BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213 – Object Oriented Programming

Lab 5: Arrays and Strings

Time:	2 hours
Objective:	Teach students the concepts of arrays, passing arrays to a method, working with multi-dimensional arrays. Strings, String functions, StringBuffer and StringTokenizer
Concepts Covered:	Arrays, Passing arrays, Multi-dimensional arrays, Strings, StringBuffer, StringTokenizer
Prerequisites:	Basic java syntax and programming concepts, understanding of classes and objects, data types

1. Arrays

An array is a data structure that stores a collection of values of the same type. In Java, arrays are defined as Objects, which is how they differ from C arrays. We will now see how to work with arrays.

1.1. Declaring and initializing arrays

Arrays may be initialized in any of the following ways:

```
int [] arr1
// This only declares a new variable called arr1 but doesn’t initialize it with an array yet

int [] arr2 = new int [5];
// This uses the new operator to initialize arr2 with a new integer array of size 5.

int [] arr3 = {1, 2, 3, 4, 5};
// This statement initializes arr3 to an array with the given values
```

1.2. Accessing elements of an array

Elements in an array are numbered with index values, started from 0 and ending at 1 less than the length of the array. Elements are accessed using these index values.

```
int[] arr = {1, 2, 3, 4, 5};
System.out.println(arr[1]);
// This statement prints the value with index 1 i.e. at the second position in the array
```

In addition to iterating through an array to access its elements, Java has a “for each” function which allows you to access array elements without index values. Its use is as follows.

```
int[] arr = {1, 2, 3, 4, 5};
for (int element : arr)
    System.out.println(element);
```

1.3. Copying arrays

```
int[] arr = {1, 2, 3, 4, 5};
int[] arrCopy = arr;
arrCopy[2] = 6;
System.out.println("Original: " + arr[2] + "\nCopy:  " + arrCopy[2]);
```

Run the code above and observe the output. When we try to copy `arr` to `arrCopy` using `arrCopy = arr`, we make both arrays point to the same variable. Hence, any changes made in `arrCopy` using `arrCopy[2] = 6` are reflected in the original `arr` as well.

To copy all the values of an array into another array, we can use the `copyOf` method from the `Arrays` class. Please note that the `Arrays` class is a part of the `util` package, so it must be imported as follows:

```
import java.util.Arrays;
```

```
int[] arr = {1, 2, 3, 4, 5};
int[] arrCopy = Arrays.copyOf(arr, arr.length);
```

Run the code above and observe what happens to the original array when you change the copied array.

The second parameter is the length of the new array, see what happens if this parameter is given as less or more than the length of the original array.

1.4. Multi-dimensional arrays

Multi-dimensional arrays are used for table or matrix arrangements. As the name suggests, elements of these arrays have multiple dimensions, like a row and a column for a two-dimensional array. Hence, more than one index is used to access elements of such arrays. We will stick to two-dimensional arrays as these are the most commonly used ones.

1.4.1. Declaring and initialising two-dimensional arrays

Two-dimensional arrays are declared and initialised in a similar manner to single-dimensional arrays.

```
int [][] arr1;
arr1 = new int[5][10];

int [][] arr2 = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

1.4.2. Accessing elements of two-dimensional arrays

Elements of a two-dimensional array can be iterated over as follows:

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

for (int[] row : arr) {
    for (int x : row) {
        System.out.print(x + "\t");
    }
    System.out.println();
}
```

To access elements explicitly using their indices, it can be done as follows:

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

for (int i = 0; i<arr.length; i++) {
        for (int j = 0; j<arr[i].length; j++) {
            System.out.print(arr[i][j] + "\t");
        }
        System.out.println();
}
```

1.5. Ragged arrays

Ragged arrays are essentially a kind of multi-dimensional array, the only difference being that the lengths of all rows are not common. Hence, they can be visualised in a kind of ragged shape. Java multi-dimensional arrays are stored as an "array of arrays", so the implementation of ragged arrays becomes easy.

For example, to initialise a triangular array in which the i^{th} row has (i+1) elements:

```
int n = 5;
int [][] arr = new int[n][];

for (int i = 0; i < n; i++) {
        arr[i] = new int[i+1];
}
```

Each individual element of the array can be accessed and changed as we have done before. Remember that as the row lengths are not common, you cannot use a single variable as row size while iterating.

1.6. Exercises

1.6.1. Complete the method given below and run it to get the output.

```
public class RetailStore {
        private int[] itemId = {1001, 1002, 1003, 1004, 1005};
        private double[] price = {13.50, 18.00, 19.50, 25.50};

        private double computePrice(int value) {
            int itemPrice;
            /*
             * TODO: Write code for the method to return the
             * price of the item whose item ID is  equal to value.
             */
            return itemPrice;
        }

        private double totalPrice () {
            /*
```

```
             * TODO: Write code for the method to return the total price of
             * all the items in the list. Use for each, instead of a
             * regular for loop.
             */
      }

      public static void main(String[] args) {
            RetailStore retailOne = new RetailStore();
            System.out.println("Price of item ID 1002 is " + retailOne.computePrice(1003));
            System.out.println("Price of item ID 1004 is " + retailOne.computePrice(1004));
            System.out.println("Total price of all items is " + retailOne.totalPrice());
      }
}
```

1.6.2. Complete the code given below and run it to get the output.

```
public class CompoundInterest {
      public static void main(String[] args) {
            final double STARTRATE = 10;
            final int NRATES = 6;
            final int NYEARS = 10;

          double[] interestRate = new double[NRATES];
          for (int j = 0; j < interestRate.length; j++)
            /*
             * TODO: Set interest rates to 10%, 11%. . .15%
             * Hint: STARTRATE has been defined as 10
             */

          double[][] balances = new double[NYEARS][NRATES];
          //Balances is a two-dimensional array
          //Each year is represented by a row in this array


          for (int j = 0; j < balances[0].length; j++) {
            /*
             * TODO: Write code here to set all balances for the first year to 10000
             */
          }

          for (int i = 1; i < NYEARS; i++) {
            for (int j = 0; j < NRATES; j++) {
                        /*
                         * TODO: Write code here to update balances for
                         * all the subsequent years as follows:
                         * interest = prevYearBalanceForThatRate * interestRate
                         * newBalance = prevYearBalanceForThatRate + interest
                         */
            }
          }

          // Printing interest rates in one row
          for (int j = 0; j < interestRate.length; j++)
            System.out.printf("%9.0f%%", 100 * interestRate[j]);
          System.out.println();

          /*
           * TODO: Write code below to print out the complete balances table in the
           * appropriate format i.e. each year in a separate row.
           */
      }
}
```

1.6.3. Complete the code given below and run it to get the output.

```
public class LotteryArray {
      public static void main(String[] args) {
            final int NMAX = 10;

            // Allocating triangular array
            int[][] odds = new int[NMAX + 1][];
            int n;
            for(n = 0; n <= NMAX; n++)
                  // TODO: initialise the nth row of odds to an array of (n+1) elements


            for (n = 0; n < odds.length; n++)
                  for (int k = 0; k < odds[n].length; k++){
            /*
             * TODO: Compute binomial coefficient:
             * n*(n-1)*(n-2)...*(n-k+1)/(1*2*3*...*k)
             * and store in the kth element of the nth row
             */
            }

            // TODO: Print triangular array odds
      }
}
```

1.6.4. Write a program that takes 2 matrices and their orders as input from the user. Check for compatibility for matrix multiplication, show appropriate error message in case of incompatibility. Otherwise, pass both matrices and orders to a method that should return the **result of their matrix multiplication**. Print the result.
Also, write a method that takes a 3x3 matrix and **returns its inverse**.

1.6.5. You are given the skeleton code for four incomplete classes named `Name`, `Student`, `StudentList` and a driver class named `Test`. You have to complete the code for all the classes as per the following specifications:

(a) `class Name`: `Name` class encapsulates the three attributes of a person's name as first name, middle name and last name. This class supplies only one constructor which receives a string value, and this string value contains the values for all the three attributes either in comma (,) or semicolon (;) separated format.

If the values are comma separated then the three attribute values are in the following order:<First name>, <Middle Name>, <Last Name>

If the values are semicolon separated, then the three attribute values are in the following order: <Last name>;<Middle Name>;<First Name>

For example: If the value supplied for constructor parameter is "Rajesh,Kumar,Khanna" the first name is "Rajesh", middle Name is "Kumar" and last name is "Khanna". Assume string parameter for constructor either contains comma or semicolon in its value but not both. There is no need to check for validity of the string parameter. The `Name` class supplies accessor methods for every instance field. The class also supplies a method `getName`() for retrieving the full name of a person. The `getName` () method returns the full name after concatenating and adding spaces between first name, middle name and last name fields in order. The class also supplies `toString`() method which returns value after simply concatenating the values of first name, middle name and last name fields.
The skeleton code for the class `Name` is given below:

```
class Name {
    private String fname; // First Name
    private String mname; // Middle Name
    private String lname; // Last Name
    // TODO: Provide accessor methods as per the given specification

    // TODO: Provide implementation for getName() method as per the specification

    Name(String name) {
       /*
        * TODO: Complete the constructor by extracting the values
        * of three name fields. Note that name value may be either
        * comma separated or semicolon separated
        */

    }
} // End of class Name
```

(b) class `Student:` `Student` class has two attributes: `name` of type `Name` [`Name` class as defined above] and `age` of type **`int`**. The class supplies only one parameterised constructor which receives the values for all instance fields of the class as parameters. First parameter is of `Name` type and second is of type **`int`**. The class supplies accessor method for every instance field and `toString()` method which returns a string after concatenating and adding spaces between values of first name, middle name, last name and age attributes for this instance. Provide the implementation for the class `Student` as mentioned below as per the specification given above.

```java
class Student {
    private Name name; // name attribute of student
    private int age;// age attribute of student

    /*
     * TODO: Complete the Student class by adding proper
     * constructor, accessor methods and by adding any
     * other methods which are required as per specification
     */

    // Write Your Code From Here

}// End of Student class
```

(c) class `StudentList` : This class encapsulates a list of size 10 of type `Student`. This class contains only static fields and methods. The list of students is maintained as an array of type `Student[]`. The skeleton code is given as follows:

```java
class StudentList {

    public static Student[] list = new Student[10]; // list of students
    public static int count = 0; // count of students added in the list

    public static void addStudent(Student std) {
        if(count >= 20)
            return; // if count is already 20 or more, then return
        // TODO: Add student std to list if count is less than 20
    }

    public static Student[] getStudentsWithAge(int age) {
       /*
        * TODO: This method returns an array of all the students whose age is
        * equal to age parameter passed to this method. If no such
        * student is found then it returns null.
        */
    }
    public static Student[] getStudentsWithLastName(String lastName) {

       /*
        * TODO: This method returns an array of all the students whose last name attribute
        * value matches with lastName parameter of this method. If no such student
        * is found then it returns null.
        */
    }

    public static Student[] getStudentsInRange(int minAge, int maxAge) {
       /*
        * TODO: This method returns an array of all the students whose age falls between
        * minAge and maxAge (both parameters inclusive).
        */
    }
}// End of class StudentList
```

(d) class `Test`: This class is the driver class. The incomplete code for the class is given below. You have to complete methods `readStudent()` and `main()` of this class as per commented specification.

```
import java.io.*;

class Test{
      public static Student readStudent() throws IOException{
            /*
             * TODO: This method reads the student details from the
             * user and returns the Student instance.
             * Values to be read from System.in are:
             * 1. First name of Student, 2. Middle name of student,3. Last name of Student,
             * 4. Name format (1 for comma(,) separated and 2 for semicolon separated),
             * 5. age of student
             */
      } // End of readStudent() Method

      public static void main(String args[]) throws IOException{
            /*
             * TODO:
             * 1. Write java code for reading details of 10 students and add them
             * in the static list ofStudentListclass.
             *
             * 2. Write java code for displaying the all the students with age 20
             * from static list field of StudentList class
             *
             * 3. Write java code for displaying the student details for all students
             * having last name “Sharma” from static list of StudentList class
             *
             * 4. Write java code for displaying all the students whose age falls in
             * the range minAge = 16 and maxAge = 20 from static list of StudentList class
             */
      }// End of main() Method
}// End of Test class
```

2. Strings

Strings are sequences of Unicode characters. Java does not have a built-in String type, instead the Java library contains a predefined class called `String`. All Java Strings are instances of this class. The String class is a part of the `java.lang` package which is always imported by default.

Java Strings are immutable, meaning that you cannot directly change a character in an existing string. In the case that you need to do so, you change the contents of the variable and make it point to a different string.

Empty strings are strings with length equal to 0. A String variable holding the value **`null`** indicates that no object is currently associated with the variable. Both these conditions can be checked as follows:

```
// Check if string is empty
if (str.length() == 0)
// or
if (str.equals(""))

// Check if string is equal to null

if (str == null)
```

2.1. String class methods

The Java String class contains close to 100 methods. We will look at some of the most useful ones below.

- `char charAt(int index)`
  returns the character at the specified `index` in the String, assuming 0-based indexing

- `int compareTo (String other)`
  returns a negative value if the string comes before `other` in dictionary order, a positive value if the string comes after `other` in dictionary order, or 0 if the strings are equal

- `boolean equals (String other)`
  returns **`true`** if the string equals `other`

- `boolean equalsIgnoreCase (String other)`
  returns **`true`** if the string equals `other`, ignoring case distinction

- `boolean startsWith (String prefix)` *OR* `boolean endsWith (String suffix)`
  returns **`true`** if the string begins with `suffix` or ends with `suffix`

- `int indexOf (String str)` *OR* `int indexOf (String str, int fromIndex)`
  returns the start of the first substring equal to the string `str`, starting at index 0 or at `fromIndex` if specified, or -1 if `str` does not occur in this string

- `int lastIndexOf (String str)` *OR* `int lastIndexOf (String str, int fromIndex)`
  returns the start of the last substring equal to the string `str`, starting at the end of the string or at

`fromIndex` if specified, or -1 if `str` does not occur in this string

- `int length()`
  returns the number of characters in the string

- `String replace (String oldString, String newString)`
  returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`

- `String substring (int beginIndex)` *OR* `String substring (int beginIndex, int endIndex)`
  returns a new string consisting of all characters from `beginIndex` until the end of the string or until (`endIndex` − 1), both inclusive

- `String toUpperCase()` OR `String toLowerCase()`
  returns a new string containing all characters in the original string, with uppercase characters converted to lowercase, or lowercase characters converted to uppercase

- `String strip()` OR `String stripLeading()` OR `String stripTrailing()`
  return a new string by eliminating leading and trailing, or just leading or trailing whitespace in the original string

- `String join (String delimiter, String. . .elements)`
  returns a new string joining all `elements` with the given `delimiter`

2.2. StringBuffer and StringBuilder

StringBuffer and StringBuilder are two classes provided by Java to enable implementation of mutable Strings. This means that strings declared as StringBuffer and StringBuilder objects allow for their character sequences to be modified without creating a new object. StringBuffer and StringBuilder both use the same methods, the difference between them is in their efficiency. StringBuffer is thread-safe, meaning that it allows multiple threads to edit it using synchronisation. StringBuilder is not thread-safe but is faster and more efficient for single-threaded processes. Some of the useful methods in these classes are:

- `StringBuilder append(String str)` OR `StringBuilder append(char c)`
  appends `str` or `c` and returns **`this`**

- `void setCharAt(int i, char c)`
  sets the character at index `i` to `c`

- `StringBuilder insert (int offset, String str)` OR `StringBuilder insert (int offset, char c)`
  inserts string `str` or character `c` at index `offset` and returns **`this`**

- `StringBuilder delete (int startIndex, int endIndex)`
  deletes all characters of the string having index `startIndex` to (`endIndex` − 1) and returns **`this`**

- `String toString ()`
  returns a `String` object with the same contents are the `StringBuffer` or `StringBuilder` used to call the method

2.3. StringTokenizer

`StringTokenizer` is a class present in the `util` package, so it must be imported as follows:

```
import java.util.StringTokenizer;
```

This class allows the application to break a String object into tokens. The set of characters that separate tokens, called delimiters, are specified at the time of instantiation of the `StringTokenizer` object. The default delimiter used is whitespace. Please note that the `StringTokenizer` class is a legacy class, and the `String split()` method is preferred for most modern applications.
Given below is a simple example demonstrating the use of `StringTokenizer.`

```
import java.util.StringTokenizer;

public class Demo {
    public static void main(String[] args) {

        // Input string
        String s = "Object Oriented Programming Concepts";

        // Create a StringTokenizer object with space as the delimiter
        StringTokenizer st = new StringTokenizer(s, " ");

        // Tokenize the string and print each token
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

Output:

```
      Object
      Oriented
      Programming
      Concepts
```

2.4. Exercises

2.4.1. The source code of the `RetailStore` class from the first exercise is modified to include a new String parameter called ItemName. This parameter is in the format “<BrandName> <Sport> <Item>-<Price>”. Complete the code given below as per the comments.

```
public class RetailStore {
      private int[] itemId;
      private double[] price;
```

```
	private String itemName[];

	/* The constructor is used here for the initialization purpose*/
	public RetailStore() {
		itemId = new int[] { 1001, 1002, 1003, 1004, 1005 };
		price = new double[] { 950.00, 750.00, 450.00, 350.00, 250.00 };
		itemName = new String[] {
		    "Yonex Tennis Racket-950","Yonex Badminton Racket-750",
		    "Silvers Badminton Racket-450","Cosco Badminton shuttle-350",
		    "Cosco Tennis Racket-250"
		 };
	}

	protected double computePrice(int value) {
		/*
		 * TODO: Complete this method exactly as per the first exercise
		 */
	}

	protected String fetchDescription(int value) {
		/*
		 * TODO: Complete the method so that it returns the
		 * description (itemName) of the product with price equal
		 * to the value parameter passed to the method
		 */
	}
}
```

Below is the class whose main method must be run.

```
public class RetailStoreExample extends RetailStore {

	public static void main(String[] args) {
		RetailStore retailOne = new RetailStore();

		String description = retailOne.fetchDescription(1004);

		// below line illustrates the use of split function of String class
		String StringArray[];
		/*
		 * TODO: Use the StringTokenizer class to split the String description
		 * using whitespace delimiter, and store the tokens in the array StringArray
		 */

		String type;
		/*
		 * TODO: Store the third token that has the item and price in the variable type
		 */

		char c;
		/*
		 * TODO: Store the first character of the variable type in c
```

```
             */

            System.out.println("The type of the item is " + type);
            System.out.println("The character at starting position is " + c);

            int index;
            /*
             * TODO: Store the location of the symbol "-" in index
             */

            String stringFromSubstring;
            /*
             * TODO: Use the substring method to store the price in the variable
             * stringFromSubstring (Hint: Use the index variable)
             */
            System.out.println("Price using substring method: " + stringFromSubstring);

            String stringFromDouble;
            /*
             * TODO: Save the price of the product with ID 1004 as a
             * String in stringFromDouble by calling the computePrice method
             */

            System.out.println("Price using computePrice method: " + stringFromDouble);

            boolean flag;
            /*
             * TODO: Compare the strings stringFromDouble and stringFromSubstring
             * and use flag to store true if they are equal, false if not.
             * Print flag with an appropriate message.
             */
      }
}
```

2.4.2. (a) Consider a class `Address` which stores the address of a user in the following attributes:

- line1 : String
- line2 : String
- line3 : String
- city : char[]
- state : char[]
- pin : String

The class supplies one parameterised constructor which receives only one parameter of type String in the following format: “line1$line2$line3$city$state$pin”

"$" character is used as a separator to separate the values of line1, line2, line3, city, state and pin attributes. The constructor updates values of all attributes. The class supplies accessor methods for every instance field. All accessor methods return only String type value. **Implement the `Address` class in Java as per mentioned specification.**

(b) Complete the code given below for the `AddressList` class as per the comments.

```
class AddressList{
      public static int countAddressWithCity(Address[] addressList, String city){
            /*
             * TODO: Complete this method so that it returns the count of the
             * addresses from addressList which have the city attribute
             * equals to city parameter passed for this method.
             */
      }// End of method countAddressWithCity()
      public static int countAddressWithPin(Address[] addressList, String strP){
            /*
             * TODO: Complete this method so that it returns the count of the
             * addresses from addressList which have the pin attribute
             * starting with strP parameter passed for this method.
             * For this and the above method, if the length of any passed argument
             * is zero or value of any passed argument is null then it returns -1.
             */
      }// End of method countAddressWithCity()
      public static Address[] getAddressWithCity(Address[] addressList, String city){
            /*
             * TODO: Complete this method so that it returns all the addresses from
             * addressList by storing them in String[] which have the city
             * attribute equals to city parameter passed for this method.
             * If addressList does not contain any address with city attribute value equal
             * to city parameter passed for this method even then it returns null.
             */
      }// End of method getAddressWithCity()
      public static Address[] getAddressWithPin(Address[] addressList, String strP){
            /*
             * TODO: Complete this method so that it returns all the addresses from
             * addressList by storing them in String[] which have their pin
             * attribute starting with strP parameter passed for this method.
             * For this and the above method, if the length of any passed argument is
             * zero or value of any passed argument is null then it returns null.
             * If addressList does not contain any address whose pins attribute value
             * starts with strP parameter passed for this method even then it returns null.
             */
      }// End of method getAddressWithCity()
}// End of class AddressList
```

(c) **Write a suitable driver class** named `Test` for the class `AddressList` and test all the methods.

2.4.3. Write a program to take an IP address as String input from the user and check for its validity. Conditions for validity of an IP address are:
- It must contain four decimal numbers separated by dots (".").
- All four numbers must be in the range 0-255.
- No other characters apart from numerical digits and (".") can be present.

2.4.4. Write a program to take String input from the user containing even number of digits only. Print the original String first. Then, swap pairs of characters within the same String and print the updated String.
Eg. Input : ThisIsAStatement
Output : hTsisISAatetemtn
Character pairs swapped : (T, h), (i, s), (I, s), (A, S), (t, a), (t, e), (m, e), (n, t)

2.4.5. Write a program to take input of a String from the user and print out all permutations of the String.
Eg. Input : abc
Output : abc, acb, bac, bca, cab, cb