public class SelectionSortForObjects
/*
 * Modifying Selection sort for int to a selection sort for objects requires:
 *      - the array to be an array of Comparable
 *      - comparisons to be made using the compareTo method
 *      - swapping requires a temp variable of type Object
 */
{
    public static void main(String[] args){
        Duck[] toSort = {new Duck("Hewie", 50), new Duck("Dewie", 60), new Duck("Lewie", 40)};
        System.out.println(Arrays.toString(toSort));
        selectionSortForObjects(toSort);
        System.out.println(Arrays.toString(toSort));
    }
    public static void selectionSortForObjects(Comparable[] a){
        int n = a.length;
        // n-1 passes of the unsorted sublist
        for (int i=0; i<n-1; i++) {
            // find the index of the smallest element in the unsorted sublist
            int iSmallest = i;
            // test against successive elements
            for (int j=i+1; j<n; j++) {
                // update the index of the smallest
                // for objects
                if (a[j].compareTo(a[iSmallest])<0) {
                    // found new smallest
                    iSmallest = j;
                }
            }
            // swap the smallest element with the start of the unsorted sublist
            if(iSmallest != i) {
                System.out.println("swapping "+a[i]+" "+a[iSmallest]);
                // swapping objects of type Comparable
                Comparable temp = a[i];
                a[i] = a[iSmallest];
                a[iSmallest] = temp;
            }
        }
    }
}
```

10.3 Searching Algorithms

If a phone directory was a list, and we needed to locate a contact then we would search the list for the pertinent element. Suppose we need to locate entry X. If the list was not sorted then we would need to examine elements of the list one-by-one until we find X, or, if we reach the end of the list then we know X was not there. If the list is known to be sorted then we would not need to examine all entries there are other approaches that can be taken discussed later. As with the sections on sorting, we consider ordered lists where the values are integers and in ascending sequence.

10.3.1 Searching an Unordered List

Suppose we need to determine if an element X is in a list. If the list is not sorted then we can sequentially examine elements until we find X or until we have considered all elements of the list. The algorithm to search for the value X in the list is straightforward; in pseudocode:

```
Search(list, $X$):
  1. Let $i$ reference the start of the list
  2. While $i$ <= end of list
     (a) If $list_i$ equals $X$ then return found at position $i$
     (b) Increment $i$ to reference the next element of the list
  3. Return not found
```

The search algorithm above is simple enough that we leave coding as an exercise.

10.3.2 Searching an Ordered List

When a list is sorted in accordance with the criteria of the search two obvious algorithms can be used:

- sequential search
- binary search.

Sequential Search

This approach examines every element of the list, one-by-one, until either the element is found, or the end of the list is reached. In pseudocode we express searching an ordered list for a value X:

Sequential Search (list, X):

```
1. Let i reference the first element of the list.
2. While $i <=$ end of list
    (a) If $list_i > X$ then return *not found*
    (b) If $list_i$ equals $X$ then return *found at position i*
    (c) Increment $i$ to reference the next element of the list
3. Return *not found*
```

This algorithm is very similar to the previous one - searching an unordered list. There is one important point though. Step $2.a$) recognizes a case where the current element is larger than the search value. If this is true then the search value cannot appear in the list since the list is ordered.

Again, we leave the coding of this method as an exercise.

Binary Search

This search algorithm considers the list to be sorted and differs from the Sequential Search above - this algorithm does not examine entries sequentially.

Instead, the binary search partitions the list into a left sublist and a right sublist that are separated by the value at the mid point of the list. If the search value is less than the midpoint value then the binary search is called recursively on the left sublist. If the search value is greater than the midpoint value then the binary search is called recursively on the right sublist. Lastly, if the search value equals the midpoint value the element has been found and the search ends successfully. If the method is called with an empty list then the search ends unsuccessfully - the search value is not in the list.

We express this algorithm in pseudocode, followed its implementation in Java.

BinarySearch (list, X):

```
1. If the list L is empty then return not found
2. Determine the midpoint of the list
3. Determine the value at the midpoint of the list
4. if $X <$ midpoint value call BinarySearch(left sublist, $X$)
5. otherwise if $X >$ midpoint value call BinarySearch(right sublist, $X$))
6. otherwise if $X =$ midpoint value then return found at position midpoint
```

The Java code for Binary Search appears as a method in Listing 10.7. Writing Binary Search as a non-recursive method is left as an exercise.

Listing 10.7: The method `binarySearch` searches for `searchValue` in the array `a`.

```java
import java.util.Arrays;
public class BinarySearch
{
    public static void main(String[] args){
        // the sorted array to be searched
        int[] sorted = {1, 3, 5, 7, 9};
        System.out.println("array to search: "+Arrays.toString(sorted));
        // exhaustive set of search values
        performSearch(sorted,0);
        performSearch(sorted,2);
        performSearch(sorted,4);
        performSearch(sorted,6);
        performSearch(sorted,8);
        performSearch(sorted,10);
        performSearch(sorted,1);
        performSearch(sorted,3);
        performSearch(sorted,5);
        performSearch(sorted,7);
        performSearch(sorted,9);
    }

    /*
     * Utility method to call binary search
     * and report results
     */
    public static void performSearch(int[] a, int searchFor){
        int result = binarySearch(a, 0, a.length-1, searchFor);
        if (result >= 0)
            System.out.println("found "+searchFor+" in position "+result);
        else
            System.out.println(searchFor+" not found");
    }

    /*
     * binary search of a sorted array
     *   - continuously bisect a list
     *     looking for search value
     *     - Return -1 if the list is empty,
     *       ... value not found
     *     - determine midpoint of list
     *     - get value at midpoint
     *     - compare midpoint value & search value
     *       - Return midpoint ...value found
     *       - recursive call with left or
     *         right sublist
     */
    public static int binarySearch ( int[] a, int
       start, int end, int searchValue){
        // when called with a sublist that is
           empty, end is greater than start
        if (start > end) return -1;
        // determine midpoint of sublist
        // and the value at the midpont
        int midPoint = (start + end) / 2;
        int midValue = a[midPoint];
        // check midpoint for value
        if (midValue == searchValue)
            return midPoint;  // found
        else if (midValue < searchValue)
            // search left sublist
            return binarySearch(a, midPoint + 1,
               end, searchValue);
        else // (midValue > searchValue)
            // search right sublist
            return binarySearch(a, start, midPoint
               - 1, searchValue);
    }
}
```

10.4 Exercises

1. Show using a diagram the iteration(s) required to sort the list
   `{ 6, 5, 4, 3, 2, 1, 0 }` using
   (a) Selection sort
   (b) Insertion sort
   (c) Bubble sort
   (d) Quicksort
2. Show using a diagram the iteration(s) required to sort the list
   `{ 4, 5, 6, 3, 2, 1, 0 }` using
   (a) Selection sort
   (b) Insertion sort
   (c) Bubble sort
   (d) Quicksort
3. Develop a program that generates 1,000 integers placing them in an array and then uses Selection Sort to sort the array.
4. Modify the program in the previous question so it counts the number of comparisons and the number of swaps that are performed.
5. Develop a program that generates 1,000 integers placing them in an array and then uses Insertion Sort to sort the array.
6. Modify the program in the previous question so it counts the number of elements that are shifted.
7. Develop a program that generates 1,000 integers placing them in an array and then uses Bubble Sort to sort the array.
8. Modify the program in the previous question so it counts the number of comparisons and the number of swaps that are performed.
9. Develop a program that generates 1,000 integers placing them in an array and then uses Quicksort to sort the array.
10. Using the pseudocode for a guide, develop the search method for an unordered list.

11. Using the pseudocode for a guide, develop the sequential search method for an ordered list.
12. Using the pseudocode and the recursive implementations as guides, develop a non-recursive binary search method for an ordered list.