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:

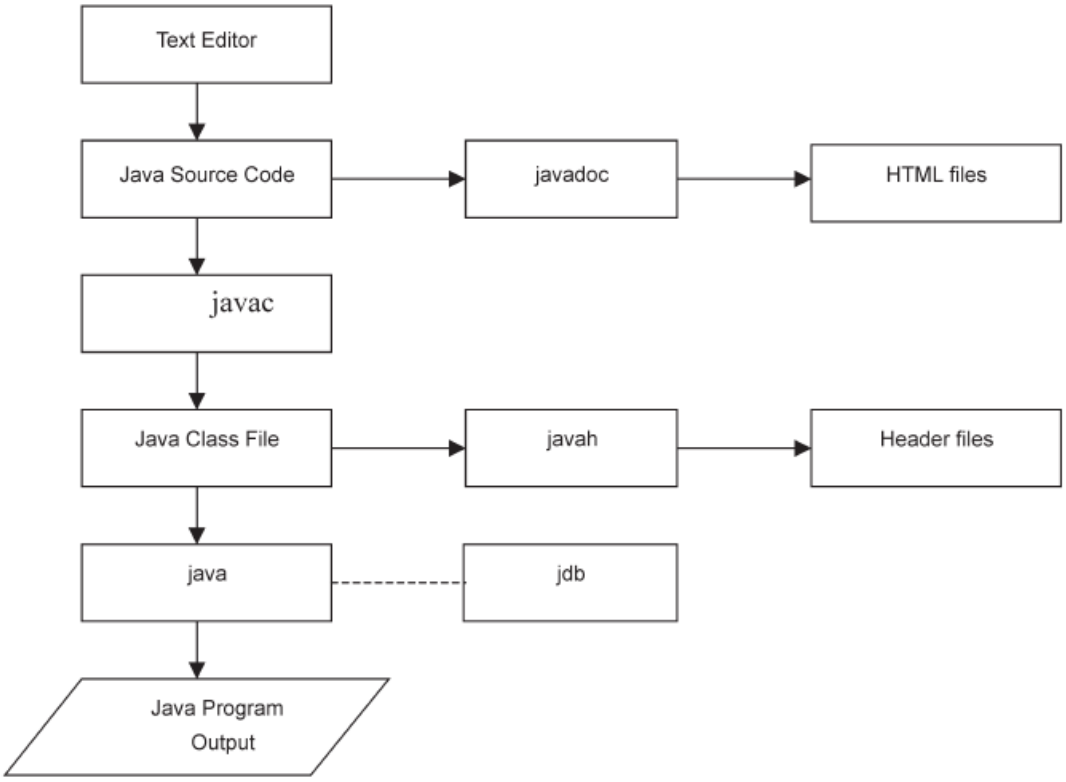

Fig 1.1. Process of building and running Java application programs.

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.

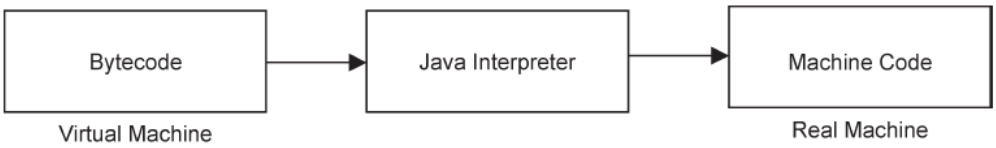

Fig 1.3. Process of converting bytecode into machine code

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.

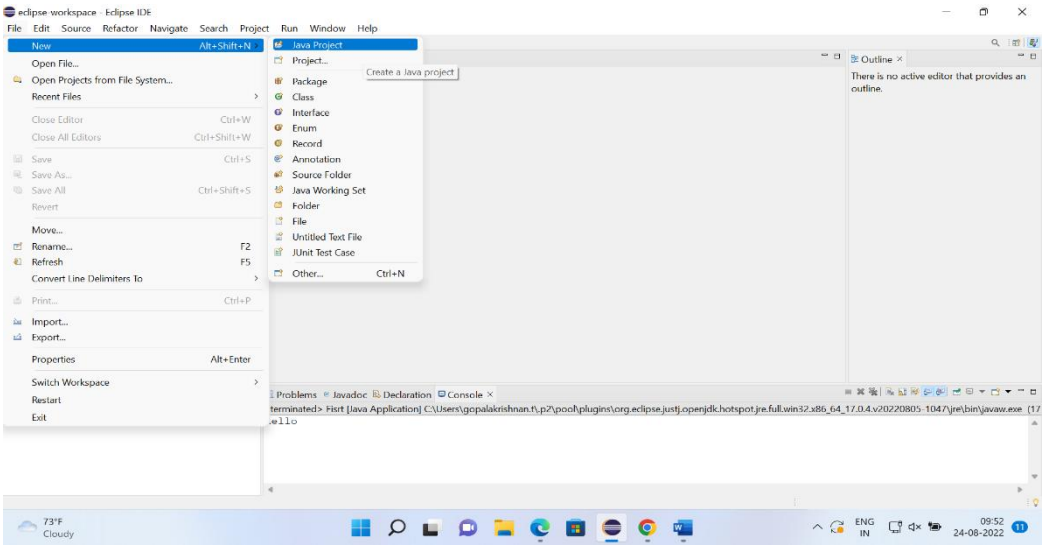

Fig 1.5. Selection of Java Application from the categories Jva.

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.

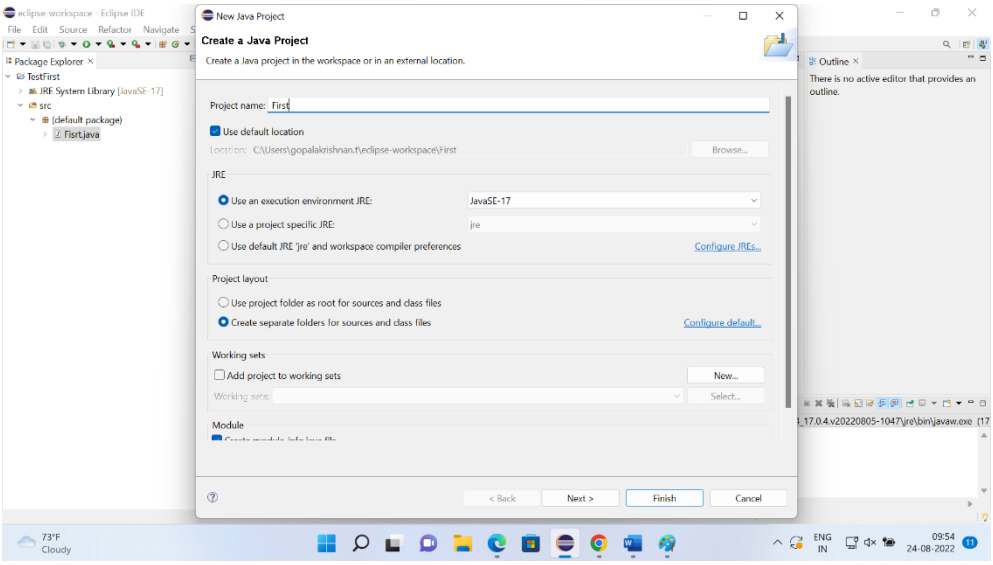

Fig1.6. Giving Name and Location to the newly created Project.

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.

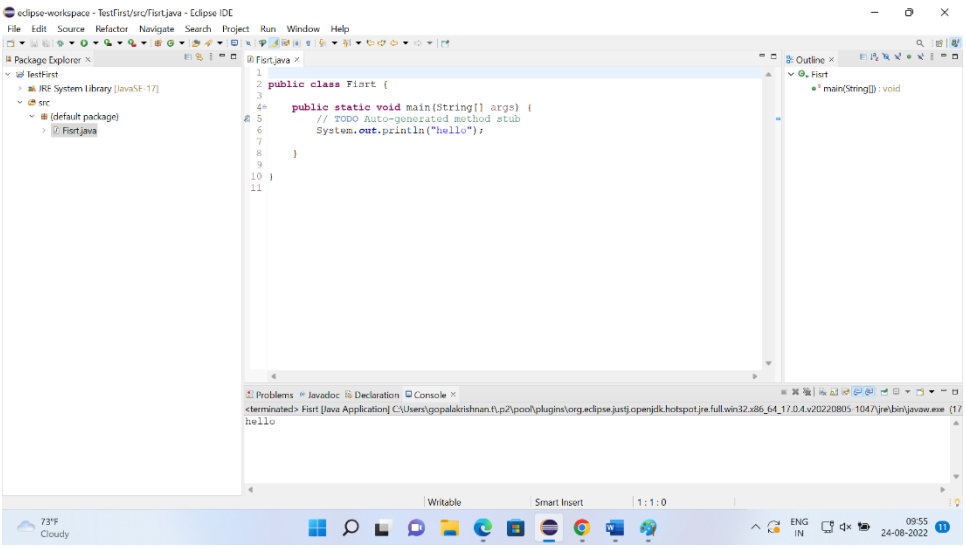

Fig 1.7. Project window,Source Editor Window,Navigator Window opens up after project creation.

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

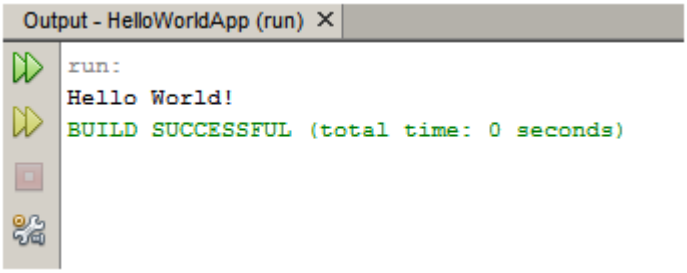

Fig 1.8. Output of the program

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 program 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

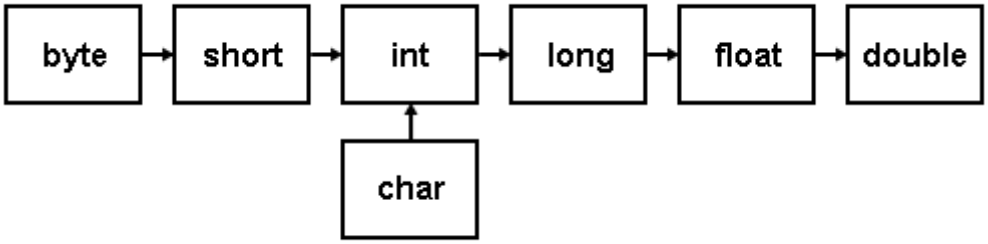

Fig 2.1. Type Promotion Rules

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

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;
        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);
    }
}
```

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 find area and circumference of a rectangle.
3. (Hint: circumference = 2 (length + breadth) ; area= length x breadth)
4. 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
5. 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.)
6. 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. Sin(x) =x - (x^3/3!) + (x^5/5!)-…

   b. Sum=1 + (1/2)2 + (1/3)3 +…

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 give 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)]