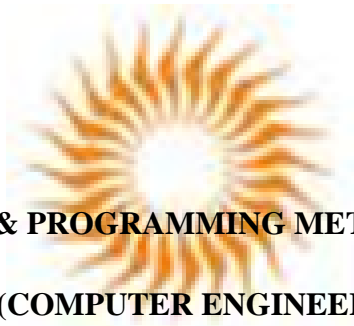




SIES GST, NERUL, NAVI MUMBAI

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL



OBJECT ORIENTED & PROGRAMMING METHODOLOGY (CSL304)

S.E. (COMPUTER ENGINEERING)

SEMESTER-III

(R-2016)-Ver1

COMPUTER ENGINEERING DEPARTMENT

DEPARTMENT'S VISION

To be a centre of Excellence in Computer Engineering to fulfill the rapidly growing needs of the Society.

DEPARTMENT'S MISSION

M1: To Impart quality education to meet the professional challenges in the area of Computer Engineering.

M2: To create an environment for research, innovation, professional and social development.

M3: To nurture lifelong learning skills for achieving professional growth.

M4 To strengthen the alumni and industrial interaction for overall development of students.

PROGRAMME EDUCATIONAL OBJECTIVES(PEOs)

PEO1: Practice Computer engineering in core and multi-disciplinary domains.

PEO2: Exhibit leadership skills for professional growth.

PEO3: Pursue higher Studies for career advancement

PROGRAMME OUTCOMES (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics science engineering fundamentals and an mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual and as a member or leader in diverse teams and individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: long learning: Recognize the need for and have the Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs)

PSO1: To apply computational and logical skills to solve Computer engineering problems.

PSO2: To develop interdisciplinary skills and acquaint with cutting edge technologies in software industries.

GENERAL INSTRUCTIONS (Do's And Don'ts)

1. Wearing ID-Card is compulsory.
2. Keep your bag at the specified place.
3. Shut down the system after use.
4. Place the chairs in proper position before leaving the laboratory.
5. Report failure/Non-working of equipment to Faculty In-charge / Technical Support staff immediately.
6. Know the location of the fire extinguisher and the FIRST-AID Box and how to use then in case of an emergency.
7. Do not eat or drink in the laboratory.
8. Do not litter in the laboratory.
9. Avoid stepping on electrical wires or any other computer cables.
10. Do not open the system unit casing or monitor casing particularly when the power is turned ON.
11. Do not insert metal objects such as clips, pins and needles into the computer casing. They may cause fire.
12. Do not remove anything from laboratory without permissions.
13. Do not touch, connect or disconnect any plug or cable without permission.

LAB OUTCOMES (LO)

1. To apply fundamental programming constructs.
2. To illustrate the concept of packages, classes and objects.
3. To elaborate the concept of strings, arrays and vectors.
4. To implement the concept of inheritance and interfaces.
5. To implement the notion of exception handling and multithreading.
6. To develop GUI based application.

LAB ARTICULATION MATRIX (MAPPING WITH PO & PSO)

	Lab Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
LO1	To apply fundamental programming constructs.	3		2										2	
LO2	To illustrate the concept of packages, classes and objects.		2		2									2	
LO3	To elaborate the concept of strings, arrays and vectors.	2		3	3									2	
LO4	To implement the concept of inheritance and interfaces.	3	3		2									2	
LO5	To implement the notion of exception handling and multithreading.			3										2	
LO6	To develop GUI based application.	3		3		2								2	
Average															

Laboratory Assessment

Academic Year: _____

Class/Sem: Div: _____

Batch: _____

Student Name: _____

Roll No: _____

Course Name: _____

Course Code: _____

1. Preparedness and Efforts/ Preparation and Knowledge		
3: Well prepared and puts efforts.	2: Not prepared but puts efforts or prepared but doesn't put efforts	1: Neither prepared nor puts efforts
2. Presentation of output/ Accuracy and Neatness of Documentation		
3: Uses all perfect Instructions / Interrupts/Presented well.	2: Uses some perfect Instructions / Interrupts/moderate presentation.	1: Doesn't use any of the proper Instruction / Interrupt/ Not presented properly.
3. Results/ Participation in Practical Performance		
3: Participate and gets proper results	2: Participate but doesn't get proper result or gets result but with the help of faculty in-charge	1: Neither Participate nor gets the results
4. Punctuality		
3: Get the experiment checked in-time and is always in-time to the lab sessions	2: Some time delays the experiment checking or is late to the lab sessions for few times	1: Most of the time delays experiment checking and / or comes late for lab sessions
5. Lab Ethics		
3: Follows proper lab ethics by keeping the lab clean and placing things at their right place	2: Sometimes doesn't follow the lab ethics	1: Most of the times makes the lab untidy and keeps things at wrong place

EVALUATION:

Performance Indicator/ Expt. No —————→	1	2	3	4	5	6	7	8	9	10
Lab outcomes- CO's	1	1	2	2	2	3	3	3	4	4
1. Preparedness and Efforts/ Preparation and Knowledge										
2. Presentation of output/ Accuracy and Neatness of Documentation										
3. Debugging and results/ Participation in Practical Performance										
4. Punctuality										
5. Lab Ethics										
Total										
Average										

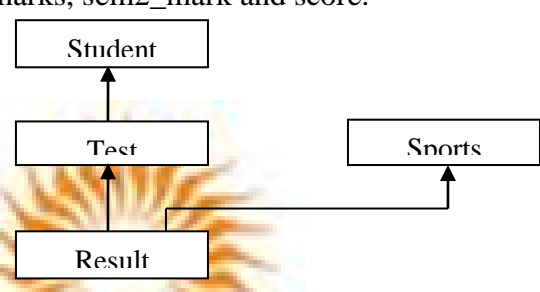
Performance Indicator/ Expt. No —————→	11	12	13	14	15	16	17
Lab outcomes- CO's	4	5	5	5	6	6	6
1. Preparedness and Efforts/ Preparation and Knowledge							
2. Presentation of output/ Accuracy and Neatness of Documentation							
3. Debugging and results/ Participation in Practical Performance							
4. Punctuality							
5. Lab Ethics							
Total							
Average							

Exceed Expectations (3), Meet Expectations (2). Below Expectations (1)

Faculty In-charge

LIST OF EXPERIMENTS

SR NO	EXPERIMENT TITLE
1	<p>Program on various ways to accept data through keyboard.</p> <p>a. WAP to calculate Factorial of a number using manual input.</p> <p>b. WAP to perform addition of two numbers using command line arguments.</p> <p>c. WAP for conversion of temp using scanner class.</p>
2	<p>a. WAP to print the grade for an input test score: using the if-else ladder</p> <ul style="list-style-type: none"> • Percentage • 0-60 • 61-70 • 71-80 • 81-90 • 91-100 • Grade • F • D • C • B • A <p>b. Switch-case: menu driven calculator.</p> <p>c. WAP to display the following pattern</p> <pre> A B C D C B A A B C B A A B A A </pre>
3	<p>a. WAP to design a class to represent bank account. It should include the following members; Name of depositor, Account Number, Type of Account and balance amount in the account. It should also has methods to</p> <ol style="list-style-type: none"> i. To assign initial values. ii. To deposit an amount. iii. To withdraw an amount after checking balance. iv. To display the name and balance. <p>b. WAP to define employee class containing name ,age ,phone number ,basic salary and number of present days as data members and suitable methods to take input and display all values along with gross salary.</p>
4	<p>a. WAP on method overloading to calculate distance between two points.</p> <p>b. Program on constructor and constructor overloading</p>
5	<p>Create a package (say) ABC, WAP operation.java in ABC (add (), subtract t(), multiply (), divide ()). WAP Imppackage.java that imports the package ABC.</p>

6	a. WAP to perform Insertion sort. b. WAP to perform matrix multiplication.
7	a. WAP to count number of occurrences of given character using string class. b. WAP to reverse a word using string class.
8	a. WAP to reverse a complete sentence using string buffer class. b. WAP to use methods of vector class.
9	<p>WAP to implement three classes namely Student, Test and Result. Student class has member as rollno, Test class has members as sem1_marks and sem2_marks and Result class has member as total. Create an interface named sports that has a member score. Derive Test class from Student and Result class has multiple inheritance from Test and Sports. Total is formula based on sem1_marks, sem2_mark and score.</p>  <pre> classDiagram class Student class Test class Result class Sports Student < -- Test Test < -- Result Sports < -- Result </pre>
10	Write a abstract class program to calculate area of circle, rectangle and triangle.
11	Program on Interface.
12	WAP for exception handling using try/catch,throw, and finally demonstrating different inbuilt exceptions (IO, Arithmetic...)
13	<p>a. Define an exception called “NoMatchException” that is thrown when a string is not equal to “India”. Write a program that uses this exception.</p> <p>b. WAP to create negative number exception while calculating factorial.</p>
14	WAP to print 1A2B3C4D5E6F7G8H9I using two child threads.
15	WAP to display smiling face using applet.
16	WAP to handle mouse event using GUI.
17	Case study on database connectivity.

18	Miniproject
----	-------------

Experiment No.1

Aim: Program on various ways to accept data through keyboard.

- a.WAP to calculate Factorial of a number using manual input.
- b.WAP to perform addition of two numbers using command line arguments.
- c.WAP for conversion of temp using scanner class.

Theory:

The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

String[] args within the declaration of the main method means. It represents a String array named args. Since args is nothing more than an identifier, we can replace it with any other identifier and the program will still work. We need to know now is how a String array can be passed as an argument when we execute the program. We pass it through the command line itself. Consider that we have a class named Add. The following statement normally used to execute the program.

java Add

When we wish to pass the String array, we simply include the elements of the array as simple Strings beside the class name. Enclosing the Strings in quotes is optional. Consecutive Strings are separated with a space. For example, if we wish to pass a three element String array containing the values "1", "2", and "3" any of the following lines is entered on the command prompt.

java Add 1 2 3

java Add "1" "2" "3"

Since these arguments are passed through the command line, they are known as command line arguments. The String arguments passed are stored in the array specified in the main() declaration. args[] is now a three element String array. These elements are accessed in the same way as the elements of a normal array. The following is the complete Add program which is capable of adding any number of integers passed as command line arguments.

```
public class Add {  
  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int i = 0; i < args.length; i++) {  
            sum = sum + Integer.parseInt(args[i]);  
        }  
        System.out.println("The sum of the arguments passed is " + sum);  
    }  
}
```

And here are some sample executions from the command line:

```
C:\Users\srgf\Desktop>java Add  
The sum of the arguments passed is 0  
  
C:\Users\srgf\Desktop>java Add 3 4  
The sum of the arguments passed is 7  
  
C:\Users\srgf\Desktop>java Add 3 4 7 9 34  
The sum of the arguments passed is 57
```

Parsing Numeric Command-Line Arguments:

Converting String to integer:

- A numeric string can be *converted* into an integer using the `parseInt` method in the `Integer` class.

The `parseInt` method is invoked as follows:

`Integer.parseInt(NumericString)`

The `parseInt` method returns an integer value of the numeric string.

If an application needs to support a numeric command-line argument, it must convert a `String` argument that represents a number, such as "34", to a numeric value. `parseInt` throws a `NumberFormatException` if the format of `args[0]` isn't valid. All of the `Number` classes, i.e. `Integer`, `Float`, `Double`, and so on — have `parseXXX` methods that convert a `String` representing a number to an object of their type.

There are various ways to read input from the keyboard, the `java.util.Scanner` class is one of them.

Scanner class

The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace by default. It provides many methods to read and parse various primitive values. Java Scanner class is widely used to parse text for string and primitive types using regular expression. Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Scanner class is a class in java.util, which allows the user to read values of various types. There are far more methods in class Scanner than you will need in this course. We only cover a small useful subset, ones that allow us to read in numeric values from either the keyboard or file without having to convert them from strings and determine if there are more values to be read.

Class Constructors

There are two constructors that are particularly useful: one takes an InputStream object as a parameter and the other takes a FileReader object as a parameter.

```
Scanner in = new Scanner(System.in); // System.in is an InputStream
Scanner inFile = new Scanner(new FileReader("myFile"));
```

Numeric and String Methods:

<i>Method</i>	<i>Returns</i>
int nextInt()	Returns the next token as an int. If the next token is not an integer, InputMismatchException is thrown.
long nextLong()	Returns the next token as a long. If the next token is not an integer, InputMismatchException is thrown.
float nextFloat()	Returns the next token as a float. If the next token is not a float or is out of range, InputMismatchException is thrown.
double nextDouble()	Returns the next token as a long. If the next token is not a float or is out of range, InputMismatchException is thrown.
String next()	Finds and returns the next complete token from this scanner and returns it as a string; a token is usually ended by whitespace such as a blank or line break. If no token exists, NoSuchElementException is thrown.
String nextLine()	Returns the rest of the current line, excluding any line separator at the end.
void close()	Closes the scanner.

The numeric values may all be on one line with blanks between each value or may be on separate lines. *Whitespace* characters (blanks or carriage returns) act as separators. The next method returns the next input value as a string, regardless of what is keyed. For example, given the following code segment and data

```
int number = in.nextInt();  
float real = in.nextFloat();  
long number2 = in.nextLong();  
double real2 = in.nextDouble();  
String string = in.next();
```

The following line was inserted after you organized your imports:

```
import java.util.Scanner;
```

The Java Scanner class is like any class you create, except it was created for you. Since Java already comes with it, it had to be imported.

A program in java consists of one or more class definitions. One of these classes must define a method `main()`, which is where the program starts running

// A Java Hello World Program

```
public class HelloWorld  
{  
    public static void main( String args[] )  
  
    { System.out.println( "Hello World" );  
  
    }  
}
```

A tool that comes with the JDK that produces html-based documentation from java source code. Within a Javadoc comment, various tags can appear which allow additional information to be processed. Each tag is marked by an @ symbol and should start on a new line.

Java Data types and Type Conversions

Changing a value from one data type to another type is known as data type conversion. Data type conversions are either widening or narrowing, it depends on the data capacities of the data types involved. There are different ways of, implicitly or explicitly, changing an entity of one data type into another data type. An important consideration with a type conversion is whether the result of the conversion is within the range of the destination data type.

If a value of narrower (lower size) data type converted to a value of a broader (higher size) data type without loss of information, is called Widening conversion. This is done implicitly by the JVM and also known as implicit casting. For example an integer data type is directly converted to a double.

```
int a = 100;  
double b = a;  
System.out.println(b);
```

- ☐ byte can be converted to short, int, long, float, or double
- ☐ Short can be converted to int, long, float, or double
- ☐ char can be converted to int, long, float, or double
- ☐ int can be converted to long, float, or double
- ☐ long can be converted to float or double
- ☐ float can be converted to double

Conclusion: Thus implemented various ways of accepting data through keyboard.

Experiment No.2

Aim: Java Operators, control statements and loops

- i. WAP to calculate the maximum and minimum of two/three numbers using ternary / conditional operators.
- ii. WAP to print the grade for an input test score: using the if-else ladder.

- | | |
|--------------|---------|
| • Percentage | • Grade |
| • 0-60 | • F |
| • 61-70 | • D |
| • 71-80 | • C |
| • 81-90 | • B |
| • 91-100 | • A |

Theory:

Java supports two types of castings – primitive data type casting and reference type casting. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules.

Java data type casting comes with 3 flavors.

1. Implicit casting
2. Explicit casting
3. Boolean casting.

1. Implicit casting (widening conversion)

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

Examples:

```
int x=10;           //occupies 4 bytes
```

```
double y=x;         // occupies 4 bytes
```

```
System.out.println(y); // Prints 10.0
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

2. Explicit casting (narrowing conversion)

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires **explicit casting**; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x=10.5;       //occupies 4 bytes
```

```
int y=x;             // occupies 4 bytes; raises compilation error.
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x=10.5;       //occupies 4 bytes
```

```
int y= (int) x;
```

The double **x** is explicitly converted to int **y**. The thumb rule is, on both sides, the same data type should exist.

Produce results of type boolean

Comparisons use 9 operators:

Equal == Not equal !=

Less than < Less than or equal <=

Greater than > Greater than or equal >=

Logical and && Logical or ||

Assignment is not the same as equality

= is not the same as ==

assignment places right hand side into left hand side

• Assignments are expressions:

int x, y;

x = y = 5; // Same as x = (y = 5); associate from R to L

T

three methods of processing a program:

- In sequence
- Conditional Branching
- Conditional Looping

Conditional Branch: Altering the flow of program execution by making a selection or choice.

Conditional Loop: Altering the flow of program execution by repeating statements.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Increment and Decrement Operator
- Conditional Operator
- Special Operator



The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

SR.NO	Operator and Description
1	& (bitwise and) Binary AND Operator copies a bit to the result if it exists in both operands. Example: (A & B) will give 12 which is 0000 1100
2	 (bitwise or) Binary OR Operator copies a bit if it exists in either operand. Example: (A B) will give 61 which is 0011 1101
3	^ (bitwise XOR) Binary XOR Operator copies the bit if it is set in one operand but not both. Example: (A ^ B) will give 49 which is 0011 0001
4	~ (bitwise compliment) Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. Example: (~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
5	<< (left shift) Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand Example: A << 2 will give 240 which is 1111 0000
6	>> (right shift) Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. Example: A >> 2 will give 15 which is 1111

7	<p>>>> (zero fill right shift) Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. Example: A >>>2 will give 15 which is 0000 1111</p>
---	---

Switch-case: menu driven calculator.

Program logic :

In addition to the nested if statement, Java provides a second method for choosing from many alternative actions. Compare the following code, each of which accomplishes the same task. The default case, at the end of the switch statement, acts similarly to the else at the end of the nested if. It will catch any value that doesn't have an exact match in the cases. Although it is not necessary to include the default case (just as it is not necessary to include the else), it is good programming practise to account for any unexpected values.

One of the significant limitations of the switch statement is the expression that can be used to control it. With an if statement, anything can be compared to produce an true or false result (e.g., primitive data such as int or float, more complex data such as Strings, or even objects created by the user).

In a switch statement, however, the expression controlling the switch must have an integer representation, which limits it to the following data types: byte, short, int, long, or char.

Example :

```

switch (score)
{
case 1:
grade = 'A';
break;
case 2:
grade = 'B';
break;
case 3:
grade = 'C';
break;
default:
System.out.println("Invalid score – grade of ? assigned");
grade = '?';
break;
}

```

WAP to generate the prime numbers between1-100.

Program logic :

With this example we are going to demonstrate how to generate prime numbers with a simple for loop. A prime number is a number that has no positive divisors other than 1 and itself. In short, to generate a prime number using a for loop you should:

- Create a for statement with an int i variable from 1 to a max int number, and step equal to 1.
- For each one of the numbers in the loop create a boolean isPrimeNumber equal to true and create another loop where the number is divided to other numbers from 2 up to the number, and if the result is zero, then the boolean isPrimeNumber is set to false.

WAP to find whether the given number is Armstrong or not.

Program logic :

Armstrong number c program: c programming code to check whether a number is armstrong or not. A number is armstrong if the sum of cubes of individual digits of a number is equal to the number itself. For example 371 is an armstrong number as $3^3 + 7^3 + 1^3 = 371$. Some other armstrong numbers are: 0, 1, 153, 370, 407.

WAP to display the following pattern

```

A B C D C B A
A B C B A
A B A
A

```



Program logic :

These program prints various different patterns of alphabets. Most of these c programs involve usage of nested loops and space. A pattern of numbers, star or characters is a way of arranging these in some logical manner or they may form a sequence. Some of these patterns are triangles which have special importance in mathematics. Some patterns are symmetrical while other are not.

We have to show five rows above, in the program you will be asked to enter the numbers of rows you want to print in the pyramid.

Conclusion: Thus we have implemented basic java programs and studied basic syntax of Java programming language.

Experiment No.3

Aim: To understand the working of a class.

- a. WAP to design a class to represent bank account. It should include the following members; Name of depositor, Account Number, Type of Account and balance amount in the account. It should also has methods to
 - v. To assign initial values (constructor).
 - vi. To deposit an amount.
 - vii. To withdraw an amount after checking balance.
 - viii. To display the name and balance.(Use constructor overloading for creating different account)

Theory:

Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object. Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor

Rules for creating java constructor:

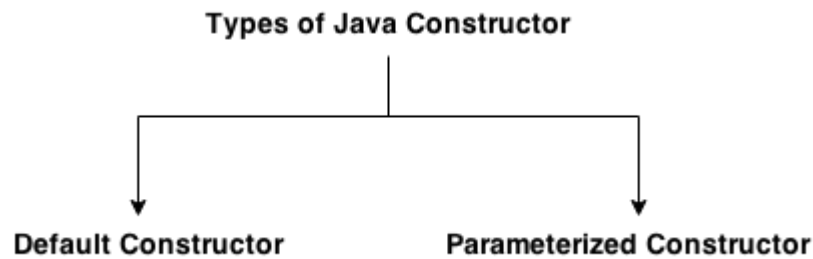
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



1. Java Default Constructor: A constructor that have no parameter is known as default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type

Syntax of default constructor:

```
<class_name>(){ }
```

2. Java parameterized constructor: A constructor that have parameters is known as parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.

Program logic :

Step 1: Create an object b of the class bank account.

Step 2: Read the number of records.

Step 3: call the init member function of the class bank account through the object created.

Step 4: Assign 500 to the balance in the records.

Step 5: Display a menu with the following options create deposit withdraw display.

Step 6: Read the choice.

Step 7: If the choice is create then call the create member function using the object.

Step 8: prompt and read the user's name, account number and account type for all the n records.

Step 9: If the choice is deposit call the deposit member function using the function.

Step 10: Read the account number.

Step 11: Check whether this account number is already exiting in the list. If true

Step 12: Read the amount to be deposited.

Step 13: Update the balance by adding the amount deposited to the balance.

Step 14: Display the balance.

Step 15: If the choice is withdraw, call the withdraw member function. Using object

Step 16: Read the account number.

Step 17: If the account number existing in the list then.

Step 18: Read the amount to be withdrawn.

Step 19: Retrieve the amount from balance. 5

Step 20: If the balance is less than 500 then

Step 21: Calculate balance + withdraw amount-500.

Step 22: Assign 500 to the balance.

Step 23: Display the withdrawn amount.

Step 24: If the choice is display, invoke the display member function using the Object.

Step 25: Read the account number.

Step 26: If this account number is existing in the list then.

Step 27: Display the details such as name, account number, account type and balance in the given account number.

b) WAP to arrange the names of students in descending order of their total marks, input data consists of students details such as names, ID.no, marks of maths, physics, chemistry.(Use array of objects).

Program logic :

Elements of an array can be of any type, and when applied to objects, this can prove very practical and useful. For example, consider collecting data on a group of children. We might have multiple arrays to store their names, ages, heights, and genders.

```
String[] name = new String[MAX_CHILDREN];  
int[] age = new int[MAX_CHILDREN];
```

```
double[] height = new double[MAX_CHILDREN];  
char[] gender = new char[MAX_CHILDREN];
```

Arrays of the same length, carrying data on different aspects of a problem, are sometimes called parallel arrays. Rather than four parallel arrays for our study, we could use a single array of objects.

There are a number of advantages to using an array of objects rather than multiple arrays of data.

1. A method to process data about one object can be given a single parameter rather than many parameters.
2. If the program is later modified to include different kinds of data, we only need to change the fields in the object class. The parameter lists of any methods remain unchanged.
3. By making the class fields private, we can ensure the data will be handled appropriately through the use of accessor and mutator methods.

Conclusion: Thus we have implemented objected oriented concept i.e class and object in java.

Experiment No.4

Aim: WAP on method overloading to calculate distance between two points.

b. Program on constructor and constructor overloading

Theory: In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java implements polymorphism.

If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Constructor Overloading in Java:

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Method Overloading in Java:

If a class have multiple methods by same name but different parameters, it is known as Method Overloading. If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b (int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading: Method overloading increases the readability of the program.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java , Method Overloading is not possible by changing the return type of the method.

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for one integer parameter.
    void test(int a) {
        System.out.println("a: " + a);
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // overload test for a double parameter
    double test(double a) {
```



```
System.out.println("double a: " + a);  
return a*a;  
}  
}
```

Conclusion: Thus we have implemented programs on concept of constructor and method overloading.

Experiment No.5

Aim: To illustrate package

a) Create a package (say) ABC, WAP operation.java in ABC (add (), subtract t(), multiply (), divide ()). WAP Imppackage.java that imports the package ABC.

Theory:

A **package** is a group of similar types of classes, interfaces and sub-packages. Package can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc. Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Package

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.

```
package mypack;  
public class Simple  
{  
    public static void main(String args[]){  
        System.out.println("Welcome to package");  
    }  
}
```

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

Using package name.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

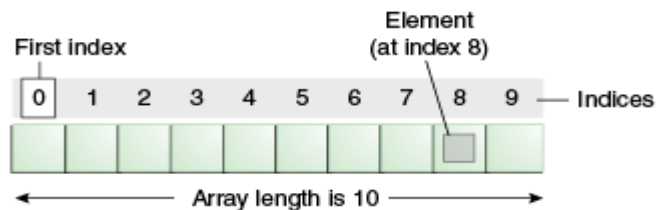
Experiment No.6

Aim: Write a program to perform Insertion Sort using one dimensional array. Write a program to perform matrix multiplication.

Theory:

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object that contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array. Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array
- Declaring Array Variables:

The syntax for declaring an array variable:

```
datatype arrayRefVar[];
```

example of this syntax:

```
double myList[];
```

- Creating Arrays:

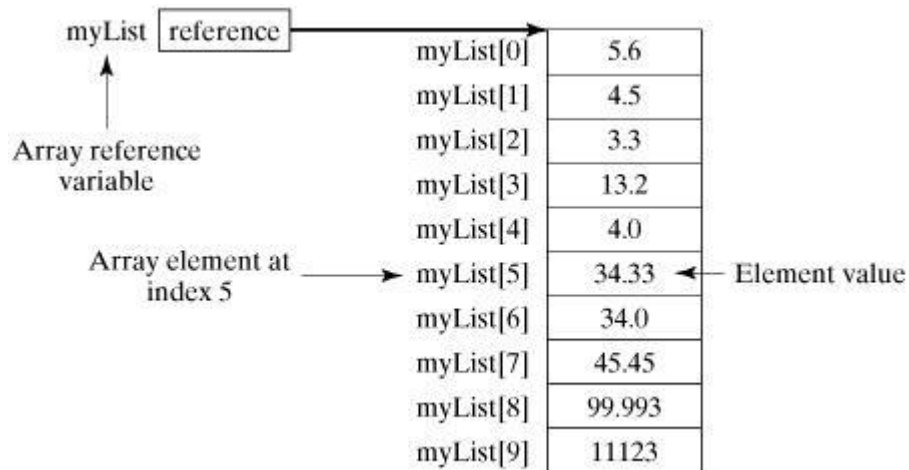
```
dataType arrayRefVar[] = new dataType [arraySize];
```

Example:

```
double myList[] = new double { 10};
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9





Insertion Sort: It is a simple Sorting algorithm which sorts the array by shifting elements one by one.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
    key = a[i];
    j = i-1;
    while(j>=0 && key < a[j])
    {
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = key;
}
```

Multidimensional array in java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

datatype arrayRefVar[][];

Example to instantiate Multidimensional Array in java

```
int arr[][]=new int[3][3];
```

Example to initialize Multidimensional Array in java

```
arr[0][0]=1;  
arr[0][1]=2;  
arr[0][2]=3;  
arr[1][0]=4;  
arr[1][1]=5;  
arr[1][2]=6;  
arr[2][0]=7;  
arr[2][1]=8;  
arr[2][2]=9;
```

Addition of Diagonal Elements:

```
for(i=0;i<m;i++){  
    for(j=0;j<n;j++){  
        if(i==j)  
            sum=sum+a[i][j];  
    }  
}
```

Row wise Addition:

```
int rowsum=0; //row-wise sum  
  
for (int i=0;i<m;i++)  
{  
    for (int j=0;j<n;j++)  
    {  
        rowsum+=arr[i][j];  
    }  
    System.out.println(rowsum);  
    rowsum=0;  
}
```

Conclusion : Thus we have implemented array in java.

Experiment No.7

Aim: a. WAP to count number of occurrences of given character using string class.

b. WAP to reverse a word using string class.

Theory:

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc. Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable. Once string object is created its data or state can't be changed but a new string object is created.

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	<u>char charAt(int index)</u>	returns char value for the particular index
2	<u>int length()</u>	returns string length
3	<u>String substring(int beginIndex)</u>	returns substring for given begin index
4	<u>String substring(int beginIndex, int endIndex)</u>	returns substring for given begin index and end index

5	<u>boolean equals(Object another)</u>	checks the equality of string with object
6	<u>boolean isEmpty()</u>	checks if string is empty
7	<u>String concat(String str)</u>	concatinates specified string
8	<u>String replace(char old, char new)</u>	replaces all occurrences of specified char value
9	<u>String replace(CharSequence old, CharSequence new)</u>	replaces all occurrences of specified CharSequence
10	<u>String trim()</u>	returns trimmed string omitting leading and trailing spaces
11	<u>String intern()</u>	returns interned string
12	<u>int indexOf(int ch)</u>	returns specified char value index
13	<u>int indexOf(int ch, int fromIndex)</u>	returns specified char value index starting with given index
14	<u>int indexOf(String substring)</u>	returns specified substring index
15	<u>int indexOf(String substring, int fromIndex)</u>	returns specified substring index starting with given index
16	<u>String toLowerCase()</u>	returns string in lowercase.
17	<u>String toUpperCase()</u>	returns string in uppercase.
18	<u>String trim()</u>	removes beginning and ending spaces of this string.

Conclusion: Thus we have implemented programs on string in java.

Experiment No.8

Aim: a. WAP to reverse a complete sentence using string buffer class.

b. WAP to use methods of vector class.

Theory:

StringBuffer is a peer class of **String** that provides much of the functionality of strings. **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writeable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. Java uses both classes heavily, but many programmers deal only with **String** and let Java manipulate **StringBuffers** behind the scenes by using the overloaded + operator.

StringBuffer defines these three constructors:

```
StringBuffer()  
StringBuffer(int size)  
StringBuffer(String str)
```

Let's see the important methods of String class.

charAt() and **setCharAt()**

The value of a single character can be obtained from a **StringBuffer** via the **charAt()** method. You can set the value of a character within a **StringBuffer** using **setCharAt()**. Their general forms are shown here:

```
char charAt(int where)  
void setCharAt(int where, char ch)
```

getChars()

To copy a substring of a **StringBuffer** into an array, use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ],  
int targetStart)
```

append()

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object.

```
StringBuffer append(String str)  
StringBuffer append(int num)  
StringBuffer append(Object obj)
```

String.valueOf() is called for each parameter to obtain its string representation. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append()**.

`insert()`

The **insert()** method inserts one string into another.

```
StringBuffer insert(int index, String str)  
StringBuffer insert(int index, char ch)  
StringBuffer insert(int index, Object obj)
```

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

`reverse()`

You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:

```
StringBuffer reverse( )
```

This method returns the reversed object on which it was called.

`delete()` and `deleteCharAt()`

Java 2 added to **StringBuffer** the ability to delete characters using the methods **delete()** and **deleteCharAt()**. These methods are shown here:

```
StringBuffer delete(int startIndex, int endIndex)  
StringBuffer deleteCharAt(int loc)
```

The **delete()** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*-1. The resulting **StringBuffer** object is returned.

`replace()`

It replaces one set of characters with another set inside a **StringBuffer** object. Its signature is shown here:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

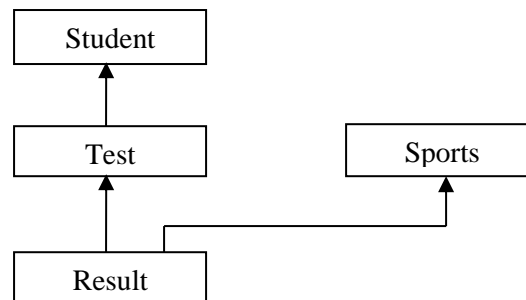
The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*-1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

Conclusion: Thus we have implemented programs on string buffer and vector in java.

Experiment No. 9

I.Aim: Inheritance-Extending Classes, interfaces and method overriding

WAP to implement three classes namely Student, Test and Result. Student class has member as rollno, Test class has members as sem1_marks and sem2_marks and Result class has member as total. Create an interface named sports that has a member score. Derive Test class from Student and Result class has multiple inheritance from Test and Sports. Total is formula based on sem1_marks, sem2_mark and score.



Theory:

Interface : It defines a standard and public way of specifying the behavior of classes. All methods of an interface are abstract methods .Defines the signatures of a set of methods, without the body (implementation of the methods).

A concrete class must implement the interface (all the abstract methods of the Interface) . It allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.

```

public interface Relation {
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
  
```

```

}
```

Extending classes :A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

Example :

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**. In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as Method Overriding.

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

```
class Vehicle{
    void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
```

```
public static void main(String args[]){  
    Bike obj = new Bike();  
    obj.run();  
}  
}
```

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

`package pkg;`

Here, pkg is the name of the package. For example, the following statement creates a package called **MyPackage**.

`package MyPackage;`

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

`package pkg1[.pkg2[.pkg3]];`

e.g. `package java.awt.image;`

Conclusion: Thus we have implemented inheritance in java.

Experiment No.10

Aim: Write abstract class program to calculate area of circle, rectangle and triangle.

Theory:

abstract class: A specification of a collection of data and the operations that can be performed on it. Describes what a collection does, not how it does it.

Java's collection framework describes ADTs with interfaces:

Collection, Deque, List, Map, Queue, Set, SortedMap

An ADT can be implemented in multiple ways by classes:

ArrayList and LinkedList implement List

HashSet and TreeSet implement Set

LinkedList, ArrayDeque, etc. implement Queue

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Helene Martin");  
names.add(42); // compiler error
```

Conclusion : implemented abstract class program in java.

Experiment No.11

Aim: Program on Interface.

Theory:

Interface : It defines a standard and public way of specifying the behavior of classes. All methods of an interface are abstract methods. Defines the signatures of a set of methods, without the body (implementation of the methods).

A concrete class must implement the interface (all the abstract methods of the Interface). It allows classes, regardless of their locations in the class hierarchy, to implement common behaviors.

```
public interface Relation {  
    public boolean isGreater( Object a, Object b);  
    public boolean isLess( Object a, Object b);  
}
```

```
public boolean isEqual( Object a, Object b);
```

```
}
```

Extending classes :A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Excepting Object, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to Object.

Example :

```
public class Bicycle {
```

```
// the Bicycle class has three fields
```

```
public int cadence;  
public int gear;  
public int speed;
```

```
// the Bicycle class has one constructor
```

```
public Bicycle(int startCadence, int startSpeed, int startGear) {  
    gear = startGear;  
    cadence = startCadence;  
    speed = startSpeed;  
}
```

```
// the Bicycle class has four methods
```

```
public void setCadence(int newValue) {  
    cadence = newValue;  
}
```

```
public void setGear(int newValue) {  
    gear = newValue;  
}
```

```
public void applyBrake(int decrement) {  
    speed -= decrement;  
}
```

```
public void speedUp(int increment) {  
    speed += increment;  
}
```

```
}  
  
}
```

Conclusion : Thus we have implemented interface in java.

Experiment No.12

Aim: WAP for exception handling using try/catch,throw, and finally demonstrating different inbuilt exceptions (IO, Arithmetic...)

Theory:

An *exception* is an abnormal condition that arises in a code sequence at run time.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

try-catch block

This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors
```



```
}  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed before try block ends  
}
```

Here, *ExceptionType* is the type of exception that has occurred.

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause.

```
type method-name(parameter-list) throws exception-list  
{
```



```
// body of method  
}
```

finally When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

Conclusion: Thus we have implemented exceptions in java.

Experiment No. 13

Aim: Define an exception called “NoMatchException” that is thrown when a string is not equal to “India”. Write a program that uses this exception.

b. WAP to create negative number exception while calculating factorial.

Theory:

Creating Your Own Exception Subclasses

Department of Computer Engineering , SIESGST

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString() method that displays the value of the exception.

```
class MyException extends Exception {  
    private int detail;  
    MyException(int a) {  
        detail = a;  
    }  
    public String toString() {  
        return "&quot;MyException[&quot; + detail + &quot;]&quot;;  
    }  
}  
class ExceptionDemo {  
    static void compute(int a) throws MyException {  
        System.out.println("&quot;Called compute(&quot; + a + &quot;)&quot;;  
        if(a &gt; 10)  
            throw new MyException(a);  
    }  
}
```

```
System.out.println(&quot;Normal exit&quot;);  
}  
public static void main(String args[]) {  
    try {  
        compute(1);  
        compute(20);  
    } catch (MyException e) {  
        System.out.println(&quot;Caught &quot; + e);  
    }  
}  
}
```

Conclusion : Thus implemented user defined exception in java.

Experiment No. 14

Aim: WAP to print 1A2B3C4D5E6F7G8H9I using two child threads.

Theory:

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more

parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code.

This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run ()**, which is declared like this:

```
public void run( )
```

Inside **run()**, you will define the code that constitutes the new thread.

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread. Here is the preceding program rewritten to

```
class NewThread extends Thread

// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
// Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }
// This is the entry point for the second thread.

    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }

        System.out.println("Exiting child thread.");
    }
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super()** inside **NewThread**. This invokes the following form of the **Thread** constructor:

```
public Thread(String threadName)
```

Here, *threadName* specifies the name of the thread.

Thread Synchronization

To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*.

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

This prevents other threads from entering **call()** while another thread is using it.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive( )
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join()**, shown here:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join()** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive()** method.

```
class DemoJoin {  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
        NewThread ob3 = new NewThread("Three");  
        System.out.println("Thread One is alive: "  
        + ob1.t.isAlive());  
        System.out.println("Thread Two is alive: "  
        + ob2.t.isAlive());  
        System.out.println("Thread Three is alive: "  
        + ob3.t.isAlive());  
        // wait for threads to finish  
        try {  
            System.out.println("Waiting for threads to finish.");  
            ob1.t.join();  
  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Main thread Interrupted");  
        }  
    }  
}
```

after the calls to **join()** return, the threads have stopped executing.

Conclusion: Thus implemented thread in java.

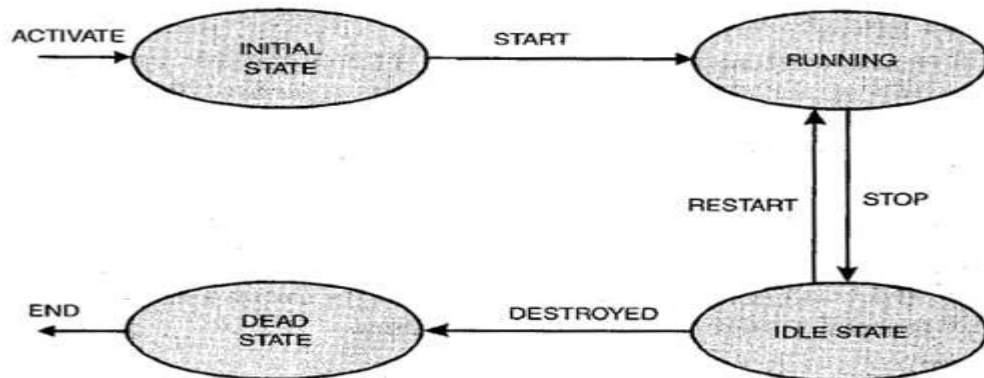
Experiment No. 15

Aim: a. WAP to display smiling face using applet.

Theory:

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Java applet inherits features from the class *Applet*. Thus, whenever an *applet* is created, it undergoes a series of changes from initialization to destruction. Various stages of an *applet* life cycle are depicted in the figure below:



Life Cycle of an Applet

Initial State

When a new *applet* is born or created, it is activated by calling *init()* method. At this stage, new objects to the *applet* are created, initial values are set, images are loaded and the colors of the images are set. An *applet* is initialized only once in its lifetime. It's general form is:

```
public void init( )
//Action to be performed
}
```

Running State

An *applet* achieves the running state when the system calls the *start()* method. This occurs as soon as the *applet* is initialized. An *applet* may also start when it is in idle state. At that time, the *start()* method is overridden. It's general form is:

```
public void start( )
{
//Action to be performed
}
```

Idle State

An *applet* comes in idle state when its execution has been stopped either *implicitly* or *explicitly*. An *applet* is *implicitly* stopped when we leave the page containing the currently running applet. An *applet* is *explicitly* stopped when we call *stop()* method to stop its execution. It's general form is:

```
public void stop()
{
//Action to be performed
}
```

Dead State

An *applet* is in dead state when it has been removed from the memory. This can be done by using *destroy()* method. It's general form is:

```
public void destroy( )  
{  
    //Action to be performed  
}
```

Apart from the above stages, Java applet also possess *paint()* method. This method helps in drawing, writing and creating colored backgrounds of the applet. It takes an argument of the graphics class. To use The graphics, it imports the package java.awt.Graphics

```
/*  
    Basic Java Applet Example  
    This Java example shows how to create a basic applet using Java Applet class.  
*/  
  
import java.applet.Applet;  
import java.awt.Graphics;  
  
/*  
    <applet code = "BasicAppletExample" width = 200 height = 200>  
    </applet>  
*/  
public class BasicAppletExample extends Applet{  
  
    public void paint(Graphics g){  
        //write text using drawString method of Graphics class  
        g.drawString("This is my First Applet",20,100);  
    }  
}
```

Compile the Source File

Compile the source file using the Java compiler.

Run the Applet

Run the applet using appletviewer. e.g. appletviewer BasicAppletExample.java.

Conclusion: thus implemented applet in java.

Experiment No. 16

Aim: GUI Application with Event Handling

Theory: Abstract window toolkit is package which consist of methods to create GUI in java.

GUI consist of button, label, scroll bar etc. Event handling consist of methods to handle events such as press button, enter text into text field etc.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MyFrame extends JFrame {
    private JButton btnExit = new JButton("Exit");
    private JButton btnAdd = new JButton("Add");
    private JTextField txtA = new JTextField();
    private JTextField txtB = new JTextField();
    private JTextField txtC = new JTextField();
    private JLabel lblA = new JLabel("A :");
    private JLabel lblB = new JLabel("B :");
    private JLabel lblC = new JLabel("C :");
    public MyFrame(){
        setTitle("GUI SAMPLE");
        setSize(400,200);
        setLocation(new Point(300,200));
        setLayout(null);
        setResizable(false);
        initComponents();
        initEvent();
    }
    private void initComponents(){
        btnExit.setBounds(300,130, 80,25);
        btnAdd.setBounds(300,100, 80,25);
        txtA.setBounds(100,10,100,20);
        txtB.setBounds(100,35,100,20);
        txtC.setBounds(100,65,100,20);
        lblA.setBounds(20,10,100,20);
        lblB.setBounds(20,35,100,20);
        lblC.setBounds(20,65,100,20);
        add(btnExit);
        add(btnAdd);
        add(lblA);
        add(lblB);
        add(lblC);
        add(txtA);
        add(txtB);
    }
}
```




```

        add(txtC);
    }

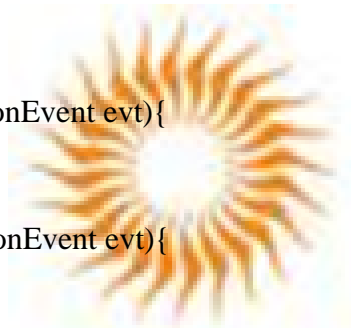
    private void initEvent(){
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(1);
            }
        });
        btnExit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btnExitClick(e);
            }
        });
        btnAdd.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                btnAddClick(e);
            }
        });
    }

    private void btnExitClick(ActionEvent evt){
        System.exit(0);
    }

    private void btnAddClick(ActionEvent evt){
        Integer x,y,z;
        try{
            x = Integer.parseInt(txtA.getText());
            y = Integer.parseInt(txtB.getText());
            z = x + y;
            txtC.setText(z.toString());
        }catch(Exception e){
            System.out.println(e);
            JOptionPane.showMessageDialog(null,
                e.toString(),
                "Error",
                JOptionPane.ERROR_MESSAGE);
        }
    }
}

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

```



```
MyFrame f = new MyFrame();  
f.setVisible(true);  
}  
}
```

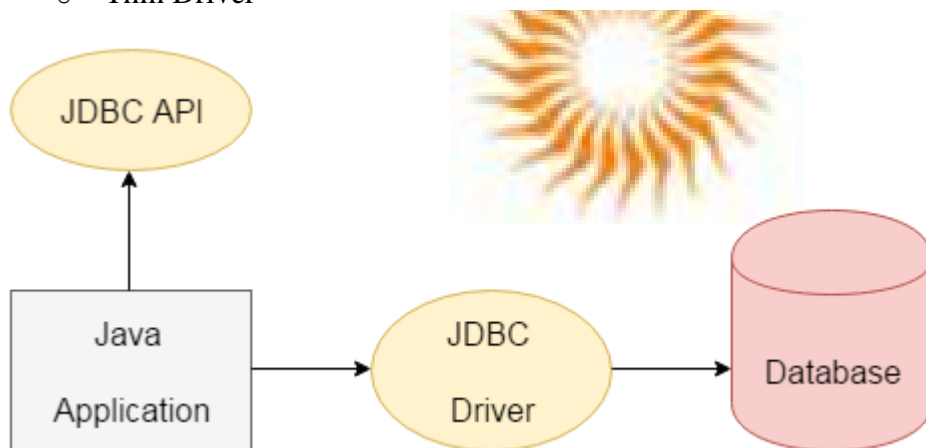
Experiment No. 17

Aim: Case study JDBC Connectivity

Theory :

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver



There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

JDBC Folder is inside\\dataserver->Commom->Java->jdbc

and refer sent mail of jdbc connectivity.

Take printout of Test java file and snapshot of Microsoft access file.

Conclusion : thus studied connectivity of excel with java.



