JV – Object-Oriented Programming in Java

Seminar 6 – Arrays and Data structures

We have now reached a new plateau where we can stop and take stock of what we've done so far. With what we now know, we can write fairly sophisticated programs consisting of many methods and interacting classes. Individual methods may perform simple arithmetic tasks or define alternative paths (choice points) or cause a sequence of statements to be repeated many times.

What is holding us back now is the nature of the data items we are working with. These are all scalar, and what we would like to do is to be able to use more complicated data structures. In this seminar we concentrate on a particular data structure known as an array. I posted a document in the readings thread using another data structure - a vector - that can be quite useful because it can hold objects of different data types at the same time. You may use either vectors or arrays for the DQs and assignments, so long as you understand how each of the structures work and what the differences between them are. There are plenty of other data structures (hash files, linked lists, queues, stacks, etc.) that you are welcome to use if you would like to research them, but that we will not cover in class.

After having a look at arrays, we will then move on and introduce file handling. Being able to read and write to and from a file will further allow us to make more sophisticated applications.

I. Overview of Arrays

The most straightforward (and oldest) form of data structure is the *array*. All the data types we have considered so far have been scalar, i.e. data items that can only have one value. In an array, a data item consists of a numbered collection of similar components. As such, it can be viewed as simply a series of data items, all of the same type, stored in a series of locations in memory such that a single value is held at each location (see **Figure 6-1**).

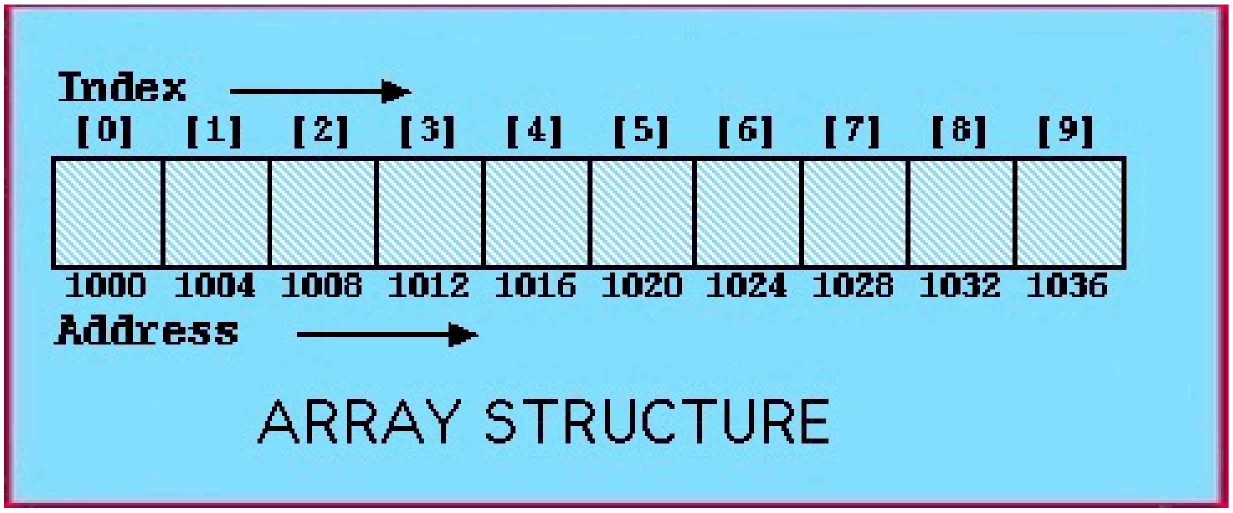


Figure 6-1: *Array structure*

Items in arrays are called *elements* (some authors use the term *cell* to describe the locations within an array). The array in **Figure 6-1** has 10 elements, so we say that the array is of *length* 10. Specific elements in an array can be identified through the use of an *index*. In Java, the index is always of the type int (this is not the case in all programming languages). In Java, the first index is always the int 0 which is referred to as the *lower bound*, and the last index number is then referred to as the *upper bound*. The value for the upper bound of the example array presented in **Figure 6-1** is 9, and the general representation for any array is from 0 to n.

A. Arrays and Their Relationships with Strings

If you think about it a string is an array, albeit a specialised one, in that it consists of a sequence of one or more data items of type char. Therefore, we have in fact already met the array data structure although we did not know it as such at the time.

II. Array Declaration

When declaring arrays we are doing two things:

1. Declaring the type of the array, i.e. the type of the elements to be contained within it.

2. Declaring the array length (and by extension the upper bound which is the array length minus one - indexing starts at 0).

In Java, to declare an array data item, we use a declaration statement of the form:

< TYPE_NAME > [ ]arrayName;

If we omit the [ ] this would tell the Java compiler to create space for a <type_name> data item called arrayName only, which is not what we want to do in this case. The [ ] indicates that this is actually an unspecified array of the datatype we declare, as in String [ ] args in the main() method. So that an appropriate amount of memory can be set aside to hold the elements in the array, we must also tell the compiler how many elements to expect. We do this as follows:

<TYPE_NAME> [ ]arrayName = new <TYPE_NAME> [n]

where n is the number of elements. Let us consider an example of an array declaration:

int [ ] temperatureArray = new int[10];

Here we have created an array of 10 integers called temperatureArray (presumably to store a sequence of temperatures in). Alternatively, if we wish to store our temperatures as doubles we would declare the array as follows:

double [ ] temperatureArray = new double[10];

Note that if we had not used an array to store our temperatures we would have had to declare 10 distinct data items:

double temperature0;
double temperature1;

```
double temperature2;
double temperature3;
double temperature4;
double temperature5;
double temperature6;
double temperature7;
double temperature8;
double temperature9;
```

The above approach is very tedious for small numbers of elements and completely impractical for any number of elements beyond about 50.

III. Array Assignment

Given knowledge of the index for an array element, we can assign a value to that element or change its value using an assignment operation. For example:

```
myArray[0] = 4;
```

Here we have an array data item (named myArray) and we have assigned a value of 4 to the first element of this array. Note how we have specified the element we are interested in by *indexing into the array*.

Java (along with many other programming languages) also supports the concept of *compound values*, sometimes referred to as *array aggregates*, which allow all the elements of an array to be assigned to simultaneously on initialization:

```
int[ ] myArray = {9,8,7,6,5,4,3,2,1,0};
```

IV. First Array Example Program

The class presented in **Figure 6-2** outputs the contents of a five element array. Note that:

1. The array length is defined as a constant (we do not like magic numbers!).

2. The values for the array are assigned upon declaration.

3. To process the array, we loop through it using the loop counter to index into the array.

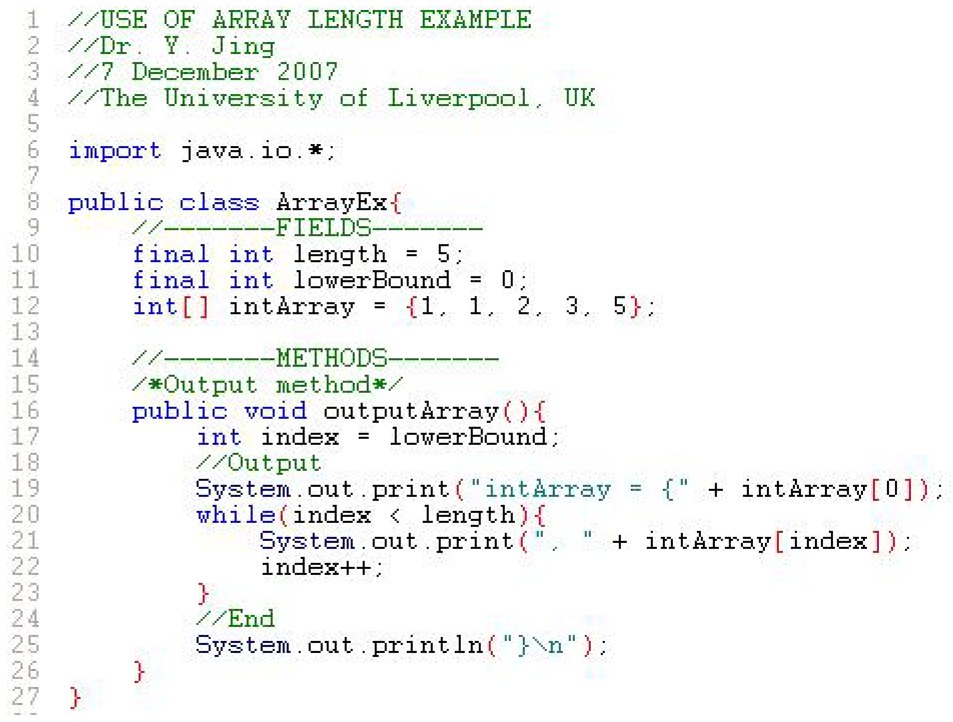
4. When the loop counter is no longer less than the value of the length constant the loop terminates.

5. The first element in a Java array always has the index 0.

6. The output from this code will be as follows:

   intArray = {1, 1, 2, 3, 5};

7. We will need an application class with a main() method to make use of the code.



```java
//USE OF ARRAY LENGTH EXAMPLE
//Dr. Y. Jing
//7 December 2007
//The University of Liverpool, UK

import java.io.*;

public class ArrayEx{
    //--------FIELDS--------
    final int length = 5;
    final int lowerBound = 0;
    int[] intArray = {1, 1, 2, 3, 5};

    //--------METHODS--------
    /*Output method*/
    public void outputArray(){
        int index = lowerBound;
        //Output
        System.out.print("intArray = {" + intArray[0]);
        while(index < length){
            System.out.print(", " + intArray[index]);
            index++;
        }
        //End
        System.out.println("}\n");
    }
}
```

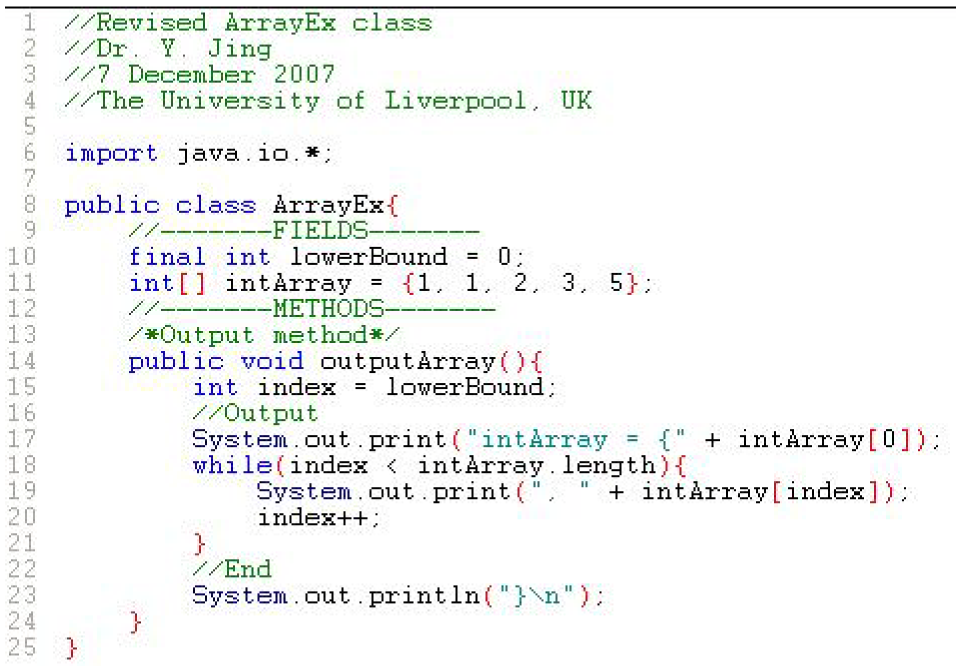
Figure 6-2: *First array example program*

V. Array Variables

Some programming languages also supply a set of predefined variables, sometimes called *attributes*, describing various features of arrays and indeed other data types. Typically, such variables describe features such as the lower bound, upper bound and length of an array. In Java, the only array variable available is length. To use this, we must use the dot operator to link it to an appropriate array. So:

int numberOfElements = myArray.length;

will assign the number of elements in the array myArray to the integer data item numberOfElements. The length variable can be useful when processing arrays. For example, the code given in **Figure 6-2** can be revised, as shown in **Figure 6-3**, so that the array may be processed without explicitly needing to know its length (i.e. we can dispense with the length constant).



```
//Revised ArrayEx class
//Dr. Y. Jing
//7 December 2007
//The University of Liverpool, UK

import java.io.*;

public class ArrayEx{
    //--------FIELDS--------
    final int lowerBound = 0;
    int[] intArray = {1, 1, 2, 3, 5};
    //--------METHODS--------
    /*Output method*/
    public void outputArray(){
        int index = lowerBound;
        //Output
        System.out.print("intArray = {" + intArray[0]);
        while(index < intArray.length){
            System.out.print(", " + intArray[index]);
            index++;
        }
        //End
        System.out.println("}\n");
    }
}
```

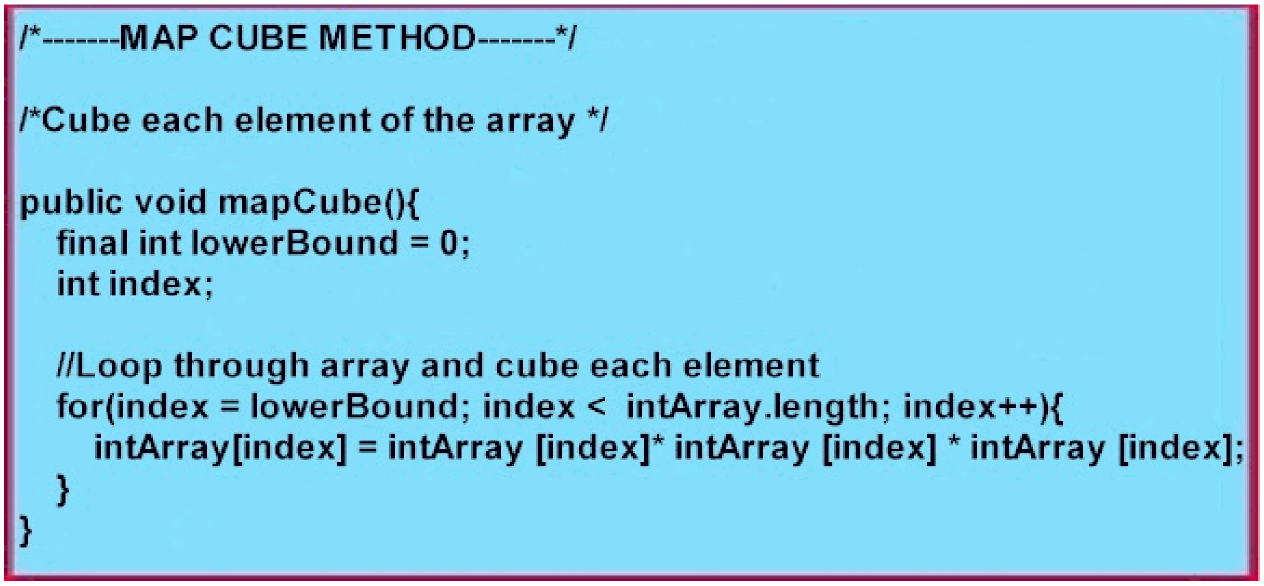
Figure 6-3: *Use of length variable*

VI. Array Processing

We can identify a number of common operations that we may wish to perform on arrays: (1) mapping, (2) filtering, (3) folding and (4) zipping. The terminology used here is that used in declarative languages (logic and functional) to describe list handling operations. However, as arrays can be thought of as lists of elements, it seems appropriate to adopt this classification of list operations to describe common array operations. We will consider each of these operations in the following sub-sections. In each case we will give an example method to be included in the ArrayEx class presented in **Figures 6-2 and 6-3** above.

A. Mapping

It is often necessary to carry out a particular operation on all the members of an array. This is called *mapping*, because we map the desired operation onto the elements of the array. For example, we may wish to cube all the elements of an array as shown in the method presented in **Figure 6-4**.

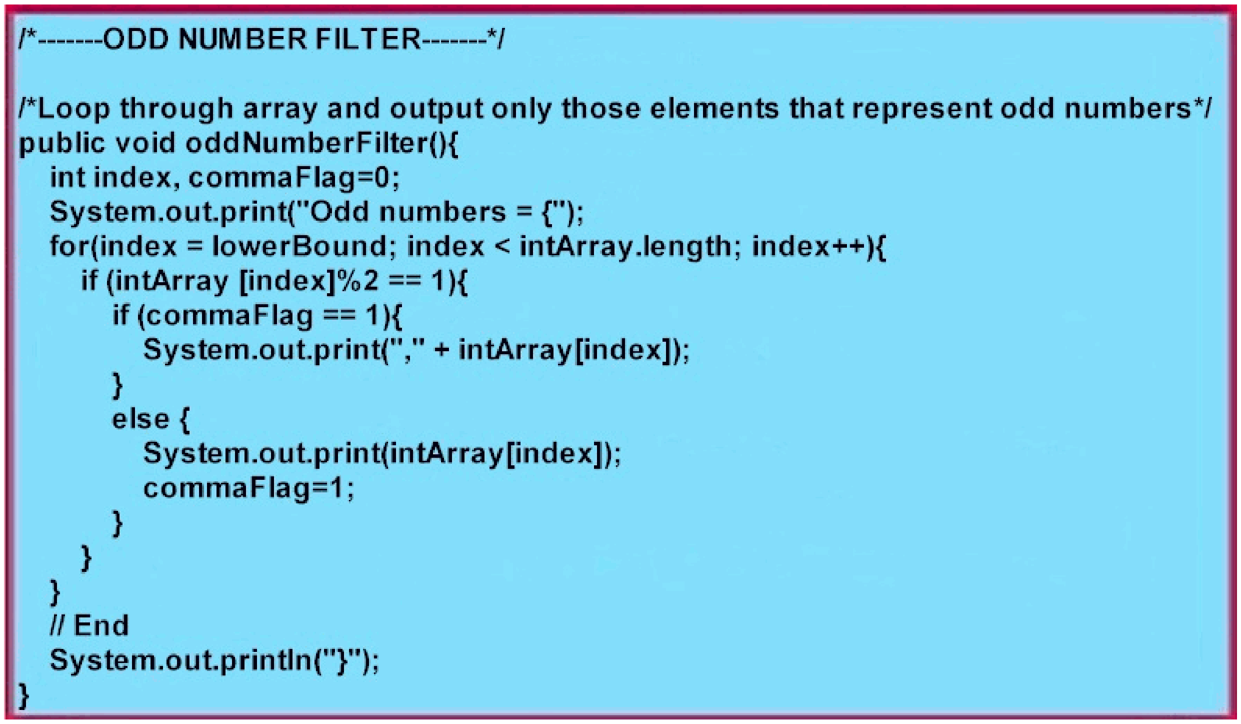


```
/*-------MAP CUBE METHOD-------*/

/*Cube each element of the array */

public void mapCube(){
   final int lowerBound = 0;
   int index;

   //Loop through array and cube each element
   for(index = lowerBound; index <  intArray.length; index++){
      intArray[index] = intArray [index]* intArray [index] * intArray [index];
   }
}
```

Figure 6-4: *Cube mapping method*

B. Filtering

The process of running through a list and keeping only those elements which pass some test is referred to as filtering. For example, identifying the odd numbers in an integer array and keeping them is filtering the array. A suitable method to carry this out is presented in **Figure 6-5**.

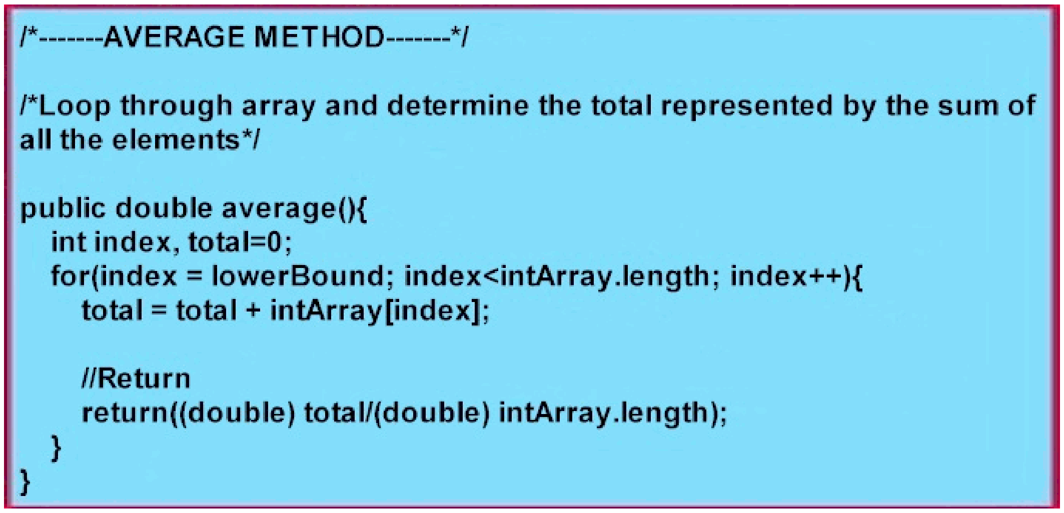


```
/*-------ODD NUMBER FILTER-------*/

/*Loop through array and output only those elements that represent odd numbers*/
public void oddNumberFilter(){
   int index, commaFlag=0;
   System.out.print("Odd numbers = {");
   for(index = lowerBound; index < intArray.length; index++){
      if (intArray [index]%2 == 1){
         if (commaFlag == 1){
            System.out.print("," + intArray[index]);
         }
         else {
            System.out.print(intArray[index]);
            commaFlag=1;
         }
      }
   }
   // End
   System.out.println("}");
}
```

Figure 6-5: *Odd number filter method*

C. Folding

The process of putting an operator between each pair of elements in an array to produce a single data item is called folding. For example, adding up all the elements of a numeric array for the purpose of finding an average value or a total value folds the array. An example average method is presented in **Figure 6-6**.

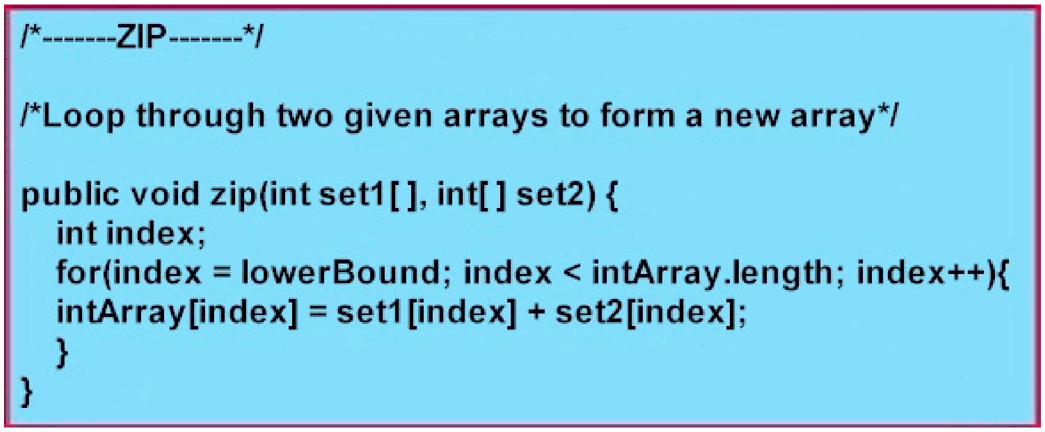


```
/*-------AVERAGE METHOD-------*/

/*Loop through array and determine the total represented by the sum of
all the elements*/

public double average(){
   int index, total=0;
   for(index = lowerBound; index<intArray.length; index++){
      total = total + intArray[index];

      //Return
      return((double) total/(double) intArray.length);
   }
}
```

Figure 6-6: *Average method*

D. Zipping

The process of combining two arrays to form a third is called zipping. For example, we may wish to add the elements of two arrays to each other to form a third array (see **Figure 6-7**). Note that the two arrays to be zipped have been passed in as arguments (we assume here that these arrays are both of length 5).



```
/*-------ZIP-------*/

/*Loop through two given arrays to form a new array*/

public void zip(int set1[ ], int[ ] set2) {
   int index;
   for(index = lowerBound; index < intArray.length; index++){
   intArray[index] = set1[index] + set2[index];
   }
}
```

Figure 6-7: *Zip method*

VII. Constrained and Unconstrained Arrays

A constrained array is an array where the upper bound is specified. We say that the bounds are static, so constrained arrays are sometimes referred to as *static* arrays. So far, we have only looked at constrained arrays. Most programming languages (including Java) support the concept of unconstrained arrays. When an unconstrained array is declared, the number of elements is not supplied until run time - typically by user input.

In **Figures 6-2 and 6-3** we defined five element integer arrays and a method – outputArray(), which can be used to output the contents of such an array. However, because the array used in the illustrations was represented as a five element static array, the class could only be used in relation to integer arrays of 5 elements. The class, and by extension the methods within it, would be much more useful if it operated with respect to integer arrays of any size from 1 up to some maximum n. This would make the class much more generic and therefore much more useful.

To do this we must amend the code given in **Figures 6-2 and 6-3** so that the length constant becomes a variable for which a value can be supplied (perhaps using a constructor) at run time. The proposed changes are presented in **Figure 6-8**. Code for an application test class to exercise the revised ArrayEx class is presented in **Figure 6-9**.

```java
//USE OF ARRAY LENGTH EXAMPLE
//Dr. Y. Jing
//7 December 2007
//The University of Liverpool, UK
import java.io.*;
public class ArrayEx{
    //--------FIELDS--------
    int length;
    final int lowerBound = 0;
    int[] intArray = {1, 1, 2, 3, 5};
    //Create BufferedReader class instance
    public static BufferedReader keyboardInput = new
            BufferedReader(new InputStreamReader(System.in));
    // --- Constructors ---
    public ArrayEx(int size){
        length = size;
        intArray = new int[length];
        }
    //--------METHODS--------
    /*Output method*/
    public void inputArray() throws IOException{
        int index = lowerBound;
        //Input
        System.out.println("Input" + length+"array elements");
        while(index < intArray.length){
            intArray[index]=new Integer(keyboardInput.readLine()).intValue();
            index++;
        }
    }
    public void outputArray(){
            int index = lowerBound;
            //Output
            System.out.print("intArray = {");
            while(index < intArray.length){
                System.out.print(intArray[index] + ", ");
                index++;
            }
            //End
            System.out.println("}\n");
    }
}
```

Figure 6-8: *Unconstrained array example*

```java
//Array example test Application
//Dr. Y. Jing
//7 December 2007
//The University of Liverpool, UK
import java.io.*;
class ArrayExApp{
// --- Fields ---
// Create BufferedReader class instance
public static BufferedReader keyboardInput = new
    BufferedReader(new InputStreamReader(System.in));
// --- Methods ---
// --- Main method ---
public static void main (String[] args) throws IOException{
    int arraySize;
    // Input value for upper bound and then create instance of arrayEx class
    System.out.println("Input array length");
    arraySize = new Integer(keyboardInput.readLine()).intValue();
    ArrayEx newArray = new ArrayEx(arraySize);
    // Input for set
    newArray.inputArray();
    // Output values for set
    newArray.outputArray();
    }
}
```

Figure 6-9: *Application class for code presented in* ***Figure 6-8***

VIII. Dynamic Memory Allocation

There are many other complex data structures; they are lists, stacks, queues, and trees.

1. Linked lists

You may want to use a linked list when it is difficult to predict the number of data elements that need be used in the list. A linked list can be accessed using a reference to the first node of the list (see figure 6-10). Each subsequent node can be accessed using the pointer stored in the previous node.

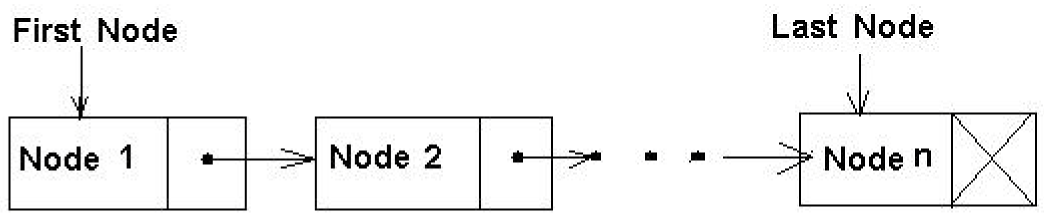


Figure 6-10: A Linked list graphical representation

The pointer (link reference) in the last node of a list is normally set to null. A node can contain any type of data. Linked lists are not static, this is because the length of a list can be increased or decreased as necessary. This is a key difference from arrays since it is difficult to change the size of arrays since they are fixed at creation time.

2. Stacks

A stack is a last-in, first-out (LIFO) data structure (see figure 6-11). Push and pop are two methods used to manipulate a stack. Method push can add a new node to the top of the stack. Method pop is a ble to remove a node from the top of the stack and returns the data in that top node.

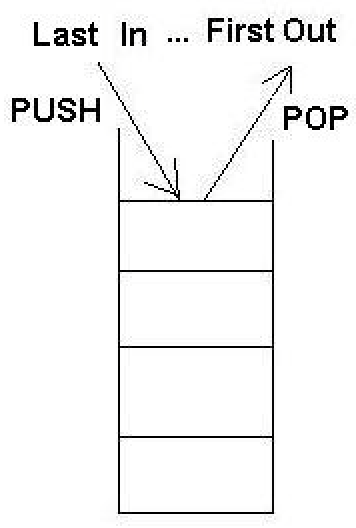


Figure 6-11: A Stack graphical representation

Why might you use stacks? Sometimes, you want to remember all the things that have happens (press them to the stack one by one as it happens), but you might want to deal with the things that happened more currently first. Think about how you put your music CDs in a spine, and pull the one on the top first.

3. Queues

Think about waiting at the checkout line in supermarket — the first person in line is serviced first, and other customers enter the line only at the end and wait to be serviced (see figure 6-12). Queue nodes can be removed only from the top of the queue and are inserted only at the rear of the queue. For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.

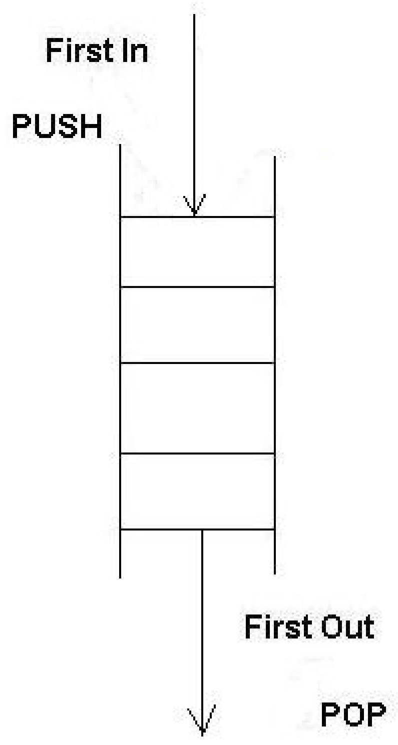


Figure 6-12: A queue graphical representation

4. Trees

Imagine a nature tree structure. It has one root, and many branches and leaves. A tree structure (see figure 6-13) is a nonlinear, two-dimensional data structure. A tree has one root note, and each tree node contains one or more links.

A binary tree is a tree with nodes all contains two links. The root node is the first node in a tree. Each link in the root node refers to a child. The left child is the first node in the left subtree, and the right child is the first node in the right subtree. The children of a node are called siblings. A node with no children is called a leaf node.

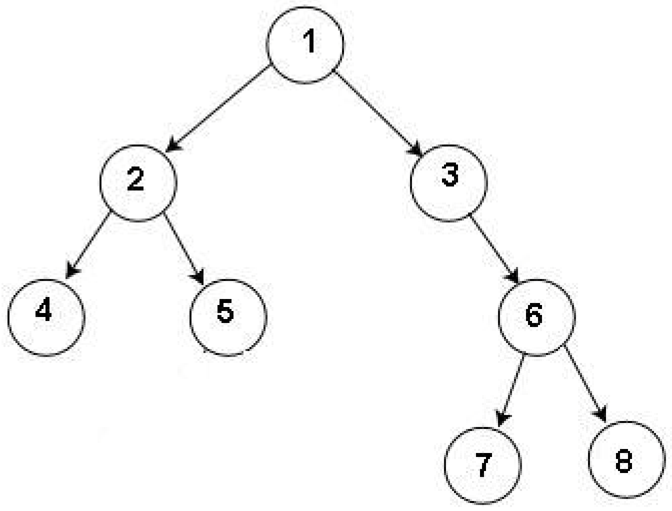


Figure 6-13: A tree graphical representation

Required Reading

In **Chapter 7** of the text, read the following sections:

7.1 - 7.9
7.11 - 12
Skim 7.14

In **Chapter 17** of the text, read the following sections:

17.4 - 17.8

Helpful Additional Readings

I've included a very simple example of a vector here (VectorDemo.java and USport.java) as another means of working with array-type structures. This is only for your benefit - you're not required to use vectors anywhere. This is simply a good complementary piece because where an array can only hold one data type of objects at a time, a vector can hold several different data types and so might be useful to practice if you have time. The extra document also provides more information about vector usage.

If you happen to get really stuck on a concept we've introduced in this week's material, please post a message as a response to this thread in the main folder to let me know. I have a ton of extra materials and instruction guides that I can post that may help clarify things for you. **All you have to do is ask! :-)**

Helpful Websites

Array Information:

http://java.sun.com/docs/books/tutorial/java/data/arrays.html

http://developer.java.sun.com/developer/TechTips/2000/tt0815.html

http://www.janeg.ca/scjp/lang/arrays.html

http://java.sun.com/j2se/1.4.1/docs/api/java/util/Arrays.html

Appendix 1: Input and Output to a File

Up until this point, all of our input has been as strings taken in through the keyboard and output as strings printed to the screen of a monitor. However, what happens if our program requires a very large amount of input, or produces output that maybe required by another program? In the first case, we cannot spend hours typing in thousands (maybe millions!) of records that our program may need. In the second case, we cannot manually transfer all our output from one program to another every time the programs are to be run. Both the above situations would be far too time consuming to even attempt in many situations, and they would introduce the possibility of human errors into many of our programs that rely on large amounts of input data.

To overcome the problems of dealing with large amounts of input data and output data, and to simply allow our programs to store data for future use, we can read and write data to files. A file must be opened before we can gain access to it for reading or writing. In Java, when a file is opened, an object is created and a stream is associated with the object. We already know that the term *stream* refers to any input source or output destination for data.

A. Reading From a File

The class FileReader contains a constructor that requires the pathname of an input file, for example:

```
FileReader file = new
FileReader(“C:\\myProgFiles\\fileNo1.txt”);
```

In the above code, a file will be opened called fileNo1.txt in the myProgFiles directory which is a sub directory of the root directory C. Note the double backslash (\\) and not the usual single backslash (\) separating the directories within the path. This is to avoid any confusion with an escape character within the string. If we used a single backslash, it would simply escape the character in front of it and would not be treated as part of the string. By using the double backslash, the escape character is escaped and therefore is used as part of the string.

However if we wish to open a file in the current directory of a program, we do not need to specify the whole path. For example, if our class file wishing to open a file is already in the above myProgFiles directory, we can simply use the following:

```
        FileReader file = new FileReader("fileNo1.txt");
```

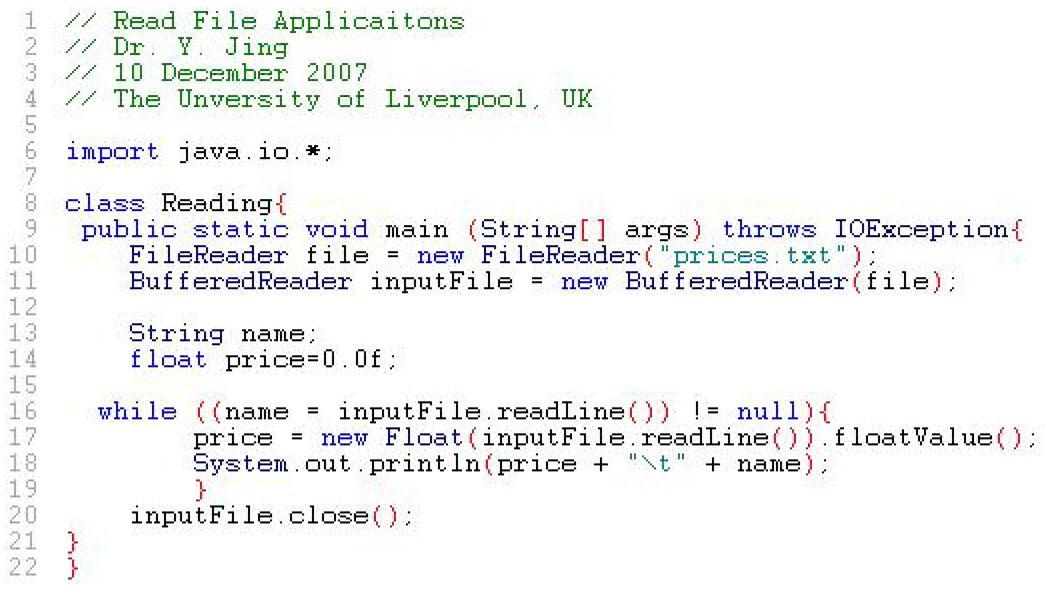
To provide the same method that was used with keyboard input **readLine()**, it is necessary to use the object *file* in the constructor of the BufferedReader class, creating a stream object *inputFile* that is associated with reading from the file with the path name C:\myProgFiles\fileNo1.txt like:

```
        BufferedReader inputFile = new BufferedReader(file);
```

When a file is no longer required it must be closed. The method defined in the class BufferedReader is **close()**. The following example illustrates how one can read from a file. First, I have created a text file called prices.txt of the following format:

```
300
television
50
table
500
music center
200
dvd player
0
```

displaying the price of a house hold item followed by the item name. Then I have created a program that will open this file and display the information on to the screen (**Figure A1-1**).



```
// Read File Applicaitons
// Dr. Y. Jing
// 10 December 2007
// The Unversity of Liverpool, UK

import java.io.*;

class Reading{
 public static void main (String[] args) throws IOException{
    FileReader file = new FileReader("prices.txt");
    BufferedReader inputFile = new BufferedReader(file);

    String name;
    float price=0.0f;

  while ((name = inputFile.readLine()) != null){
        price = new Float(inputFile.readLine()).floatValue();
        System.out.println(price + "\t" + name);
        }
    inputFile.close();
}
}
```

Figure A1-1: *Program to read information from a file*

B. Writing to a File

The class FileWriter contains a constructor that also requires the pathname of the output file:

FileWriter file2 = new FileWriter(“C:\\myProgFiles\\fileNo2.txt”);

or as in the same circumstances explained for the FileReader:

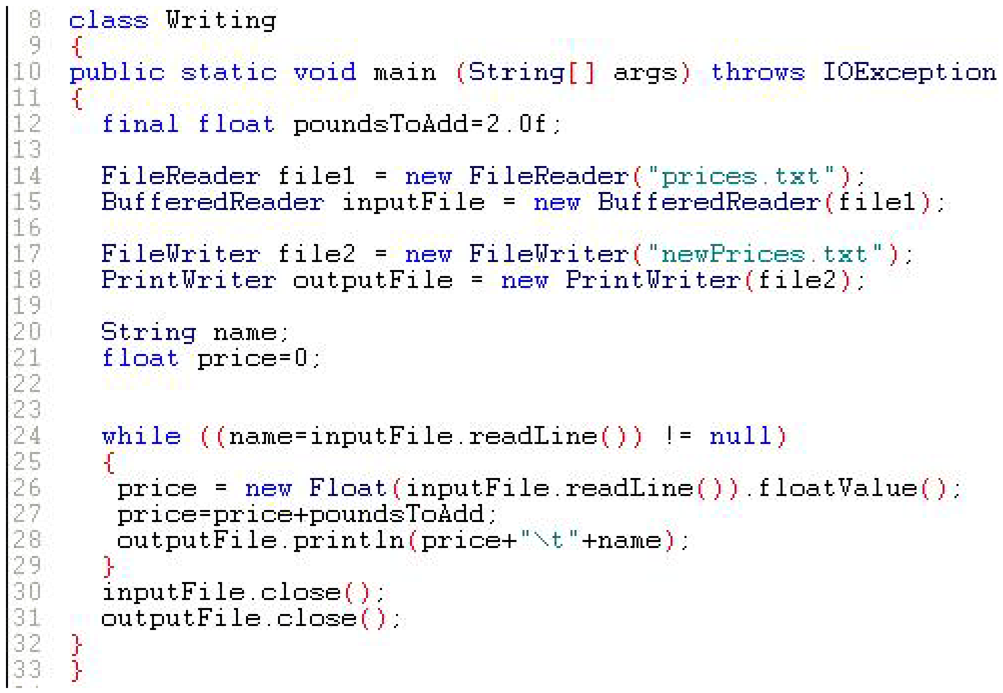
FileWriter file2 = new FileWriter(“fileNo2.txt”);

If this file does not exist it will be created. To provide the same methods that were used with screen output (print, println, flush) it is necessary to use the object *file2* in the constructor of the PrintWriter, thus creating a stream object outputFile that is associated with writing to the file fileNo2:

PrintWriter outputFile = new PrintWriter(file2);

Now when using file-writing, output does not need to be directed to the screen only. Instead, it can be directed to a text file with a specified name and path if necessary. The following example

(**Figure A1-2**) modifies the prices of the previous example and writes the new prices to a file instead of the screen (although it could do both if desired).



```
class Writing
{
public static void main (String[] args) throws IOException
{
  final float poundsToAdd=2.0f;

  FileReader file1 = new FileReader("prices.txt");
  BufferedReader inputFile = new BufferedReader(file1);

  FileWriter file2 = new FileWriter("newPrices.txt");
  PrintWriter outputFile = new PrintWriter(file2);

  String name;
  float price=0;


  while ((name=inputFile.readLine()) != null)
  {
   price = new Float(inputFile.readLine()).floatValue();
   price=price+poundsToAdd;
   outputFile.println(price+"\t"+name);
  }
  inputFile.close();
  outputFile.close();
}
}
```

Figure A1-2: *Writing to a file*

After running this program you can then open the newPrices.txt file and compare them to the old prices.

C. Tokenizing

The practice of creating a text file with one item of data per line does not always convey what a group of items represents. For example, it would be better to be able to read from a text file containing data that was grouped together in lines (like the output from the program in **Figure 11**).

Given that the grouping of items of data or words on one line is a more natural way of creating a text file, how can we program the computer to split up a string into the individual items (tokens) of data on a line?

To do this we use methods from the class StringTokenizer found in the package **util**. A partial listing of the constructors and methods of this class is as follows:

```
public class StringTokenizer implements Enumeration{
   public StringTokenizer(String str);
   public StringTokenizer(String str, String delim);
   public String nextToken();
   public String nextToken(String delim);
   public int countTokens();
}
```

As usual a complete listing can be found within the Java API.

Before a line of text can be split up, an object of type StringTokenizer must be instantiated from the input string e.g.:

```
StringTokenizer data = new
StringTokenizer(inputFile.readLine());
```

To split a line of text into tokens, we must know what the delimiting character is for each token. For example, in a line of text where items of data are separated by a space and the end of the line by a return character, both the space and the return character are the token delimiters. If the delimiter is not specified then it is assumed to be any white spaced character such as space '\u0020\', horizontal tab '\u0009\' and new line '\u000A\'.

Both the nextToken methods return a token of type String. The method nextToken() will return a token delimited by any white space character while the method nextToken(String delim) will return a token specifically delimited by the character delim e.g.

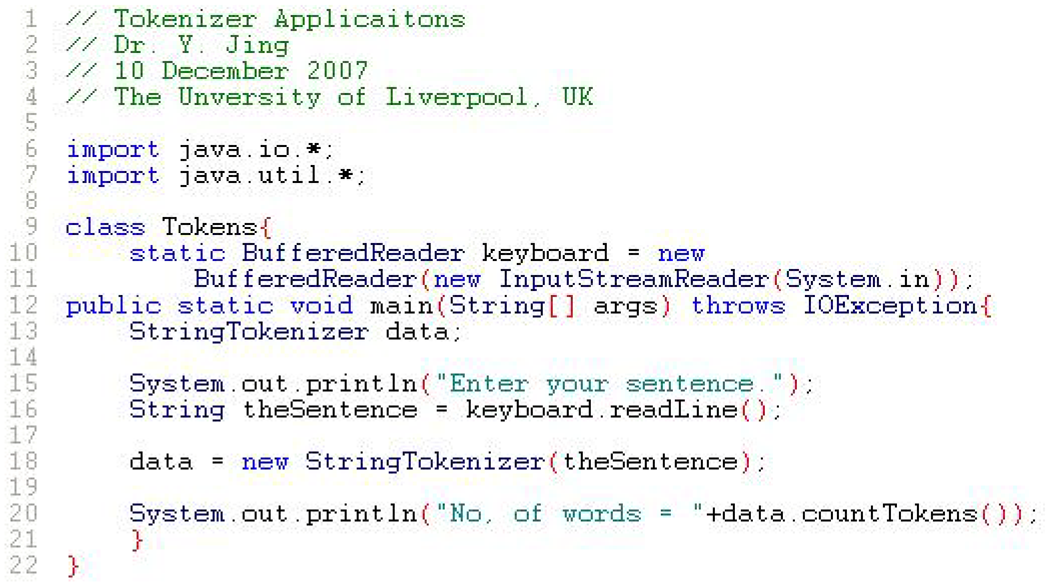
```
String token = data.nextToken();
```

Will return a token delimited by any white space character while:

```
String token = data.nextToken("\u000A");
```

Will return a token specifically delimited by a new line character. The method countTokens() will return the number of tokens in a string that are delimited by any white spaced character.

The following (and final program for this lecture) is a very simple program (**Figure A1-3**) that will ask the user to type in a sentence at the keyboard, after which it will display the number of words within the sentence. Although this example is quite simple, it demonstrates well how tokenizing strings works.



```java
// Tokenizer Applicaitons
// Dr. Y. Jing
// 10 December 2007
// The Unversity of Liverpool, UK

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

class Tokens{
    static BufferedReader keyboard = new
        BufferedReader(new InputStreamReader(System.in));
public static void main(String[] args) throws IOException{
    StringTokenizer data;

    System.out.println("Enter your sentence.");
    String theSentence = keyboard.readLine();

    data = new StringTokenizer(theSentence);

    System.out.println("No. of words = "+data.countTokens());
    }
}
```

Figure A1-3: *Tokenizing a string*