Java with BlueJ Part 2

Ron McFadyen

January 3, 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:

```
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.

```
1  /**
2   * Display number of days in each month
3   */
4  public class MonthLengths
5  {
6      public static void main(String[] args){
7          int[] daysInMonth =
8              {31,28,31,30,31,30,31,31,30,31,30,31};
9          System.out.println("Days for each of "
10             +daysInMonth.length+" months ");
11         for (int i = 0; i< daysInMonth.length; i++)
12             System.out.print(daysInMonth[i]+"  ");
13     }
14 }
```

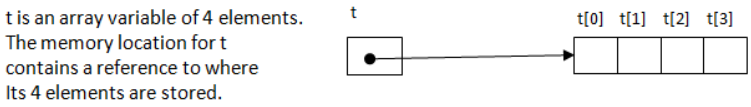
The output:

Options

```
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

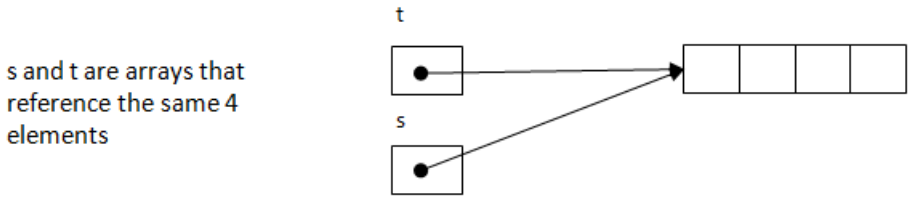
Arrays are objects in Java and so the memory location for the array variable contains a reference to the actual storage locations holding the array's values. For instance the memory allocations for an array can be visualized as:



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
```

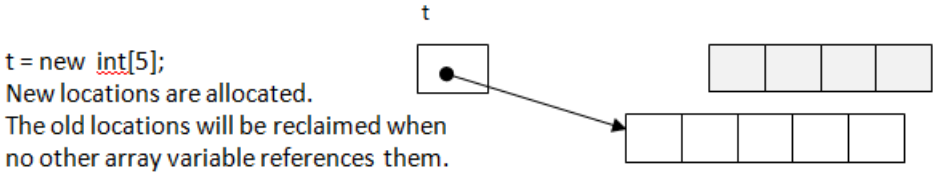
what we end up with is two storage locations for `s` and `t` that reference the same 4 elements. We haven't created a copy, rather we have two array variables that reference the same 4 elements:



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.



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.

```
1  import java.util.Scanner;
2  /**
3   * Display average of 7 values
4   */
5  public class AverageTemperature
6  {
7      public static void main(String[] args){
8          Scanner kb = new Scanner(System.in);
9          double[] temperature = new double[7];
10         System.out.println("Enter 7 temperatures:");
11         for (int i=0; i<7; i++)
12             temperature[i] = kb.nextDouble();
13         double sum = 0.0;
14         for (double t:temperature) sum +=t;
15         System.out.println("average= "+sum/7.0);
16     }
17 }
```

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.

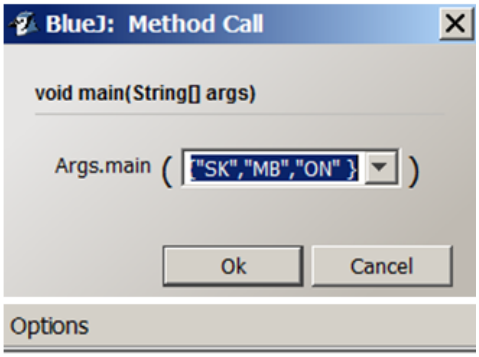
Example 3 Passing arguments to main()

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

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

```
1  /**
2   * Print the values passed into the program
3   */
4  public class Args
5  {
6     public static void main(String[] args){
7        System.out.println("The elements of args:");
8        for (String s: args) System.out.print(" "+s);
9     }
10 }
```

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.

```
1  import java.util.Scanner;
2  /**
3   * Student information is in two arrays.
4   * Find student number and report name.
5   */
6  public class StudentInfo
7  {
8      public static void main(String[] args){
9          String[] name =
               {"Joe","Linda","Mary","Peter","Lee"};
10         int[] number = {123, 222, 345, 567, 890};
11         Scanner kb = new Scanner(System.in);
12         System.out.println("Enter student number:
               ");
13         int toFind = kb.nextInt();
14         for (int i=0; i<number.length; i++)
15             if (toFind==number[i])
16                 System.out.println(name[i]);
```

```
17        }
18  }
```

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.

```
1  import java.util.Scanner;
2  /**
3   * From monthly sales calculate monthly average.
4   */
5  public class MonthlySales
6  {
7      public static void main(String[] args){
8          double[] sales = new double[12];
9          Scanner kb = new Scanner(System.in);
10         System.out.println("Enter monthly sales"
11             +" enter -1 after last value");
12         int numberMonths=0;
13         double aSale = kb.nextDouble(); //1st month
14         while(aSale != -1) {
15             sales[numberMonths++] = aSale;
16             aSale = kb.nextDouble();
17         }
18         double sum = 0;
19         for (int i=0; i<numberMonths; i++)
20             sum+=sales[i];
21         if (numberMonths>0) System.out.println(
22             "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).

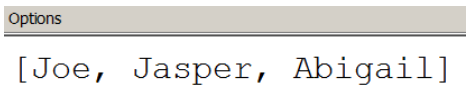
Arrays and ArrayLists

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.

```
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`.



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},
```

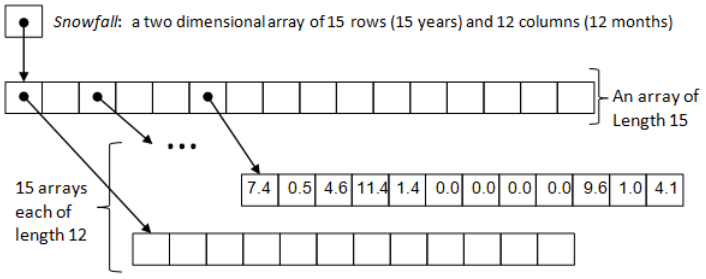
```
{7.4,0.5,4.6,11.4,1.4,0,0,0,0,9.6,1,4.1},
{3.6,3,8.6,0.3,0.2,0,0,0,0,9.8,4.4,29.4},
{15.9,5.5,6.7,0.9,0,0,0,0,0,3,2.5,20.9},
{3.1,5.1,5.4,2.9,3.4,0,0,0,0,0,1.7,10.3},
{4.9,0,13.8,7.4,0,0,0,0,0,17.2,9.3,11.1},
{2.6,5.8,12.8,0.5,1.3,0,0,0,0,0,1.5,3.3},
{8,5.3,2.5,1.2,1,0,0,0,0,8.5,4.5,16.5},
{4.9,20.2,0,1,0,0,0,0,0,5.5,1.7,5.2},
{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.



When a program references an array element the JVM accesses the stor-

age 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 i^{th} row is referenced by `snowfall[i]` ... the number of elements in the i^{th} row is `snowfall[i].length()`.
  - In line 22 the value read is stored in the i^{th} row and j^{th} 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 j^{th} element of the i^{th} 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)
```

```
8       throws FileNotFoundException {
9           // an array of 15 rows and 12 columns
10          double [][] snowfall = new double[15][12];
11          // get data from file
12          Scanner f = new Scanner(
13              new File("SnowfallInInches.txt"));
14          System.out.println("\nData read from "
15              +"SnowfallInInches.txt "+
16              "\nby year from 2000 to 2014, and for "+
17              "\neach year from January to December");
18          // outer loop controls the row subscript
19          for (int i=0; i<snowfall.length; i++){
20              // inner loop controls the column
21              for (int j=0;j<snowfall[i].length;j++){
22                  snowfall[i][j] = f.nextDouble();
23              }
24          }
25          // display the contents of the table by year
26          System.out.println("Data obtained is:");
27          for (int i=0; i<snowfall.length; i++){
28              System.out.println();
29              for (int j=0;j<snowfall[i].length;j++){
30                  System.out.print(snowfall[i][j]+"\t");
31              }
32          }
33      }
34  }
```

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

```
1  import java.util.Scanner;
2  public class DriversTrips
3  {
4      public static void main(String[] args ){
5          // 2D array with varying number
6          //    of elements per row
7          int[][] trips ={
8              {25, 29, 30, 40},
9              {44, 25},
10             {22, 27, 55, 33, 80},
11             {55, 57, 45},
12             {31, 42, 49, 46}
13         };
14         System.out.println("\n\t\tDriver Trips");
```

```
15          // number of drivers = number of rows
16          //    is trips.length
17          for (int i=0; i<trips.length; i++){
18              System.out.print("driver: "+i+"\t");
19              // number of trips for ith driver
20              //    is trips[i].length
21              for (int j=0;j<trips[i].length;j++){
22                  System.out.print(trips[i][j]+"\t");
23              }
24              System.out.println();
25          }
26      }
27
28  }
```

Figure 2.5: Output: DriversTrips.

```
Options

                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

```
1  import java.util.Scanner;
2  import java.io.File;
3  import java.io.FileNotFoundException;
4
5  public class DisplaySnowfall
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         // prompt user ... display snowfall
20         System.out.println("Enter the year as "+
21             "an integer, "+
22             "\nthen the name of the month:");
23         Scanner kb = new Scanner(System.in);
24         // convert year to proper subscript
25         int year = kb.nextInt()-2000;
26         String month = kb.next();
27         // convert month to proper subscript
28         int monthInt = convertMonth(month);
29         System.out.print("The snowfall for "+
```

```
30              month+" in "+ (2000+year)+" is "+
31              snowfall[year][monthInt]+" inches");
32      }
33
34      public static int convertMonth(String month){
35          int monthInt;
36          switch (month.toLowerCase()){
37              case "january": monthInt = 0; break;
38              case "february": monthInt = 1; break;
39              case "march": monthInt = 2; break;
40              case "april": monthInt = 3; break;
41              case "may": monthInt = 4; break;
42              case "june": monthInt = 5; break;
43              case "july": monthInt = 6; break;
44              case "august": monthInt = 7; break;
45              case "september": monthInt = 8; break;
46              case "october": monthInt = 9; break;
47              case "november": monthInt =  10; break;
48              case "december": monthInt = 11; break;
49              default: monthInt = -1; // bad month
                    name
50          }
51          return monthInt;
52      }
53  }
```

Figure 2.6: Output: Sample run for DisplaySnowfall.

```
Options
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++)
```

```
26              for (int j=0; j<a[i].length; j++)
27                  c[i][j] =a [i][j] + b[i][j];
28          // display the 3 matrices
29          displayMatrix("A = ",a);
30          displayMatrix("B = ",b);
31          displayMatrix("C = ",c);
32      }
33      public static void displayMatrix(
34          String heading,int[][] m){
35          System.out.println(heading);
36          for (int i=0; i<m.length; i++){
37              for (int j=0; j<m[i].length; j++)
38                  System.out.print(m[i][j]+"\t");
39              System.out.println();
40          }
41      }
42  }
```

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 ith row of A and the jth 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

Data Validation

3.1 Character

3.2 String

3.3 StringBuilder

Chapter 4

Enumeration Classes (Enums)

To enumerate is to list all values. There are many situations where a Java application is works with a well-defined set of values for some entity. 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, Two, ...
6. states of a door: open, closed, ...
7. four directions: north, south, east, west
8. grades: A, B, ...

When Java 1.5 was released Enums were added to the Java language. Enum is a variation on the Java class that provides for cases where the programmer needs a small number of values. In order to create an Enum when you use BlueJ you click the button New Class and in the pop-up you select Enum instead of Class, give a name to the Enum, and click the OK button. See Figure 4.1 where a new Enum named Day is to be created.

Figure 4.1: Create a new Enum

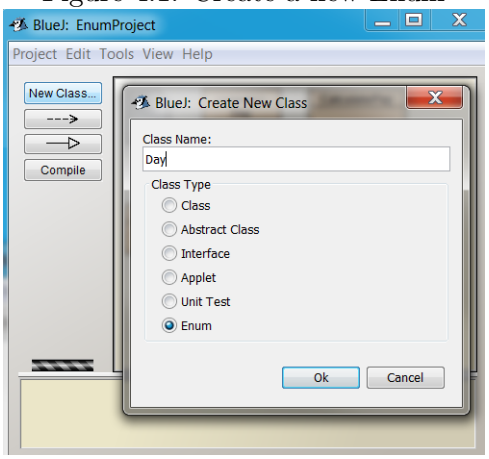
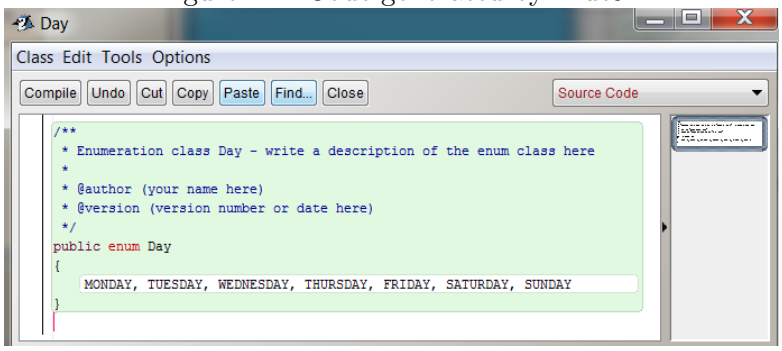


Figure 4.2 shows the default code placed in an Enum by the BlueJ editor. This code is generated for any Enum and happens to be exactly what someone would use (or start with) for an Enum to represent the 7 days of the week. If we were creating a different Enum we would need to edit the code appropriately.

Figure 4.2: Code generated by BlueJ



Note how the declaration of an Enum begins in a way 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
{
}
```

In Java code the way these constants are specified is usually done by including a prefix "Day." where the constants are said to be fully-qualified: Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, etc. 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
```

There is Java feature called *static import* (see exercises) that will enable code to specify an enum simply, for example:

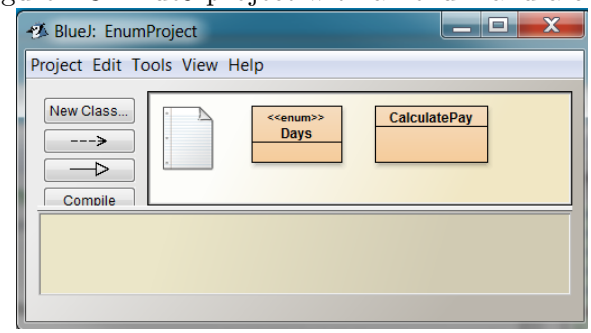
```
today = Monday;
```

We will now show how simple enums can be used in code.

Example 1 Using an enum for days of the week

In this example we have a project with an enum Day, and a Java class CalculatePay - see Figure 4.4. The example is a simple gross pay application that requires 3 things: rate of pay, hours worked, and the day of the week to be used for the calulation - see the following listing. If the current day is a weekend day then the rate of pay is doubled. Note in the code how the if statement uses == to compare two values.

Figure 4.3: BlueJ project with an enum and a class



Listing 4.1: Use Enums for the day of the week.

```
1  /**
2   * Calculate gross pay for one day
3   * based on hours worked and rate of pay.
4   * For weekends the rate of pay is doubled.
5   */
6  public class CalculatePay
7  {
8      public static void main(String[] args){
9          // set today to be Saturday
10         Day today = Day.SATURDAY;
11         double rate = 15.00;
12         int hours = 6;
13         double gross;
14         if (today == Day.SATURDAY
15           ||today == Day.SUNDAY)
16             gross = 2 * rate * hours;
17         else
18             gross = rate * hours;
19     }
20 }
```

4.1 values() Method

In its basic form an enum is a set of distinct values. Java provides several methods that are pre-defined for enums. In this section we consider the values() method. This method returns an array of values. For instance if we coded

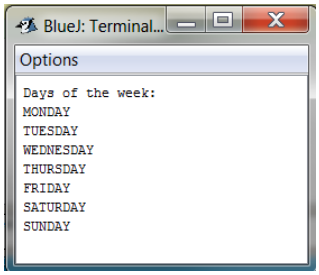
```
Day[] mydays = Day.values();
```

we would have the set of days as elements of an array named *mydays*. In the following listing we see that we can iterate through and display the list, in this case, one value per line. The output follows.

Listing 4.2: Use valueOf() to get Enum constants.

```
1  /**
2   * Use values() to get the enum values
3   */
4  public class EnumValues
5  {
6      public static void main(String[] args){
7          Day[] daysOfWeek = Day.values();
8          System.out.println("Days of the week:");
9          for (Day d: daysOfWeek)
10             System.out.println(d);
11     }
12 }
```

Figure 4.4: Displaying the elements returned from values()



4.2 Enum constants are objects

An Enum is a special type of java class. It is different from 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.

Because enums are represented by objects they can have data and methods. In this section two important points are made pertaining to:

1. the way values for fields in an enum are specified
2. that enum instances are created automatically by the JVM and not through the use of the *new* operator.

Here, we consider a simple example where each Enum constant has an associated piece of information. Consider an educational environment where a student receives a grade in each course they are taking. Suppose these grades will be one of A, B, C, D, and F; these grades can be represented by an enum *Grade*. Suppose also that each grade has a corresponding grade point value (A has a grade point value of 4, B ...3, C ...2, D ...1, and F ...0). When we define the enum Grade we specify the values for fields (grade points) in parentheses following each constant, as in:

```
public enum Grade{A(4), B(3), C(2), D(1), F(0);
```

One small matter to notice in the above line is that Java requires a semicolon at the end of the list of constants with field values- this was not necessary in the previous examples. The next two things specified are fields and constructors. In this example we need a field for the grade point values:

```
 private int gradePoint;
```

and then the constructor ...the constructor is called automatically by the JVM. Since we have defined constants above with integer values in parentheses the constructor must have an `int` parameter:

```
Grade(int gradePoint){
    this.gradePoint = gradePoint;
}
```

The process of instantiating enum types is handled by the JVM automatically; it is not possible for a programmer to purposely instantiate an enum. If you were to try and compile a statement such as:

```
Day d = new Day();
```

Java compiler will reject your code; your program will not compile.

Chapter 5

Hierarchies

5.1 Object class

5.2 Inheritance

5.3 Abstract classes

5.4 Interface classes

Chapter 6

File Processing

6.1 Binary files

Chapter 7

Exception Handling

7.1 File handling

Chapter 8

Recursion

Chapter 9

Sorting and Searching

Chapter 10

Map Class