MANIPAL INSTITUTE OF TECHNOLOGY

Manipal – 576 104

DEPARTMENT OF INFORMATION TECHNOLOGY

Certificate

This is to certify that Ms./Mr. ……...…… Reg. No. …..…… Section: …… Roll No: ……... has satisfactorily completed the lab exercises prescribed for Object Oriented Programming Lab [ICT-2142] of Third Semester B. Tech. [IT/CCE] Degree at MIT, Manipal, in the academic year 2024-2025.

Date: …….....

Signature of the faculty

Signature
Head of the Department

INDEX

Course Objectives

- Get practical experience of Arrays, Strings and Control structures
- Get practical experience to Create classes and objects for a given problem
- Develop Reuseable, Reliable and Polymorphic code
- Develop Graphical User Interface & Event driven programs for a given application using JavaFX

Course Outcomes

At the end of this course, students will be able to

1. Explain the Object-oriented paradigm of software development.
2. Achieve reusability using Inheritance, Packages and Generics.
3. Appreciate the use of exception handling, achieve concurrency through multithreading and implement small Java applications using JavaFX.

Evaluation plan

Laboratory – Conduction and Evaluation Pattern

Marks Distribution

Internal Evaluation (60 Marks)	Record – 16 Marks	Record Evaluation-1 = 8 Marks
		Record Evaluation-2 = 8 Marks
	Mid-Term - 20 Marks	Write-up = 5 Marks
		Execution = 15 Marks
	Quiz - 14 Marks	Quiz or Project work = 14 Marks
	Program Check- 10 Marks	Program checking before Test-1 = 5 Marks
		Program checking before Test-2 = 5 Marks
Final Exam (40 Marks)	Exam - 40 Marks	Write-up = 10 Marks
		Program Execution = 30 Marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance.
5. Adhere to the rules and maintain the decorum.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
    - Comments should be used to give a statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
    - Statements within the program should be properly indented.
    - Use meaningful names for variables and functions.
    - Make use of constants and type definitions wherever needed.
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
    - Solved exercise - to be used as a reference to understand the concept
    - Lab exercises - to be completed during lab hours

- Additional Exercises - to be completed outside the lab or in the lab to enhance the skill

- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class in case of genuine reason (medical certificate approved by HOD) with the permission of the faculty concerned.
- Questions for lab tests and examinations are not necessarily limited to the questions in the manual but may involve some variations and / or combinations of the questions.
- A sample note preparation is given as a model for observation.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

LAB NO: 1 **Date:**

JAVA FEATURES & SIMPLE PROGRAMS USING CONTROL STRUCTURES

Objectives:

1. To know the features of Java
2. Understand the Java Development Kit (JDK)
3. To write, compile, and run a Java program
4. To know the execution steps using Netbeans/Eclipse IDE & Command Prompt
5. Write Java programs using control structures

1.1 Features of Java Language

Java is truly object oriented programming language mainly used for Internet applications. It can also be used for standalone application development. Following are the main features of Java:

Simple: Java was designed to be easy for the professional programmer to learn and use effectively. Design goal was to make it much easier to write bug free code. The most important part of helping programmers write bug-free code is keeping the language simple. Java has the bare bones functionality needed to implement its rich feature set. It does not add unnecessary features.

Object Oriented: Java is a true object oriented language. Almost everything in Java is an object. The program code and data are placed within classes. Java comes with an extensive set of classes and these classes are arranged in packages.

Robust: Memory management can be a difficult and tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and de-allocation. (de-allocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can and should be managed by the program.

Multithreaded: Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows to write programs that do many things simultaneously. The java run-time system comes with a sophisticated solution for multi-process synchronization that enables users to construct smoothly running interactive systems.

Compiled and Interpreted: Java is a two stage system because it combines two approaches namely, compiled and interpreted. First Java compiler translates source code into what is known as bytecode instructions. Bytecodes are not machine instructions and therefore in the second stage, Java interpreter generates machine code that can be directly executed by the machine that is running the Java program. Thus the Java is both a compiled and interpreted language.

Platform Independent and Portable: Java programs can be easily moved from one computer system to another, anywhere and anytime. Changes in upgrades in operating systems, processors and system resources will not force any changes in Java programs. Java ensures portability in two ways. First, Java compiler generates bytecode instructions that can be implemented on any machine. Secondly, the sizes of the data types are machine independent.

Dynamic: Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

Security: JVM is an interpreter which is installed in each client machine that is updated with latest **security** updates by internet . When this byte codes are executed , the JVM can take care of the **security**. So, **java** is said to be more **secure** than other programming languages.

1.2 Understand the Java Development Kit (JDK)

The JDK comes with a collection of tools that are used for developing and running Java programs which include:
_ appletviewer (for viewing Java applets)
_ javac (Java compiler)
_ java (Java interpreter)
_ javap (Java disassembler)
_ javah (for C header files)
_ javadoc (for creating HTML documents)
_ jdb (Java debugger)

Following table 1.1 lists these tools and their descriptions:

Table 1.1 : The JDK tools

Tool	Description
Javac	Java compiler, which translates Java source code to bytecode files that the interpreter can understand

Java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javadoc	Creates HTML format documentation from Java source code files.
Javah	Produces header files for use with native methods.
Javap	Java disassembler, which enables us to convert bytecode files into a program
Jdb	Java debugger, which finds errors in programs.
applet-viewer	Enables us to run Java applets (without using a Java compatible browser)

The way these tools are applied to build and run application programs are shown below:

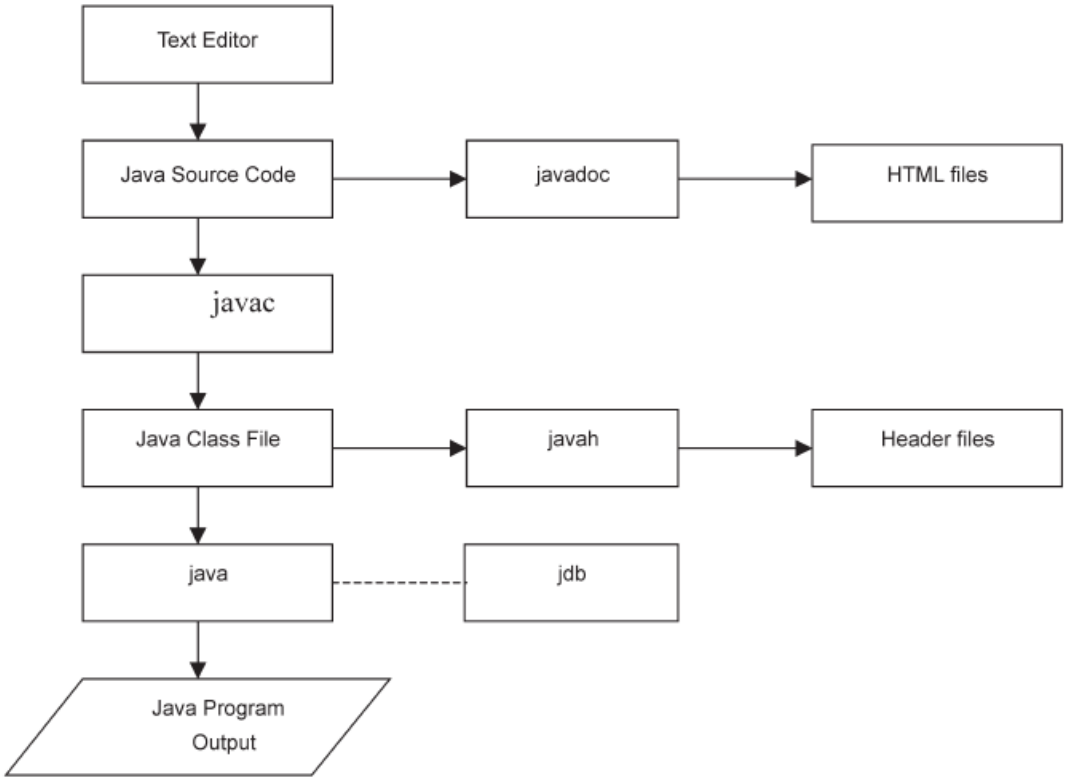


Fig 1.1. Process of building and running Java application programs.

To create a Java program it needs to create a source code file using a text editor. The source code is compiled using javac and executed using Java interpreter. The Java debugger jdb is used to find errors. A compiled Java program can be converted into a source code using Java disassembler javap.

Java Virtual Machine (JVM)

All language compilers translate source code into machine code for a specific computer. Java compiler produces an intermediate code known as bytecode for a machine that does not exist. This machine is called as Java Virtual Machine and it exists only inside the computer memory.

Following figure shows the process of compiling a Java program into bytecode which is also called as Java Virtual Machine code.

Fig. 1.2 Process of Compilation

The Java Virtual Machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediate between the virtual machine and the real machine as shown in following Fig 1.3. The interpreter is different for different machines.

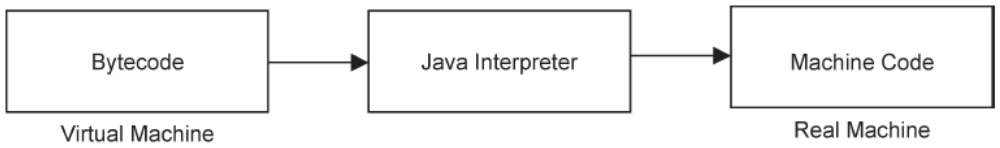


Fig 1.3. Process of converting bytecode into machine code

1.3 Write, compile and run a Java program

First Sample Program: Program to display the message "Hello World"

Aim: To write a program in Java that displays a message "Hello World"

```
/*
        This is a simple a program.
        Call this file "HelloWorld.java".
*/

        class HelloWorld{

        // program begins with a call to main()
                public static void main(String args[]){
                        System.out.println("Hello World");
                }
        }
```

Sample output:
Hello World

BUILD SUCCESSFUL (total time: 7 seconds)

Entering the Program

The first thing about Java is that the name given to a source file is very important. For the example given above, the name of the source file should be **HelloWorld.java**. In Java, a source file is officially called a *compilation unit.* It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the **.java** file name extension.

The name of the class defined by the program is also **HelloWorld**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. It should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the program
To compile the **HelloWorld** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown below:

C:\\>javac HelloWorld.java

The **javac** compiler creates a file called **HelloWorld.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of program that contains instructions the Java interpreter will execute. Thus, the output of **javac** is not code that can be directly executed.

To actually run the program, the Java interpreter is used which is , called **java**. To do so, pass the class name **HelloWorld** as a command-line argument, as shown below:

C:\\>java HelloWorld
When the program is run, the following output is displayed:
Hello World

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give the Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When the Java interpreter executes as just shown, by actually specifying the name of the class that the interpreter will execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

A Closer Look at the First Sample Program

The program begins with the following lines:

```
/*
        This is a simple Java program.
        Call this file "HelloWorld.java".
*/
```

This is a *comment.* Like most other programming languages, Java allows to enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds that the source file should be called **HelloWorld.java**. In real applications, comments generally explain how some part of the program works or what a specific feature does.

Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment.* This type of comment must begin with **/*** and end with ***/**. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown below:

class HelloWorld{

This line uses the keyword **class** to declare that a new class is being defined. **HelloWorld** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). The use of the curly braces in Java is identical to the way they are used in C++.

The next line in the program is the *single-line comment,* shown here:

// program begins with a call to main().
This is the second type of comment supported by Java. A *single-line comment* begins with a **//** and ends at the end of the line. As a general rule, programmers use multi line comments for longer remarks and single-line comments for brief, line-by-line descriptions.

The next line of code is shown here:

public static void main(String args[]) {
This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. (This is just like C/C++.) Since most of the programs will use this line of code, let's take a brief look at each part.

The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value.

As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, if Main is typed **Main** instead of **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method. Any information which is needed to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters*. If there are no parameters required for a given method, it still need to include the empty parentheses. In **main( )**, there is only one parameter, **String args[ ]** that declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when

the program is executed. This program does not make use of this information, but other programs may use this to enter inputs through command line arguments. The last character on the line is the **{**. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace.

NOTE: **main( )** is simply a starting place for the interpreter. A complex program will have many classes, only one of which will need to have a **main( )** method to get things started. When it begin creating applets, Java programs that are embedded in web browsers, it won't use **main( )** at all, since the web browser uses a different means of starting the execution of applets.

The next line of code is shown here. Notice that it occurs inside **main( )**.

This line outputs the string "Hello World." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. As seen , **println( )** can be used to display other types of information, too. The line begins with **System.out**. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple, utility programs and for demonstration programs. Notice that the **println( )** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements.

The first **}** in the program ends **main( )**, and the last **}** ends the **HelloWorld** class definition.

Second Sample Program

Program to display the area of a rectangle (Hint: area=length x breadth)

Aim: To write a program in Java to find the area of a rectangle and verify the same with various inputs(length, breadth).

Program:

```
//RectangleArea.java
```

```
//program to find area of a rectangle

class RectangleArea {
   public static void main(String args[]){
        int length,breadth;
        length=Integer.parseInt(args[0]);  //command line arguments
        breadth=Integer.parseInt(args[1]); //convert string to integer
        int area=length *breadth;
        System.out.println("length of rectangle ="+ length);
        System.out.println("breadth of rectangle ="+ breadth);
        System.out.println("area of rectangle ="+ area);
}}
```

Sample input and output:

```
C:\\>javac RectangleArea.java
C:\\> java RectangleArea 10 8
        length of rectangle = 10
        breadth of rectangle = 8
        area of rectangle = 80;

C:\\> java RectangleArea 12 15
        length of rectangle = 12
        breadth of rectangle = 15
        area of rectangle =180;
```

NOTE: The **Integer** class provides **parseInt( )** method that returns the **int** equivalent of the numeric string with which it is called. (Similar methods and classes also exist for the other data types)

1.4 Execution steps using Eclipse

Eclipse is a sophisticated integrated development environment (IDE) that aims to help developers build any type of application. It allows us to quickly and easily develop desktop, mobile and web applications with Java, HTML5, PHP, C/C++ and more. Eclipse IDE is free, open source, and has a worldwide community of users and developers.

Following are step-by-step instructions to get started developing Java applications with NetBeans IDE. The basic steps described are as follows.

1. Create a new project
2. Set application as Java
3. Give name and location for the project
4. Compile and run a Java program

Setting Up the Project

1. To create an IDE project:

- Start Eclispe IDE.
- In the IDE, choose File > New Project, as shown below:

Fig 1.4. To create a new project in Eclispe IDE.

2. In the New Project wizard, expand the Java category and select “Java Application” as shown in the Fig 1.5 below. Then click Next.

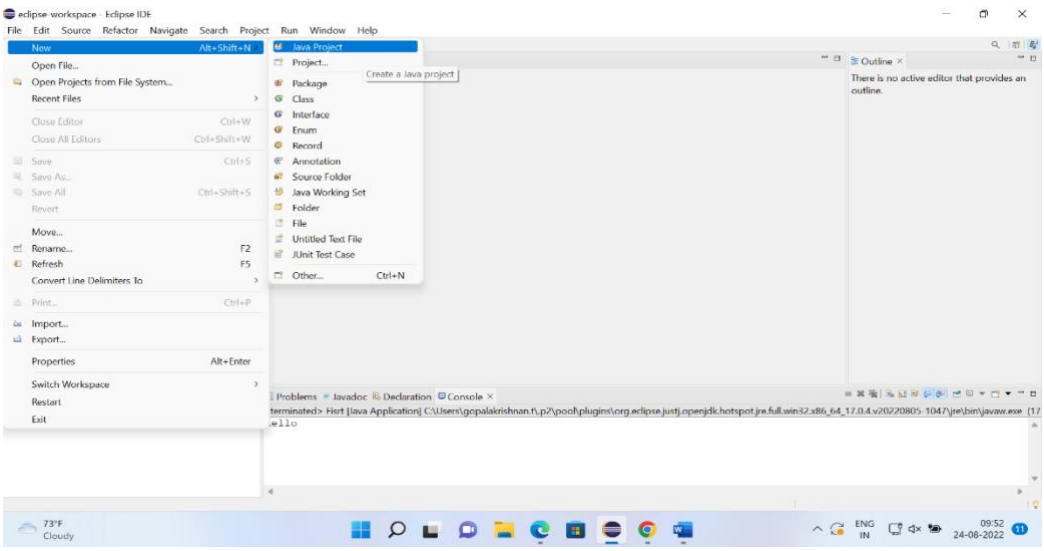


Fig 1.5. Selection of Java Application from the categories Jva.

3. “Name and Location” page of the wizard, do the following (as shown in the fig 1.6. below):
    - In the Project Name field, type HelloWorldApp.
    - Leave the Use Dedicated Folder for Storing Libraries checkbox unselected.
    - In the Create Main Class field, type helloworldapp.HelloWorldApp.

4. Click Finish.

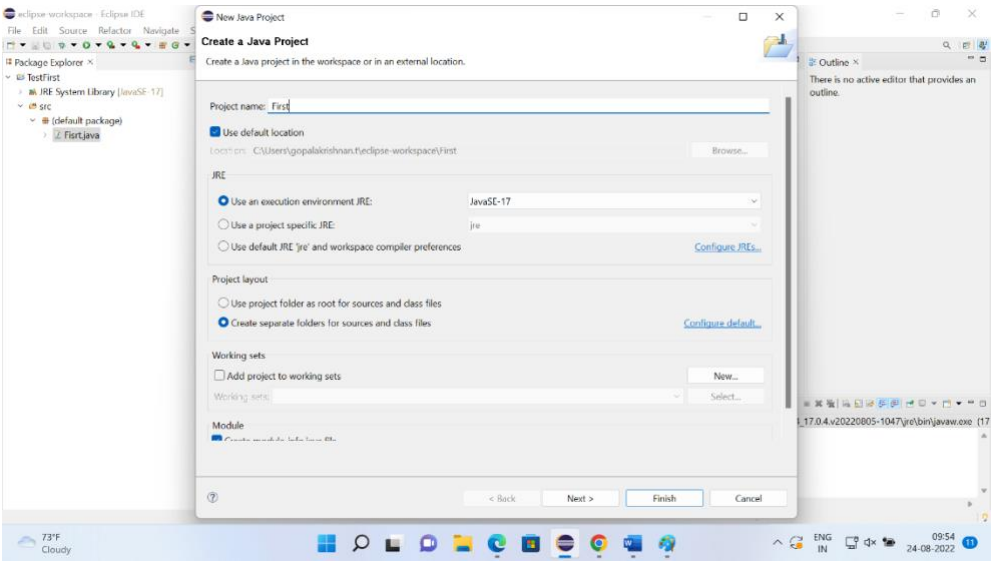


Fig1.6. Giving Name and Location to the newly created Project.

The project is created and opened in the IDE with the following components:

- The Projects window, which contains a tree view of the components of the project, including source files, libraries that your code depends on, and so on.
- The Source Editor window with a file called HelloWorldApp open.
- The Navigator window, can use to quickly navigate between elements within the selected class.

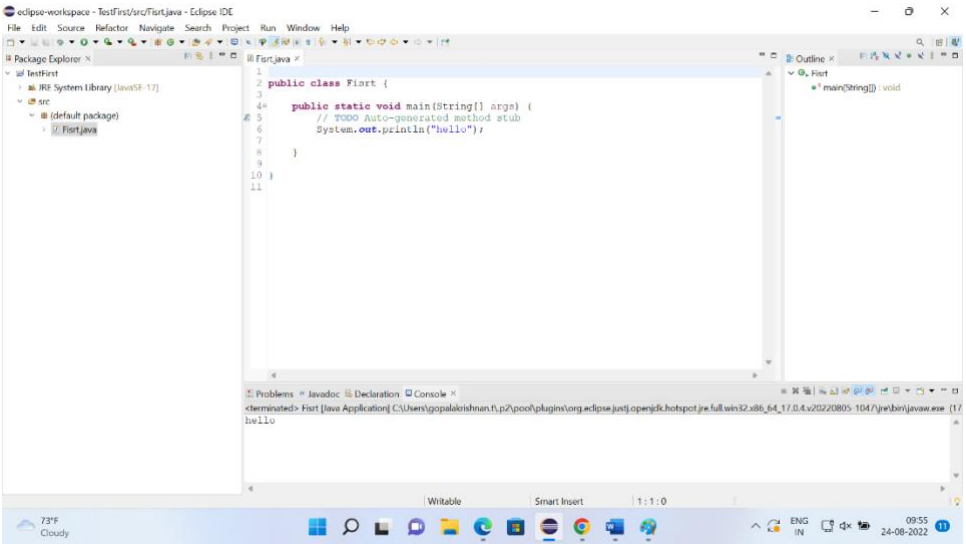


Fig 1.7. Project window,Source Editor Window,Navigator Window opens up after project creation.

Compiling and Running the Program

The IDE has Compile on Save feature. The user need not manually compile the project in order to run it in the IDE. When saved as a Java source file, the IDE automatically compiles it.

The Compile on Save feature can be turned off in the Project Properties window. Right-click your project, select Properties. In the Properties window, choose the Compiling tab. The Compile on Save checkbox is right at the top. Note that in the Project Properties window it can configure numerous settings for your project: project libraries, packaging, building, running, etc.

To run the program:

- Choose Run > Run Project.

For the above program output appears as follows in Fig 1.8

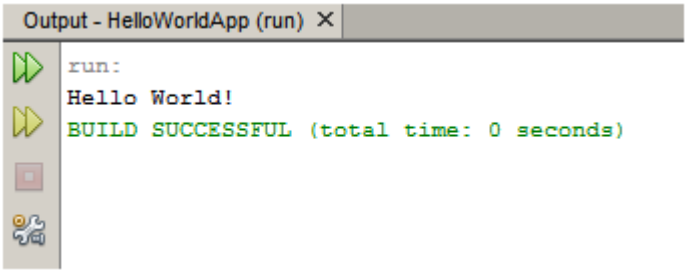


Fig 1.8. Output of the program

Lab exercises

1. Write a Java program to find area and circumference of a rectangle.
   (Hint: circumference = 2 (length + breadth) ; area= length x breadth).
2. Write a Java program to enter 10 numbers and display the number of positive,negative and zeros number.
3. Write a Java program to generate odd numbers from 1 to 100.

Additional exercises

1. Write a program to check whether a number is palindrome or not.
2. Write a Java programs to print factorial of a given no.
3. Write a Java program to print table of number entered by user .

LAB NO: 2 **Date:**

DATA TYPES, TYPE CONVERSION, OPERATORS

Objectives:

1. To learn the different data types in Java
2. To understand Java type conversion and casting
3. To be familiar with bit-wise, arithmetic, Boolean, logical and relational operators
4. To write simple Java programs to demonstrate the usage of taking input from keyboard, data types, type conversion, and operators

2.1 Java data types

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

i) Integers: This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	– 32,768 to 32,767
byte	8	– 128 to 127

ii) Floating-point numbers: This group includes **float** and **double**, which represent numbers with fractional precision.

Name	Width in Bits	Range
double	64	1.7e–308 to 1.7e+308
float	32	3.4e–038 to 3.4e+038

iii) Characters: This group includes **char**, which represents symbols in a character set, like letters and numbers. Java uses unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII ranges from 0 to 127

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[]) {
        char ch1, ch2;
        ch1 = 88; // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
        }
}
```

This program displays the following output:
ch1 and ch2: X Y

iv) Boolean: This group includes **boolean**, which is a special type for representing true/false values. This is the type returned by all relational operators, such as **a < b**. **Boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.

2.2 Java type conversion and casting

i) Automatic type conversion: When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with **char** or **Boolean**. Also, **char** and **Boolean** are not compatible with each other.

ii) Casting incompatible types: Although the automatic type conversions are helpful, they will not fulfill all needs. For example, if it wants to assign an **int** value to a **byte** variable, the conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion*, since explicitly making the value narrower so that it will fit into the target type. A *cast* is simply an explicit type conversion. It has this general form: (*target-type*) *value*

```
int a;
byte b;
// ...
b = (byte) a;
```

iii) Type promotion rules: First, all **byte** and **short** values are promoted to **int**. Then, if one operand is a **long operand**, the whole expression is promoted to **long**. If one operand is a **float** operand, the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.It is illustrated in the below fig.2.1

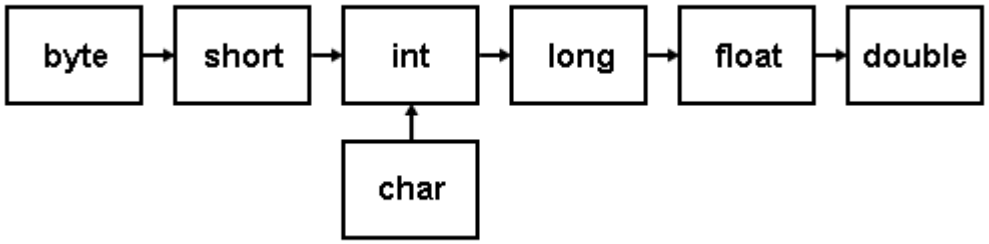


Fig 2.1. Type Promotion Rules

2.3 Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in most other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As seen from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If operator results in false when A is false, no matter what B is. If used the | | and && forms, rather than the | and & forms of these operators, java will not bother to evaluate the right-hand operand alone. This is very useful when the right-hand operand depends on the left one being true or false in order to function properly. For example, the following code fragment shows how it can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if ( denom != 0 && num / denom >10)
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code were written using the single & version of AND, both sides would have to be evaluated, causing a run-time exception when denom is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if ( c==1 & e++ < 100 ) d = 100;
```

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

2.4 Bit-wise operators

Java defines several *bitwise operators* which can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized as given below:

Operator	Description
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

2.5 Reading keyboard input

Java provides Scanner class to get input from the keyboard which is present in java.util package. Therefore this package should be imported to the program. First create an object of Scanner class and then use the methods of Scanner class.

Scanner a = new Scanner(System.in);

Here "Scanner" is the class name, "a" is the name of object, "new" keyword is used to allocate the memory and "System.in" is the input stream. Following methods of Scanner class are used in the program below :-

1) nextInt to input an integer
2) nextFloat to input a float
3) nextLine to input a string
4) nextDouble to input a double
5) next().charAt(0)

This program firstly asks the user to enter a string followed by an integer number and a float value. Immediately after entering each input, the value entered by the user will be printed on the screen.

```
import java.util.Scanner;
class GetInputFromUser{
  public static void main(String args[])  {
        int a;
        float b;
        String s; char c;
        Scanner in = new Scanner(System.in);
```

```
        System.out.println("Enter a string");
        s = in.nextLine();
        System.out.println("You entered string "+s);
        System.out.println("Enter an integer");
        a = in.nextInt();
        System.out.println("You entered integer "+a);
        System.out.println("Enter a float");
        b = in.nextFloat();
        System.out.println("You entered float "+b);
        c=in.next().charAt(0);
        System.out.println("You entered character "+c);

    }
}
```

Lab exercises

1. Write a Java program to find whether a given year is leap or not using boolean data type. [Hint: leap year has 366 days;]
   Algorithm:
   if (*year* is not exactly divisible by 4) **then** (it is a common year)
   else
   if (*year* is not exactly divisible by 100) **then** (it is a leap year)
   else
   if (*year* is not exactly divisible by 400) **then** (it is a common year)
   else (it is a leap year**)**

2. Write a Java program to read an int number, double number and a char from keyboard and perform the following conversions:- int to byte, char to int, double to byte, double to int

3. Write a Java program to multiply and divide a number by 2 using bitwise operator. [Hint: use left shift and right shift bitwise operators]

4. Write a Java program to execute the following statements. Observe and analyze the outputs.

   a.
   ```
   int x =10;
   double y = x;
   System.out.println(y);
   ```

   b.
   ```
   double x = 10.5;
   int y = x;
   System.out.println(y);
   ```

   c.
   ```
   double x=10.5;
   int y = (int) x
   System.out.println(y);
   ```

5. Create the equivalent of a four-function calculator. The program should request the user to enter a number, an operator, and another number. (Use floating point.) It should then carry out the specified arithmetic operation: adding, subtracting, multiplying, or dividing the two numbers. Use a switch statement to select the operation. Finally, display the result. When it finishes the calculation, the program should ask if the user wants to do another calculation. The response can be 'y' or 'n'. [Hint: use do-while loop]

   Example

Enter first number, operator, second number: 10 / 3

Answer = 3.333333

Do another (y/n)? n

Additional exercises

1. Write a Java program to find the result of the following expressions for various values of a & b:
   a. (a << 2) + (b >> 2)
   b. (b > 0)
   c. (a + b * 100) / 10
   d. a & b

2. Write a Java program to find largest and smallest among 3 numbers using ternary operator.

3. Write a Java program to execute the following statements. Observe and analyze the outputs

   a.
   ```
   boolean x =true;
   int y = x;
   ```

   b.
   ```
   boolean x =true;
   int y =(int)x;
   ```

LAB NO: 3 **Date:**

CONTROL STATEMENTS

Objectives:

1. To learn the syntax & usage of control statements in Java
2. To write simple Java programs to demonstrate the usage of Selection ,Iteration and Jump statements in Java

3.1 Java Selection Statement

(i) Simple if – else

```
if (condition) statement1;
else statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed, otherwise statement2 (if it exists) is executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero.

(ii) Nested if

A nested if is an if statement that is the target of another if or else. When nested ifs are used, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

(iii) If – else – if ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder.

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

```
// Demonstrate if-else-if statements.
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}}
```

Output:April is in the Spring.

(iv) Switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of the code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. The general form of a switch statement is given below:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN:

// statement sequence
break;
default:
// default statement sequence }
```

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

The switch statement works like this: The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of "jumping out" of the switch.

```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++) {
switch(i) {
case 0:
System.out.println("i is zero.");
break;

case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
```

```
case 3:

System.out.println("i is three.");
break;

default:
System.out.println("i is greater than 3.");
} // switch
}// for
}// main
}//SampleSwitch
```

Output:
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.

As can be seen, each time through the loop, the statements associated with the case constant that matches i are executed. All others are bypassed. After i is greater than 3, no case statements match, so the default statement is executed. The break statement is optional. If the break is omitted, execution will continue with the next case. It is sometimes desirable to have multiple cases without break statements between them.

3.2 Iteration Statement

(i) While

The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
// body of loop
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

```
// Demonstrate the while loop.
class While {
public static void main(String args[]) {
int n = 10;
while(n >5) {
System.out.println("tick " + n);

n—;
} // while
```

```
} // main
}//While class
```

Output:
tick 10
tick 9
tick 8
tick 7
tick 6

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.

(ii) do – while

If the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a while loop at least once, even if the conditional expression is false to begin with. In other words, there are times when to test the termination expression at the end of the loop rather than at the beginning. The do-while loop always executes its body at-least once, because its conditional expression is at the bottom of the loop. Its general form is:

```
do {
// body of loop
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n—;
} while(n >5); }}
```

Output:
tick 10
tick 9
tick 8
tick 7
tick 6

(iii) for

```
for(initialization; condition; iteration) {
```

```
// body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
// Demonstrate the for loop.
class ForTick {
public static void main(String args[]) {
int n;
for(n=10; n>5; n—)
System.out.println("tick " + n);}}
```

Output:
tick 10
tick 9
tick 8
tick 7
tick 6

(iv) The for-each loop introduced in Java5.
It is mainly used to traverse array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable.
Advantage of for-each loop:

- It makes the code more readable.
- It elimnates the possibility of programming errors.)

Syntax of for-each loop:
for(data_type variable : array | collection){}
Example of for-each loop for traversing the array elements:

```
class ForEachExample1{
 public static void main(String args[]){
  int arr[]={12,13,14,44};


  for(int i:arr){
    System.out.println(i);
  }
```

```
}
}
```

o/p:- Output:12
13
14
44

(v) nested loops

Java allows loops to be nested. That is, one loop may be inside another.

```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<5; i++) {
for(j=i; j<5; j++)
System.out.print(".");
System.out.println();
}
}
}
```

Output:

.....
.....
.....
.....
.....

3.3 Jump Statement

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of the program

i) Break: In Java, the break statement has three uses. First, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

ii).Continue: Sometimes it is useful to force an early iteration of a loop. That is, to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

iii).Return: The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

Lab exercises

1. Write a program to compute whether a no . is an Armstrong number or not.Use any of the iteration statements.
2. Write a Java program to display the numbers in the following format
    a. using nested for loop.
    b. using for-each loop.

    1
    2 2
    3 3 3
    4 4 4 4
    5 5 5 5 5
3. Write a Java program to generate prime numbers between n and m.(Hint: A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Eg: 2, 3, 5,7,11 etc.)
4. Write a java program to search for a value in a 1 dimensional array using for each loop construct. Assume that the array is initialized at the time of declaration and user enters the value to be searched on request.(1 mark)
    Input: a[]={1,2,3,1,2,1,5,6,7} searchValue= 1
    Expected Output : The value is found at locations: a[0] ,a[3],a[5] .

Additional exercises

1. Write a program to print all combinations of four digit number. A four digit number is generated using only four digits {1, 2, 3, 4}.
    - Case 1: Duplication of digit is allowed.
    - Case 2: Duplication of digit is not allowed.
2. Write a Java programs to evaluate the following series
    a. $\text{Sin}(x) = x - (x^3/3!) + (x^5/5!) - \ldots$
    b. $\text{Sum} = 1 + (1/2)2 + (1/3)3 + \ldots$
3. Write a Java program to display the numbers in the following format (Hint: use nested for loop).

    1

2 3

4 5 6

7 8 9 10

LAB: 4 **Date:**

ARRAYS

Objectives:

1. To learn the syntax & usage of arrays in Java
2. To write simple Java programs to demonstrate the usage of arrays in Java.

4.1 ARRAYS

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

i) One-dimensional arrays: A one-dimensional array is, essentially, a list of like-typed variables. The general form of a one dimensional array declaration is : *type var-name[ ];*

int month_days[]; // declares an array named month_days with the type "array of int".

Although this declaration establishes the fact that month_days is an array variable, no array actually exists. The value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, it must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

array-var= new type[size];
month_days = new int[12];
It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown below:

int month_days[] = new int[12];

// example to know the usage of array

```
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };
System.out.println("April has " + month_days[3] + " days."); }}
```

ii)Multi-dimensional arrays: Multidimensional arrays are arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

```
int twoD[ ][ ] = new int[4][5];
```

```
// Demonstrate a two-dimensional array.

class TwoDArray {
public static void main(String args[]) {
int twoD[ ][ ]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

Ouput:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Alternate array declaration syntax: type[ ] var-name;

```
int al[ ] = new int[3];
int[ ] a2 = new int[3];
```

Lab exercises

i. Write a Java program to display non diagonal elements and find their sum. [Hint: **Non Principal diagonal**: The diagonal of a diagonal matrix from the top right to the bottom left corner is called non principal diagonal.]
ii. Write a Java program to display principal diagonal elements and find their sum. [Hint: Principal Diagonal: The principal diagonal of a rectangular matrix is the diagonal which runs from the top left corner and steps down and right, until the right edge or the bottom edge is reached].
iii. Find whether a given matrix is symmetric or not. [Hint: A = AT]
iv. Write a program to add and multiply two integer matrices. The algorithm for matrix multiplications are given below:

a) To multiply two matrixes sufficient and necessary condition is "number of columns in matrix A = number of rows in matrix B".
b) Loop for each row in matrix A.
c) Loop for each columns in matrix B and initialize output matrix C to 0.
d) This loop will run for each rows of matrix A.
e) Loop for each columns in matrix A.
f) Multiply A[i,k] to B[k,j] and add this value to C[i,j]
g) Return output matrix C.

v. Write a Java program to find whether the matrix is a magic square or not. [Hint: Compare the sum for every row, the sum with every column, the sum of the principal diagonal and the sum of the non-principal diagonal elements. If they are all same, then the matrix is a magic square matrix].

Additional exercises

1. Print all the prime numbers in a given 1D array.
2. Find the largest and smallest element in 1D array.
3. Search for an element in a given matrix and count the number of its occurrences.
4. Write a program to merge two arrays in third array. Also sort the third array in ascending order.
5. Find the trace and norm of a given square matrix. [Hint: Trace= sum of principal diagonal elements; Norm= Sqrt(sum of squares of the individual elements of an array)]

LAB NO: 5 **Date:**

CLASSES AND METHODS

Objectives:

1. To understand the fundamentals of class
2. To know the usage and importance of constructors
3. To be familiar with method overloading and constructor overloading
4. To write simple Java programs to demonstrate the usage of classes, constructors and overloading concepts of Java

5.1 Fundamentals of class

A class defines the structure and behavior (data and code) that will be shared by a set of objects. A class is a template for an object, and an object is an instance of a class. When a class is created

it will specify the code and data constituting that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. The methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data.

General form of a class is as shown below:

```
class classname{
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

```
/* A simple Java program to illustrate the class concept.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
}
}
```

```
C://>javac BoxDemo.java
C://>java BoxDemo
Voulme is 3000
```

5.2 Constructors

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
/*  Box uses a constructor to initialize the dimensions of a box.*/
class Box {
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
        }
// compute and return volume
double volume() {
return width * height * depth;
} }

class BoxDemoConstructor {
        public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }}
```

```
Output:
Constructing Box
```

```
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

5.3 Method overloading and Constructor overloading

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.

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.

```
// 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;
        }
}

class Overload {
public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.2);
        System.out.println("Result of ob.test(123.2): " + result);
        }
```

```
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.2
Result of ob.test(123.2): 15178.24
```

In this example, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

```
/* sample program for constructor overloading
 Box defines three constructors to initialize the dimensions of a box various ways.
*/
class Box {
        double width;
        double height;
        double depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }
// constructor used when no dimensions specified
        Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
        }
        // constructor used when cube is created
        Box(double len) {
        width = height = depth = len;
        }
        // compute and return volume
        double volume() {
        return width * height * depth;
}
}

class OverloadCons {
public static void main(String args[]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
```

```
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
}}
```

The output produced by this program is shown here:
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

Lab exercises

1. Create a class Box that uses a parameterized method to initialize the dimensions of a box. (dimensions are width, height, depth of double type). The class should have a method that can return volume. Obtain an object and print the corresponding volume in main() function.

2. Define a class Employee with data members: employee name, city, basic salary, dearness allowance (DA%) and house rent (HRA%). Define getdata(), calculate(), and display() functions. Calculate method should find the total salary and display method should display it.
   Total = basic+basic*da/100+basic*hra/100;

3. Create a Time class that has separate integer member data for hours, minutes and seconds. One constructor should initialize these data to zero and another should initialize to fixed value. A method should display time in hh:mm:ss format. Finally a method should add 2 objects of time passed as argument.

4. Create a complex class. Use method overloading to find the sum.
   add(integer, complex number)
   add(complex number, complex number)

5. Create class Number with only one private instance variable as a double primitive type. Include the following methods (include respective constructors) isZero( ), isPositive(), isNegative( ), isOdd( ), isEven( ), isPrime(), isArmstrong(). The above methods return boolean primitive type.

6. Write a program to define a class called Book with title,author and edition fields. Define suitable constructors for the Book class. Create a list of 6 Book objects in ascending order. Display only those books' details written by an author taken as an user input.

Additional exercises:

1. The annual examination results of 3 students are tabulated as follows:-

Roll No.	Subject 1	Subject 2	Subject 3

Create a class Result with 2D array and 1D array as its data members. And write methods to perform the following tasks:-

   a. Store marks of 3 subjects obtained by 3 students in 2D array
   b. To store total marks obtained by each student in 1D array.
   c. To find the highest marks in each subject and the roll number of the student who secured it.
   d. To find the student who obtained the highest total marks.

2. Create a class with integer array of size 10 and write methods to perform following:-
   a. Input values into an array
   b. Display the values
   c. Display the largest value
   d. Display the average
   e. Sort the array in ascending order

3. Swap two values using call by value and call by reference.

4. Write a Java program to implement stack class.

5. Write a JAVA program which contains a method square() such that square(3) returns 9, square(0.2) returns 0.04.

LAB NO: 6 **Date:**

CLASS INHERITANCE

Objectives:

1. To understand the basics of inheritance
2. To learn the concept of method overriding
3. To study the dynamic method dispatch
4. To write Java programs using the concepts of inheritance, method overriding and dynamic method dispatch

6.1 Basics of inheritance

Inheritance allows the creation of hierarchical classifications. Using inheritance, it can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass.* The class that does the inheriting is called a *subclass.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass, and add its own unique elements. A keyword called "***extends***" is used for inheritance.

The general form of a **class** declaration that inherits a superclass is shown below:

```
class subclass-name extends superclass-name {
        // body of class
}
```

It can only specify one superclass for any subclass because Java does not support the inheritance of multiple super classes into a single subclass. But, it is possible to create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

NOTE:

- A sub class cannot access those members of the superclass that have been declared as **private**.
- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

- **super** has two general forms. The first form calls the superclass constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.
- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass constructor, this order is the same whether or not super( ) is used.

```
// A simple example of inheritance.
// Create a superclass.
class A {
        int i, j;
        void showij() {
        System.out.println("i and j: " + i + " " + j);
        }
}

// Create a subclass by extending class A.
class B extends A {
        int k;
        void showk() {
        System.out.println("k: " + k);
}
void sum() {
        System.out.println("i+j+k: " + (i+j+k));
        }
}

class SimpleInheritance {
public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();
        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
        /* The subclass has access to all public members of
        its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
        }
```

```
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

6.2 Method overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden

```
// Method overriding.
class A {
        int i, j;
A(int a, int b) {
        i = a;
        j = b;
}

// display i and j
void show() {
        System.out.println("i and j: " + i + " " + j);
}}

class B extends A {
        int k;
        B(int a, int b, int c) {
        super(a, b);
        k = c;
}
// display k – this overrides show() in A
void show() {
        System.out.println("k: " + k);
}}

class Override {
public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
}}
```

Output:
k = 3

6.3 Dynamic method dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through superclass reference variable, different versions of the method are executed.

```
// Dynamic Method Dispatch
class A {
    void callme() {
    System.out.println("Inside A's callme method");
    }
 }

 class B extends A {
    // override callme()
    void callme() {
    System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
    System.out.println("Inside C's callme method");
    }
}

class Dispatch {
public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C
    A r; // obtain a reference of type A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme
    r = b; // r refers to a B object
```

```
        r.callme(); // calls B's version of callme
        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
    }
    }
```

The output from the program is shown below:
Inside A's callme method
Inside B's callme method
Inside C's callme method

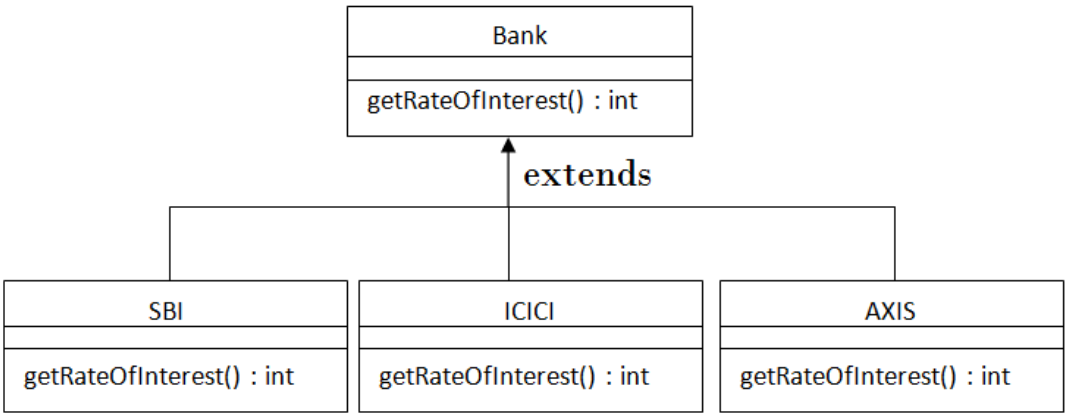
Lab exercises

1. Create an Account class that stores customers name, acc-no and type of account. From this derive class current account and savings bank account. Include necessary methods in order to achieve following tasks:-
   a) Accept the deposit from a customer and update the balance
   b) Display the balance
   c) Compute and deposit interest
   d) Permit withdraw and update the balance
   e) Check for minimum balance impose penalty if necessary and update the balance

   For savings bank account the facilities given are computing interest, withdraw facility. No interest on current bank account and should maintain a minimum balance and if balance is less than this level, service tax is imposed.

2. Create a base class for student having registration number, name and age. From this class create two new class UG and PG student with semester and fees as its data members. Use proper member function for demonstrating inheritance. Display the details of students who took admission to UG course and PG course separately, total number of UG admissions and PG admissions.

3. Create a base class called Bank that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest. Write a Java program to calculate the interest for SBI, ICICI and AXIS banks, by demonstrating the concept of method overriding and dynamic method dispatch.



Additional exercises:

1. Create a base class “Game” with method called “type” that prints “indoor & outdoor games”. Create a subclass cricket with a method called “type” that prints “cricket is an outdoor game”. Create one more subclass of “Game” called “chess” with a method “type” that prints “chess is an indoor game”. Write a complete Java program for the above, to understand the Dynamic method dispatch concept.

2. Create two classes Bike and Splendar. Splendar class extends Bike class and overrides its run() method. Write a complete Java program to implement the runtime polymorphism. Include a member called “speedlimit” in both the classes with different values. The run() method should give the information of speed limit. Check whether the runtime polymorphism can be achieved through the data members.

LAB NO: 7 **Date:**

CLASSES-ACCESS CONTROL, STATIC KEYWORD, NESTED & INNER CLASS, FINAL ,WRAPPER CLASS

Objectives:

1. To know the different access specifiers used with the class
2. To learn the concept of nested, inner and wrapper classes
3. To study the variations of static keyword when used with methods and variables
4. To write Java programs using the concepts of access specifiers, nested, inner, wrapper class, usage of static keyword with methods and variables.

7.1 Access Specifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are −

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).
  Default Access Modifier - No Keyword

7.2 Variation of using static keyword

The static keyword is used in java mainly for memory management. It is used with variables, methods, blocks and nested class. It is a keyword that are used for share the same variable or method of a given class. This is used for a constant variable or a method that is the same for every instance of a class. The main method of a class is generally labeled static.

No object needs to be created to use static variable or call static methods, just put the class name before the static variable or method to use them. Static method can not call non-static method.

In java language static keyword can be used for following

- variable (also known as class variable)

- method (also known as class method)
- block
- nested class

Static variable

Any variable declared as static is known as **static variable**.

Static variable is used to fulfill the common requirement. For Example company name of employees, college name of students etc. Name of the college is common for all students.
The static variable allocate memory only once in class area at the time of class loading.

Advantage of static variable

Using static variable a program memory can be made efficient (i.e it saves memory).

When and why to use static variable

For example to store record of all employee of any company, in this case employee id is unique for every employee but company name is common for all. When a static variable such as a company name is created then only once memory is allocated otherwise it allocate a memory space each time for every employee.

Syntax to declare static variable:
public static variableName;

Syntax for declare static method:
public static void methodName()
{
.....
.....
}

Syntax for access static methods and static variable:

className.variableName=10;
className.methodName();
Example
public static final double PI=3.1415;
public static void main(String args[])
{
.....
.....
}

Difference between static and final keyword

static keyword always fixed the memory that means that will be located only once in the program where as final keyword always fixed the value that means it makes variable values constant.

7.3 Nested,Inner,Wrapper Class

a) Nested & Inner Class

The Java programming language allows it to define a class within another class. Such a class is called a nested class and is illustrated here:

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```

Terminology: Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called static nested classes. Non-static nested classes are called inner classes.

```
class OuterClass {
   ...
   static class StaticNestedClass {
      ...
   }
   class InnerClass {
      ...
   }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or package private. (Recall that outer classes can only be declared public or package private.)

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B

within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

b) Wrapper Class

Wrapper classes are used to convert any data type into an Object type. The primitive data types are not objects. They do not belong to any class. They are defined in the language itself.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.

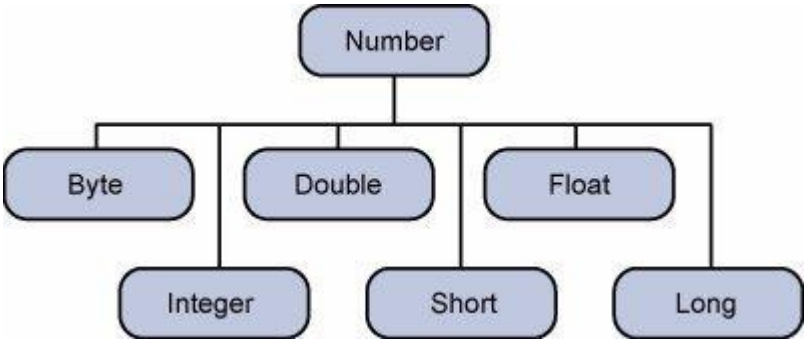


Fig 8.1 The hierarchy of the class Number.

Converting primitive data types into object is called boxing, and this is taken care by the compiler. Therefore, while using a wrapper class pass the value of the primitive data type to the constructor of the Wrapper class to convert primitive data into an Object of its owen type.

The Wrapper object can be converted back to a primitive data type, and the process is called unboxing. The Number class is part of the java.lang package.

Following is an example of boxing and unboxing −

Example

```
public class Test {

   public static void main(String args[]) {
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

o/p:15

Lab exercises:

1. Write a java program to store student record of college named “MIT”. Class Student_Detail should contain name, id, college_name as its members.display_details() method should display the details of the students.
2. Write a counter program to count the number of objects created.
3. Write a java program to illustrate autoboxing and unboxing by considering all data types.

Additional Exercises:

1) Write output of the following and analyze the code.

```
class Example1{
  //Static class
  static class X{
     static String
str="Inside Class X";
  }
  public static void
main(String args[])
  {
     X.str="Inside Class
Example1";

System.out.println("Stri
ng stored in str is- "+
X.str);
  }
}
```

```
class Example2{
   int num;
   //Static class
   static class X{
      static String str="Inside
Class X";
      num=99;
   }
   public static void
main(String args[])
   {
      Example2.X obj = new
Example2.X();

System.out.println("Value
of num="+obj.str);
   }
}
```

```
class Example3{
   static int num;
   static String mystr;
   static{
      num = 97;
      mystr = "Static keyword
in Java";
   }
   public static void
main(String args[])
   {

System.out.println("Value
of num="+num);

System.out.println("Value
of mystr="+mystr);
   }
}
```

```
class Example4{
   static int num;
   static String mystr;
   //First Static block
   static{

System.out.println("Stat
ic Block 1");
      num = 68;
      mystr = "Block1";
  }
  //Second static block
  static{

System.out.println("Stat
ic Block 2");
      num = 98;
      mystr = "Block2";
  }
  public static void
main(String args[])
```

```
class Example5{
   static int i;
   static String s;
   public static void
main(String args[]) //Its a
Static Method
   {
      Example5 obj=new
Example5();
      //Non Static variables
accessed using object obj

System.out.println("i:"+obj.
i);

System.out.println("s:"+obj.
s);
   }
}
```

```
class Example6{
  static int i;
  static String s;
  //Static method
  static void display()
  {
     //Its a Static method
     Example6 obj1=new
Example6();

System.out.println("i:"+obj1
.i);

System.out.println("i:"+obj1
.i);
  }

  void funcn()
  {
     //Static method called in
non-static method
```

{
 	
System.out.println("Value of num="+num);
 	
System.out.println("Value of mystr="+mystr);
}
}
		display();
		}
		public static void main(String args[]) //Its a Static Method
		{
		//Static method called in another static method
		display();
		}
		}

LAB NO: 8 **Date:**

INTERFACE & ABSTRACT CLASS

Objectives:

1. To learn the use of abstract class
2. To understand the interfaces
3. To write Java programs to illustrate usage of interfaces and abstract classes

8.1 Abstract class

An abstract class is a superclass that cannot be instantiated and is used to state or define general characteristics. An object cannot be formed from a Java abstract class; trying to instantiate an abstract class only produces a compiler error. The abstract class is declared using the keyword abstract.

Abstract classes serve as templates for their subclasses. For example, the abstract class Tree and subclass, Banyan_Tree, has all the characteristics of a tree as well as characteristics that are specific to the banyan tree.

Declaring an Abstract Class in Java

In Java a class can be made an abstract by adding the abstract keyword to the class declaration. Here is a Java abstract class example:

```
public abstract class MyAbstractClass {
}
```

If it is required that certain methods be overridden by subclasses by specifying the abstract type modifier, then these methods are referred to as sub-classes responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them. It cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

Any class that contains one or more abstract methods must also be declared abstract.

```
abstract class A {
        abstract void callme();
        // concrete methods are still allowed in abstract classes
        void callmetoo() {
        System.out.println("This is a concrete method.");
        }
}

class B extends A {
        void callme() {
        System.out.println("B's implementation of callme.");
        }
}

class AbstractDemo {
        public static void main(String args[]) {
        B b = new B();
        b.callme();
        b.callmetoo();
        }
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

8.2 Interfaces

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that an interface defines which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows to fully utilize the "one interface, multiple methods" aspect of polymorphism.

An interface is defined much like a class. This is the general form of an interface:

access interface *name* {
return-type method-name1(*parameter-list*)*;*
return-type method-name2(*parameter-list*)*;*
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(*parameter-list*)*;*
type final-varnameN = value;
}

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

access class *classname*[extends *superclass*]
[implements *interface* [,*interface...*]] {
// class-body
}

//simple program to demonstrate the interfaces

```
interface Callback {
        void callback(intparam);
}
```

```
class Client implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
        System.out.println("callback called with " + p);
        }
}
```

Note: When an interface's method is implemented , it must be declared as ***public***.

The variables can be declared as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be stored in such a variable. When a method is called through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
        public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
        }
}
```

The output of this program is shown here:

```
callback called with 42
```

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Lab exercises:

1. Imagine a company that markets both hardware and software. Create an interface that will be implemented by two classes hardware and software, where hardware item holds the category of item and its original manufacturer and software holds type of software and operating system under which it works. Calculate the total sales for hardware as well as software item recorded for last 3months

2. Write a program to compute the areas of a rectangle and a circle by using abstract class.

3. Use interface for the previous question instead of abstract class.

Additional exercises:

1. Write a program to compute the area of a square and a triangle by using abstract class.

2. Use interface for the previous question instead of abstract class.

3. Write a java program to create an interface called “sports” with methods getNumberOfGoals and dispTeam. Create classes Hockey and football that uses the interface “sports”. Write the appropriate code for the methods to display the goals and the wining team considering at least two teams.

LAB NO: 9 **Date:**

STRING HANDLING

Objectives:

1. To understand the basics of String Handling and String Class

2. To develop string based applications.

9.1 Basics of String Handling and String Class

Java implements strings as objects of type **String**. It has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed in number of ways, making it easy to obtain a string when needed. Once a **String** object has been created, that cannot be changed. To modify the string a new **String** object is created and the original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.

Both the **String** and **StringBuffer** classes are defined in **java.lang**. Thus, they are available to all programs automatically.

Following are the constructors of String class:

String(char chars[])
String(char *chars*[ ], int *startIndex*, int *numChars*)
String(String *strObj*)
String(byte *asciiChars*[ ])
String(byte *asciiChars*[ ], int *startIndex*, int *numChars*)

The table 9.1 shows the methods of String Class

Table 9.1 Methods of String Class

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
int	**codePointAt**(int index) Returns the character (Unicode code point) at the specified index.
Modifier and Type	**Method and Description**

int	**compareTo**(String anotherString) Compares two strings lexicographically.
int	**compareToIgnoreCase**(String str) Compares two strings lexicographically, ignoring case differences.
String	**concat**(String str) Concatenates the specified string to the end of this string.
boolean	**contains**(CharSequence s) Returns true if and only if this string contains the specified sequence of char values.
boolean	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
boolean	**equals**(Object anObject) Compares this string to the specified object.
boolean	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
static String	**format**(Locale l, String format, Object... args) Returns a formatted string using the specified locale, format string, and arguments.
byte[ ]	**getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
void	**getChars**(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
boolean	**isEmpty**() Returns true if, and only if, length() is 0.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.

Note: The strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

```
// Construct one String from another.
class MakeString {
public static void main(String args[ ]) {
        char c[ ] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2); } }
```

The output produced by the above program is shown below:

```
Java
Java

BUILD SUCCESSFUL (total time: 2 seconds)
```

In the above example strings s1 and s2 are created using character array. So s1 and s2 contains the same string.

Lab exercises

(Hint: use the appropriate built in methods of String class to write the following Java programs)

1. Write a Java program to count and display the number of characters, words, lines, and vowels in a String.

2. Write a Java program to replace an entered word to all repeating characters.

3. Write a menu driven program to do the following:-
   a. To check whether a string is palindrome or not
   b. Write the string in an alphabetical order
   c. Reverse the string
   d. Concatenate the original string and the reversed string

4. Write a menu driven program to do the following:-
   a. To compare two strings
   b. To convert the uppercase character to lower and vice-versa
   c. To display whether an entered string is a substring of the other or not
   d. If the entered string is a substring of the other, replace it with “Hello”

Additional exercises:

1. Write a program to accept an array of strings and arrange them in alphabetical order.

2. Develop a program for searching a student from a class. Assume a set of 10 students with their details as Registration number, First Name, Last Name and Degree in an array of Student objects. Search a student either by First Name or Last Name.

3. Write a program to accept five strings separately, concatenate and display them as a single string.

4. Write a program that inputs a telephone number as a string in the form **(555) 555-5555**. Use **String** method to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. Display area code and seven digit phone number separately.

5. Write a program that reads a five-letter word from the user and produces all possible three letter words that can be derived from the letters of the five letter word. For example, the three letter words produced from the word "bathe" include the commonly used words "ate," "bat," "bet," "tab,", "hat," "the" and "tea."

LAB NO: 10 **Date:**

EXCEPTION AND FILE HANDLING

Objectives:

1. To understand the basic concept of exception handling in Java.
2. To know the exception types.
3. To write simple Java programs that handle exceptions.
4. To understand data handling using streams.
5. To explore File class and its constructors and methods.
6. To develop an application which takes an input from a file and sends its output to another file.

10.1Basics of Exception Handling

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. Exceptions can be generated by the Java run time system, or they can be manually generated by the code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Program statements that want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. The code can catch this exception (using **catch**) and handle it in some rational manner. 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.

The general form of an exception handling block is given below:

```
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.

10.2 Exception Types

Java defines several exception classes inside the standard package **java.lang**. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeExceptions** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 10.1. Table 10.2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.* Java defines several other types of exceptions that relate to its various class libraries.

Table 10.1. Java's Unchecked RuntimeException Subclasses

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide by zero.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
ArrayIndexOutOfBoundsException	Array index is out of bounds.
LlegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out of bounds.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10.2. Java's Checked Exceptions defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

10.3 Simple Java program that handles exception

```
//Program which handles Arithmetic Exception (division by 0)\

import java.util.Scanner;
class ArithmeticExceptionTest
{
 public static void main(String args[]){
            Scanner sc=new Scanner(System.in);
            int a,b,x=0;
            System.out.println("DIVISION");
            System.out.print("Enter a number - ");
            a=sc.nextInt();
            System.out.print("Enter a number - ");
            b=sc.nextInt();
            try{
                  x=a/b;
              }catch(ArithmeticException A){
                  System.out.println("Error");
              }
            finally{
                  System.out.println("a/b="+x);
             }
 }}
```

The output produced by the above program is shown below:

```
DIVISION
Enter a number - 2
Enter a number - 4
a/b=0
BUILD SUCCESSFUL (total time: 13 seconds)

DIVISION
```

Enter a number - 5
Enter a number - 0
Error
a/b=0
BUILD SUCCESSFUL (total time: 11 seconds)

Lab exercises

1. Write a program that handles NumberFormatException. [Hint: Invalid conversion of a string to a number]
2. Write a program that handles ArrayOverflowException. [Hint: Consider the array size]
3. Write a program to enter student's Name, Roll Number and Marks in three subjects and find the percentage, grade and handle NumberFormatException.
4. Create a user defined exception class which displays error message.
5. Write a program for validating matrix. [Hint: Square matrix]
6. Write a program for checking the negative root of a number. [Hint: Input negative number]

Additional exercises:

1. Design a class Input_Exception for validating an input taken from the user during runtime and continue to perform the sum of the numbers entered until a -1 is input. In case the user enters a -1, the sum is calculated and displayed. In case the user enters any floating point numbers, then the method that takes the input should raise the user defined exception and the same should be handled in the main.

2. Write a program to throw the NegativeArraySizeException, when the size of an array is negative.

3. Write a program to raise EvenNumberException if the number is even.

4. Write a program to find the exception marks out of bounds. In this program create a class called Student. If the mark is greater than 100 it must create an exception called MarkOutofBoundsException and throw it.

LAB NO: 11 **Date:**

MULTITHREADED PROGRAMMING & SWINGS

Objectives:

1. To understand basics of multitasking and multithreading.
2. To develop multitasking applications using threads.
3. To understand basic concepts of a java swing.
4. To develop simple java application using swings.

11.1 Basics of Multitasking and Multithreading

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously. Java's easy-to-use approach to multithreading allows us to think about the specific behavior of program. 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 (performs more than one task at a time).

There are two distinct types of multitasking: process-based and thread-based. Process-based multitasking is the feature that allows computer to run two or more programs concurrently. For example, process-based multitasking enables to run the Java compiler at the same time that are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. 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.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
There are two ways to create a thread.

1. By extending a Thread class
2. By implementing Runnable Interface.

Following code is the general form of thread creation block:

```
class NewThread implements Runnable {
        public void run()
        {  //---
        }
}

class Threadex {
        public static void main(String args[]) {
        new NewThread(); // create a new thread
        //-----
        }
 }

//Example to create a thread.
        class SampleThread implements Runnable
        {
                public void run()
                {
                        for(int i=0;i<5;i++)
                        {
                                System.out.println("Hello World");
                        }
                }
        }

        public class MyThread
        {
                public static void main(String ar[])
                {
                        SampleThreadthrd=new SampleThread();
                        Thread newThread=new Thread(thrd);
                        newThread.start();
                }
        }
```

The output produced by the above program is shown below:

```
        Hello World
        Hello World
        Hello World
        Hello World
        Hello World

        BUILD SUCCESSFUL (total time: 3 seconds)
```

In the above example thread is created by implementing Runnable interface. Start() method is called to start the execution of a thread.

Lab exercises:

1. Write a menu driven program to create thread using runnable interface and inheriting thread class.[Hint : Make use of Extends and Implements keywords]

2. Write a program to define a matrix class. Create 5 threads – t1 displaying elements in a matrix form, t2- display the transpose of the matrix, t3-display maximum value in the matrix, t4- to display principal diagonal elements, t5-display non diagonal elements [Hint : Multiple instances of thread]

3. Demonstrate isAlive() and join() methods in checking status of a thread in the above program.

4. Write a program to set priority and check for an InterruptedException.

5. Write a program to synchronize two different threads
    a. Using synchronized method.

    b. Using synchronized statements.

Generics

Objectives:

In this lab, student will be able to:

1. Understand the benefits of generics
2. Create generic classes and methods

Introduction to Generics:

Generics are a powerful extension to Java because they streamline the creation of type-safe, reusable code. The term *generics* means *parameterized types*. Parameterized types are important because they enable the programmer to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

Solved exercise:

1. Program defines two classes, the generic class Gen, and the second is GenDemo, which uses Gen. Here, T is a type parameter that will be replaced by a real type when an object of type Gen is created.

```
class Gen<T> {
        T ob;           // declare an object of type T
        // Pass the constructor a reference to an object of type T.
        Gen(T o) {
                ob = o;
           }
     // Return ob.
        T getob() {
                returnob;
           }
     // Show type of T.
```

```
void showType() {
    System.out.println("Type of T is " +ob.getClass().getName());
        }
}
// Demonstrate the generic class.
Class GenDemo {
        public static void main(String args[]) {
         // Create a Gen reference for Integers.
                Gen<Integer>iOb;
          // Create a Gen<Integer> object and assign its
          // reference to iOb. Notice the use of autoboxing
          // to encapsulate the value 88 within an Integer object.
                iOb = new Gen<Integer>(88);
          // Show the type of data used by iOb.
                iOb.showType();
          // Get the value in iOb. Notice that no cast is needed.
                int v = iOb.getob(); System.out.println("value:
                " + v); System.out.println();
          // Create a Gen object for Strings.
                Gen<String>strOb = new Gen<String>("Generics Test");
           // Show the type of data used by strOb.
                strOb.showType();
          // Get the value of strOb. Again, notice that no cast is needed.
                String  str  =  strOb.getob();
                System.out.println("value: " + str);
            }
}
```

Output:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Here, T is the name of a type parameter. This name is used as a placeholder for the actual type that will be passed to Gen when an object is created. The GenDemo class demonstrates the generic Gen class. It first creates a version of Gen for integers, as shown here:

Gen<Integer>iOb;

The type Integer is specified within the angle brackets after Gen. In this case, Integer is a type argument that is passed to Gen's type parameter, T. This effectively creates a version of Gen in which all references to T are translated into references to Integer. Thus, for this declaration, ob is of type Integer, and the return type of getob( ) is of type Integer.

The next line assigns to **iOb**a reference to an instance of an **Integer** version of the **Gen**class:

iOb = new Gen<Integer>(88);

Generics Work Only with Objects. Therefore, the following declaration is illegal:

Gen<int>strOb = new Gen<int>(53); // Error, can't use primitive type

Lab exercises:

1. Write a generic method to exchange the positions of two different elements in an array.
2. Define a simple generic stack class and show the use of the generic class for two different class types Student and Employee class objects.
3. Define a generic List class to implement a singly linked list and show the use of the generic class for two different class types Integer and Double class objects.
4. Write a program to demonstrate the use of wildcard arguments.

Additional exercises:

1. Write a generic method that can print array of different type using a single Generic method:
2. Write a generic method to return the largest of three Comparable objects.

JavaFX and Event Handling

Objectives:

Objectives:

In this lab, student will be able to:

1. Understand importance of light weight components and pluggable look-and-feel
2. Create, compile and run JavaFX applications
3. Know the fundamentals of event handling and role of layout managers

JavaFX:

JavaFX components are lightweight and events are handled in an easy-to-manage, straightforward manner. The JavaFX framework is contained in packages that begin with the javafx prefix. The packages we will use in our code are : javafx.application, javafx.stage, javafx.scene, and javafx.scene.layout.

The Stage and Scene Classes:

Stage is a top-level container. All JavaFX applications automatically have access to one Stage, called the primary stage. The primary stage is supplied by the run-time system when a JavaFX application is started.

Scene is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of Scene.

Layouts:

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the FlowPane class provides a flow layout and the GridPane class supports a row/column grid-based layout.

The Application Class and Life-Cycle methods:

A JavaFX application must be a subclass of the Application class, which is packaged in javafx.application. Thus, your application class will extend Application. The Application class defines three life-cycle methods that your application can override.

1. **void init( ) -** The init( ) method is called when the application begins execution. It is used to perform various initializations.

2. **abstract void start(Stage primaryStage) -** The start( ) method is called after init( ). This is where the application begins and it can be used to construct and set the scene. This method is abstract and hence must be overridden by the application.

3. **void stop( ) -** When the application is terminated, the stop( ) method is called. It can be used to handle any cleanup or shutdown chores.

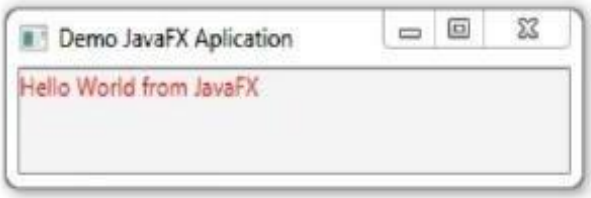
Solved exercise:

1. Write a JavaFX Application program to display a simple message.

```
import javafx.scene.control.*;
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;

public class HelloWorld extends Application {

  public void start(Stage primaryStage) { // entry point for the application
    primaryStage.setTitle("Demo JavaFX Aplication"); // set the title of top level
    //container.
    Label lbl = new Label();    // create a label control
    lbl.setText("Hello World from JavaFX");
    lbl.setTextFill(Color.RED);

    FlowPane root = new FlowPane(); // create a root node.
    root.getChildren().add(lbl); // add the label to the node
    Scene myScene = new Scene(root, 300, 50); // create a scene using the node
    primaryStage.setScene(myScene); // set the scene on the stage
    primaryStage.show(); // show the stage
    }
```

```
public static void main(String[] args) {
      launch(args); // Start the JavaFX application by calling launch( ).
  }
 }
```

Output:



Event Handling:

The JavaFX controls that respond to either the external user input events or the internal events need to be handled. The delegation event model is the mechanism of handling such events. The advantage of this model is that application logic that processes the events is cleanly separated from the User Interface logic that generates the event.

Solved exercise:

2. Write a JavaFX-Application program that handles the event generated by Button control.

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
public class JavaFXEventDemo extends Application {
Label response;
public static void main(String[] args) {
// Start the JavaFX application by calling launch().
launch(args);
}

// Override the start() method.
public void start(Stage myStage) {
```

```
// Give the stage a title.
myStage.setTitle("Use JavaFX Buttons and Events.");
// Use a FlowPane for the root node. In this case,
// vertical and horizontal gaps of 10. FlowPane
 rootNode = new FlowPane(10, 10);
// Center the controls in the scene.
rootNode.setAlignment(Pos.CENTER);
// Create a scene.
Scene myScene = new Scene(rootNode, 300, 100);

// Set the scene on the stage.
myStage.setScene(myScene);
// Create a label.
response = new Label("Push a Button");
// Create two push buttons.
Button btnUp = new Button("Up");
Button btnDown = new Button("Down");
// Handle the action events for the Up button.
btnUp.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) {
response.setText("You pressed Up."); } });
// Handle the action events for the Down button.
btnDown.setOnAction(new EventHandler<ActionEvent>() {
public void handle(ActionEvent ae) { response.setText("You
pressed Down."); } });
// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(btnUp, btnDown, response);
 // Show the stage and its scene.
myStage.show(); } }
```

Output:

JAVA FX UI Controls:

JavaFX:

JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs. The JavaFX framework has all of the good features of Swing.

JavaFX control classes:

Button	ListView	TextField
CheckBox	RadioButton	ToggleButton
Label	ScrollPane	TreeView

- Button: Button is JavaFX's class for push buttons. The procedure for adding an image to a button is similar to that used to add an image to a label. First obtain an ImageView of the image. Then add it to the button.
- ListView: List views are controls that display a list of entries from which you can select one or more.
- TextField: It allows one line of text to be entered. Thus, it is useful for obtaining names, ID strings, addresses, and the like. Like all text controls, TextField inherits TextInputControl, which defines much of its functionality.
- CheckBox: It supports three states. The first two is checked or unchecked, as you would expect, and this is the default behavior. The third state is indeterminate (also called undefined). It is typically used to indicate that the

state of the check box has not been set or that it is not relevant to a specific situation. If you need the indeterminate state, you will need to explicitly enable it.

- RadioButton: Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the RadioButton class, which extends both ButtonBase and ToggleButton.
- ToggleButton: A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up).
- Label: Label class encapsulates a label. It can display a text message, a graphic, or both.
- ScrollPane: JavaFX makes it easy to provide scrolling capabilities to any node in a scene graph. This is accomplished by wrapping the node in a ScrollPane. When a ScrollPane is used, scrollbars are automatically implemented that scroll the contents of the wrapped node.
- TreeView: It presents a hierarchical view of data in a tree-like format.

Lab exercises:

1. Write a JavaFX application program to do the following:
   a. Display the message "Welcome to JavaFX programming" using Label in the Scene.
   b. Set the text color of the Label to Magenta.
   c. Set the title of the Stage to "This is the first JavaFX Application".
   d. Set the width and height of the Scene to 500 and 200 respectively.
   e. Use FlowPane layout and set the hgap and vgap of the FlowPane to desired values.
2. Write a JavaFX program to accept an integer from the user in a text field and display the multiplication table (up to number *10) for that number. Use FlowPane layout for the application.
3. Write a JavaFX program to display a window as shown below. Use TextField for UserName and PasswordField for Password input. On click of "Sign in" Button the message "Welcome UserName" should be displayed in a Text Control. Use GridPane layout for the application.

Fig 11.1 Welcome Window

4. Write a JavaFX program that obtains two positive integers passed from the user in two text fields and displays the numbers and their GCD as the result on a Label.

Additional exercises:

1. Define a class called Employee with the attributes name, empID, designation, basicPay, DA, HRA, PF, LIC, netSalary. DA is 40% of basicPay, HRA is 15% of basicPay, PF is 12% of basicPay. Display all the employee information in a JavaFX application.
2. Design a simple calculator to show the working for simple arithmetic operations. Use Grid layout.

Lab Exercises:

1. Write a JavaFX application program that obtains two floating point numbers in two text fields from the user and displays the sum, product, difference and quotient of these numbers using Canvas on clicking compute button with a calculator image placed on it.
2. Write a JavaFX application program to create your resume. Use checkbox to select the languages which you can speak. On clicking the Submit button all the details of the resume should be displayed using Canvas.
3. Write a JavaFX application program that creates a thread which will scroll the message from right to left across the window or left to right based on RadioButton option selected by the user.
4. Write a JavaFX application program that displays a Circle whose diameter is entered by the user in a text field and calculates and displays the area, radius, diameter and circumference using Canvas based on the option chosen in List View (Area or radius and so on).

Additional exercises:

1. Write a JavaFX program to simulate a static analog clock whose display is controlled by a user controlled static digital clock.
2. Write a JavaFX program to implement a simple calculator using the Toggle buttons.

REFERENCES

1. Herbert Schildt and Dale Skrien, *Java Fundamentals – A Comprehensive Introduction*,1st Edition, McGrawHill, 2015
2. Herbert Schildt, *The Complete Reference JAVA 2,* 10th Edition, Tata McGrawHill, 2017
3. Dietel and Dietel, *Java How to Program*, 9th Edition, Prentice Hall India, 2012