



MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

(A constituent unit of MAHE, Manipal)

DEPARTMENT OF _____

CERTIFICATE

This is to certify that Ms./Mr.

Reg. No.: Section: Roll No.:

has satisfactorily completed the lab exercises prescribed for Introduction to Object Oriented Programming Lab CSS-of Second Semester B.Tech, Degree at MIT, Manipal, in the academic year 2025-2026.

Date:

Signature of the Faculty

Signature Head of the Department

i

INDEX

LAB NO.	TITLE	Remarks
	Course Objectives and Outcomes	
	Evaluation Plan	
	Instructions To the Students	
1	Basics of Java programming -Illustration of Data types, Variable and arrays, Type conversion and casting, Operators	
2	Control Statements	
3	Classes, Objects and Methods	
4	Constructors And Static Members	
5	Nested and inner classes	
6	String, Arrays, Array of objects	
7	Mid-Term Lab Exam	
8	Inheritance and method overriding	
9	Abstract Classes & Interfaces	
10	Package & Access Modifiers	
11	Exception Handling	
12	End Sem Lab Exam	
	References	

At the end of this course, the student should be able to:		No. of Contact Hours	Marks	Program Outcomes (POs)	Learning Outcomes (LOs)	BL (Recommended)
CLO1	Write, compile and execute simple Java program containing primitive type variables, literals and arrays	3	1+1+1 (R+V+E)	1,2,3,5,9	1	≥ 3
CLO2	Effectively use different operators and control structures to manage program execution flow	3	1+1+1	1,2,3,5,9	1	≥ 3
CLO3	Demonstrate programs by identifying objects, strings designing classes and suitably overloading methods and constructor	15	8+5+1+20	1,2,3,5,9	2	≥ 3
CLO4	Illustrate programs on inheritance and demonstrate dynamic dispatch of overridden methods	3	2+2+1	1,2,3,5,9	2	≥ 3
CLO5	Demonstrate solutions to programs on abstract classes packages interfaces and exception handling	12	8+5+2	1,2,3,5,9	2	≥ 3
	Total	36	60	1,2,3,5,9	1,2	≥ 3

COURSE OUTCOMES (COS)

Evaluation Plan:

1. Continuous Evaluation	60%	
WEEK-5: EVAL1 L1 TO L4 Record: 8M Viva/Exec: 8M	WEEK-7: EVAL 2 Record: 5M Viva/Exec: 4M AND MID-TERM EVAL: 15M Program exec: 10M Write-up: 5M	WEEK-11: EVAL3 L8 TO L11 Record: 12M Viva: 8M
EXPERIMENT CHECK UP DONE IN REGULAR LAB TIME RANDOMELY FOR 6M , IN ANY LAB AND FOR ANY STUDENT. SO, BACKUP OF PROGRAMS IS NOT NECESSARY.		
2. Lab Examination	40%	
<ul style="list-style-type: none">End Semester Lab evaluation: 40 Marks, write Up: 15 Marks, execution: 25 Marks, Total: 15+25 = 40 Marks. <p>Examination of 2 hours duration (Max. Marks: 40)</p>		

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 the 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.
- 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

OOP LAB MANUAL

- Questions for lab tests and examination 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.

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 lists these tools and their descriptions:

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:

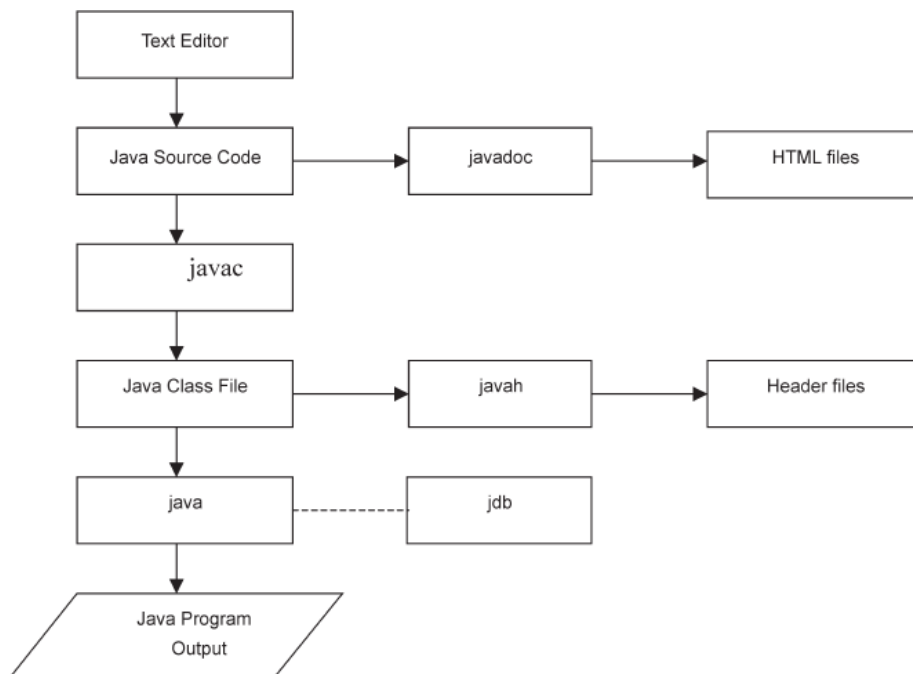


Fig. 1.1. Process of building and running Java application programs.

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.

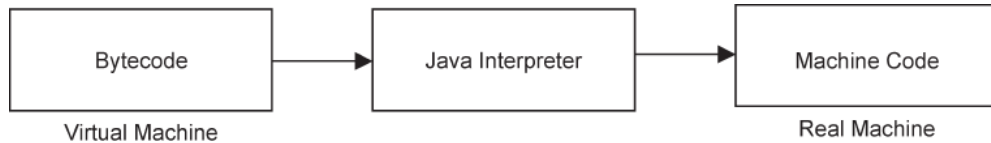


Fig. 1.3: Process of converting bytecode into machine code

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 IDE

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 Eclipse 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 Eclipse IDE.
- In the IDE, choose File > New > Java Project, as shown below:

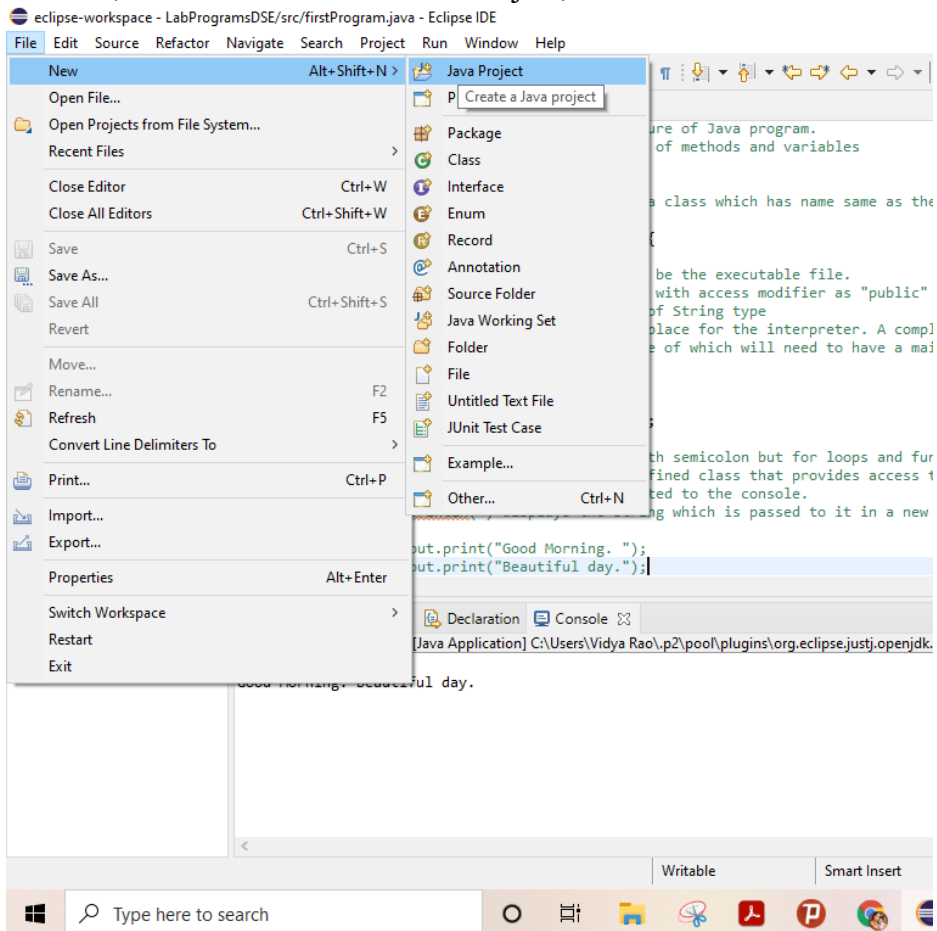


Fig. 1.4. To create a new project in Eclipse IDE.

2. In the New Project wizard, provide a name for your project, as shown in the Fig 1.5 below. Then click Finish.

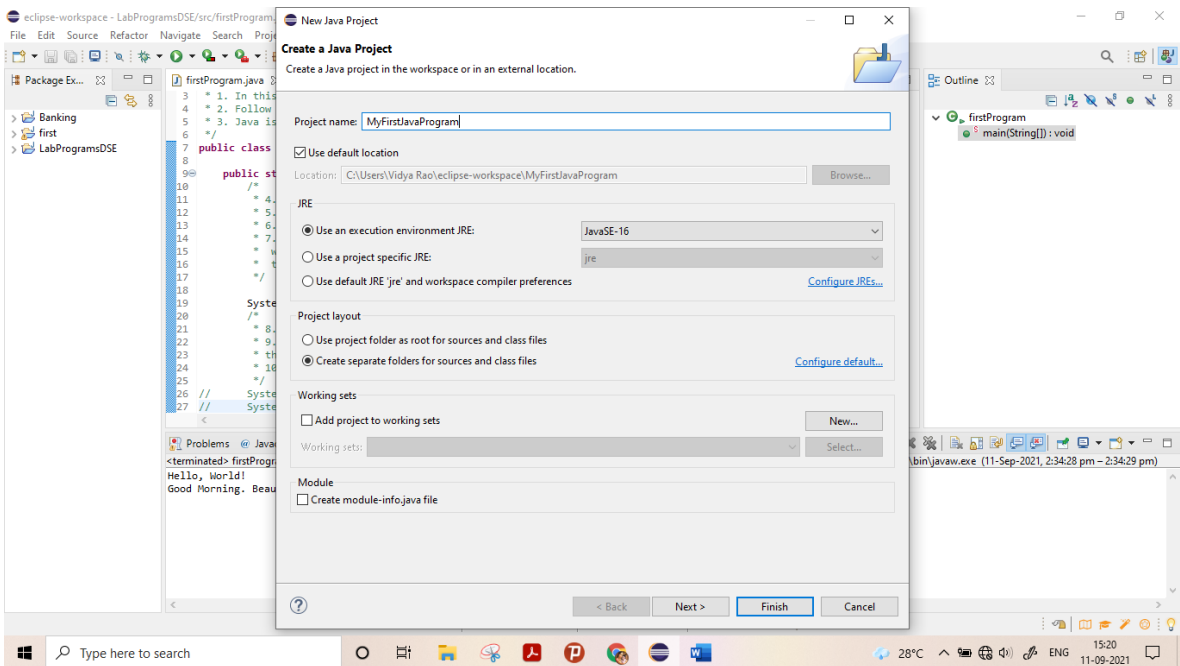


Fig 1.5 Creating Java Project

3. You will see your project displayed on the left panel of Eclipse IDE as in Fig 1.6

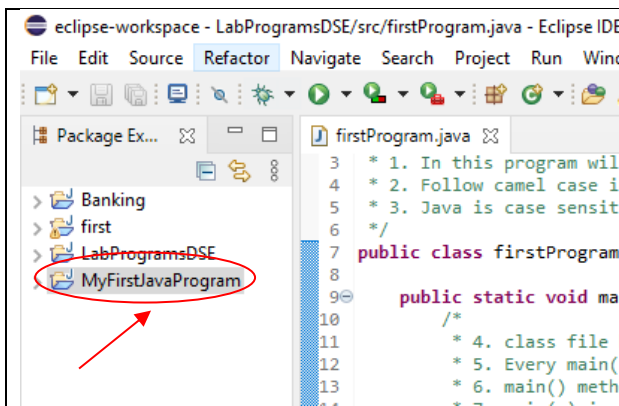


Fig 1.6 Project created and displayed on the directory panel

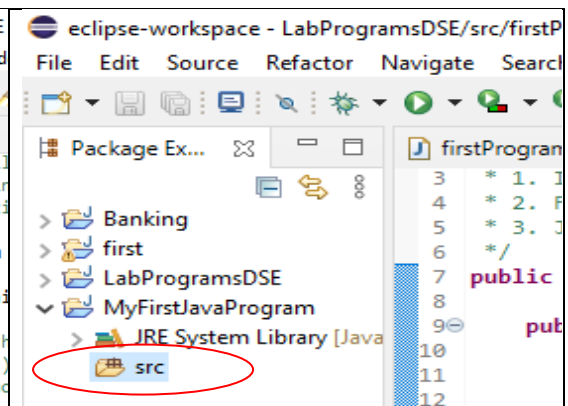


Fig 1.7 Display of packages and source directory

- Click on the project name and we get a drop down folder directory as in Fig 1.7. and the click right click on the “src” folder and select “New > class”

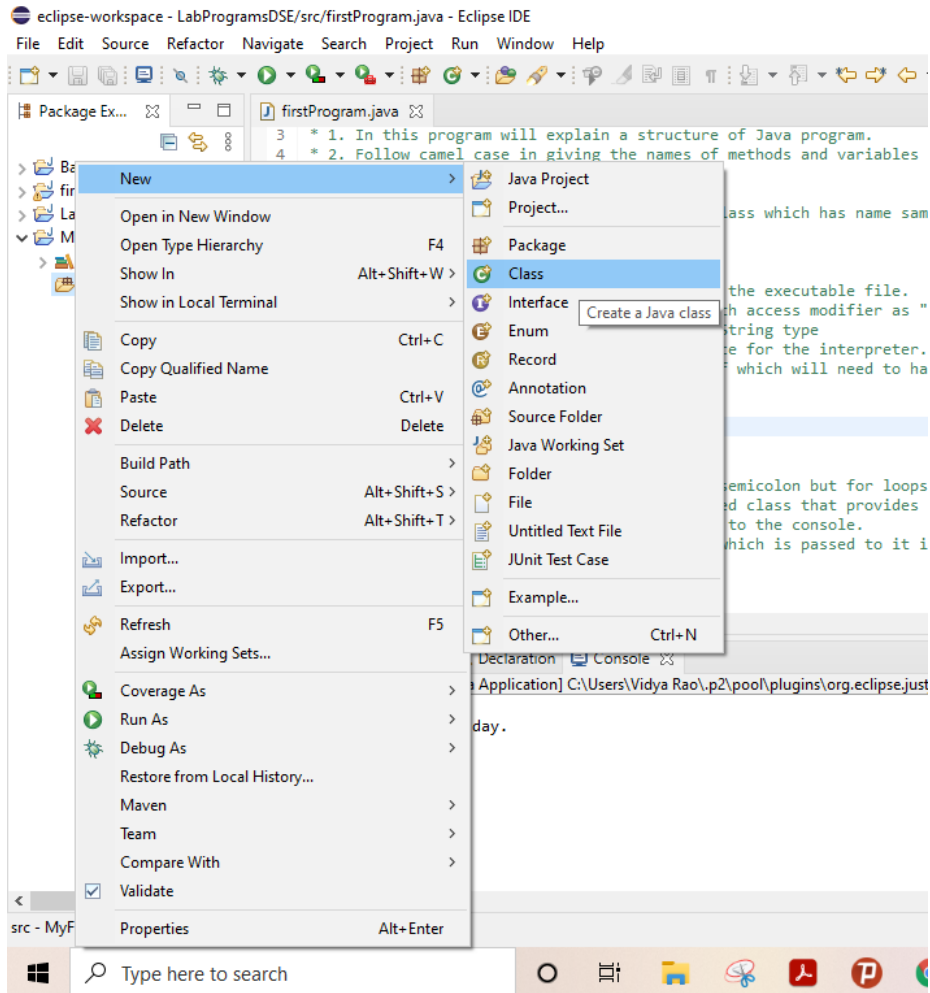


Fig 1.8. To create a class under “src” folder.

- The provide a name to the class and check the “public static void main(String args[])” which creates a class file with default main() method and click “Finish”. As in Fig 1.9

OOP LAB MANUAL

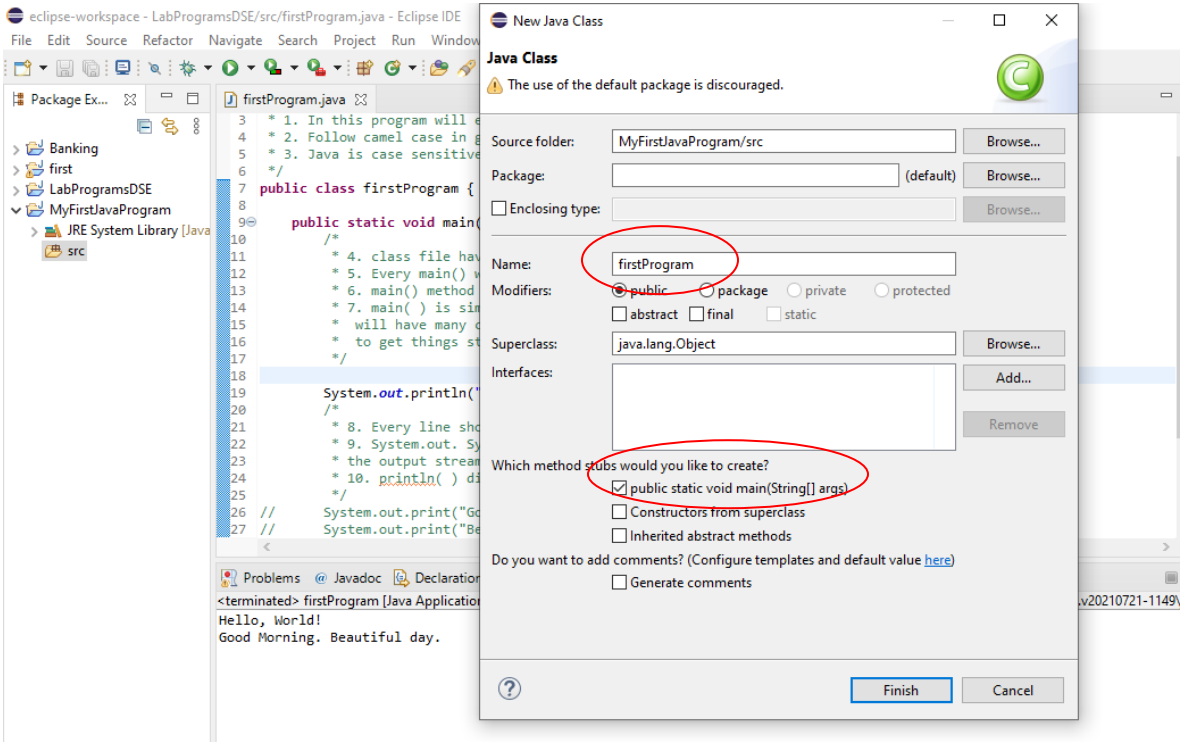


Fig 1.9. providing class name under “src” folder.

6. We can see the default structure of our program as below in Fig 1.10

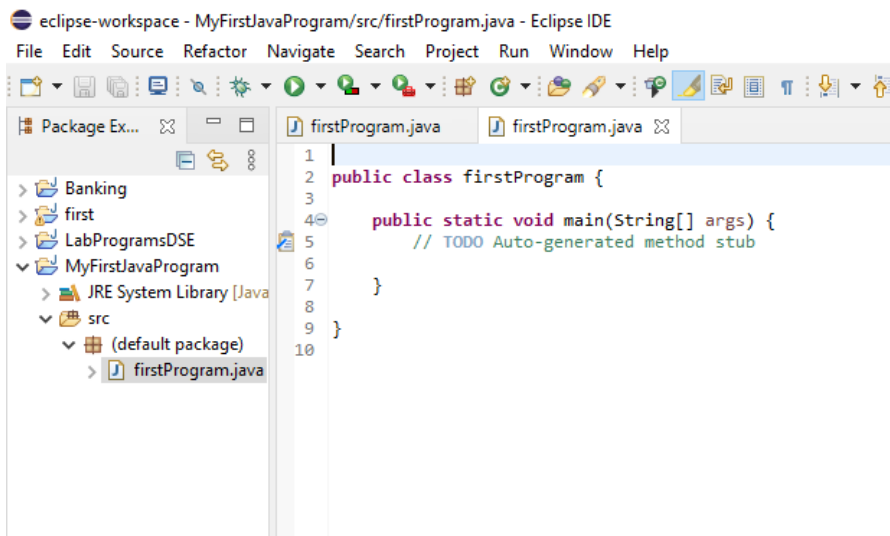


Fig. 1.10: Project window, Source Editor Window.

Compiling and Running the Program

Type the “System.out.println(“Hello, World!”);” within main() as below.

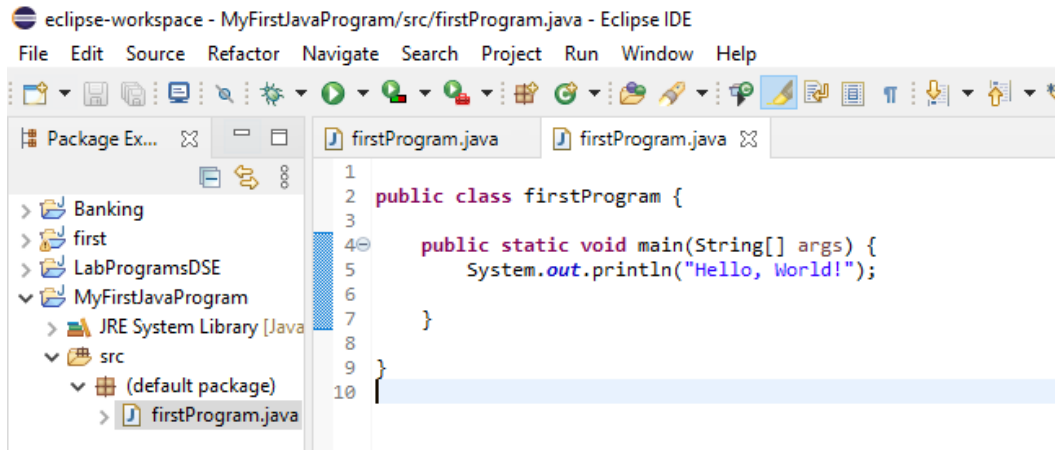



Fig 1.11 Program need to displayed within the main()

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. Click the Green Color arrow button on top of the Window  which will automatically compile and execute the program. The output is displayed below in the Console window as below.

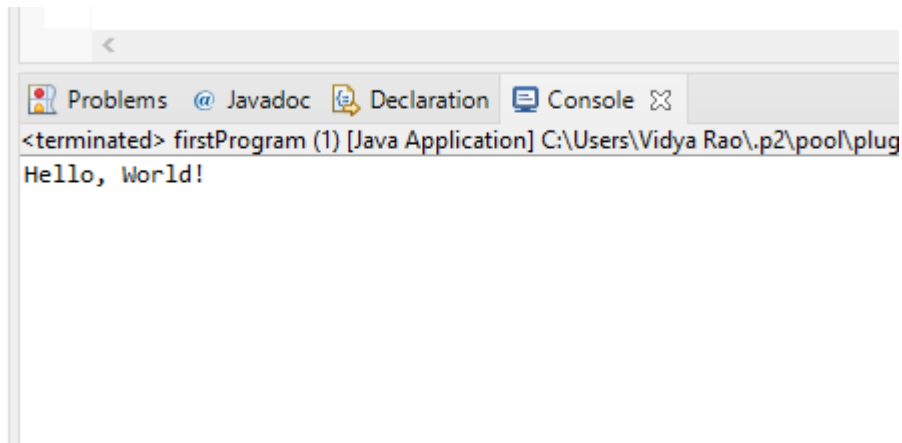


Fig 1.12 Output displayed at console window.

LAB NO.: 1

Date:

Basics of Java programming -Illustration of Data types, Variable and Arrays, Type conversion and casting, 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
5. To learn the syntax & usage of arrays in Java

1.1. 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 **chars**. 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

- i) **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**.

1.2. 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. 1.1

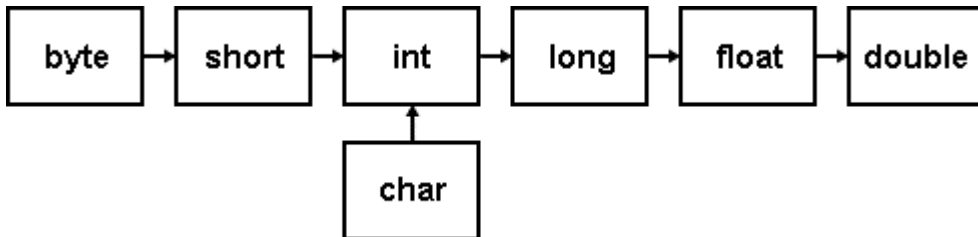


Fig. 1.1: Type Promotion Rules

1.3. 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.

1.4. 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

1.5. 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);`**

OOP LAB MANUAL

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. Example member methods of scanner class are as follows:

Sl. No.	Method	Descripti on
1.	String next()	Returns the next token from the scanner.
2.	String nextLine()	Moves the scanner position to the next line and returns the value as a string.
3.	byte nextByte()	Scans the next token as a byte.
4.	short nextShort()	Scans the next token as a short value.
5.	int nextInt()	Scans the next token as an int value.
6.	long nextLong()	Scans the next token as a long value.
7.	float nextFloat()	Scans the next token as a float value.
8.	double nextDouble()	Scans the next token as a double value.

The program given below 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);
    }
}

```

1.6. 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.

OOP LAB MANUAL

```
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 a1[ ] = new int[3];
int[ ] a2 = new int[3];
```

iii) non symmetric column size arrays (All rows need not have same number of columns):

```
class TwoDArray {
public static void main(String args[]) {
int twoD[ ][ ]= new int[2][ ];
twoD[0]=new int[3]; twoD[1]=new int[2];
int i, j, k = 0;
for(i=0; i<twoD.length; i++)
for(j=0; j<twoD[i].length; j++) {
twoD[i][j] = k;
k++;
}
}
```

OOP LAB MANUAL

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

Output:

```
0 1 2  
3 4
```

Lab Exercises:

1. Write a Java program to accept the number of hours worked, hourly rate and calculates the salary for an employee according to the following criteria: The company pays straight time for the first 40 hours worked by each employee and time and a half for all hours worked in excess of 40 hours.
2. Write a java program to add two numbers using the bitwise operator and check if the output is an even or odd number. [Hint: use left shift and right shift bitwise operators].
3. Write a Java program to execute the following statements. Observe and analyze the outputs.

a. `int x = 10;`

b. `double x = 10.5;`

c. `double x=10.5;`

`double y = x;`

`int y = x;`

`int y = (int) x`

`System.out.println(y);`

`System.out.println(y);`

`System.out.println(y);`

4. Write a Java program to rotate the elements of an array to the right/left by a given number of steps. The program should handle arrays of different sizes and should be able to rotate the array in both directions (left and right).
5. Write a Java program to manage stock information for multiple products. The program should store the product name, product price, and quantity in separate one-dimensional arrays for n items. The user should be able to specify which products they want to purchase and the desired quantity for each. Based on the user's input, the program should generate and display the total bill.

Additional Exercises:

1. Write a Java program to find the result of the following expressions for various values of a & b:
 - a. $(a \ll 2) + (b \gg 2)$
 - b. $(b > 0)$

OOP LAB MANUAL

- c. $(a + b * 100) / 10$
 - d. $a \& b$
 2. 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)
 3. Write a Java program to read an int number, double number, and a char from the keyboard and perform the following conversion: int to byte, char to int, double to byte, double to int.
 4. Perform multiplication and division by 2^n using shift operators.
 5. Write a Java program to find largest and smallest among 3 numbers using ternary operator.
 6. Write a Java program to execute the following statements. Observe and analyze the outputs
 - a. **boolean** x =**true**; b. **boolean** x =**true**;
 int y = x; **int** y =(int)x;
-

LAB NO.: 2

Date:

CONTROL STATEMENTS

Objectives: 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

2.1. 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) Simple 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 can 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.

v) Advanced Switch Statement

Enums in Java are a powerful feature used to represent a fixed set of constants. They can be used in switch statements for better type safety and readability.

Example: Use of Enum in Switch in Java

**// Java Program to Illustrate Use of Enum
// in Switch Statement**

// Class

```
public class GFG {
```

```
    // Enum
```

```
    public enum Day { Sun, Mon, Tue, Wed, Thu, Fri, Sat }
```

```
    // Main driver method
```

```
    public static void main(String args[])
    {
```

```
        // Enum
```

```
        Day[] DayNow = Day.values();
```

```
        // Iterating using for each loop
```

```
        for (Day Now : DayNow) {
```

```
            // Switch case
```

```
switch (Now) {

// Case 1
case Sun:
    System.out.println("Sunday");

    // break statement that halt further
    // execution once case is satisfied
    break;

// Case 2
case Mon:
    System.out.println("Monday");
    break;

// Case 3
case Tue:
    System.out.println("Tuesday");
    break;

// Case 4
case Wed:
    System.out.println("Wednesday");
    break;

// Case 5
case Thu:
    System.out.println("Thursday");
    break;

// Case 6
case Fri:
    System.out.println("Friday");
    break;

// Case 7
case Sat:
    System.out.println("Saturday");
}
}
```

```
}  
}
```

Use of Strings in Switch Statement

// Java Program to Demonstrate use of String to

// Control a Switch Statement

// Main class

```
public class GFG {
```

// Main driver method

```
public static void main(String[] args)
```

```
{
```

// Custom input string

```
String str = "two";
```

// Switch statement over above string

```
switch (str) {
```

// Case 1

```
case "one":
```

// Print statement corresponding case

```
System.out.println("one");
```

// break keyword terminates the

// code execution here itself

```
break;
```

// Case 2

```
case "two":
```

// Print statement corresponding case

```
System.out.println("two");
```

```
break;
```

// Case 3

```
case "three":
```

```

        // Print statement corresponding case
        System.out.println("three");
        break;

    // Case 4
    // Default case
    default:

        // Print statement corresponding case
        System.out.println("no match");
    }
}
}

```

2.2.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 eliminates 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:

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

2.3.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. A Taxi service offers a new service based on travel distance. Write a Java program to calculate the total distance traveled by considering the following charges. First 5 km = INR 10/km, Next 15 km = INR 8/km, Next 25 km = INR 5/km.
2. For given a 9-digit registration number of a CSE student, identify the year of joining. Assuming the first two digits specify the year of joining.
3. For a given date of birth of a person, calculate the date of retirement by taking years of service as input. (assume service periods as 60 years).

4. 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.]
5. Write a Java programs to print factorial of a given no recursively.
6. Write a Java program to compute the electricity bill for an industry using a switch-case statement. The program should take the daily consumption in units for 7 days as input. Based on the total consumption, the program should calculate and display the total electricity bill according to the following pricing table:

Units	Price per Unit (INR)
0 - 100	7.00
101 - 200	8.00
>= 201	10.00

Additional Exercises:

1. Write a program to check whether a number is palindrome or not.
2. Write a Java program to print table of number entered by user.
3. Write a Java program to display the numbers in the following format
 - using nested for loop.
 - using for-each loop.

```

1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
    
```

4. 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.)
5. 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] .
6. 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.
7. Write a Java programs to evaluate the following series
- $\text{Sin}(x) = x - (x^3/3!) + (x^5/5!)-\dots$
 - $\text{Sum} = 1 + (1/2)^2 + (1/3)^3 + \dots$

CLASSES, OBJECTS 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, objects and method overloading concepts of Java

3.1 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.

OOP LAB MANUAL

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

Volume is 3000

1.2 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;
```

OOP LAB MANUAL

```
        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
```

2.2 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 call. Thus, overloaded

OOP LAB MANUAL

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 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

OOP LAB MANUAL

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);
}
```

OOP LAB MANUAL

```
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. Define a Class STUDENT having following

Members: sname, marks_array, total, avg and provide the following methods:

- a. assign(): to assign initial values to the STUDENT object
- b. display(): to display the STUDENT object
- c. compute(): to Compute Total, Average marks

Write a Java program Illustrating Class Declarations, Definition, and Accessing Class Members to test the class defined.

2. Define a class EMPLOYEE having following members: Ename, Eid, Basic, DA, Gross_Sal, Net_Sal and following methods:
 - a. read(): to read N employee details
 - b. display(): to display employee details
 - c. compute_net_sal(): to compute net salary

Write a Java program to read data of N employee and compute and display net salary of each employee Note: (DA = 52% of Basic, gross_Sal = Basic + DA; IT = 30% of the gross salary)

3. Define a class Mixer to merge two sorted integer arrays in ascending order with

OOP LAB MANUAL

the following instance variables and methods:

instance variables:

```
int arr[]           //to store the elements of an array
```

Methods:

```
void accept()       // to accept the elements of the array in ascending order  
                    without any duplicates
```

```
Mixer mix(Mixer A) // to merge the current object array elements with the  
                    parameterized array elements and return the resultant object
```

```
void display()      // to display the elements of the  
array Define the main() method to test the class.
```

4. Create a Die class with one integer instance variable called sideUp. Give it a getSideUp() method that returns the values of sideUp and a void roll() method that changes sideUp to a random value from 1 to 6. Then create a DieDemo class with a method that creates two Die objects, rolls them, and prints the sum of the two sides up.

Additional Exercises

1. Create a class with integer array of size 10 and write methods to perform following. Use Switch case to accept choice from the user.
 - 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
2. The International Standard Book Number (ISBN) is a unique numeric book identifier which is printed on every book. The ISBN is based upon a 10-digit code. The ISBN is legal if:

$1 \times \text{digit1} + 2 \times \text{digit2} + 3 \times \text{digit3} + 4 \times \text{digit4} + 5 \times \text{digit5} + 6 \times \text{digit6} + 7 \times \text{digit7} + 8 \times \text{digit8} + 9 \times \text{digit9} + 10 \times \text{digit10}$ is divisible by 11.

example: For an ISBN 1401601499:

Sum = $1 \times 1 + 2 \times 4 + 3 \times 0 + 4 \times 1 + 5 \times 6 + 6 \times 0 + 7 \times 1 + 8 \times 4 + 9 \times 9 + 10 \times 9 = 253$ which is divisible by 11.

OOP LAB MANUAL

Write a program to implement the following methods:

inputISBN() to read the ISBN code as a 10-digit integer.

checkISBN() to perform the following check operations:

- a. If the ISBN is not a 10-digit integer, output the message “ISBN should be a 10 digit number” and terminate the program.
- b. If the number is 10-digit, extract the digits of the number and compute the sum as explained above. If the sum is divisible by 11, output the message, “Legal ISBN”; otherwise output the message, “Illegal ISBN”

CONSTRUCTORS AND STATIC MEMBERS

Objectives:

In this lab student will be able to:

- Utilize various types of constructors
- Overloading constructors
- Understanding static

4.1 Introduction:

4.1.1 Constructor:

A constructor initializes an object when it is created. It has the same name as that of class and has no return type (not even void). Constructors are utilized to give initial values to the instance variables defined by the class, or to perform any other start up procedures required to create a fully formed object. In general there are two different types of constructors:

1. Default
2. Parameterized

The following example shows how they are created and called.

Solved exercise

1. Program to illustrate default and parameterized constructors.

```
import java.util.*;

class
Student{ int
id;
    String name;
Student(){           //      zero      argument      constructor
    System.out.println("inside      default      constructor");
    System.out.println("the default values are "+id+"
"+name);

}
```

```

Student(int i,String n){           //           Parameterized
    constructor id = i;
    name = n;
    System.out.println("inside           parameterized
    constructor"); System.out.println("the values are
    "+id+" "+name);
}
public static void main(String args[]){
    Student s1 = new Student(); //calling default constructor
    Student s2 = new Student(111,"Karan"); //calling parameterized constructor
}
}

```

Output

4.2. Static variables and methods

Normally a class member must be accessed through an object of its class, but it is possible to create a member that can be used without reference to a specific instance. To create this type of member, precede its declaration with the keyword `static`. When a member is declared `static`, it can be accessed before any objects of its class are created without reference to any object. Both methods and variables can be declared as `static`.

Following example illustrates the usage of static variable and static method:

```

class StaticMeth{
    static           int
    val=1024;
    static           int
    valDiv2(){
        return val/2;
    }
}
class SDemo2{
    public static void main(String[] args){
        System.out.println("val           is

```

```

        "+StaticMeth.val);
        System.out.println("StaticMeth.valDiv2()           :
        "+StaticMeth.valDiv2()); StaticMeth.val=4;
        System.out.println("val           is           "+StaticMeth.val);
        System.out.println("StaticMeth.valDiv2()           :
        "+StaticMeth.valDiv2());
    }
}

```

Output

```

val is 1024
StaticMeth.valDiv2() : 512
val is 4
StaticMeth.valDiv2() : 2

```

Since the method `valDiv2()` is declared static, it can be called without any instance of its class `StaticMeth` being created, but by using class name.

Lab Exercises:

1. Consider the already defined `STUDENT` class. Provide a default constructor and parameterized constructor to this class. Also provide a display method. Illustrate all the constructors as well as the display method by defining `STUDENT` objects.
2. Consider the already defined `EMPLOYEE` class. Provide a default constructor, and parameterized constructor. Also provide a display method. Illustrate all the constructors as well as the display method by defining `EMPLOYEE` objects.
3. Define a class to represent a **Bank account**. Include the following members.

Data members:

- a. Name of the depositor
- b. Account number.
- c. Type of account.
- d. Balance amount in the account.
- e. Rate of interest (static data)

Provide a default constructor and parameterized constructor to this class. Also provide Methods:

- a. To deposit amount.
- b. To withdraw amount after checking for minimum balance.

- c. To display all the details of an account holder.
- d. Display rate of interest (a static method)

Illustrate all the constructors as well as all the methods by defining objects.

- 4. Create a class called Counter that contains a static data member to count the number of Counter objects being created. Also define a static member function called showCount() which displays the number of objects created at any given point of time. Illustrate this.

Additional Exercises

- 1. Define a class **IntArr** which hosts an array of integers. Provide the following methods:
 - 1. A *default constructor*.
 - 2. A *parameterized constructor* which initializes the array of the object.
 - 3. A method called *display* to display the array contents.
 - 4. A method called *search* to search for an element in the array.
 - 5. A method called *compare* which compares 2 **IntArr** objects for equality.
- 2. Define a class called Customer that holds private fields for a customer ID number, name and credit limit. Include appropriate constructors to initialize the instance variables of the Customer Class. Write a main() method that declares an array of 5 Customer objects. Prompt the user for values for each Customer, and display all 5 Customer objects.

NESTED & INNER CLASSES

Objectives:

By the end of this experiment, students will:

- Understand the concepts of nested and inner classes and their practical applications.
- Differentiate between static nested classes and non-static inner classes.
- Implement and utilize nested classes (both static and non-static) for logical grouping and encapsulation.
- Apply inner classes in scenarios requiring tight coupling with an enclosing class instance.

5.1 Introduction:

5.1.1. Nested Classes

In Java, a **nested class** is any class defined inside another class. Nested classes are a way to logically group classes that are only used in one place. Nested classes improve encapsulation and readability, allowing related functionality to be tightly coupled.

Syntax:

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

Types of Nested Classes

1. Static Nested Classes:

- Declared with the **static** modifier.
- Do not have access to instance members (non-static members) of the enclosing class.
- Can directly access the static members of the outer class.

2. Non-static Inner Classes:

- A nested class without the **static** modifier.
- Have access to all members (static and non-static) of the enclosing class.

OOP LAB MANUAL

- Require an instance of the outer class to be instantiated.

Example 1:

```
class Bank {
    private static String bankName = "Global Bank";

    static class Account {
        private String accountNumber;
        private double balance;

        Account(String accountNumber, double balance) {
            this.accountNumber = accountNumber;
            this.balance = balance;
        }

        void displayAccountDetails() {
            System.out.println("Bank Name: " + bankName);
            System.out.println("Account Number: " + accountNumber);
            System.out.println("Balance: $" + balance);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Bank.Account account = new Bank.Account("12345", 2500.75);
        account.displayAccountDetails();
    }
}
```

Output:

```
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Bank Name: Global Bank
Account Number: 12345
Balance: $2500.75
Press any key to continue . . .
```

Example 2:

```
class University {
```

OOP LAB MANUAL

```
private String universityName;

University(String universityName) {
    this.universityName = universityName;
}

class Department {
    private String departmentName;

    Department(String departmentName) {
        this.departmentName = departmentName;
    }

    void displayDetails() {
        System.out.println("University: " + universityName);
        System.out.println("Department: " + departmentName);
    }
}

public class Main {
    public static void main(String[] args) {
        University university = new University("MAHE");
        University.Department department = university.new Department("Computer
Science");
        department.displayDetails();
    }
}
```

Output:

```
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
University: Manipal University
Department: Computer Science
Press any key to continue . . .
```

5.1.2. Inner Classes

An **inner class** in Java is a class defined within another class. Inner classes are primarily used for logically grouping classes that are only used in one place, improving encapsulation and readability.

Syntax:

```
class OuterClass {
```

```

...
static class StaticNestedClass {
    ...
}
class InnerClass {
    ...
}
}

```

Types of inner classes include:

1. **Non-static inner classes:** Have access to the enclosing class's instance members and methods.
2. **Local inner classes:** Defined within a method or block; they exist only within that scope.
3. **Anonymous inner classes:** Declared and instantiated simultaneously, often used for event handling or functional interface implementations.

Difference Between Inner and Nested Classes

Feature	Inner Class	Nested Class (Static Nested Class)
Definition	Non-static class inside another class.	Static class inside another class.
Association with Outer	Requires an instance of the outer class.	Does not require an instance of the outer class.
Access to Outer Class	Can access all outer class members, including private ones.	Can only access static members of the outer class.
Use Case	To model "has-a" relationships.	Used for helper or utility classes.

Example 1: Non-static Inner Class

```

class Library {
    private String name;
    private String address;

    Library(String name, String address) {
        this.name = name;
        this.address = address;
    }

    void displayLibraryDetails() {
        System.out.println("Library Name: " + name);
        System.out.println("Library Address: " + address);
    }
}

```

OOP LAB MANUAL

```
}

class Book {
    private String title;
    private String author;

    Book(String title, String author) {
        this.title = title;
        this.author = author;
    }

    void displayBookDetails() {
        System.out.println("Book Title: " + title);
        System.out.println("Author: " + author);
    }
}

public class Main {
    public static void main(String[] args) {
        Library library = new Library("Central Library", "Downtown");
        Library.Book book = library.new Book("Java Programming", "John Doe");

        library.displayLibraryDetails();
        book.displayBookDetails();
    }
}
```

Output:

```
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Library Name: Central Library
Library Address: Downtown
Book Title: Java Programming
Author: John Doe
Press any key to continue . . .
```

Example 2: Anonymous Inner Class

```
interface Greeting {
    void sayHello();
}
```

```
public class Main {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() { // Anonymous Inner Class
            @Override
            public void sayHello() {
                System.out.println("Hello, welcome to the Java world!");
            }
        };
        greeting.sayHello();
    }
}
```

Output:

```
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Hello, welcome to the Java world!
Press any key to continue . . .
```

Example 3: Local Inner Class

```
class Calculator {
    void calculateSquare(int number) {
        class Square {
            int getSquare() {
                return number * number;
            }
        }
        Square square = new Square();
        System.out.println("Square of " + number + " is: " + square.getSquare());
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.calculateSquare(5);
    }
}
```

Output:

```
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Hello, welcome to the Java world!
Press any key to continue . . .
```

Lab Exercises:

1. Student Class

Enhance the STUDENT class by adding an **inner class** named Subject that handles details of individual subjects. Where:

- a) The **inner class Subject** should:
 - a. Contain fields for subjectName and marks.
 - b. Provide methods to assign marks and display subject details.
- b) The STUDENT class should maintain an array of Subject objects.
- c) The STUDENT class should:
 - a. Provide methods to add subjects.
- d) Calculate total and average marks by iterating over the Subject objects.

2. Employee Class

Implement an EMPLOYEE class with an **inner class** named Department that handles department-related details. **The inner class Department should:**

- a. Contain fields for departmentName and location.
- b. Provide methods to set and display department details.

The EMPLOYEE class should:

Contain fields for eName (employee name), salary, and an array of Department objects.

Provide methods to:

- c) Add departments.
- d) Display employee details along with department information.

3. Implement a ShoppingCart class that contains an **inner class** Item.

- a) The Item class should have fields like itemName, quantity, and price.
- b) The ShoppingCart class should provide methods to add items, calculate the total price, and display the cart contents.

4. Design a Weather class with a **static nested class** Forecast.

- a) The Forecast class should predict weather conditions (Sunny, Rainy, Cloudy) based on input data like temperature and humidity.

- b) Use the nested class to predict and display the weather forecast for different cities.

Additional Exercises:

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

<pre> 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("Strin g stored in str is- "+" X.str); } } </pre>	<pre> 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); } } </pre>	<pre> 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); } } </pre>
<pre> class Example4 { static int num; static String mystr; //First Static block static { System.out.println("Static Block 1"); num = 68; mystr = "Block1"; } //Second static block static { System.out.println("Static Block 2"); num = 98; mystr = "Block2"; } public static void main(String args[]) { </pre>	<pre> 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); } } </pre>	<pre> 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 display(); } } </pre>

OOP LAB MANUAL

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

STRING, ARRAYS, ARRAY OF OBJECTS

Objectives:

1. Know different ways of creating String objects and constants
2. Learn and use string handling methods
3. Know the difference between String and StringBuffer classes

6.1 Basics of String Handling using String and StringBuffer classes

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, the character sequence that make up the string cannot be replaced, deleted or extended. Hence, String objects are immutable. 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)

Table 9.1 lists the methods of String Class.

Table 9.1: Methods of String Class

Modifier and Type	Method and Description
char	<u>charAt</u> (int index) Returns the char value at the specified index.
int	<u>codePointAt</u> (int index) Returns the character (Unicode code point) at the specified index.
int	<u>compareTo</u> (<u>String</u> anotherString) Compares two strings lexicographically.
int	<u>compareToIgnoreCase</u> (<u>String</u> str) Compares two strings lexicographically, ignoring case differences.
<u>String</u>	<u>concat</u> (<u>String</u> str) Concatenates the specified string to the end of this string.
boolean	<u>contains</u> (<u>CharSequence</u> s) Returns true if and only if this string contains the specified sequence of char values.
boolean	<u>endsWith</u> (<u>String</u> suffix) Tests if this string ends with the specified suffix.
boolean	<u>equals</u> (<u>Object</u> anObject) Compares this string to the specified object.
boolean	<u>equalsIgnoreCase</u> (<u>String</u> 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.

```
// 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

Lab Exercises:

1. Write a menu driven program to do the following
 - i. To check whether a string is palindrome or not
 - ii. Write the string in an alphabetical order
 - iii. Reverse the string
 - iv. Concatenate the original string and the reversed string
2. To the already defined Employee class, add the following string processing methods:
 - i. `formatEmployeeName()`: A method that formats the employee's name by capitalizing the first letter of each word and converting the remaining letters to lowercase. For example, if the employee's name is "JOHN DOE", this method would transform it to "John Doe".
 - ii. `generateEmail()`: A method that generates an email address for the employee based on their name. For example, if the employee's name is "John Doe", this method would generate an email address like jdoe@example.com.

Illustrate the above methods upon creating an array of Employee objects. The details of each Employee object must be read from the user and initialized using the parameterised constructor.

3. To the already defined Student class, add the following methods:
 - i. `extractInitials()`: A method that extracts the initials from the student's name. For example, if the student's name is "John Doe", this method would return "JD".
 - ii. `removeWhitespace()`: A method that removes any whitespace characters from the student's name. For example, if the student's name is "John Doe", this method would transform it to "JohnDoe".
 - iii. List all the student names containing a particular substring.

iv. Sort the students alphabetically

In a separate class that contains main(), read the students details from console input. After displaying the details of all the students, illustrate the use of the above methods.

Additional Exercises:

1. Create a class ServiceRequest with two String fields, namely, request (that represents a verbal complaint such as “Streetlights are not working”) and sr_date (that records the date on which the complaint was lodged). Assume any date is of the form DD-MMM-YEAR (E.g., 01-NOV-2016 or 15-AUG-2000). Create an array of service requests to store complaints lodged by citizens on different occasions. After displaying all the requests entered, list all the service requests in month-wise manner using displayByMonth(ServiceRequest[] sr), a static method defined inside the ServiceRequest class.

INHERITANCE AND METHOD OVERRIDING

Objectives:

1. Understand the basics of Inheritance
2. Use *super* keyword to access super class members and constructors
3. Demonstrate dynamic polymorphism by overriding methods

Inheritance

Java supports inheritance by allowing one class to incorporate another class into its declaration. This is done using the *extends* keyword. Thus, the subclass adds (extends) to the superclass.

Example Program:

Write a program that creates a superclass called TwoDShape which stores the width and height of a two-dimensional object, and a subclass called Triangle extends from it. Create different Triangle objects in main() and display their details.

```
class TwoDShape{
    double width, height;
    TwoDShape(double h, double w){
width = w;
hieght = h;
    }
    void show(){
        System.out.println("width and height are "+width+" and "+height);
    }
}
class Triangle extends TwoDShape{
    String style;
    Triangle(String s, double h, double w){
super(h, w);
style=s;
    }
    double area(){
        return width*height/2;
    }
}
//show () method here overrides the one in TwoDShape class which is the base class
void show() {
```

```

        super.show();
        System.out.println("Triangle is" + style ); }
    }
}
class inheritanceEx{
    public static void main(String[] args){
        Triangle t1=new Triangle("outlined",8.0,12.0);
        Triangle t2=new Triangle("filled", 4.0,4.0);
        System.out.println("Details of triangle, T1:");
        t1.show();
        System.out.println("Details of triangle, T1:");
        t2.show();
        System.out.println("Area of triangle T1:"+t1.area());
        System.out.println("Details of triangle, T2:"+t2.area());
    }
}

```

Lab Exercises

1. To the already defined STUDENT class, add two subclasses ScienceStudent and ArtsStudent and implement the following:
 - i) Add a data member practicalMarks (int) to the ScienceStudent class that represents the marks obtained by the student in the laboratory subject. The ScienceStudent class should override the compute() method to include the practical marks in the total marks and average marks calculation. Additionally, the ScienceStudent class should provide a method displayPracticalMarks() to display the practical marks obtained by the science student.
 - ii) Add a data member electiveSubject (String): to the ArtsStudent class that represents the elective subject chosen by the arts student. Also, add appropriate constructors to the subclasses.

In main(), create objects of STUDENT, ScienceStudent, and ArtsStudent, and demonstrate the keyword 'super' and other functionalities of the classes by assigning values, computing marks, and displaying the information of the students. Also, demonstrate dynamic polymorphism.

2. To the already defined EMPLOYEE class, add two subclasses FullTimeEmp and PartTimeEmp and implement the following:
 - i) Include the following data members to the PartTimeEmp class-hoursWorked (int) that represents the number of hours worked by the part-time employee and hourlyRate (double, static and final) that represents the hourly rate at which the part-time employee is paid. Also, override calculateSalary() and displayEmployeeDetails() splayEmployeeDetails() method of the base class to display the part-time employee's details, including the hours worked and hourly rate.

ii) The FullTimeEmployee subclass includes the data members bonus and deductions as additional data members and are of type double and overrides the calculateSalary() and displayEmployeeDetails() methods to incorporate these factors.

In main(), create various objects to illustrate the functionality of all the classes.

3. Create a Building class with two subclasses namely, House and School. The Building class contains fields for square footage and stories. The House class contains additional fields for number of bedrooms and baths. The School class contains additional fields for number of classrooms and grade level (for example, elementary or junior high). All the classes contain appropriate overloaded constructors and methods to display the details. In a separate class that contains main(), demonstrate the working of this hierarchy.
4. 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 to achieve following tasks
 - i) Accept the deposit from a customer and update the balance
 - ii) Display the balance
 - iii) Compute and deposit interest
 - iv) Permit withdraw and update the balance
 - v) Check for minimum balance impose penalty if necessary and update the balance

For savings bank account, the facilities provided include computing interest and withdrawal. No interest can be computed on current bank account and a minimum balance must always be maintained. In any instance when it goes below this level, service tax must be imposed.

Additional Exercises:

1. Design a base class called Student with the following 2 fields:- (i) Name (ii) Id. Derive 2 classes called Sports and Exam from the Student base class. Class Sports has a field called s_grade and class Exam has a field called e_grade which are integer fields. Derive a class called Results which inherit from Sports and Exam. This class has a character array or string field to represent the final result. Also it has a method called display which can be used to display the result. Illustrate the usage of these classes in main().
2. Create a class Book with data members title and ISBN. This class is inherited by Journal and Novel with subject and genre as their respective data members. Both the subclasses must implement compare(Book b1, Book b2) that returns true if two journals are from the same subject area/if two novels are of the same genre. Create an array of journals and novels in a separate class that contains main(). After displaying the journal and novel details on the console output, compare any two journals and any two novels using the compare() method and display the result.

ABSTRACT CLASSES & INTERFACES**Objectives:**

1. To differentiate between a concrete class and an abstract class
2. To understand the fundamentals of interfaces
3. To demonstrate run time polymorphism using interface references

8.1 Abstract class

Any class that contains one or more abstract methods is deemed to be abstract. An abstract method is one whose concrete implementation can not be defined inside the class. The method as well as class can use abstract keyword in their definition as follows:

```
abstract class class_name{  
    //data members  
    return_type method1(){  
        //implementation  
    }  
    return_type method2(){  
        //implementation  
    }  
    abstract return_type method_name(param_list);  
    //other methods  
}
```

Note that an abstract method does not have method body. It must be mandatorily overridden by every subclass of the abstract class. These subclasses upon overriding, provide a concrete implementation.

Example program:

```
    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

An interface is a blueprint of a class. It has static constants and abstract methods only. There are mainly three reasons to use interface. They are given below.

- i) It is used to achieve absolute abstraction.
- ii) Interface can be used to achieve the functionality of multiple inheritances.
- iii) It can be used to achieve loose coupling.

Example Program:

```

interface Printable{ void print(); }
interface Showable{ void show(); }
class A implements Printable,Showable{
    public void print(){
        System.out.println("Hello");
    }
    public void show(){
        System.out.println("Welcome");}
}

class InterfaceDemo{
    public static void main(String args[]){
        A obj = new A();
        obj.print();
        obj.show();
    }
}

```

Lab Exercises

1. Create an abstract class Figure with an abstract method area and two integer variables that represent x and y dimensions. Create three more classes Rectangle, Triangle and Square that extend Figure and implement the area method appropriately. Illustrate how the method area can be computed at run time for Rectangle, Square and Triangle to achieve dynamic polymorphism.
2. Design an interface called Series with the following methods
 - i) getNext (returns the next number in series)
 - ii) reset(to restart the series)
 - iii) setStart (to set the value from which the series should start)Design a class named ByTwos that implements Series such that it generates a series of numbers, each two greater than the previous one. Also design a class which will include the main method for referencing the interface.
3. Design a class Student with the methods, getRollNum() and putRollNum() to read and display the Roll No. of each student; and getMarks() and putMarks() to read and display their marks. Create an interface called Sports with a method putSportsScore() that will set the score obtained by a student in physical education examination. Design a class called Result that will implement this interface to generate the result by considering both the marks and sports score.

Additional Exercises

1. Create a class Phone{string brand, int memCapacity}, which contains an interface (Nested interface) Callable{makeAudioCall(string cellNum), makeVideoCall(string cellNum)}. Create subclasses BasicPhone and SmartPhone and implement the methods appropriately. Demonstrate the creation of both subclass objects by calling appropriate constructors which accept values from the user. Using these objects call the methods of the interface.
2. Create an interface InfInterest with rate of interest, r and calculateInterest() as its members. Implement two classes SimpleInterest and CompoundInterest that suitably implement InfInterest. Provide parameterized constructor to initialize the data members of the classes, if any. Also, create a class InterestDemo that determines either of the interest amounts (based on user's choice) by invoking calculateInterest() only once.

PACKAGE & ACCESS MODIFIERS

Objectives:

In this lab students will be able to:

1. Know the purpose and creation of packages
2. Know how to import packages and how packages affect access
3. To know the different access specifiers used with the class
4. To write Java programs using the concepts of access modifiers

9.1: Introduction to Packages:

While naming a class, the name chosen for a class is unique and does not collide with class names chosen by other programmers. Packages are containers for classes used to keep the class name space compartmentalized. The package has both a naming and a visibility control mechanism. It is possible to define classes inside a package that are not accessible bytecode outside that package.

Simple example of java package:

The package keyword is used to create a package in java.

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

To compile java package:

Give the command with the following format in the terminal:

```
javac -d directory javafilename
```

For example, *javac -d . Simple.java*

The -d switch specifies the destination where to put the generated class file. Any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. can be used. To keep the package within the same directory, use .(dot).

To run java package program:

Use fully qualified name to run the class.

e.g., *Java mypack.Simple*

Output: Welcome to package

To access a package from another package:

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

- `import package.*;`
- `import package.classname;`
- fully qualified name

Using `packagename.*`

If `package.*` is used, then all the classes and interfaces of this package will be accessible but not subpackages. The `import` keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the `packagename.*`

```
//save by A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");
    }
}

//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output: Hello

Using `packagename.classname`

If `package.classname` is imported, then only declared class of this package will be accessible.

Example of package by import `package.classname`

```
//save by A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");}
}

//save by B.java
```

```

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

```

Output: Hello

Using a fully qualified name

If the fully qualified name is used then only the declared class of this package will be accessible. Now there is no need to import. But the fully qualified name must be used every time while accessing the class or interface.

It is used when two packages have the same class name e.g., java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```

//save by A.java
package pack;
public class A{
    public void msg(){
        System.out.println("Hello");
    }
}
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.Aobj = new pack.A();//using fully qualified name obj.msg();
    }
}

```

Output: Hello

9.2: Access Modifiers:

Java provides four types of access modifiers to control the visibility and accessibility of classes, methods, variables, and constructors. These modifiers are default, private, protected, and public.

Default Access Modifier:

If no access modifier is specified, Java applies the default access modifier, accessible only within the same package.

Private Access Modifier:

The private modifier makes members accessible only within the class where they are declared.

Protected Access Modifier:

The protected modifier allows access within the same package and to subclasses in other packages.

Members with protected access can be accessed in subclasses outside their package, but not directly through an object of the parent class.

Public Access Modifier:

The public modifier makes members accessible from any class, package, or subclass.

//SuperClass.java

```
package packA;
public class SuperClass {
    void defaultMethod() {
        privateMethod(); // can access
        System.out.println("inside defaultMethod!");
    }
    private void privateMethod() {
        System.out.println("inside privateMethod!");
    }
    protected void protectedMethod() {
        System.out.println("inside protectedMethod!");
    }
    public void publicMethod() {
        System.out.println("inside publicMethod!");
    }
}
```

//SubClass.java

```
package packA;
public class SubClass extends SuperClass {
    public static void main(String[] args) {
        SuperClass superObj = new SuperClass();
        superObj.privateMethod(); // cannot access
        superObj.defaultMethod(); // can access
        superObj.protectedMethod(); // can access
        superObj.publicMethod(); // can access
        SubClass myObj = new SubClass();
        myObj.privateMethod(); // cannot access
        myObj.defaultMethod(); // can access
        myObj.protectedMethod(); // can access
        myObj.publicMethod(); // can access
    }
```



```

    }
}
//OtherClass.java
package packA;
public class OtherClass {
    public static void main(String[] args) {
        SuperClass superObj = new SuperClass();
        superObj.privateMethod(); // cannot access
        superObj.defaultMethod(); // can access
        superObj.protectedMethod(); // can access
        superObj.publicMethod(); // can access
    }
}

//SubClass.java
package packB;
import packA.SuperClass;
public class SubClass extends SuperClass {
    public static void main(String[] args) {
        SuperClass superObj = new SuperClass();
        superObj.privateMethod(); // cannot access
        superObj.defaultMethod(); // cannot access
        superObj.protectedMethod(); // cannot access
        superObj.publicMethod(); // can access
        SubClass myObj = new SubClass();
        myObj.privateMethod(); // cannot access
        myObj.defaultMethod(); // cannot access
        myObj.protectedMethod(); // can access
        myObj.publicMethod(); // can access
    }
}

//OtherClass.java
package packB;
import packA.SuperClass;
public class OtherClass {
    public static void main(String[] args) {
        SuperClass superObj = new SuperClass();
        superObj.privateMethod(); // cannot access
        superObj.defaultMethod(); // cannot access
        superObj.protectedMethod(); // cannot access
        superObj.publicMethod(); // can access
    }
}

```

Modifier	Same Class	Subclass in same package	Another class in same package	Subclass in another package	Another class in another package
Private	Yes	No	No	No	No
Default	Yes	Yes	Yes	No	No
Protected	Yes	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes	Yes

Lab Exercises:

Lab Exercise 1: Student Management System

Problem Statement: Develop a Java application to manage student data, incorporating the principles of packages and access modifiers. The application should consist of the following components.

Student Class: Should be inside the edu.manipal.mit.student package.

Attributes:

name (String, private): Student's name.

rollNumber (int, private): Unique roll number.

department (String, protected): Student's department (e.g., CSE, AI/ML, FinTech etc.).

gpa (double, public): Grade point average.

Course Class: Should be inside the edu.manipal.mit.course package.

Attributes:

courseName (String, private): Name of the course.

instructor (String, protected): Course instructor.

credits (int, public): Number of credits.

Registration Class: Should be inside the edu.manipal.mit.registration package.

Attributes:

student (Student, private): Student object

course (Course, private): Course object

Methods:

registerStudent() (public): this method is used to register the student to the course. Print student and course information in System.out

Main Class: Should be inside the edu.manipal.mit.main package

Test the application by creating student , course and registration objects and registering the student to the course.

Requirements:

- Utilize appropriate access modifiers (private, protected, public) for attributes and methods to control visibility and access.
- Implement necessary constructors, getter methods for all classes as required

Lab Exercise 2: Employee Management System

Problem Statement:

Develop a Java application for managing employee data within a company, demonstrating the use of packages and access modifiers. The application should consist of the following components:

Employee Class: Located in the edu.manipal.mahe.employee package.

Attributes:

employeeId (int, private): Unique employee ID.

name (String, private): Employee's name.

age (int, protected): Employee's age.

basicSalary (double, public): Employee's basic salary.

Department Class: Located in the edu.manipal.mahe.department package.

Attributes:

departmentName (String, private): Name of the department.

manager (String, protected): Department manager.

location (String, public): Department location.

Payroll Class: Located in the edu.manipal.mahe.payroll package.

Attributes:

employee (Employee, private): Employee object

department (Department, private): Department object

Methods:

- calculateNetSalary() (private): Calculates an employee's net salary as per following calculation.

DA = 52% of Basic Salary, Gross Salary= Basic Salary + DA, IT = 30% of the Gross Salary, Net Salary = (Gross Salary – IT)

- generatePaySlip() (public): Generates an employee's pay slip showing following output (for basic salary of 50000.0)

Payslip for January 2025

Employee name: Arun

Employee age: 53200.0

Employee department name: IT Department

Employee Net Salary : 53200.0

Main Class: Should be inside the edu.manipal.mahe.main package

Test the application by creating employee, department and payroll objects, calculating net salaries, and generating pay slips.

Requirements:

- Utilize appropriate access modifiers (private, protected, public) for attributes and methods to control visibility and access.
- Demonstrate the use of packages to organize classes logically.
- Implement necessary constructors, getter methods for all classes as required

Additional Exercises:

1. Create a package edu.manipal.mit. Create a Student class to store their name and roll number, define a method to display these details. Both variables and method should use the default access modifier. Create another class DefaultAccessDemo in the same package to set the student details and display them. Create a new package edu.manipal.kmc. Demonstrate that the default access modifier restricts access when you

try to use the Student class from a different package and note the compilation errors.

2. Update the Student class mark the attributes name and rollNumber as private, and student details as public. Create public setter methods to assign values to name and rollNumber. Create a PrivateAccessDemo class in the same package, that uses the setter methods to set the values for name and rollNumber and displays the details. Try to access the name and rollNumber attributes directly from the PrivateAccessDemo class and note the errors.
3. Create a Department class to store department details, add departmentName(protected) attribute, departmentCode(public). Add two methods to set and display department details. Create a Course class in the same package that inherits from the Department, add private attributes courseName and courseDuration to store course specific information. Add public methods to set course details and display course with department details. Create a Subject class in a different package:
Store subjectName and subjectCredits as private attributes.
Provide public methods to set and display subject details.
Implement a ProtectedAndPublicDemo class in another package to:
Create a Course object and set course details, including department information & display this information.
Create a Subject object and set subject details and display them.
Try to directly access departmentName (protected) in ProtectedAndPublicDemo and note down errors and explain why it is restricted.
Attempt to directly access private attributes of Subject in ProtectedAndPublicDemo and note down the error.

Exercise 4:

Create a Employee class that should serve as the base class, containing details like name and employeeId. Use the protected modifier to allow access to these details in subclasses. Add 2 public methods setEmployeeDetails , displayEmployeeDetails to set and display the details.

Create a subclass PermanentEmployee to store additional information like salary(private). This class shall have setSalary and displayPermanentEmployeeDetails(display salary and employee information) methods with public access modifiers.

Create a subclass ContractEmployee to store information like contractDuration (in months). Add two methods setContractDuration , displayContractEmployeeDetails(to display contractDuration and employee information) information with public access modifiers.

Create ProtectedAccessDemo class in a separate package.

Add main method, create instances of PermanentEmployee and ContractEmployee, setting and displaying their details.

Test and write down your answers for following:

Try to access the name and employeeId attributes of PermanentEmployee object and

document your observations.

Can you access and display `employeeId` inside the `displayPermanentEmployeeDetails` method of `PermanentEmployee` class? Explain the reasons

LAB NO.: 10

Date:

EXCEPTION HANDLING

Objectives:

In this lab, students will be able to:

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.

Theory Related to the lab 10.1 10.2

10.1 Introduction to 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 Runtime Exception 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.
IllegalStateException	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.
StringIndexOutOfBoundsException	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	The requested field does not exist.
NoSuchMethodException	The requested method does not exist.

10.3 Simple Java program that handles exception

```
//Program which handles Arithmetic Exception (division by 0)\nimport java.util.Scanner;\nclass 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 to validate the age of a student during their registration. If the age is not between 18 and 60, throw an `IllegalArgumentException`. Create Student class with Private attributes name and age. Add method `registerStudent(String name, int age)` that throws an `IllegalArgumentException` if the age is invalid (that is, not between 18 and 60) Write `StudentAgeValidationDemo` class to create instance of student class and invoke `registerStudent` method with valid and invalid data Catch the exception and display an error message for invalid input.
2. Write a program to manage course enrollment. If a course exceeds its capacity, throw a custom checked exception called `CourseFullException`. Create a `Course` class with

private attributes `courseName`, `capacity`, and `enrolledStudents`. A method `enrollStudent()` that throws `CourseFullException` if the course is already full. Create a custom checked exception `CourseFullException`. Write a `CourseCapacityDemo` class to enroll students in a course, Handle the `CourseFullException` when the course is full.

- Write a program to calculate the average marks of a student. If the total marks are zero or the number of subjects is zero, throw an `ArithmeticException` to avoid division by zero. Create a `Student` class , add a method `calculateAverage(int totalMarks, int numberOfSubjects)` that throws `ArithmeticException` if `numberOfSubjects` is zero. Write a `MarksValidationDemo` class to invoke the `calculateAverage` method with valid and invalid data. Handle the exception and display an appropriate error message.

Additional Exercises:

- Throw an `InvalidSalaryException` if the salary of an employee is less than the minimum wage (50,000). Throw a `ContractLimitExceededException` if a contract employee's work duration exceeds the maximum allowed period (24 months). Create these exception classes by extending `Exception`.
Create `Employee` base class with attributes `name` and `salary` Add a method `setSalary(double salary)` that throws `InvalidSalaryException` if the salary is below 50,000. Provide methods to display employee details. Create `ContractEmployee` inherits `Employee` class. Add an attribute `contractDuration` (in months). Add a method `setContractDuration(int months)` that throws `ContractLimitExceededException` if the duration exceeds 24 months. Provide methods to display contract details. Create `EmployeeExceptionDemo` class with main method. Also, demonstrate the working of the custom exceptions using try-catch blocks.

REFERENCES:

1	Schildt H., <i>Java-The Complete Reference</i> , 12 th Edition, Tata McGraw-Hill 2020.
2	Schildt H., & Skrien D., <i>Java Programming A Comprehensive Introduction</i> , 1 st Edition, McGraw-Hill 2013
3	Horstmann C. S. & Cornell G., <i>Core Java Volume I – Fundamentals</i> , 13 th Edition, Oracle 2024.
4	Horstmann C. S. & Cornell G., <i>Core Java Volume II – advanced Java</i> , 13 th Edition, Oracle 2024.
5	https://onlinecourses.nptel.ac.in/noc20_cs58/preview