Module : Core Java 8

Module Overview

In this module, students will be able to familiarize with Java and Core Java. Apart from above, they will be able to understand Java Architecture and Advantages of the same.

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

- Implementing OOPs features in Java
- Developing Java script
- Use of JDK, eclipse
- Testing using Junit
- Implementing Multithreading

Introduction to JAVA programming

Java Introduction

- Java is one of the most used programming languages.

- Java is a programming language and a computing platform for application development.

- Java was developed by **James Gosling, Patrick Naughton, Chris Warth, Ed Frank,** and **Mike Sheridan** at Sun Microsystems, Inc. in 1991.

- Originally, it was called '**Oak**' by James Gosling (one of the developers), then it was renamed as '**Java**', 1995.

- It was first released by Sun Microsystem in 1995 and later acquired by Oracle Corporation.

- The most striking feature of the language is that it is a platform-neutral language.

- Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Introduction to JAVA programming

- Java is the first programming language that is not tied to any particular hardware or operating system.

- Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language.

- Java follows the principle of **WORA (Write Once, Run Anywhere)** and is platform-independent.

- **Simple syntax** to print "Hello world" in JAVA

```java
public class MyFirstJavaProgram {

    /* This is my first java program.
     * This will print 'Hello World' as the output
     */

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

JAVA Architecture

- As one of the main advantages of Java is that it is platform-independent.

- There are three main components of Java architecture

    1. JVM

    2. JRE

    3. JDK

- In Java, **JVM** (Java Virtual Machine) is a component that provides an environment for running Java Programs.

- At the runtime, JVM interprets the bytecode into machine code which will be executed the machine in which the Java program runs.
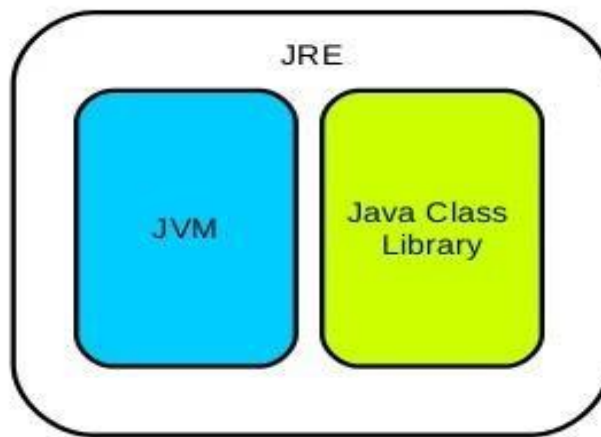
Java Component- JVM

- Java combines both the approaches of compilation and interpretation.

- First, java compiler compiles the source code into bytecode.

- It is very important as a developer that we should know the Architecture of the JVM, as it enables us to write code more efficiently.

- Every Java developer knows that bytecode will be executed by JRE (Java Runtime Environment).

- But many don't know the fact that JRE is the implementation of Java Virtual Machine (JVM), which analyzes the bytecode, interprets the code, and executes it.

- JVM stands for Java Virtual Machine, which is used to interpret an entire program line by line.

- This can handle the run time exception.

- Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems.

- That is why we call java as platform independent language.

- JVM has the following tasks:

    - Load code

    - Verify code

    - Execute code

Java Component- JRE

- JRE stands for Java Runtime Environment.

- It plays a key role while executing any java application.

- It is a collection of tools that together allow the development of applications and provide a runtime environment.

- The JVM is a part of JRE.

- This is like JVM, platform-dependent.

Java Component- JDK

- JDK stands for Java Development Kit.

- It includes Development Tools to provide an environment to develop your Java programs and JRE to execute your java code.

- In order to create, compile and run Java program you would need JDK installed on your computer.

Java Platform Editions

- **Java Platform Editions includes :**

    - **Java Micro Edition :** It is useful for developing small devices like mobile phones.

    - **Java Standard Edition :** It is useful for GUI, database access, networking, and security.

    - **Java Enterprise Edition :** It is an enterprise platform which is mainly used to develop web and enterprise applications.

Applications of Java

- Below are a few applications that can be created using Java programming:

    Standalone Application:-

    - Standalone applications are also known as desktop applications.
    - These are traditional software that we need to install on every machine. **Examples** are Media player, antivirus, etc.
    - AWT (Abstract Window Toolkit) and Swing are used in Java for creating tandalone applications.

    Web Application:-

    - An application that runs on the server side and creates a dynamic page is called a web application.
    - Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.
    -

    Enterprise Application:-

- An application that is distributed in nature, such as banking applications, etc. is called enterprise application.

- It has advantages of the high-level security, load balancing, and clustering.
- In Java, EJB is used for creating enterprise applications.

Mobile Application:-

- An application that is created for mobile devices is called a mobile application.
- Currently, Android and Java ME are used for creating mobile applications.
- Android framework is closely associated with Java.

Real-World Applications of Java

- An application developed by **NASA** using Java called **World Wind** allows you to zoom into the outer space. Also, you can explore any location on earth.

- An open-source framework associated with big data called **Hadoop** is written in Java.

- By mingling the science of Robotics with Java programming and Artificial Intelligence, **self-driving cars** are being developed.

Popular Java Editors

- To write Java programs, we will need a text editor.

- There are even more IDEs available. But for now, you can consider one of the following :-

    Notepad – On a Windows machine, you can use any simple text editor like Notepad, or TextPad.

    Netbeans – A Java IDE that is open-source and free.

    Eclipse – A Java IDE developed by the eclipse open-source community

- Trainer will make group of participants and ask them to explain Java architecture and advantages with the help of Mind-Mapping on paper/word/ppt/paint/etc.

Eclipse IDE for Java

- Java IDE lets you **edit, build, debug, and test** your systems with ease and grace

- Every Java developer needs a programming editor or IDE.

- Deciding which editor or IDE will best suit you depends on several things, including the nature of the projects, the process used by the development team, and your level and skills as a programmer.

- **Eclipse** is the most popular Java IDE, it is free and open-source and is written mostly in Java, although its plugin architecture allows Eclipse to be extended in other languages.

Download Eclipse IDE

- Eclipse is the most popular IDE for developing Java applications.

- Click the following link to download Eclipse:

  http://www.eclipse.org/downloads/eclipse-packages

- Click on the link **32-bit** or **64-bit** (depending on the bit version of your operating system) to start downloading the package.

- In order to write and run a Java program, you need to install a software program called JD

Lab Activity

- Trainer to ask Participants to search for videos which shows Installation for JDK and Eclipse

- Once that is done, participants will install JDK and Eclipse in their respective PC's under Trainer's guidance.

Create a Java Project

- To execute First Program in Java and Eclipse.

- Create a new Java project in Eclipse, go to

    File > New > Java Project.

- Enter project name: HelloWorld.

- Leave the rest as it is, and click Finish.
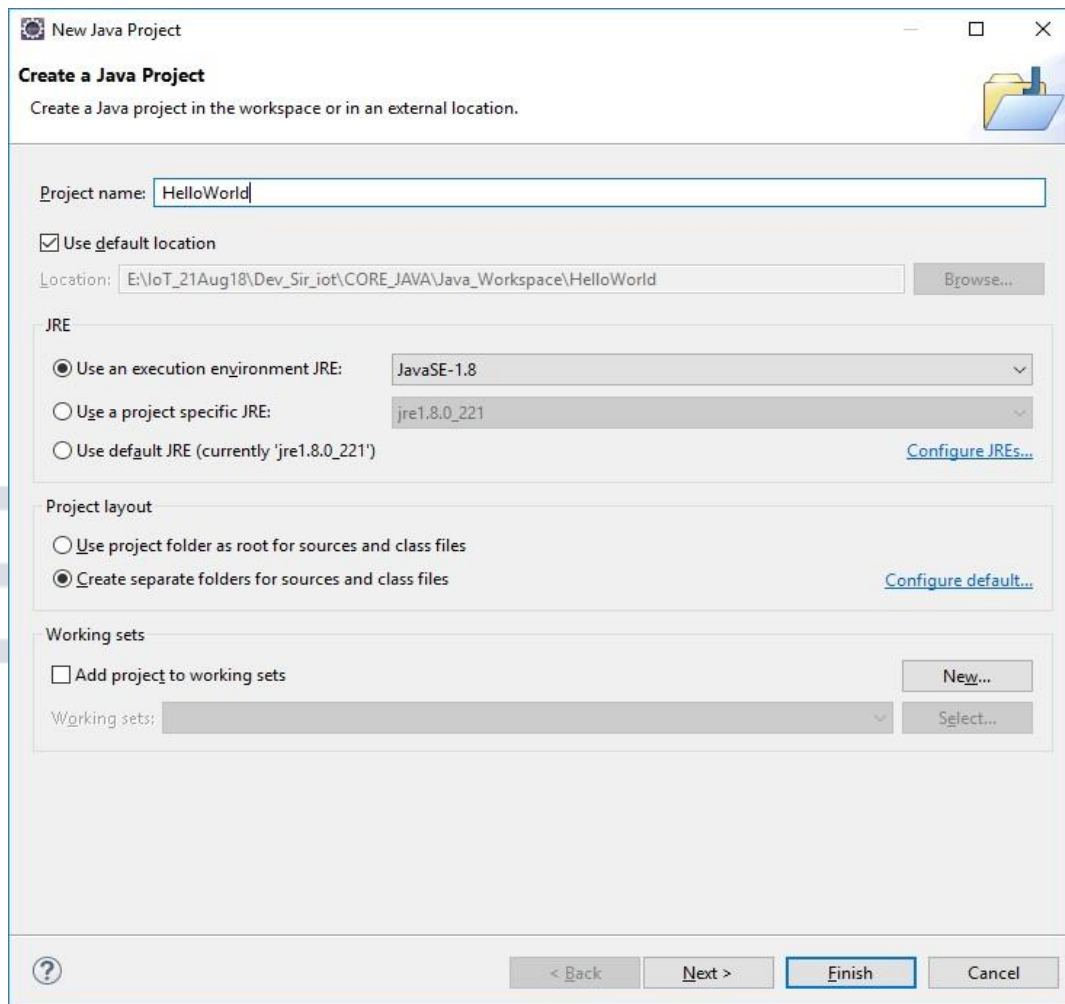
Create a package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.

- There are many built-in packages such as java, lang, awt, javax, etc.

- To create a package

- Right click on the project, and select **New > Package** from the list menu.

- Enter the name of your package (type any name of your choice).

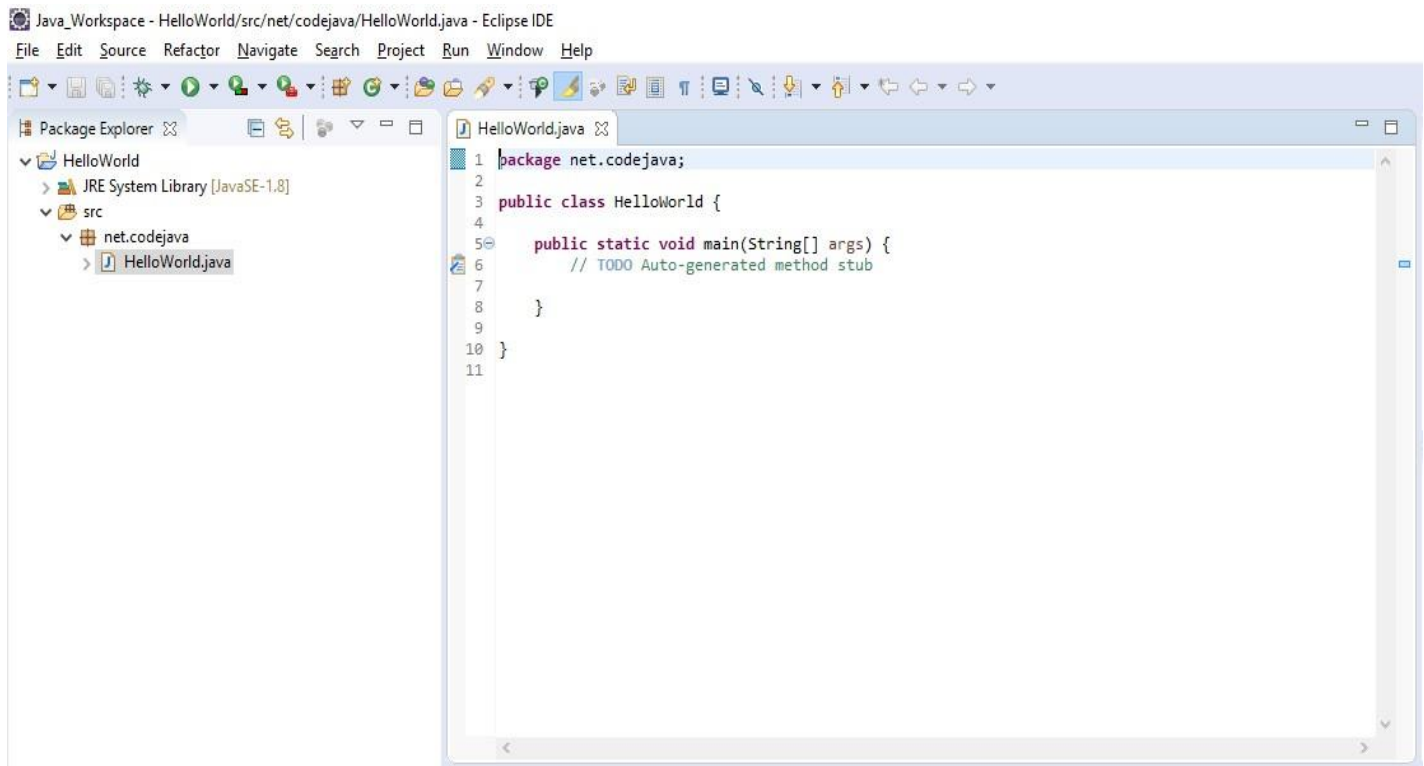- Click **Finish**. You should see the newly created package appears.
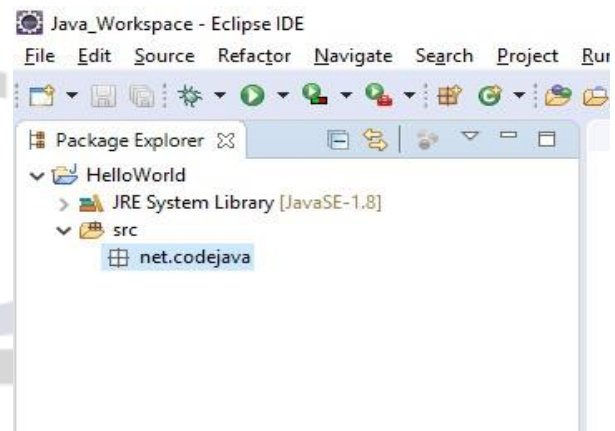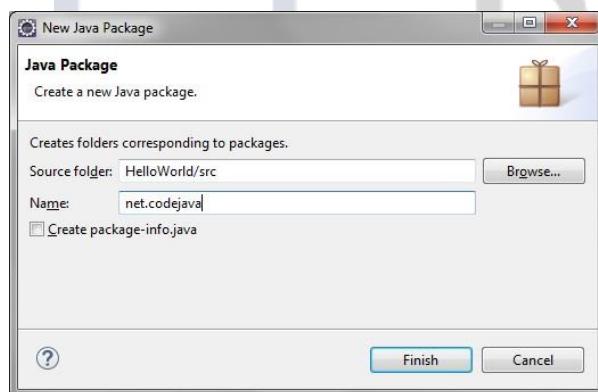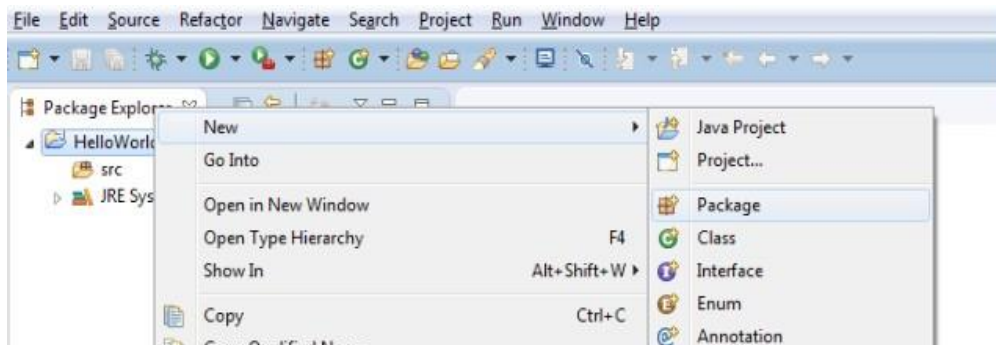
First Java Program

- Lets begin with first java program
- To create a new Java class under a specified package, right-click on the package (net.codejava) and select **New > Class** from the list menu.
- The **New Java Class** dialog appears, type the name of class as HelloWorld and choose the option to generate the main() method and click **Finish.**

Java_Workspace - HelloWorld/src/net/codejava/HelloWorld.java - Eclipse IDE

```java
package net.codejava;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

- Now type following code to print the message "Hello World"
- By default, Eclipse compiles the code automatically as you type. And it will report compile errors in the Problems view at the bottom
- Now, let's run the hello world java code.
- Click tab **Run > Run** (or press **Ctrl + F11**), Eclipse will execute the application and show the output in the Console view.
- Java code has run and printed the output **"Hello World"** and terminates.

```
*HelloWorld.java

1  package net.codejava;
2
3  public class HelloWorld {
4
5⊖     public static void main(String[] args) {
6          System.out.println("Hello World");
7      }
8
9  }
10
```

```
Problems  @ Javadoc  Declaration  Console
<terminated> HelloWorld [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (Sep 17, 2019, 5:12:45 PM)
Hello World
```

Parameters used in First Java Program

- Lets understand what above program consists of and its keypoints.

- **class** keyword is used to declare a class in java.

- **public** keyword is an access modifier. It means it is visible to all.

- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.

- **void** means it doesn't return any value.

- **main** represents the starting point of the program.
- **String[] args** is used for command line argument.

- **System.out.println()** is used to print statement.

- Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class.

Lab Activity:

Create an account in GitHub

- Trainer to ask participants' to create an account in GitHub referring participant guide, and share the details with the trainer.

- Trainer to explain the participants' how to create an account in GitHub, create repositories and push the code.

- After creating an account on GitHub, trainer to ask participants to create repositories and push the code and share it with the trainer.

Learners will push their respective projects in their own repositories as and when informed by the trainer.

Once pushed, learners will share the respective GitHub link with the trainer.

Instructions to create an account and repositories:

1. Initially we have to open this link - https://github.com/login.

2. Next we need to create new account (public).

3. Then the student needs to clear one small puzzle activity to avoid "I am robot".

GitHub will then ask the following:

Step-1 what kind of work do you do, mainly? We will select "STUDENT".

Next step-2 How much programming experience do you have?

Next step-3 What do you plan to use GitHub for? So, we will give "Host a project (repository)".

Step - 4 I am interested in: So give your interest area like framework, network or etc...

Step - 5 Then click completing setup.

4. Please verify your email address. You will have received one GitHub confirmation mail in your registered mail.

5. Then click create new repository. (create 6 like this)

6. Then give your Repository name (HTML, CSS JavaScript, Bootstrap, Python, Ruby on Rails).

13. Inside your repository, you can create your own program or else you can add existing file from your PC.
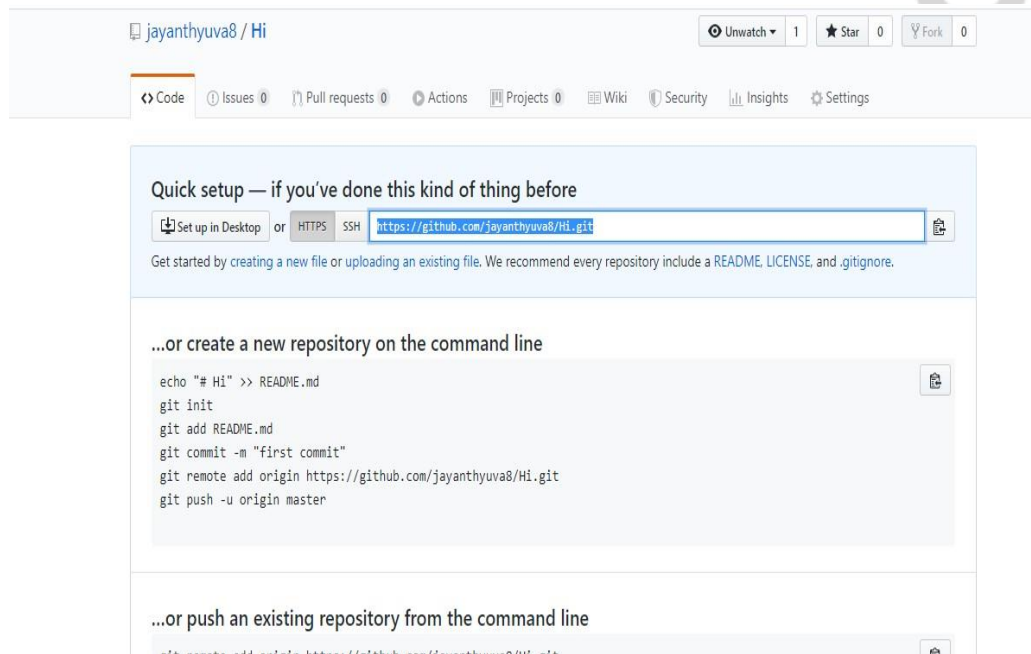
15. If you share that URL to anyone that person also can add program files and download your files directly from their systems.

Comments

- comments can be used to provide information or explanation about the variable, method, class or any statement in simple words comments are used in programs to make the code more understandable.

- The java comments are statements that are not executed by the compiler and interpreter.

- The comments can be used to provide information or explanation of any statement.

- In Java there are three types of comments:

    1. Single line comment (**//**)

    2. Multiline comment (/* ………… */)

    3. Documentation comment (Begins with " **/* *** " and ends with " ***/** ")

ADVANTAGES OF JAVA

The advantages of Java are as follows:

1. Java is easy to learn.

2. Java is object-oriented.

3. Java is platform-independent.

4. Java is Powerful Open source rapid development tools

5. Java has Open-source Libraries

6. Java is Free

7. Java is a secure language

8. Java is cheap and economical to maintain

9. Java supports portability feature

10. Java supports Multithreading

Introduction to JAVA programming

Data Types

- Data types in Java define the type of information that a variable can hold.

- In simple words, when we declare a variable in java then it is necessary to tell the data type of that variable means which kind of value that variable is going to store.

- The data types defines the type of values that a variable can take, for example, if a variable has a float data type, it can only take float values.

- In Java data types are divided into two types

    1. Primitive data types

    2. Non - primitive data types

- Each data type reserve different size of space in memory and they can store only that type of information which is their data type.

- As a programmer we should make minimum memory usage, because of this reason many type of data types are created.
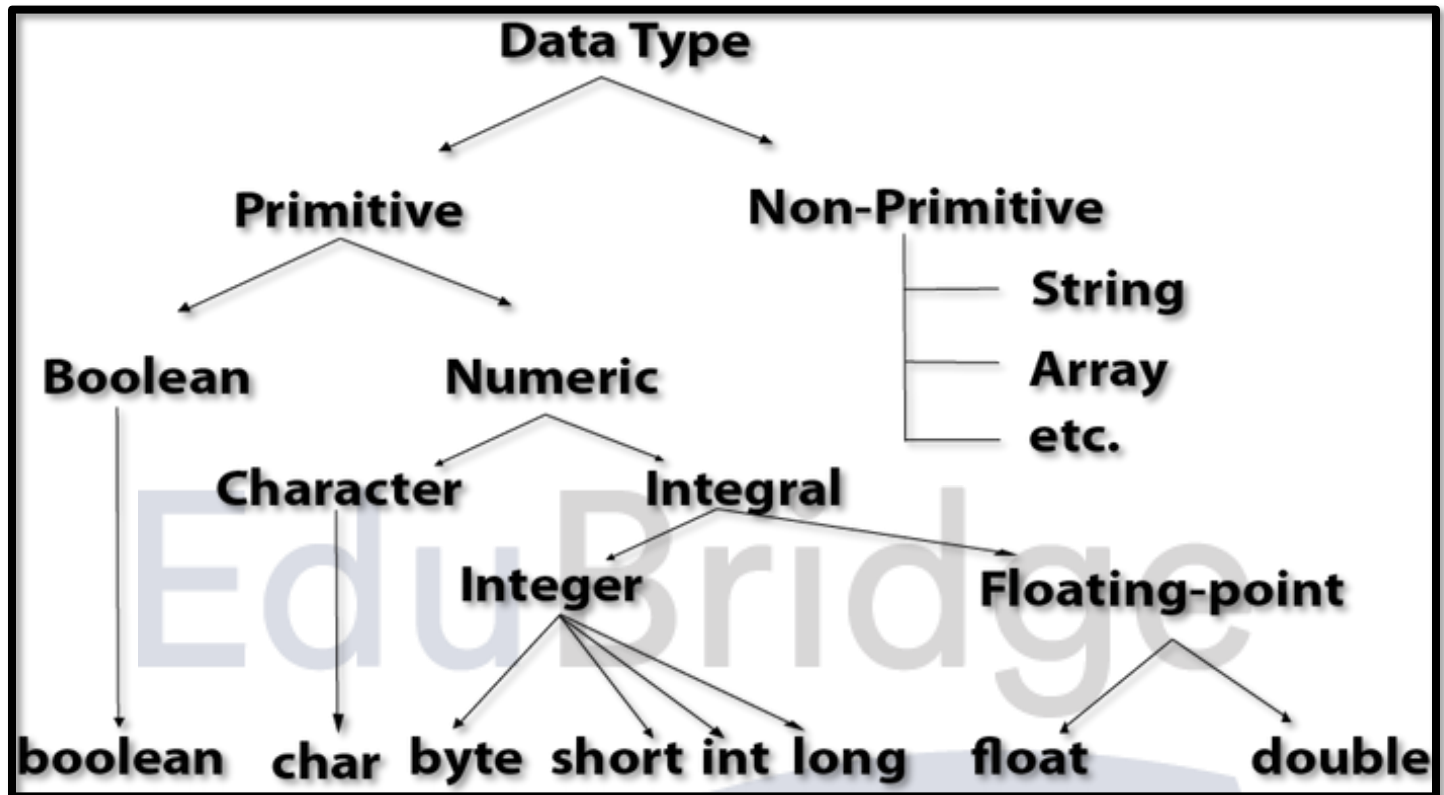
Primitive Data Types

- There are **eight primitive data types** in Java.

    1. Byte
    2. Short
    3. Int
    4. Long
    5. Char
    6. Float
    7. Double
    8. Boolean

- Java developers included these data types to maintain the **portability of java** as the size of these primitive data types does not change from one operating system to another.

- They are also helpful in **data manipulation**.

- The data types byte, short, int, and long are integer primitive data types.

- Java does not support unsigned integers so all these integers are signed, positive and negative values.

- Float and double are floating-point numbers primitive data types.

- Data type char is used to store characters in Java.

- In Java for logical values, we use boolean data types.

- In the table listed below will help to know about the data types like its default value and default size in the Java.

S.NO	Data Type	Size	Dfault Value
1	Byte	1 Byte	0
2	Short	2 Byte	0
3	Int	4 Byte	0
4	Long	8 Byte	0
5	Float	4 Byte	0.0
6	Double	8 Byte	0.0
7	Char	2 Byte	Empty
8	Boolean	1 Byte	FALSE

Integer Primitive Data Type

- Integers data type are used to store whole number they can be either positive or negative.

- The data type byte, short, int and long are integer primitive data types can be used to store an integer number.

- we can choose any among from them based on our requirement.

- There are only difference of size in these four data types, let's see in detail about each.

Integer Primitive Data Type : Byte

- The range of byte variable is from -128 to 127 and is a signed 8 bit type.

- Suppose we want to store an integer number 27, then it is best to use byte because it will reserve only 1 byte space in memory, since a byte is four times smaller than an integer.

- If we use int data type then it will reserve 4 byte space in memory, so we are wasting memory.

- Example: **byte b = 119**

Integer Primitive Data Type : Short

- Its default size is 2 byte.

- Its value range lies between -32,768 to 32,767

- The default value is 0.

- It is used to save memory as it is 2 times smaller than an integer.

- Example : **short a = 101**

Integer Primitive Data Type : Int

- This is the default integer type and most programmers by default use this data type even though they can be managed with byte or short.

- The programmers can use 10 digit number with int type.

- It occupies 4 bytes of memory.

- The range of integer lies between 2,147,483,648 to 2,147,483,647.

- Its default value is 0.

- Example : **int a = 2828**

Integer Primitive Data Type : Long

- An integer can store 10 digit values but if we want a value bigger than this range, we can use the long data type which can store up to 19 digit number.

- The long value needs to include the **letter 'l" or 'L' in its suffix.**

- The long data type takes 8 bytes of memory.

- The range lies between 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

- It's default value is 0.

- Example : **long a = 2828L**

Floating Primitive Data Type

- Floating point numbers are also known as real numbers.

- For example, floating point or decimal numbers are used when expressing measurements (feet and inches), time (hours, minutes, and seconds), etc.

- Decimal number is a number which contain decimal point (.), for example 66.7, 89.8 etc.,

- In java there are two floating data type which can be used to store a decimal value.

  1. **Float**

  2. **Double**

- Decimal values are useful because they allow more accuracy and precision.

Floating Primitive Data Type : Float

- The float requires less space than double, but it can store a smaller range of values than double.

- float is less precise than double.

- The float primitive data types are used to store decimal numbers.

- Sufficient for holding 6 to 7 decimal digits

- It occupies 4 bytes of memory.

- Declared float value should end with a "f".

- Float data type can not be used when you need precession like in currency.

- Example : **float b=8.2f**

Floating Primitive Data Type : Double

- The double Java data type is a default choice when it comes to decimal, as it is a double precision 64 bit.

- Double data type should never be used for precise values such as currency

- Double is preferred over a float in regular programming because it is sufficient for holding 15 decimal digits size.

- It occupies 8 bytes of memory.

- Double requires more space than float.

- It is good practice to end value with a "d".

- Its value range is unlimited.

- Example : **double m = -41358486.9d**

Character Primitive Data Type

- Java has Char Data type that takes 2 bytes of memory to support Unicode Characters.

- There is no negative value is used in the char data type.

- The first 256 (numbered from 0 to 255) characters of Unicode are the ASCII set of characters only.

- Char data type is used to store a single character.

- The value must be surrounded by single quotes.

- We can also use ASCII value as character variable value.

- Char data type is single 16-bit Unicode character.

- The range of char is 0 to 65,536.

- Example : **char b = 'H'**

Boolean Primitive Data Type

- It occupies 1 bit of memory.

- Data type boolean can have only true or false values.

- It specifies one bit of information.

- Boolean data type is used in condition testing.

- It's used in scenarios where only two states can exist.

- Boolean data type represents one bit of information.

- Default value is false.

- Example : **boolean b = true**

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the exercise one.

Exercise 1:

```java
package Hello.World;


public class Hello_World {


    public static void main(String[] args) {
        //boolean example
        boolean IoT = true;
    System.out.println(IoT);


    //byte example byte
    range; range = 127;
     System.out.println(range);


     //short example
     short range1;
     range1 = -127;
     System.out.println(range1);
```

```java
//int example

int range2 = 300000000;

System.out.println(range2);


//long example

long range3 = 30000000000L;

System.out.println(range3);


//double example double

number = 99.9;

System.out.println(number);


//float example

float number1 = 99.9f;

System.out.println(number1);


//char example (Unicode value of A is '\u0041')

char letter = '\u0041';

System.out.println(letter);



char letter2 = 65;   // ASCII value of 'A' is 65

System.out.println(letter2);
```

```
char letter3 = 'A';

System.out.println(letter3);
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Non Primitive Data Type

- Primitive types are predefined in Java.

- Non primitive types are created by the user and is not defined by Java (except for String).

- Non primitive data types are simply called "objects" because they are created, rather than predefined.

- Examples of non primitive data types are Strings, Arrays, Classes, Interface, etc.

- Non primitive data types are derived from primitive data types.

- Example String data type, it is a group of character data type.

- Non primitive data types are called **reference types** because they refer to objects.

- Non primitive types starts with an uppercase letter.

- Default value of this data type is null.

- Some widely used non primitive data types in Java are String, Arrays, Classes, Interface, etc.

Non Primitive Data Type : String

- **String** is a sequence of characters.

- It is used to store group of characters known as string.

- The value must be surrounded by double quotes.

- For example **"Hello World".**

- The java.lang.String class is used to create string object.

- string is an immutable object which means it is constant and cannot be changed, once it has been created.

- There are some more terms associated with string data type, are as follows String Comparison, String Concatenation, Concept of Substring, String class methods and its usage, StringBuffer class, StringBuilder class, Creating Immutable class, toString() method, StringTokenizer class, etc.

- There are two ways to create a String in Java

  1. String literal
  2. Using new keyword

- **String literal** : It is created by using double quotes.

  - **Example : String a = "JAVA";**

- **New keyword** : Java String is created by using a keyword "new".

  - Example : **String b = new String("JAVA");**

Non Primitive Data Type : Arrays

- An array in java is an object which is used to store multiple variables.

- It stores the similar elements.

- First element of the array is stored at index 0.

- Example : **int a[]; / int[] a;**

- There are 2 types of arrays supported in Java.

  1. Single Dimensional Array

  2. Multi-Dimensional Array

- Class is a blueprint for object.

- It is also considered as user defined data type.

- A class consists of variables and methods.

- It represents the set of properties or methods that are common to all objects of one type.

- The class body is surrounded by braces, { }.

- The name should begin with an initial letter

Non Primitive Data Type : Interface

- The interface is a blueprint of the class.

- Allow different objects to interact easily.

- It specifies a set of methods that the class has to implement.

- Interfaces are declared by specifying a keyword **"interface"**.

- A class can implement more than one interface.

- An interface is declared like a class but the only difference is that it contains only final variables and method declarations. It is fully abstract class.

Variables in JAVA

- Variable is the reserved memory location in the main memory, in simple words variable is the name of memory locations where information is stored.

- Example : **int num = 96**

    Here **variable name is "num"** which is associated with

    Value 96, **int is a data type** that determines this Variable that can hold **integer values.**

- Each variable has a specific data type, which is responsible to determine the memory space and the value or data that can be store in that variable.

- There are few naming convention rules follow them while defining a variable name.

- It is user defined name, its totally depends on user for defining a variable name.

- Java has certain naming conventions for a valid variable.

- These naming conventions help other developers to understand the code written by a developer

- The rules are listed below.

    1. Variable names are case sensitive

    2. Reserved Keywords cannot be used as variable name.

    3. White spaces are not allowed in variable name.

    4. A variable name can contain numbers, alphabets, and two symbols "_" and "$".

    5. Variable name should not start from numbers.

Types of Variables

- The basic syntax to declare variables in Java is shown below:

    Data type Variable name = **Value**;

- **Example :**

    int num = **96**;

- **Explanation of Example :** Here **variable name is "num"** which is associated with **Value** 96, **int is a data type** that determines this Variable that can hold **integer values.**
- Similarly we can declare more variable with different data type.
- There are three types of variables in java:

    1. Local variables

    2. Instance variables

    3. Static variables.

Example on Types of Variables in JAVA

```
class Sample{                              //Start of class
            int num = 96;          //Instance variable
            static int num1 = 99;  //Static variable
            void method1()
            {
            int num2 = 69;         //Local variable
             }
       }                              //end of class
```

- The sample code shown here will give the overview to understand and how to use 3 types of variables in java while programming and their working.

Local Variables in JAVA

- A variable declared inside the class body surrounded by braces, { } is called local variable.

- Local variables are visible only within the declared class, method, constructor, or block.

- We can use local variables with the same name in different functions body surrounded by braces, { }.

- local variable cannot be defined with "static" keyword.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the exercise 2.

Exercise 2:

```java
package Hello.World;


public class Hello_World {
    public void StudentAge()
    {
        // Local variable
        int age = 0;
        age = age + 30;
        System.out.println("Student age is : " + age);
    }


    public static void main(String[] args) {
        //Local Variable
        Hello_World temp = new Hello_World();
        temp.StudentAge();
    }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Instance Variables in JAVA
- A variable declared within the class but outside the body of the method is called instance variable.

- The instance variables are visible for all methods, constructors and any block in the class.

- Each instance(objects) of class has its own copy of instance variable.

- Instance variables are created when an object of the class is created and destroyed when the object is destroyed.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the exercise.

Exercise 3:

```java
package Hello.World;


class SubjectMarks {

    // These variables are instance variables, defined in class.

    int ScienceMarks;

    int MathsMarks;

    int GKMarks;

}


public class Hello_World {

        public static void main(String[] args) {

                // Student 1

        SubjectMarks Subj1 = new SubjectMarks();
```

```java
        Subj1.ScienceMarks = 60;

        Subj1.MathsMarks = 80;

        Subj1.GKMarks = 70;


        // Student 2

        SubjectMarks Subj2 = new SubjectMarks();

        Subj2.ScienceMarks = 85;

        Subj2.MathsMarks = 50;

        Subj2.GKMarks = 95;


        // displaying marks for Student 1

        System.out.println("Marks for first object:");

        System.out.println(Subj1.ScienceMarks);

        System.out.println(Subj1.MathsMarks);

        System.out.println(Subj1.GKMarks);


        // displaying marks for Student 2

        System.out.println("Marks for second object:");

        System.out.println(Subj2.ScienceMarks);

        System.out.println(Subj2.MathsMarks);

        System.out.println(Subj2.GKMarks);


    }


}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Static Variables in JAVA

- Static variables are also known as class variables as they are associated with the class and can be used by every instance of the class.

- Static variables are declared in the same manner as instance variables are declared, the only difference is that static variables are declared using the static keyword within a class outside any method constructor or block.

- Static variables are declared with the static keyword.

- Static variables are created at the start of program and destroyed at the end of the program.

- Visibility is similar to instance variables.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 4:

```
package Hello.World;

class Student {
        // static variable
    public static double fees;
    public static String name = "Aanh";
}

public class Hello_World {
        public static void main(String[] args) {
                Student.fees = 9999999.99;
        System.out.println(Student.name + "'s annual fees : " + Student.fees);
        }
```

```
        }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Operator

Operator in JAVA

- In Java, operators are used for evaluation of expressions.
- **Operator** in java is a **symbol** that is used to perform operations.
- Java supports the following types of operators:

    1. Unary Operator

    2. Arithmetic Operator

    3. Shift Operator

    4. Relational Operator

    5. Bitwise Operator

    6. Logical Operator

    7. Ternary Operator

    8. Assignment Operator

Operator in JAVA : Unary

- The Java unary operators require only one operand to perform any operation like :
    - (++) Incrementing a value by one
    - (--) Decrementing a value by one
    - (!) Negating an expression
    - (~) Inverting the value of a boolean

- **Example: int X=100 ;**

  X++ , ++X,   X--, --X,   !X, ~X.

Operator in JAVA : Arithmetic

- Java arithmetic operators are used to perform basic mathematical operations.
- Basic arithmetic operators are:
    - (+)   Addition
    - (-)   Subtraction
    - (*)   Multiplication
    - (/)   Division
    - (%) Modulo

- **Example : int a = 20; int b = 10;**
- a + b,   a – b,   a*b,   a/b,   a%b.

Operator in JAVA : Shift

- Shift operators are used to shift the bits of a number left or right.
- Basic shift operators are :
    - (>>)   Signed Right shift operator
    - (>>>)   Unsigned Right shift operator
    - (<<)   Left shift operator
    - (<<<)   Unsigned Left shift operator
- **Example : int X = 10;**
    - X>>1,   X>>>1,   X << 1,   X>>>1

Operator in JAVA : Relational

- The relational operators are used to determine the relationship between the operands.
- There are following relational operators :
    - (==) Relational Operator **equals to**
    - (!=) Relational Operator **not equal to**
    - (<) Relational Operator **less than**
    - (>) Relational Operator **greater than**
    - (>=) Relational Operator **greater than or equal to**
    - (<=) Relational Operator **less than or equal to**

- **Example : int a = 20; int b = 10;**
    a==b,   a!=b,   a<b,   a>b,   a>=b,   a<=b

Operator in JAVA : Bitwise

- Bitwise operator performs bit by bit operation.
- Basic bitwise operator are :
    - (|) Bitwise OR operator
    - (&)Bitwise AND operator
    - (^) Bitwise XOR operator
    - (~) Bitwise Complement operator

- **Example : int a = 20 (0001 0100); int b = 10 (0000 1010);**
    a|b,   a&b,   a^b,   ~a

Operator in JAVA : Logical

- Logical Operators are used with binary value of variables.
- Basic logical operator are :
    - (&&) Logical operator **AND**
    - (||) Logical operator **OR**
    - (!) Logical opertor **Not**

- **Example : int a = 20 (0001 0100); int b = 10 (0000 1010);**
    a&&b,   a||b,   a!b

- This operator is used as one line replacement for if-then-else statement.
- Ternary operator consists of three operands.
- **Syntax** : **condition ? value1 : value2**.
- **Example : int num =10 ? 100 : 200;**
- If the above example is **true** than the value1 (100) before the colon (:) is assigned to the variable num else the value2 (200) is assigned to the num.

Operator in JAVA : Assignment

- It is used to assign the value on its right to the operand on its left and then storing the result to the left operator.
- Some Assignment Operator in Java are :
    - (+=) To add the right and left operand
    - (-=) To subtract the right and left operand
    - (*=) To multiply the right and left operand
    - (/=) To divide the right and left operand
    - (^=) To raise the value of left to the power of right
    - (%=) To apply divide modulus operator

- **Example :**

    a += b,   a-=b,   a*=b,   a/=b,   a^=b,   a%=b

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 5:

```
package Hello.World;


public class Hello_World {
```

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

// Assignment Operator
        int number1, number2;
    number1 = 10;
    System.out.println(number1);
    number2 = number1;
    System.out.println(number2);

// Arithmetic Operator
    double output;

    // Addition operator
    output = number1 + number2;
    System.out.println("number1 + number2 = " + output);

    // Subtraction operator
    output = number1 - number2;
    System.out.println("number1 - number2 = " + output);

    // Multiplication operator output = number1 *
    number2; System.out.println("number1 * number2 = "
    + output);

    // Division operator
    output = number1 / number2;
    System.out.println("number1 / number2 = " + output);

    // Remainder operator
    output = number1 % number2;
    System.out.println("number1 % number2 = " + output);

// Unary Operators
    double number = 10, resultNumber;
    boolean flag = false;

    System.out.println("+number = " + +number);
    // number is equal to 10.

    System.out.println("-number = " + -number);
    // number is equal to 10.
    // ++number is equivalent to number = number + 1
```

```java
        System.out.println("number = " + ++number);
    // number is equal to 11.

    // -- number is equivalent to number = number - 1
        System.out.println("number = " + --number);
    // number is equal to 10.

        System.out.println("!flag = " + !flag);
        // flag is still false.

        System.out.println(number++);
        System.out.println(number);

        System.out.println(++number);
        System.out.println(number);

// Relational Operators
        if (number1 > number2)
        {
                System.out.println("number1 is greater than number2.");
        }
        if (number1 < number2)
        {
                System.out.println("number2 is greater than number1.");
        }
        if (number1 == number2)
        {
                System.out.println("number1 is equal to number2.");
        }

// Logical Operators
        int number3 = 20;
        boolean result;
        // Logical OR Operator
        result = (number1 > number2) || (number3 > number1);
        // result will be true because number3 greater than number1
        System.out.println(result);

        // Logical AND Operator
        result = (number1 > number2) && (number3 > number1);
        // result will be false because     number1 is equal to number2
        System.out.println(result);
```

```java
// Ternary Operator
        int februaryDays = 29;
        String display;

        display =  (februaryDays == 28) ? "Not a leap year" : "Leap year";
        System.out.println(display);

// Bitwise Operator
        int a = 50;        /* 50 = 0011 0010 */
    int b = 20;   /* 20 = 0001 0100 */
    int c = 0;

    c = a & b;       /* 16 = 0001 0000 */
    System.out.println("a & b = " + c );

    c = a | b;       /* 54 = 0011 0110 */
    System.out.println("a | b = " + c );

    c = a ^ b;       /* 38 = 0010 0110 */
    System.out.println("a ^ b = " + c );

    c = ~a;          /*-51 = 1100 1101 */
    System.out.println("~a = " + c );


// Shift Operator
    c = a >> 2;      /* 12 = 0000 1100 */     // Right shift operator
    System.out.println("a >> 2  = " + c );


    c = a << 2;     /* 200 = 1100 1000 */    // left shift operator
    System.out.println("a << 2 = " + c );
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Array

Arrays in JAVA

- An array is a collection of similar types of elements.

- An array is a container that holds data of one single type. For example, you can create an array that can hold 100 values of int type.

- To make it more clear, a pictorial representation of array elements with their corresponding index values is shown below.

- The first element of the array is stored at the 0th index, second element of the array is stored at 1st index and so on as shown in given figure.
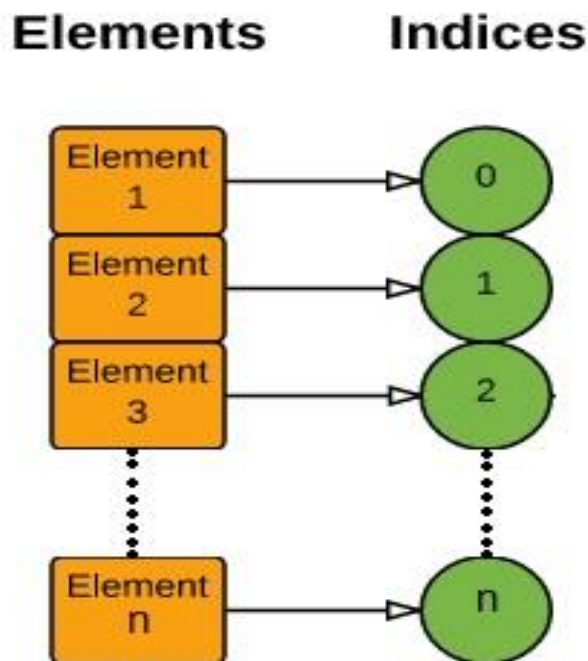
Array Syntax

- In Java, an array is an object of a dynamically generated class.

- Once the array is created, its length is fixed.

- Syntax to define array :

  data type [ ] array name;

- **Example :   int [ ] num;**

  num = new int[10];

- Here, **num** array can hold 10 values of data type int.

Arrays Declaration

- It's possible to define array in one statement. You can replace the two line in above example with a single statement.

- **Example :**

  int[ ] num = new int[10];

- Operator "new" is used to initialize an array.

- **Declaring Array :**

  int[] num = {73, 12, 25, 8, 99};

- The elements are numbered as 0, 1, 2, ..., n-1. These numbers are called as indices.

- These numbers are used to locate the positions of elements within the array.

- The data type of an array must be specified by an int value and not long or short.

Arrays Types

- There are two types of array in JAVA

  1. Single Dimensional Array

  2. Multi Dimensional Array

Single Dimension array

- You can create a single dimensional array by using the "new" keyword whose syntax is given below.

- It is an array in which each element is accessed by using only one index number. These index number represents the position of the element in the array.

- The following syntax is used to declare a single dimensional array :

data type [ ] array name;

or

data type array name [ ] ;

eg.   int[ ] num = new int[10];

Multi Dimensional Array

- A multi dimensional array is very much similar to a single dimensional array. It can have multiple rows and multiple columns unlike single dimensional array, which can have only one full row or one full column.

- In java, a jagged array means to have a multi dimensional array with uneven size of rows in it.

- The following syntax is used to declare a multi dimensional array :

data type [ ] [ ] array name ;

or

data type array name [ ] [ ] ;

- **eg.** int[ ] [ ]num = new int[10] [20];

- For both row and column i.e. [] [] the index begins from 0.

- Multi dimensional array are also called Jagged arrays in java.

Cloning of Arrays

- Cloning creates copies that are clones of the original elements or reference elements.

- Cloning arrays are of two types shallow copy and deep copy in Java.

- In a single-dimensional array, a deep copy creates the clones of the original elements or reference elements.

- In a multi-dimensional array, a shallow copy is created, which means both arrays are pointing to the same memory address.

- Cloning shallow copy and deep copy in Java are the ways of copying the attributes of one object into another of the same type.

- Deep copy means a variable would have a copy of the original array in a different memory location.

- Shallow copy means both arrays are pointing to the same memory address. Whenever you modify one of these arrays, you will be modifying both arrays.

Array Deep Copy

- Deep copy means a variable would have a copy of the original array in a different memory location.

- **Example :** int array1[] = {1,2,3,4,5};

            int array2[] = array1.clone();

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 6:

```
package Hello.World;


public class Hello_World {
        public static void main(String[] args) {
                int intArray[] = {0,1,2,3,4,5,6,7,8,9};

        int cloneArray[] = intArray.clone();

        // Deep copy is created following print statement will print false
        System.out.println(intArray == cloneArray);

        for (int i = 0; i < cloneArray.length; i++) {
           System.out.print(cloneArray[i]+" ");
        }

            }

}
```
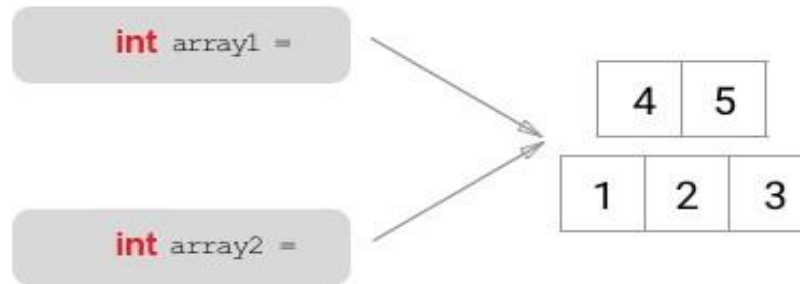
Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Array Shallow Copy

- Shallow copy means both arrays are pointing to the same memory address. If any one of these array is modified other array get auto reflected.

- **Example :** int array1[][] = {{1,2,3},{4,5}};

    int array2[][] = array1.clone();

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 7:

```java
package Hello.World;


public class Hello_World {
        public static void main(String[] args) {
                int intArray[][] = {{1,2,3,4,5},{6,7,8,9,10}};

        int cloneArray[][] = intArray.clone();

        // shallow copy is created following print statement will print true
        System.out.println(intArray[0] == cloneArray[0]);
        System.out.println(intArray[1] == cloneArray[1]);


        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Array

- **Advantages :**

    - **Code Optimization:** It makes the code optimized.

    - **Random access:** Data located at the index position.

- **Disadvantages :**

    - **Size Limit:** We can store only the fixed size of elements in the array.

String

String in JAVA

- String is a sequence of characters enclosed within double quotes (" ") is known as **String.** It is an immutable object.

- When we create a string in java, it actually creates an object of type String.

- **Java String** class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Create String in JAVA

There are two ways to create a String in Java

1. **String literal**

2. **Using new keyword**

1. **String literal :**

    - Java String literal is created by using double-quotes.
    - **Example**:

    String s = "king";

- String objects are stored in a special memory area known as the "string constant pool".
- This is the most common way of creating the string.

2. **Using new keyword:**

- String object can be created using new operator like java class.
- **Example:**

  String s = new String("king");

- It creates two objects in String pool and in heap
- Also one reference variable 's' is created that will refer to the object in the heap.

Java String Pool: Java String pool refers to collection of Strings which are stored in heap memory. So whenever a new object is created. It will check whether the new object is already present in the pool or not. If it is present, then same reference is returned to the variable else new object will be created in the String pool and the respective reference will be returned.

String concatenation in JAVA

- String concatenation means joining of two or more strings.
- We have two strings
                str1 = "Core" and str2 = "JAVA"
- If we add these two strings, we should have a result as str3= "CoreJAVA".
- There are **two methods** to perform string concatenation.
- First is by using **arithmetic "+" operator** and second is using "**concat**" method of String class.
- Both method will results in the same output.

Example

- How to perform string concatenation in java
- First example is by using **arithmetic "+" operator** and second is using "**concat**" method of String class.
- string concatenation by **operator (+) method**:

- **Example1:**

> **String str3 = "Core" + "JAVA";**

- String concatenation by **concat() method** :
- **Example2:**

> **String str3 = str1.concat(str2);**

Lab Activity:

- Following is the web link which is used as a web terminal to try and test Java through JSHELL.

- So you can use this link to check the string method examples as shown below :

> https://tryjshell.org/

- Participant will write & observe the simple hello word program, with the help of trainer

String Method

List of the some important methods available in the Java String are as follows:

- **Length() :**
  - It returns the length of the string object.
  - **Example :**

    > String s = "Core JAVA";

    > s.length()

    > **Output : 9**

- **getChars() and toCharArray()** :
  - It is used to populate a character array from the string object as source.
  - syntax :

  - getChars(int **srcBegin,** int **srcEnd,** char **dst[],** int **dstBegin**)
  - Where :
  - **srcBegin** − index of the first character in the string to copy.

- **srcEnd** – index after the last character in the string to copy.
- **dst** – the destination array.
- **dstBegin** – the start offset in the destination array.

- **Example :**

  The **Java String** class **getChars()** method copies the content of this string into a specified char array. There are four arguments passed in the getChars() method.

  Example :
  ```
  String s1 = "JAVA";
  char[] dest=new char[4];
  s1.getChars(0,4,dest,0)
  System.out.println(Arrays.toString(dest));
  ```
  Output : [J, A, V, A]

- **toCharArray():**
  - This method returns a new character array created from the string characters.
  - The **java string toCharArray()** method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

Example:

String s = "IoT";

s.toCharArray()

Output: char[3] { 'I', 'o', 'T' }

- **compareTo() and compareToIgnoreCase()** :
    - compareTo() method is used to compare two strings lexicographically.
    - The compareToIgnoreCase() method is similar to compareTo() method also performs the lexicographical comparison only it ignore case.
    - **Example :**
    -

            "Java".

        compareTo("

        Java")

        Output : 0

    -

            "Java".compareToI

        gnoreCase("JAVA")

        Output : 0

isEmpty(), isBlank() :

- isEmpty() method returns true if the string is empty.

- isBlank() method returns true if the string is empty or contains only whitespace characters like spaces and tabs.
- **Example :**

 String emptyStr = "   ";

 emptyStr.isEmpty()   // Output:

 false emptyStr.isBlank()   //

 Output: true

charAt(int index) :

- This method returns the character at the given index.

 Example :

 String s = "Java";

 s.charAt(3)

 Output: 'a'

startsWith() and endsWith():

- These methods are used to check if the string has given prefix or suffix strings or not.

 Example :

 "Coking".startsWith("king")

 Output : false

 "Coking".endsWith("king")

 Output : true

toLowerCase() and toUpperCase():

- These methods are used to create lowercase and uppercase strings.

Example:

"Java".toUpperCase()

Output : "JAVA"

"jAVa".toLowerCase()

Output : "java"

trim(), strip(), stripLeading(), and stripTrailing() :

- **trim() :** It trim all the leading and trailing whitespaces from a string.
- **strip():** This method uses Character.isWhitespace()
- method to remove leading and trailing whitespaces from a string.
- The **stripLeading() and stripTrailing()** methods also remove leading and trailing whitespaces.
- **Example:** String s = "   Java   "

    s.trim()

    Output : "Java"

repeat() :

- This method returns a new string whose value is the concatenation of this string given number of times.
- **Example:**

    String s = "Java"

    s.repeat(2)

    Output: "JavaJava"

contentEquals():

- This String method compares the content of the string.
- **Example:**

    String s1 = "Skillking"

StringBuffer s2 = new StringBuffer()

s2.append("Skillking")

s1.contentEquals(s2)

Output : true

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 8:

```
package Hello.World;
```

```java
public class Hello_World{


    public static void main(String[] args) {

            // String declaration without using new operator
    String a = "Cooking";
    System.out.println("String a = " + a);

    // String declaration using new operator
    String b = new String("IoT");
    System.out.println("String b = " + b);

    //There are many string methods available some String Methods are as follows
    System.out.println("The length of the string: " + a.length());
    System.out.println(a.toUpperCase());
    System.out.println(a.toLowerCase());
    System.out.println(a + b);
    System.out.println(a.concat(b));
    System.out.println("Character at position 5: " + a.charAt(5));
    System.out.println(a.equals(b));
    System.out.println(a.equalsIgnoreCase(b));
    System.out.println(a.compareTo(b));
    System.out.println(a.contains("X"));


        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

String Method

- We can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc. with the help of these methods.
- Like this way there are many more important string method such as

    getBytes(), equals(), hashCode() and equalsIgnoreCase(), indexOf() and lastIndexOf(), substring() and subSequence() ,matches(), replace(), replaceFirst(), and replaceAll(), split(), lines(), indent(), transform(), format(),

intern(), valueOf() and copyValueOf(), repeat(), describeConstable() and resolveConstantDesc(), formatted(), stripIndent(), and translateEscapes(), etc.

Java String Buffer class

- Java StringBuffer class is used to create mutable string.
- StringBuffer represents growable and writable character sequences.
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer( ):** It reserves space for sixteen characters without reallocation.

    Example : StringBuffer s=new **StringBuffer**();

- **StringBuffer(int size):** It accepts a whole number argument that explicitly sets the scale of the buffer.
- **Example:** StringBuffer s=new **StringBuffer**(30);
- **StringBuffer(String str):** It reserves area for sixteen characters while not reallocation and accepts a String argument that sets the initial contents of StringBuffer object.
- **Example:** StringBuffer s=new **StringBuffer**("IoT");

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 9:

```
package Hello.World;

public class Hello_World{

        public static void main(String[] args) {

                StringBuffer str = new StringBuffer("Cooking ");
                str.append("Emerging ");
                System.out.println(str);
```

```
              str.insert(17, "Technology ");
              System.out.println(str);

              str.replace( 8, 16, "IoT");
              System.out.println(str);

              str.reverse();
              System.out.println(str);

              System.out.println( str.capacity() );

       }
   }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Enum

- Enum is a one of the special data types to declare the list of constants that enable for a variable, which have similar meaning. For example, we create enum to manage the list of months, name of days, and other similar kind of values.

- We declare enum with the help of enum keyword and since this holds the constant value, the constraint is to write the value in capital letters.

- In a common real-life scenario, we can understand enum such as a short form of code word, which has a meaning.

- In programming, when we want to hide the actual data from the end users, we prefer to use enums.

- For example, in our program we want the user to enter a number so that 0 represents Sunday and 7 represents Saturday. But when we are setting the enum for those numbers, we will set with the name of the day instead of sequence 0 or 1, that will be Sunday, Monday, and so on and we further use the same in our program for evaluation.

- This way, we also mislead the hackers to identify what value is being used in the program and they cannot easily hack the program written using enum.

- We may also use the enum contact with values for mathematical formulas or assign a unique value for that constant.

- We can also declare a class as type enum, which has a different meaning in Java programming and at the time of compiling, the compiler implicitly adds some additional features to that call.

- An enum class can have methods and fields.

Example (Enum constant):

```
Public class EnumImpl {
public enum Day {
MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY, SUNDAY
}
public static void main(String[] args) {
System.out.println(Day.MONDAY);
} }
```

Output:

MONDAY

- Here, we have declared a enum Day, which holds name of the days and printed the first constant **MONDAY** as follows

Example (The Enum class type example with contacts with values)**:**

```java
public enum Day {
SUNDAY(0), MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5),
SATURDAY(6);
private int dayIndex;
Day(int name){
this.dayIndex = name;
}
public static void main(String[] args) {
System.out.println(Day.MONDAY.getDayIndex());
System.out.println(Day.TUESDAY);
}
public int getDayIndex() {
return dayIndex;
}
}
```

Output:

```
1
TUESDAY
```

- In the preceding example, **Day(int name)** is the constructor, which is assigning the constant value to a private variable

- After this, we have created a **getDayIndex** method, which helps us to get the value of the constant.

- In the main method, we have written the following statements to get the constant and its values:
  System.out.println(Day.MONDAY.getDayIndex()); // this line of code will print the value from constant MONDAY that is 1.
  System.out.println(Day.TUESDAY); // this line of code will simply print constant TUESDAY.

- **NOTE: All enums implicitly extend java.lang.Enum because a class can only extend one parent class in Java.**

Control flow statements

- Control flows are the sections of a code that gets executed in sequence as they appear in the program, but before execution.

- It checks for the condition and when the given condition becomes true, only then the section of code gets executed.

- Now, we will study and practice the use of different control flows of Java programming such as if, if-then, if-then-else, and switch.

- Using these statements, we manage executing flow of the program and make the decision at runtime regarding what flow is going to be executed next in the program.

- We will also learn about loops; loops are statements using the ones we can repeat a block of code several times based on the condition or expression.

- Until the expression returns true, the loop will keep repeating the execution of that same block of code, which is written in the body of loop.

- We will see the types of loops available in Java programming and how to implement those in the program.

JAVA Control flow - if Statement

- An **if statement** is a powerful statement, which we use to decide in a program and execute a code block when the given condition becomes true.

- This is also called if-then statement.

```
For example: if(course=="JAVA") {
        system.out.println("Course is JAVA.");
        }
```

- The following figure is a graphical representation or flow chart of the execution of an if statement program:
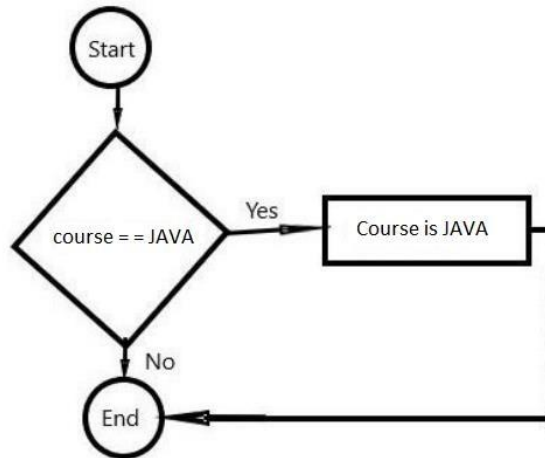
JAVA Control flow – if-then-else Statement

- If the condition which is given in the IF statement is not true, then the block of code that is written inside the if block will not get executed, and to handle the false condition, we write an else statement block of code that gets executed in place of the if statement.
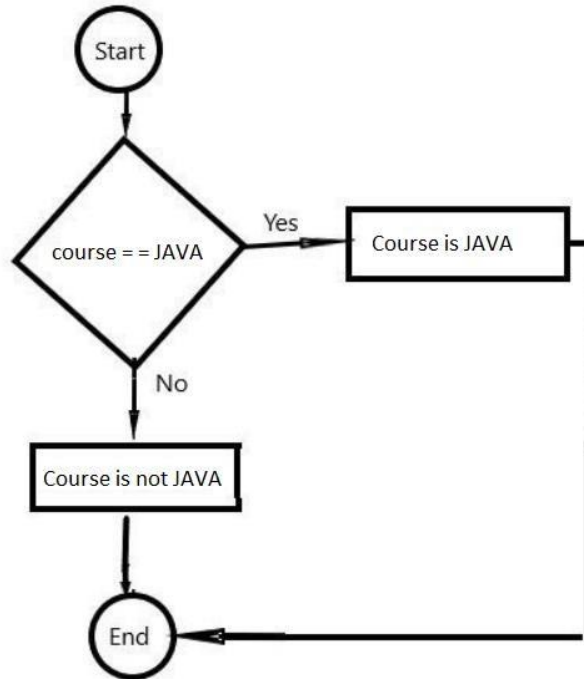
For example:

```
if(course=="JAVA") {
System.out.println("Course is JAVA.");
} else {
System.out.println("Color is not red.");
}
```

- The following flow chart shows the execution flow of an if-else statement:

- When a given if condition and expression are true, then the program will route to the **Yes** part, or else, it will route to the **No** part of the program.

JAVA Control flow – Nested if-else Statement

- Nested if-else is used to perform multiple if-then-else-if conditions in a program.

- We need this when we must handle or check multiple conditions in a program on the same or different patterns.

- Using the if statement, we can test and compare a single value and range of values

For example:

```
if(marks>= 60) {
system.out.println("You passed with first grade.");
} else if(marks >=45 || marks < 60) {
system.out.println("You passed with second grade.");
} else if(marks >=33 || marks < 45) {
system.out.println("You passed with third grade.");
} else {
system.out.println("Sorry, you are fail.");
}
```

JAVA Switch Statement

- Switch case is another way of making decisions and executing a block of code written under that case.

- This is like nested if-else.

- We use the switch keyword to create a **switch** statement in the code and pass a **variable** as an argument

- In the body of switch statement we declare cases with possible values to check and a default case to handle the flow if there is no case matched with the given expression.

- The body of switch statement is called switch block

Example:

```
switch(expression){

        case
        value1:

        //code to be

        executed break;
```

```
    case value2:

        //code to be executed

    break;

    ......

    default:

        //execute if all cases are not matched

}
```

- The general syntax of how switch case is implemented is as follows:

- We can have any number of case statements.

- Values must be of same type of expression.

- Break statement is optional.

- Case labels always end with a colon (:).

- Default statement is used when none of the cases condition is true.

- No break statement is needed in the default case.

- Duplicate case values are not allowed.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 10:

```java
package Hello.World;


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


              int
              a=3;

               //Switch
        expression switch(a){
        //Case statements
        case 1: System.out.println("I am @ Home");
        break;
        case 2: System.out.println("I am @ Office");
        break;
        case 3: System.out.println("I am @ school");
        break;
        //Default case statement
        default:System.out.println("Not @ Home, Office, or school");
        }

     }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Switch Case Vs If Statement

- Where we can use nested if or switch case both for evaluation, then we must think on actual requirements and later, decide the right method.

- If we are using nested if, then the program will execute all if conditions to find the exact evaluation, but this is not the case of a switch.

- Switch directly jumps to the matching case and does not evaluate or check the other cases from the case.

- So, this way, we make our system faster and navigate to the right block of code instead of checking every condition.

Loops in Java

- loops are used to execute a set of instructions/ functions repeatedly.

- When the condition is true loops are executed.

- There are three types of loops available in Java for performing a task such as

  1. do while

  2. While

  3. for loops

While loop in Java

- While loop first checks the body of the loop then it will be executed.

- **Syntax :**

    while (condition)

    {

    loop statements...

    }

- The while statement is used to iterate/repeat a block of code until the given condition remains true.

- The while statement returns Boolean value **true** or **false** after execution of expression.

- When it returns true, the cursor will go inside the body of the while block and executes the statements.

- When it returns it will not execute the code from the while block

---

Example
```
public class WhileLoop {
public static void main(String[] args) {
int a = 2;
while(a<=4) {
System.out.println("Current value of a is "+a);
a++;
}}}
```

Output:
Current value of a is 2
Current value of a is 3
Current value of a is 4

- **While loop** is continuously executed if the condition becomes true, until the condition becomes false to end the loop

- It can be considered as a repeating if statement.

- When it returns true, the cursor will go inside the body of the while block and executes the statements.

- When it returns it will not execute the code from the while block

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 11:

```
package Hello.World;


public class Hello_World {
        public static void main(String[] args) {
                int i = 1;

            while (i <= 10) {
              System.out.println("Skillking IoT Student " + i);
              i++;
            }
```

```
        }

    }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

do while loop in Java

- **do while** is like the while loop and is used for the same looping purpose.

- The only difference between while and do while is that in do while, we check the condition later and in while, we check the condition at the first point.

- The cursor moves inside the do while block and executes the statements written under the do block and later it checks the condition. Hence, the do while block will always execute do part at least once, whether a condition become true or false.

- And then, it executes the while to check the condition; if the condition returns true, then program will repeat and execute the do statement again and again until it remains true; otherwise, the program will exit

- The do while loop executes a part of the programs at least once and execution depends upon the given condition.

- **Syntax :**

      do

      {

      loop statement.......

      }

      while (condition);

Example

```
public class DoWhileLoop {
public static void main(String[] args) {
int a = 2;
do {
System.out.println("Current executing index :"+a);
a++;
} while(a<=1);
} }
```

Output:
Current executing index :2

- The above program will get executed and execute the do statement and block of code written under the do statement.

- Then at the end, it will check the condition.

- Since the initial value of variable a was 2, and in the do block, we have initialize the value of a with 1, now the value of a is 3.

- Hence, the condition will become and this program will no further execute the do statement.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 12:

```
package Hello.World;


public class Hello_World {
        public static void main(String[] args) {
                int i=1;
            do{
                System.out.println("Skillking IoT Student" + i);
            i++;
            }while(i<=10);


        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

for loop in Java

- For loop is a very clear and an easy way to create loop. In the for loop, we write all three parameters of loop together in a single statement. A for loop statement must have three required parameters and options, and each parameter will be separated by semicolon (;).

- Those are the following:
  For loop first initialize the variables and then the condition is tested, and accordingly increment or decrement takes place.

- If the number of iteration is fixed, for loop can be used.

- **Syntax :**

  for (initialization; condition; increment/decrement)   {

          loop statement.....

          }

Example
```
for(inti=1; i<=5; i++) {
System.out.println("Value of i "+i);
}
```

Output:
Value of i 1
Value of i 2
Value of i 3
Value of i 4
Value of i 5

- In the code shown, we have declared a **for** loop, which gets executed five times.

- We have declared and initialized an integer variable In the first parameter, we have initialized the variable **i** with 1, which means this loop will begin from initial value 1.

- In the second parameter, we have defined the termination point which says that this loop will keep executing until the value of **i** remain less than or equal to 5.

- In the last parameter, we have defined This expression is to increase the value of **i** after each iteration with **i = i +** Each time this loop is executed, the value of **i** gets increased by 1.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 13:

```
package Hello.World;
```

```
public class Hello_World {
        public static void main(String[] args) {
                for (int i = 1; i <= 10; i++)
                {
            System.out.println("Skillking IoT Student " + i);
          }

        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

nested loop in Java

- We can also write a for loop inside the body of another for loop statement, which means a loop under another loop.

- This is called the nested loop.

- Here is an example of a nested for loop.

- Explanation: Here, we will understand the execution process of nested loops. Nested for loops are executed from parent to child; it means each iteration will get executed first from parent to child, then after it will move to parent iteration. Under the parent loop, there are multiple child loops, then execution of all child loop gets completed first before moving to parent loop for next iteration.

Example
```
for(inti=1; i<=5; i++) {
System.out.println("Value of i "+i);
for(int j=1; j<=i; j++) {
System.out.println("value of j is " +j);
}
}
```

Output:
Value of i 1
value of j is 1
Value of i 2
value of j is 1
value of j is 2
Value of i 3
value of j is 1
value of j is 2
value of j is 3
Value of i 4
value of j is 1
value of j is 2
value of j is 3
value of j is 4
Value of i 5
value of j is 1
value of j is 2
value of j is 3
value of j is 4
value of j is 5

break statement in Java

- Suppose you are working with loops. It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

- In such cases, break statements are used.

- The break statement terminates the loop immediately, and the control of the program moves to the next statement following the loop.

- **Syntax :**

```
break;
```

Labeled break statement in Java

- The break statement we have discussed till now is unlabeled form which terminate inner loop, there is another form of break statement, **labeled break**, that can be used to terminate the outer loop

  Example :

```
while (testExpression) {
// codes
Jump_Here:
while (testExpression) {
// codes
while(testExpression) {
// codes
break Jump_Here;
}
}
// control jumps here
}
```

- In the above example, when the statement break **Jump_Here**; is executed, the while loop labeled as **Jump_Here** is terminated. And, the control of the program moves to the statement after the second while loop.

continue statement in Java

- The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

- **Syntax :**

  continue;

- The continue statement we have discussed till now is unlabeled form of continue, which skips the execution of remaining statement(s) of innermost for, while and do..while loop.

- There is another form of continue statement, labeled form, that can be used to skip the execution of statement(s) that lies inside the outer loop.

- In given example we can see that the label identifier **Jump_Here** specifies the outer loop. Notice the use of the continue inside the inner loop. Here, the continue statement is skipping the current iteration of the labeled statement (i.e. outer loop). Then, the program control goes to the next iteration of the labeled statement.

```
Jump_Here:
while (testExpression) {
// codes
while (testExpression) {
// codes
if(testExpression) {
// codes
continue Jump_Here;
}
// codes
}
// codes
}
```

JAVA Classes and Objects

- In object oriented programming approach, we design a program using objects and classes.

- A class is a group of objects which have common properties.

- classes are user defined data types and objects are an instance of a class.

- The methods and variables defined within a class are called members of the class.

- The "**class** " **keyword** is used to declare a class.

- A class contains variables and methods.

- The Variables, defined within a class are called instance variables.

- A class is a blueprint from which objects are created. So, an object is the instance(result) of a class.

- Objects can be tangible or intangible both.

- Objects have two characteristics **states** and **behaviors**.

- Class represents a set of objects having similar responsibilities.

- An object is a real world entity, for example chair, car, pen, bag, apple, etc.

JAVA Class Structure Diagram

The basic structure of a class is as follows:

```
┌─────────────┐
│    Class    │
├─────────────┤
│  Attributes │
├─────────────┤
│  Operations │
└─────────────┘
```

Fig a

```
┌──────────────────────────────────────────┐
│                  Course                    │
├──────────────────────────────────────────┤
│ students : List                            │
│                                            │
│ teacher : Lecturer                         │
├──────────────────────────────────────────┤
│ add Student(student: Student)              │
│ assignTeacher(teacher: Lecturer)           │
│ getNumberOf Students() : void              │
│ getTeacher() : Lecturer                    │
│ remove Student(student: Student) : void    │
└──────────────────────────────────────────┘
```
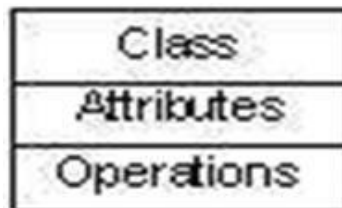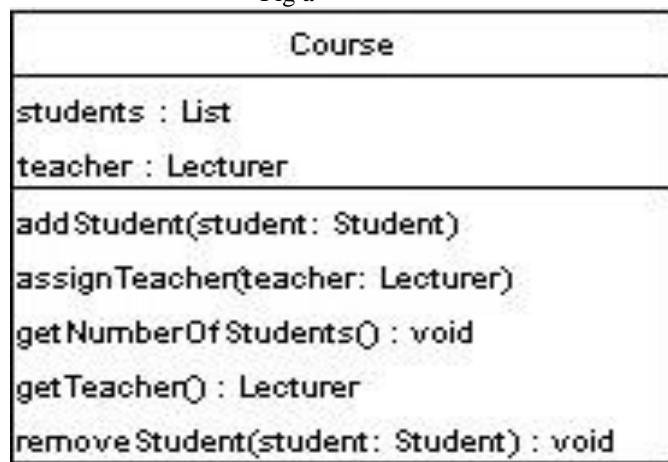
Fig.b

- Class: The top section is used to name the class.

- Attributes: The second one is used to show the attributes of the class.

- Operations: The third section is used to describe the operations performed by the class.

Course
students : List
teacher : Lecturer
addStudent(student: Student)
assignTeacher(teacher: Lecturer)
getNumberOfStudents() : void
getTeacher() : Lecturer
removeStudent(student: Student) : void
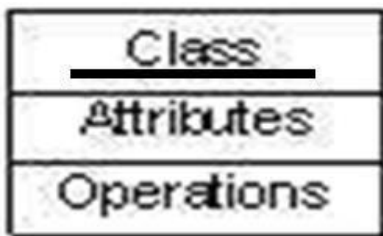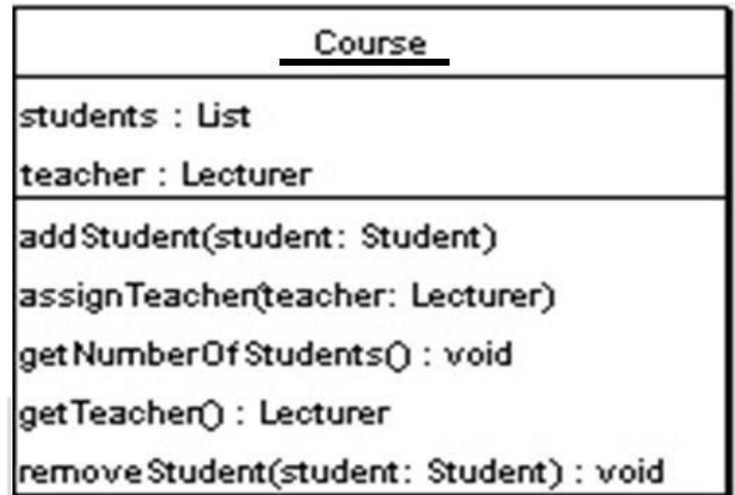
Fig a

Fig b

- The above fig.a and fig.b is divided into 3 parts the same way as the class.

- The only difference is the name which is underlined.

- As the object is an actual implementation of a class, which is known as the instance of a class, but they are used to build a prototype of a system from a practical perspective close to real-life scenarios.

Characteristics of Class

- A class is a group of objects which have common properties.
- A class in Java contains:

    - Constructors

    - Methods

    - Fields

    - Blocks

- Nested class and interface

Characteristics of Class: Fields

- A field contains things like variables, constant and other data types.

- Syntax diagram represent the field explanation to be declare in a class.

```
class <class_name>
    {
    field;
    }
```

Characteristics of Class : Methods

Methods : In Java, a method is like a function, runs when it calls. Methods can be called anywhere in the program.

Advantages:-

1. **Code reusability** methods means we can use code many times by declaring only once.

2. **Code optimization** methods means we can reduce the line of code.

- Methods should be declared inside the class.
- It should be defined by method name followed by parenthesis ().
- Some predefined methods in java, is **System.out.println**(), etc.
- A syntax for creating method is shown below.

```
public class Chair{
  static void method1() {
    // code to be executed
  }
}
```

Characteristics of Class : Constructor

Constructor: It is a block of codes similar to the method. It is a special type of method. Every class contains a minimum of one constructor. Constructor name must be the same as its class name.

A syntax for creating a constructor is shown below.

```
public class Chair
{
public Chair()
{

// initialize objects

}}
```

Characteristics of Class : Block

Block : One or more line of code statements enclosed in braces is known as block.

A block begins with an open curly brace ({) and ends with a close curly brace (}).

Syntax of a block.

```
{ //start
        int num = 50;
        num++;
} //end
```

Characteristics of Class : Nested Class & Interface

Nested Class & Interface : It is a class which is declared inside the class or interface. It can access all the fields of outer class including private members.

Syntax for nested Class & interface is shown below.

```
class OuterClass{
 //code
 class InnerClass{
  //code
 }
}
```

Characteristics of Object

- • If we consider the real world applications, we can find many objects present around us like cars, dogs, humans, chairs, books, etc.

- • Every object has some characteristics like name, method, etc.

- • An object has three characteristics :

- State

- Behavior

- Identity

- **State:** It contains the attributes and value of an object. In simple words, it represents the data of an object**.**
- **Behavior:** It represents the functionality of an object.
- **Identity:** It is used internally by Java to identify the uniqueness id of the object.
- **Explanation:** Let's take some examples to understand the term state, behavior, and identity.
- We can see car as an object example to understand the term state, behavior, identity associated with object.

  Object :- Car

  Car is an object.

- Object name is car, so uniqueness of the object is car, known as **Identity** of an object**.**

- Car has 4 wheels, wiper, brake, gears, etc., so it represents the **state** of an object**.**

- Car has braking, accelerating, gearing, etc., so it represents the **behavior** of an object.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 14:

```
//creating a class named City
public class City
{
  //declaring class variables
  public String name;
  public long population;
```

```java
//defining the method of the
class public void display()
{
  System.out.println("City name: " +name);
  System.out.println("Population: "
  +population);
}
public static void main(String args[])
{
  //declaring the objects of the class City
  City metro1,metro2;

  //Instantiating the objects of the class using the new
  keyword metro1 = new City();
  metro2 = new City();

  metro1.name ="Mumbai";
  metro1.population = 23409876;
  System.out.println("Details of metro city
  1:");
  metro1.display(); //display() method is being invoked for the object metro1

  metro2.name ="Pune";
  metro2.population = 45874294;
  System.out.println("Details of metro city
  2:");
  metro2.display(); //display() method is being invoked for the object metro2

  }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Packages in java

- Package is a kind of bundle or container or library, where we put one or more Java classes, interfaces, and other related entities/information.

- It means bundling the multiple related program files at one place. Package is the first statement of any Java program.

- Packages can be categorized into two categories, the built in package and user defined package.

- **Built-in packages:** The already predefined package by the java compiler is known as built-in packages.

- Some of the commonly used built-in packages in java are as follows:

  - **java.lang:** It contains language support classes.

  - **java.util:** It contains utility classes such as vectors, lists, hash tables, etc.

  - **java.awt:** It contains classes for the graphic user interface.

  - **java.applet:** It contains a set of classes for applets.

  - **java.net:** It contains a set of network classes.

  - **java.io:** It contains classes for input and output operation.

- **User defined packages :** These are the packages that are defined by the user. Now we will see how the packages are created and used in java.

- lets discuss user-defined packages with the help of examples.

- How to create the packages in java is shown in the figure given below :

  - Select a suitable name for the package to be created.

  - Name of the package must be same as the directory under which this file is saved.

  - Declare the name of the package with the "**package**" keyword.

  - Define a public class inside that package.

```
package packagename;

public class Classname
{
.....................
.....................
}
```

- **Importing created packages :**

    1. After the creation of packages, it can be imported using the '**import**' keyword.

    2. By importing all the classes in a package.

  e.g.   import packagename*;   or

    1. By explicitly declaring the class name to be imported

  e.g.   Import packagename.Classname;

Java Access Modifiers

- Java access modifiers are used to provide access control in java.

- Access modifiers are used with Classes as well as Class variables and methods.

- It is allowed to use only public or default access modifiers with java classes.

- Java provides three types of access control through Keywords

    1. Private

    2. Protected

    3. Public

- Access modifier is a keyword that we use to set the visibility or scope or define the boundary of variable, method, and class.

- This is also known as specifier.

- Default is the default access modifier when we do not write any modifier with class declaration.

- Default modifier makes a class accessible within the same package

Java Access Modifiers: Private

- If a class member is "**private**" then it will be accessible only inside the same class. This is the most restricted access and the class member will not be visible to the outer world.

- **Example:**

```
public class ExampleTest {
public void MethodEx() {
private String firstName = "India";
}
public static void main(String[] args)
                    {
ExampleTest exampleTest = new ExampleTest();
exampleTest.firstName = "Bharat"; // we cannot access this private variable in main
method.
} }
```

- Variable, method, and class declared private will be accessible only with the block {}.

- For example, if a private variable declares within a method, then that can only be accessible within the method body

- In the above program will generate compilation error since we are trying to access a private member of the method **firstName** inside the main method

Java Access Modifiers: Public

- If a class member is "**public**" then it can be accessed from anywhere. Also member variable or method is accessed globally.

- **Example:**

Save by foo.java

```
package jack;
public class foo{
public void msg(){System.out.println("India");}
}
```

```
package mack;
import jack.*;
```

Save by moo.java

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

```
 foo obj = new foo();
 obj.msg();
 }


}
```

- Public modifier makes everything public. The variable, method, and class declared with public modifier can be accessible to any class and method.

- There is no restriction and boundary defined for public

- The foo class of jack package is public, so can be accessed from outside the package.

- Also msg method of this package is declared as public, so it can be accessed from outside.

Java Access Modifiers: Protected

- If class member is "**protected**" then it will be accessible only to the classes in the same package and to the subclasses.

- **Example:**

```
Class ParentClass
package foo.example.moo;
public class ParentClass {
protected String cityName = "Mumbai";
protected String districtName = "CSMT";
}
```

```
Class ChildClass
package foo.example.moo;
public class ChildClass extends ParentClass {
public static void main(String[] args) {
ChildClass childClass = new ChildClass();
System.out.println(childClass.cityName);
System.out.println(childClass.districtName);
} }
```

- Access modifier **Protected** is in between private and public.

- It allows access to all classes within the same package and classes that are subclasses of other classes.

- Here, derived or subclass means protected will allow to inherit the properties from base class or parent class.

- In the above example, **ParentClass** has two protected variables **cityName** and **districtName**

- However, by using inheritance, when we inherit a class using the extends keyword, that inherited class exposes all its variables and methods to the derived class.

- This creates a relationship between two classes, as in the real-world we have a relation with our father, and we do have rights to access the assets belonging to our father.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 15:

```java
class ParentClass{
        int a = 10;
        public int b = 20;
        protected int c = 30;
        private int d = 40;

        void showData() {
                System.out.println("Inside ParentClass");
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}

class ChildClass extends ParentClass{

        void accessData() {
                System.out.println("Inside ChildClass");
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
```

```
                //System.out.println("d = " + d); // private member can't be accessed
        }

}
public class AccessModifiersExample {

        public static void main(String[] args) {

                ChildClass obj = new ChildClass();
                obj.showData();
                obj.accessData();

        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Constructor

- Constructor is a special type of method with the same name as the class name.

- Constructor is used in various ways to declare the default variable and load or make available the prerequisites of a program so that our program can execute smoothly.

- It contains a collection of statements that are executed at the time of object creation.

- The constructor is not mandatory for a programmer to write it for a class, but for the ease of program and security purposes, we make constructors.

- Let us discuss more on the type of constructors

- There are three types of constructors in Java:

    1. Default constructor

2. No argument constructor

3. Parameterized constructor.

Default Constructor

- A constructor with no parameter is treated as the default constructor.

- Every time we create an object of a class using the **new** keyword, we initialize the default constructor.

- After this, the system implicitly calls the constructor with no parameter written in the program

- If we don't write constructor then java compiler will create one by default, This constructor is known as default constructor.

- If you implement any constructor then you no longer receive a default constructor from Java compiler.

- **Example:**

```java
public class Main
{
    Main()
    {
        System.out.println("creating a default constructor");
    }
    public static void main(String[] args) {
        //calling a default constructor
        Main skillking = new Main();
    }
}
```

```
creating a default constructor
```

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 16:

```
package Hello.World;

public class Hello_World{
        Hello_World()
        {
                System.out.println("Creating a default constructor");
        }

        public static void main(String[] args) {
                //calling a default constructor
                Hello_World skillking = new Hello_World();

        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

No Argument Constructor

- Constructor with no arguments is known as **no argument constructor**.

- **Syntax** of no argument constructor is the same as default constructor, but the body can have any code, unlike default constructor.

- **Example:**

```
1
2   public class Main
3 ▾ {
4       int i;
5       Main()
6 ▾    {
7           i = 10;
8           System.out.println("i = " + i);
9       }
10 ▾   public static void main(String[] args) {
11
12          Main skillking = new Main();
13      }
14 }
```

i = 10

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 17:

```
package Hello.World;

public class Hello_World{
        int i;
        Hello_World()
        {
                i=10;
                System.out.println("Creating a No Argument constructor");
                System.out.println("i = " + i);
```

```
        }

        public static void main(String[] args) {
                Hello_World skillking = new Hello_World();


        }
    }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Parameterized Constructor

- The constructor with parameters is called a parameterized constructor.

- This is also known as constructor overloading.

- A constructor that is not identical but different in number of parameters, type of parameters, and sequence of parameters can be declared.

- We overload the constructor for different purposes and initialize the things based on the parameter or input received.

- Here, we will see in Example how to create a parameterized constructor.

```
1
2   public class Main
3 - {
4       int i;
5       Main(int i)
6 -     {
7           i = i;
8           System.out.println("Student for playing carrom: " + i);
9       }
10 -    public static void main(String[] args) {
11          Main a = new Main(1);
12          Main b = new Main(2);
13          Main c = new Main(3);
14          Main d = new Main(4);
15          Main e = new Main(5);
16      }
17  }
```

```
Student for playing carrom: 1
Student for playing carrom: 2
Student for playing carrom: 3
Student for playing carrom: 4
Student for playing carrom: 5
```

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 18:

```
class ParentClass{
        int a = 10;
        public int b = 20;
        protected int c = 30;
        private int d = 40;

        void showData() {
                System.out.println("Inside ParentClass");
                System.out.println("a = " + a);
```

```
                    System.out.println("b = " + b);
                    System.out.println("c = " + c);
                    System.out.println("d = " + d);
            }
    }

    class ChildClass extends ParentClass{

            void accessData() {
                    System.out.println("Inside ChildClass");
                    System.out.println("a = " + a);
                    System.out.println("b = " + b);
                    System.out.println("c = " + c);
                    //System.out.println("d = " + d); // private member can't be accessed
            }

    }
    public class AccessModifiersExample {

            public static void main(String[] args) {

                    ChildClass obj = new ChildClass();
                    obj.showData();
                    obj.accessData();

            }

    }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Super keyword

- The super() keyword should always be the first statement of the constructor.

- The super keyword is used to call the constructor from a super class.

- This super() will call the default constructor from the super class.

Example

```
public class SuperMethod extends SuperClass {
public SuperMethod() {
super("Java");
}
public static void main(String[] args) {
SuperMethod superMethod = new
SuperMethod();
} }
class SuperClass {
SuperClass(String name) {
System.out.println("This constructor received name
argument " + name);
} }
```

Output

```
This constructor received name argument Java
```

- In this program, we have two classes - one is **SuperMethod** and another is **SuperClass** is the parent class of the **SuperMethod** class.

- Both the classes have a constructor.

- **SuperClass** has a parameterized constructor, which is expecting an argument string to be called. On the other hand, **SuperMethod** has a default constructor.

- The above program will throw compile time exception since there is no default constructor in the parent class, but has a parameterized constructor, which we need to call explicitly.

- To fix this compilation error, call the parent class parameterized constructor with Super() keyword

this keyword

- The **this** keyword represents the members (for example, variables or methods) from same class.

- **this** works in a similar manner as super keyword.

- In the constructor, we use the **this()** keyword to call or refer a constructor, which is defined within the same class. this should be the first statement of a constructor. If you write **this()** as second statement or later, then the system will generate a compilation error.

- A class can have any number of constructors and with the help of the **this()** keyword, a constructor can communicate with other constructors within the class.

- this keyword works in a similar manner as super key

- this should be the first statement of a constructor.

Example

```
public class ConstructorThis{
ConstructorThis()
{ this(2); System.out.println("Default Constructor.");}
ConstructorThis(int i)
{ this(2,3); System.out.println("Constructor with one argument."); }
ConstructorThis(int i, int j)
{ this(2,3,4); System.out.println("Constructor with two arguments.");}
ConstructorThis(int i, int j, int k)
{ System.out.println("Constructor with three arguments.");}
public static void main(String[] args) {
ConstructorThis constructorThis = new ConstructorThis();} }
```

Output

```
Constructor with three arguments.
Constructor with two arguments.
Constructor with one argument.
Default Constructor
```

OOPS

Method Overloading

- It allows the class to have more than one method having the same name, if their argument lists are different.

- It is not possible by changing the return type of methods.

- In order to overload a method, the argument lists of the methods must differ in either of the following:

    1. **Number of argument :**

        It is allowed within the class given that the number of arguments   are not the same.

        Example :   max(int, int)   // 2 argument

        max (int, int, int)   // 3 argument

    2. **Data type of argument:**
        It is allowed within the class given that at least one pair of arguments are of different data type.

        Example :   max (int, int)   // same data type

        max(int, float)   // different data type

    Data type of argument : We have two methods with the name **max**(), one with argument of int type and another method with the argument of int & float type.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 19:

```
package Hello.World;

public class Hello_World {

        void sum(int a,long b)
                {
                System.out.println(a+b);
                }
        void sum(int a,int b,int c)
                {
                System.out.println(a+b+c);
                }

        public static void main(String[] args) {

                Hello_World obj=new Hello_World();
                obj.sum(50,50);
                obj.sum(50,50,50);
        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 20:

```
package Hello.World;

public class Hello_World {

        void sum(int a,int b)
                {
                System.out.println(a+b);
                }
        void sum(long a,long b)
                {
                System.out.println(a+b);
                }

        public static void main(String[] args) {

                Hello_World obj=new Hello_World();
                obj.sum(50,50);
                obj.sum(50,50);
        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

3. Swapping argument data type : It is allowed within the class given that the order of the data type variables are not the same.

      Example :   **max(int, float)**   // 1st int & 2nd float

                             max(float, int)   // 1st float & 2nd int

First method is having argument list as (int, float) and second is having (float, int). Both the methods have different sequence of data type in argument list

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 21:

```java
package Hello.World;

public class Hello_World {

        void sum(int a,double b)
                {
                System.out.println(a+b);
                }
        void sum(double b,int a)
                {
                System.out.println(a+b);
                }

        public static void main(String[] args) {

                Hello_World obj=new Hello_World();
                obj.sum(50,50.0);
                obj.sum(50.0,50);

        }

}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Method overriding & overridden

- Declaring a method in sub class which is already present in parent class is known as method overriding.

- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.

- In this case the method in parent class is called overridden method and the method in child class is called overriding method.

- A method declared in **child class** but it is already present in the **parent class** is known as method overriding.

- The method declared in the parent class is called overridden method and the method in the child class is called the overriding method.

- Method overriding is used for runtime polymorphism.

- A method declared static cannot be overridden but can be redeclared.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 22:

```
package Hello.World;

//Creating a parent class.
class Books{
//defining a method
        void run()
                {
                System.out.println("Book not in stock");
                }
}

//Creating a child class
```

```
public class Hello_World extends Books{
        void run()
                {
                System.out.println("Book is available");
                }
        public static void main(String[] args) {
                Hello_World obj = new Hello_World();   //creating object
                obj.run();    //calling method

        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Encapsulation

Encapsulation is the concept of hiding the data, variable, and method from external interaction. We create encapsulation for hiding the members of class by declaring the members as private to restrict other classes from directly interacting with variables. To expose and manage the member variables of an encapsulated class, we need to create public getter and setter for those variables. Hiding data lets no other developer know which variable is storing value and how manipulation of that is happening. However, a developer can use getter and setter methods to achieve and use the functionality defined in the encapsulated class.

- Let's understand the concept of encapsulation with an example.

- We have a car that has many features and functionalities to use such as start, stop, play music, change gear and many more. We are just using the trigger to perform the functionality, but we do not know about internal processing and how these things are happening internally. All these features are encapsulated into a single object that we call a car. In the same way, we are developing encapsulation in Java classes, by hiding the actual data and encapsulating everything into a class.

Abstraction

- Abstraction is a process where you show only "relevant functional details" and "hide irrelevant details" of an object from the user.

- Abstraction reduces the complexity of the view of objects to the user.

- It increases security as information is kept hidden.

- Abstraction is one of the most important features of oops.

- Abstraction is a process of hiding the implementation details and only showing the functionality to the user.

- Abstraction is created using abstract classes and interfaces.

- There are two ways to achieve abstraction in java are as follows:

  1. Abstract class

  2. Interface
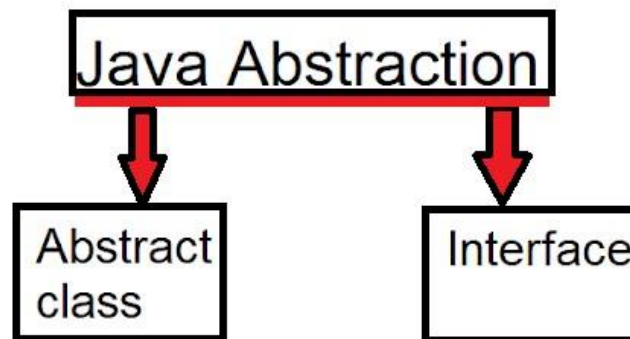
Abstract class

- A class that is declared using "**abstract**" keyword is known as abstract class.

- An Abstract class is a class that contains abstract as well as concrete methods.

- We cannot create the object of an abstract class.

- An abstract class can have static methods and constructors.

- A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

- An abstract class can have an abstract and non abstract method.

- A class can not be declared with both final and abstract keywords, because final keyword is used to prevent overriding whereas abstract methods need to be overridden.

- An abstract method does not have implementation and body.

- It defines only the signature of the method.

- Abstract classes must be declared with an abstract keyword.

- We can not create an object of abstract classes. It can only be used as a reference.

- To use an abstract class, we need to create another class which extends this class.

- An abstract method is a method that has only the method definition.

- An abstract method does not have implementation and body. It defines only the signature of the method.

```
abstract class className{
    //body of class

    abstract  methodName(); // abstract method
}
```

Abstract class Exampl

```
abstract class Parent{
    abstract void Show();  //this is abstract method
}

class Child extends Parent{
    void Show(){
        System.out.println("Show method in Child class");
    }
}

class Main{
    public static void main(String[] args ){
        //Parent p1 = new Parent();       will give error because it is defined as abstract

        Parent p2 = new Child();
        p2.Show();

        Child c1 = new Child();
        c1.Show();
    }
}
```

- In the given example, parent is the abstract class and its implementation is provided by the child class.
- We use the abstract keyword to declare an class and method.
- We have to extend abstract classes to subclass and implement all the abstract method in the subclass.
- We can not create an object of abstract class parent.

Purpose of Abstract class

- Lets assume we have a class bird that has a method sound() and the subclasses of it like Parrot, Crow, Peacock, Duck etc.

- The bird sounds differs from one bird to another, there is no point to implement this method in parent class.

- Now every child class must override this method to give its own implementation details, like duck class will say "quack", peacock class will say "scream" and so on.

- So now when we know that all the bird child classes will and should override this method, then there is no point to implement this method in parent class.

- Therefore, making this method abstract would be the good choice as by making this method abstract we force all the sub or child classes to implement this method.

- Now each bird must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method.

- This way we ensures that every bird has a sound.

- So this is the purpose of abstract keyword before any class name.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 23:

```
package Hello.World;

abstract class Bird {
        abstract void makeSound();

        }

        class Duck extends Bird {
          public void makeSound() {
            System.out.println("The sound that a Duck makes : Quack Quack");

          }
        }

        class Owl extends Bird {
          public void makeSound() {
            System.out.println("The sound that an Owl makes: Hoot Hoot");

          }
        }

public class Hello_World{

        public static void main(String[]
                args) { Duck d = new
                Duck();
          d.makeSound();

          Owl c = new Owl();
          c.makeSound();

        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Inheritance

- The process in which one class (object) acquires all the properties and behaviors (fields and methods) of another class (parent) is known as inheritance.

- When we inherit from an existing class, we can reuse fields and methods of the parent class.

- Java Inheritance is one of the most important features of object oriented programming.

- We can declare new fields in the subclass that are not in the superclass.

- In real life, a child inherits the properties from his father just like that inheritance represents a parent child relationship which is also known as IS-A relationship.

- In inheritance "extends" keyword is used to inherit a class.

- The extends keyword defines that we are making a new class that is derived from an existing class.

- HAS-A simply mean the use of instance variables that are references to other objects. For example, Maruti has Engine, or House has Bathroom.

- Some terms used in Inheritance are :

    - **Class :** It is a blueprint from which objects are created.

    - **Sub Class/Child Class:** It is a class which inherits the other class. It is also called a derived class, extended class, or child class.

    - **Super Class/Parent Class:** The class whose properties and functionalities are used (inherited) by another class is called as base class or a parent class.

    - **Reusability:** We can use the same fields and methods already defined in the previous class.

    - **Extends keyword :** It indicates that you are making a new class that derives from an existing class.

    - **Super keyword :** It is used to access methods of the parent class.

    - **This keyword :** It is used to access methods of the current class.

- The following example explains the syntax of inheritance in java:

- In Java, 'extends' keyword is used to inherit a class. Here the ParentClass is the name of parent class from which the ChildClass is acquiring the properties and ChildClass is the name of child class. The ChildClass is inheriting the properties and methods of ParentClass.

```
class ParentClass
{
//methods and variables
}
class ChildClass extends ParentClass
{
   //methods and variables
}
```

- **Types of Inheritance :**

  There are five types of Java Inheritance.

  1. Single Inheritance

  2. Multilevel Inheritance

  3. Hierarchical Inheritance

  4. Multiple Inheritance

  5. Hybrid Inheritance

- Multiple and Hybrid inheritance can be achieved only through Interfaces.

- **Note:** Java does not support Multiple and Hybrid Inheritance with classes. Multiple and Hybrid inheritance can only be achieved only through Interfaces.

Single Inheritance

- **Single Inheritance :** In single inheritance, the features and methods of the parent class are inherited by a single child class.
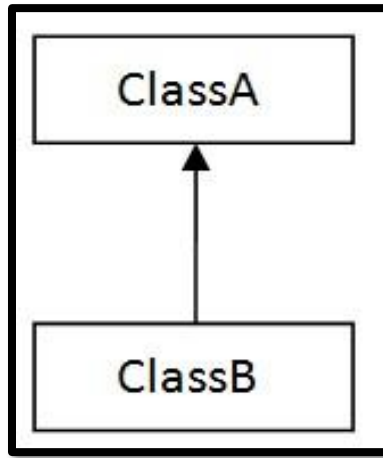
- In the given single inheritance figure, Class A is parent class, and Class B is Child Class, and class B is inheriting the properties, behavior, features and methods of class A.
- This process is known as single inheritance.
- Single inheritance enables a child class to inherit properties and behavior from a single parent class.
- It allows a child or derived class to inherit the properties and behavior of a base or parent class, thus enabling code reusability as well as adding new features to the existing code.
- In the syntax below, the class A serves as a parent class for the child class B

```
public class A{
        ............
}
public class B extends A {
        ............
}
```

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 24:

```java
package Hello.World;

class ABC
{
    public void print_ABC()
    {
        System.out.println("Cooking");
    }
}

class XYZ extends ABC
{
    public void print_XYZ()
    {
        System.out.println("Emerging Technology");
    }
}

public class Hello_World{

        public static void main(String[] args) {
                XYZ d = new XYZ();
    d.print_ABC();
    d.print_XYZ();


        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Multilevel Inheritance

- In multilevel inheritance, a child class will be inheriting a parent class as well as the child class, and also parent class for some other class.
- In the given diagram, The class A serves as a base or parent class for the derived or child class B, which in turn serves as a base or parent class for the child or derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between class A and class C.
- This process is known as multilevel inheritance.

- The image shown here explains multilevel inheritance.

```
ClassA
  ↑
ClassB
  ↑
ClassC
```
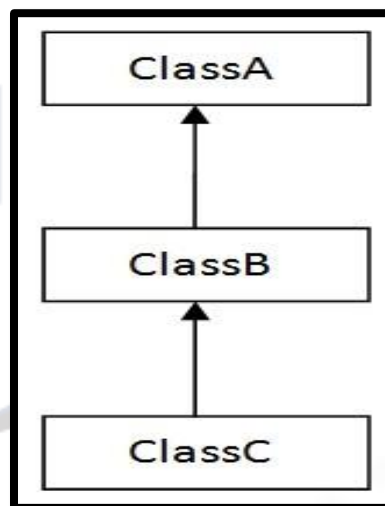
- Multilevel inheritance refers to a relationship between child and parent class where a class extends the child class.

- In the syntax shown, class C extends class B and class B extends class A.

- Therefore class A serves as a base or parent class for the derived or child class B, which in turn serves as a base or parent class for the child or derived class C.

- In the syntax below, class A serves as a parent class for the class B, and also class B serves as a parent class for class C.

```
class A{
    ...........
}
class B extends A{
    ...........
}
class C extends B{
    ...........
}
```

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 25:

```
package Hello.World;

class ABC
{
    public void print_ABC()
    {
        System.out.println("Cooking");
    }
}

class XYZ extends ABC
{
    public void print_XYZ()
    {
        System.out.println("Emerging Technology");
```

```
}
  }

  class PQR extends XYZ
  {
    public void print_PQR()
    {
      System.out.println("IoT");
    }
  }

  public class Hello_World{

        public static void main(String[] args) {
              PQR d = new PQR();
    d.print_ABC();
    d.print_XYZ();
    d.print_PQR();

    }
  }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Hierarchical Inheritance

- **Hierarchical Inheritance :** In hierarchical inheritance, more than one subclasses inherit the properties, behavior, features and methods from a single parent class.

- The following image explains hierarchical inheritance, It refers to a child and parent class relationship where more than one child classes extends the same parent class, like child class B & C extends the same parent class A.

- This process is known as hierarchical inheritance.
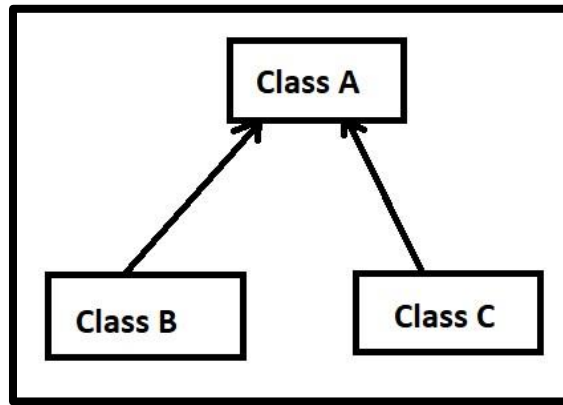
- Hierarchical inheritance includes more than one child classes or we can say that more than one child class have one same parent class from where child class can inherit all features and functionalities from a superclass.

- Therefore child class B & C extends the same parent class A.

- In the syntax above, Class B and Class C are the subclasses of parent Class A.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 26:

```
package Hello.World;

class ABC
{
    public void print_ABC()
    {
        System.out.println("Cooking");
    }
```

```
}

class XYZ extends ABC
{
   public void print_XYZ()
   {
      System.out.println("Emerging Technology");
   }
}

class PQR extends ABC
{
   public void print_PQR()
   {
      System.out.println("IoT");
   }
}

public class Hello_World{

        public static void main(String[] args) {
                PQR d = new PQR();
      d.print_ABC();
      XYZ f = new XYZ();
      f.print_XYZ();


        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Multiple Inheritance

- **Multiple Inheritance :** In Multiple inheritance one class can have more than one superclass and inherit properties, behavior, features and methods from all parent classes.
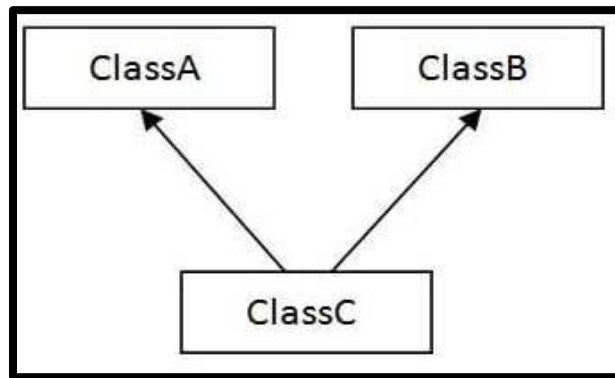
- It refers to the concept of one child class extending more than one parent classes, which means a child class has two parent classes.
- For example in the figure shown below class C extends both classes A and B.
- Java doesn't support multiple inheritance but it is supported through interface only.
- This process is known as multiple inheritance

- In the syntax below, Class C is derived from interface A and B.

```
public interface A {
        ..............
}
public interface B {
        ..............
}
public interface C extends A,B {
        ..............
}
```

- Multiple Inheritance is a feature of object oriented concept, where a child class can inherit properties of more than one parent class.
- So in the syntax given below child class C extends both parent classes A and B

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 27:

```java
package Hello.World;

interface ABC
{
    public void print_ABC();

}

interface XYZ
{
    public void print_XYZ();

}

interface PQR extends ABC, XYZ
{
        public void print_PQR();
}

class child implements PQR{
        public void print_ABC()
    {
        System.out.println("Cooking");
    }
        public void print_XYZ()
    {
        System.out.println("Emerging Technology");
    }
        public void print_PQR()
    {
        System.out.println("IoT");
```

```
        }

    }

    public class Hello_World{

            public static void main(String[]
                        args) {
                    child d = new
                        child();
        d.print_A
        BC();
        d.print_X
        YZ();
        d.print_P
        QR();

            }
    }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Hybrid Inheritance

- **Hybrid Inheritance :** Hybrid Inheritance is a combination of both Single Inheritance and Multiple Inheritance.

- Since in Java Multiple Inheritance is not supported directly we can achieve Hybrid inheritance also through Interfaces only.
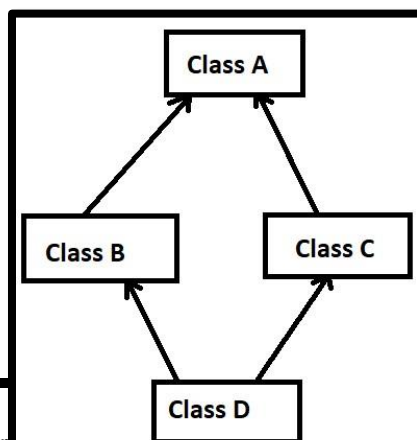
- As we can see in the above diagram Class D is the Parent class for both Class B and Class C which is single inheritance and again Class B and Class C act as Parent for Class A which is multiple inheritance.
- Java doesn't support hybrid inheritance but it can be achieved through interface only.

- This process is known as hybrid inheritance

- Hybrid Inheritance is a combination of both Single Inheritance and Multiple Inheritance.

- Java doesn't support hybrid inheritance but it can be achieved through interface only.

- As we can see in the above code the Class D has implemented both the interfaces class B and class C. In this case we didn't have ambiguity even though both the interfaces are having same method.

```
public class ClassA
{
        ...............
}
public interface ClassB
{
        ...............
}
public interface ClassC
{
        ...............
}
public class ClassD extends ClassA implements ClassB,ClassC
{
        ...............
}
```

- In the syntax below, class D inheriting both class B and class C though interface otherwise it will give error at runtime.

- Hybrid Inheritance is a combination of both Single Inheritance and Multiple Inheritance.

- Java doesn't support hybrid inheritance but it can be achieved through interface only.

- As we can see in the above code the Class D has implemented both the interfaces class B and class C. In this case we didn't have ambiguity even though both the interfaces are having same method.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 28:

```java
package Hello.World;

interface A
{
    public void A();
}
interface B extends A
{
    public void B();
}
interface C extends A
{
    public void C();
}

public class Hello_World implements B, C{

        public void A()
    {
        System.out.println("Inside A");
    }
    public void B()
    {
        System.out.println("Inside B");
    }
    public void C()
    {
        System.out.println("Inside C");
    }

        public static void main(String[] args) {
                    obj1.A(); obj1.B(); obj1.C();

    }
```

```
Hello_World
obj1= new
Hello_World();


}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants

Interface

- Interface looks like a class but it is not a class.

- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract means only method syntax is defined not body.

- Interfaces are declared by specifying a keyword "interface".

- All the interface methods are by default **abstract and public**.

- Variable names conflicts can be resolved by interface name.

- Defining an interface is similar to defining a class, but a class describes the attributes and behaviors of an object, and an interface contains behaviors that a class implements.

- An interface is not extended by a class it is implemented by a class.

- An interface can extend multiple interfaces.

- A class uses the **implements** keyword to implement an interface.

- An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface.

- Java Interfaces are used to achieve abstraction.

```
interface interfaceName{
    //declare constant fields, abstract methods

        ....................
}

class className implements interfaceName{

        ....................
    //defining all the abstract methods
}
```

Polymorphism in Java

- It is the process of representing one form in multiple forms known as **Polymorphism,** in simple words it is the **OOPs** feature that allows to perform a single action but in different ways.

- Polymorphism is derived from two greek words **poly** and **morphs**.

- The word "**poly**" means many and "**morphs**" means forms.

- So polymorphism means many forms.

- Let's understand the concept of Polymorphism in Java with a real time example.
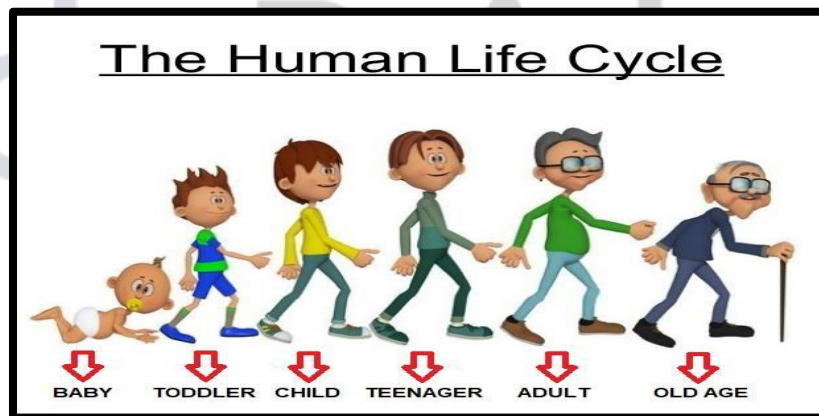
- Life cycle is the developmental stages that occur during an organism's life time.
- The major stages of the **human life cycle** include pregnancy, infancy, the toddler years, childhood, teenager, adulthood, middle age, and the senior years i.e. old age, Here one person present in different different behaviors.
- so it means having many forms, therefore polymorphism can be defined as a mechanism for getting different characteristics of the same instance.
- Polymorphism is not a programming concept but it is one of the principal of objects oriented programming language.
- For many objects oriented programming language polymorphism principle is common but implementations are varying from one objects oriented programming language to another object oriented programming language.
- There are mainly two types of polymorphism in Java as shown below:
    - compile-time polymorphism
    - Runtime polymorphism

- Compile time polymorphism is also known as static polymorphism or early binding.

- Static polymorphism in java is achieved by method or function or operator overloading.

- When there are multiple functions with same name but different arguments then these functions are said to be overloaded.

- In order to overload a method, the argument lists of the methods must differ in either of the following:

- Method Overloading allows to have more than one method having the same name, if the arguments of methods are different in number, sequence and data types of parameters.

- In the following example, we have three definitions of the same method max().

- So the max method would be determined by its arguments at the compile time that's why it is known as **Compile Time Polymorphism**.

- **Example : max(int, int)**   // 2 argument

Approach 1 : Different in number

max (int, int, int)   // 3 argument

Approach 2 : Different in sequence

max(int, float)   // different data type

Approach 3 : Different in data types

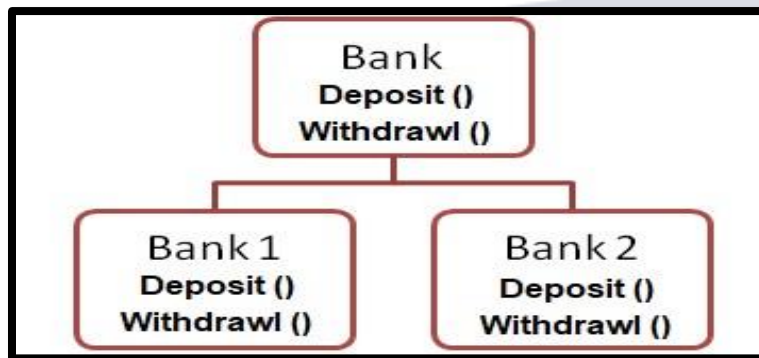**max(float, int)**   // 1st float & 2nd int

Early Binding

- It is a compile time process.

- The method definition and method call are linked during the compile time.

- Actual object is not used for binding.

- Program execution is faster.

- The binding of static, private and final methods are done at compile time.

Runtime Polymorphism

- Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism.

- Runtime polymorphism is also known as dynamic or late binding.

- Runtime polymorphism in java is achieved by method overriding.

- Overriding allows a child class to implement a method that is already provided by one of its parent class.

- Let us consider a following scenario for understanding method overriding.

- Let us consider a scenario where bank is the parent class which provides functionality of deposit & withdrawal, also there are two child classes overriding these functions.

Late Binding

- It is a run time process.

- The method definition and method call are linked during the run time.

- Actual object is used for binding.

- Program execution is slower.

- The binding of static, private and final methods are not done.

Exception Handling

- An Exception is an unwanted event that interrupts the normal flow of the program.

- Errors are generated while writing a programming code.

- So these errors are displayed at compile time.

- Some of these errors do not show up at compile time but interrupts the normal flow of execution at run time.

- These errors are known as Exceptions in programming.

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- This is something that every programmer faces at any point of coding. They can occur from different kind of scenarios like entering the wrong data by user, hardware failure, network failure, class not found, out of memory, etc.

- Exception Handling is a mechanism to handle runtime errors.

- The main advantage of exception handling is to maintain the normal flow of the application.

- All exception and errors types are sub classes of class **Throwable.**

- Suppose if an exception is not handled, it may lead to a system failure. That is why handling an exception is very important.

- Java provides specific keywords for exception handling like throw, throws, try, catch, finally.

What is Java Exception ?

Let's discuss with an example. Mr. Foo is going for a interview; suddenly, he saw a big wall of bricks in the middle of his way. This wall is an obstacle for Mr. Foo, how he will handle this situation. Mr. Foo changes his direction and saw a

hurdle which can be easily crossed and can be reach to the other side of the wall, therefore, he gets over the hurdle. Therefore exception Handling in Java is the same process.

Exceptions in JAVA	Errors in JAVA
1. Occurs at compile time or run time	1. Occurs at runtime
2. Possible to recover when exception occurs	2. Impossible to recpover when error occurs
3. Type of exception may be either checked and unchecked	3. Error are of type unchecked
4. An exception is caused because of the code.	4. An error is caused due to lack of system resources.
5. Example : Out of memeory, etc.	5. Example : Class not found

Types of Java Exceptions :

- There are mainly two types of exceptions: **checked and unchecked**.

- According to oracle an error is also considered as the type of exception, but **error** is also considered as the unchecked exception, there are three types of exceptions:

    1. Checked Exception

    2. Unchecked Exception

    3. Error

Checked Exception

- A checked exception is an exception that is checked by the compiler at the time of compilation, these are also called as compile time exceptions.

- Checked exception are directly sub class of java.lang.Exception class.

- These exceptions occur at compile time and without handling them, the program cannot be executed.

- **Example:** IOException, ClassNotFoundException, etc.

Unchecked Exception

- An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**.

- Unchecked exceptions are extended from the java.lang.RuntimeException class.

- Java Virtual Machine (JVM) handles such exceptions.

- **Example :** ArithmeticException, ArrayIndexOutOfBoundsException, etc.

Error

- Errors in Java are normally overlooked.

- They are irrecoverable.

- They occur due to some scarcity of system resources.

- Error is also one type of unchecked exception.

- They are usually created in response to catastrophic failure

- **Example:** Hardware error, NoSuchMethodError, OutOfMemoryError, etc.

Java Exception Keywords

- Below 5 keywords are used to handle exceptions in Java.

    1. try

2. catch

3. finally

4. throw

5. throws

Java Exception Keyword: try

- **try:**

The "**try**" keyword is a block where we put exception code. The "**try**" block cannot be used alone. A try block must be followed by catch blocks or finally block or both.

Syntax :

try{

      //**statements which cause an exception**

  }

Java Exception Keyword: catch

- **Catch :**

It must be used along with "**try**" block. It is used to handle the exception. Multiple catch blocks are possible in Java to handle multiple types of exceptions. The catch block catches the exception thrown by the try block.

Syntax :

try {

// **statements which cause an exception**

}

catch (exception(type) e(object))

{

//**code for handling error**

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 29:

```
package Hello.World;


public class Hello_World{


        public static void main(String[] args) {
                try
                {
                        int[] numbers = {1, 2, 3};
                        System.out.println(numbers[50]);
                }
                catch (Exception e)
                {
                        System.out.println("Some Error in code.");
            }

        }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Java Exception Keyword: finally

- **finally:** The "finally" block is used to execute the important code of the program. It will be executed irrespective of whether the exception is handled or not. There should be only one "**finally**" block even if code is having multiple try-catch block.

Syntax:

```
try
{
// statements which cause an exception
}
catch
{
// code for handling error
}
finally
{
// Statements to be executed
}
```

- A finally block appears at the end of the catch blocks and has the following syntax.
- The statements present in this block will always execute regardless of whether exception occurs in try block

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 30:

```java
package Hello.World;


public class Hello_World{


    public static void main(String[] args) {
        try
        {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[50]);
        }
        catch (Exception e)
        {
            System.out.println("Some Error in code.");
        }
        finally
        {
            System.out.println("The finally statement lets you execute code, after try...catch, "
                    + "regardless of the result");
        }


    }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Java Exception Keyword: throw

- **throw:** The "**throw**" keyword is used to throw a user defined exception from a method or any block of code. We can throw either checked or unchecked exception in java by throw keyword.

    Syntax :

    throw exception;

Example :

throw new ArithmeticException("Divide by zero");

- **throw:** The "**throw**" keyword is used to throw a user defined exception from a method or any block of code. We can throw either checked or unchecked exception in java by throw keyword.

Syntax :

throw exception;

Example :

throw new ArithmeticException("Divide by zero");

- In this example, we have created the validate method that takes number value as input in the form of arguments. If the divisor value is zero, we are throwing the ArithmeticException.
- ArithmeticException is thrown to indicate that a method has been passed an illegal or inappropriate argument when the divisor is zero.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 31:

```
package Hello.World;


public class Hello_World{

        static void Age(int age) {
          if (age < 18)
          {
            throw new ArithmeticException("Access denied must be at least 18 years old to watch movie");
```

```
        }
        else
        {
          System.out.println("Access granted to watch movie");
        }
      }

      public static void main(String[] args) {

              Age(14);

      }
  }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Java Exception Keyword: throws

- **throws:** The "**throws**" keyword is used to declare exceptions. **Throws keyword** is used for handling checked exceptions. Throws keyword can be used to declare multiple exception at a time.

  Syntax :

  type method_name(parameter_list) throws exception_list

  {

  // definition of method

  }

Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the **throws** keyword.

Java collections

Java Collections Framework

- Collections framework provides a set of interfaces and classes to implement various data structures and algorithms.

- Collections framework is contained in java.util package.

- It allows the programmers to program at the interfaces, instead of the actual implementation.

- A well designed framework can improve your productivity and provide ease of maintenance.

- The Java Collection Framework package (java.util) contains:

    1. A set of interfaces

    2. Implementation classes

    3. Algorithms (like sorting and searching)

- Java Collections are similar to containers that consists of multiple items in a single unit. for e,g. collections of books, list of names etc.

- Collections framework provides unified architecture for manipulating and representing collections.

- Java collection framework can perform following activity :

    - Add objects to collection

    - Remove objects from collection

    - Search for an object in collection

    - Retrieve/get object from collection

- Collection framework contains different types of collections such as lists, sets, maps, stacks, queues, etc.

Following is the Java Collections Framework hierarchy:

- Collection interface is the root interface from which the interfaces List, Set,Queue are extended.
- There are some other classes in collection framework which do not extend Collection Interface they implement Map interface.
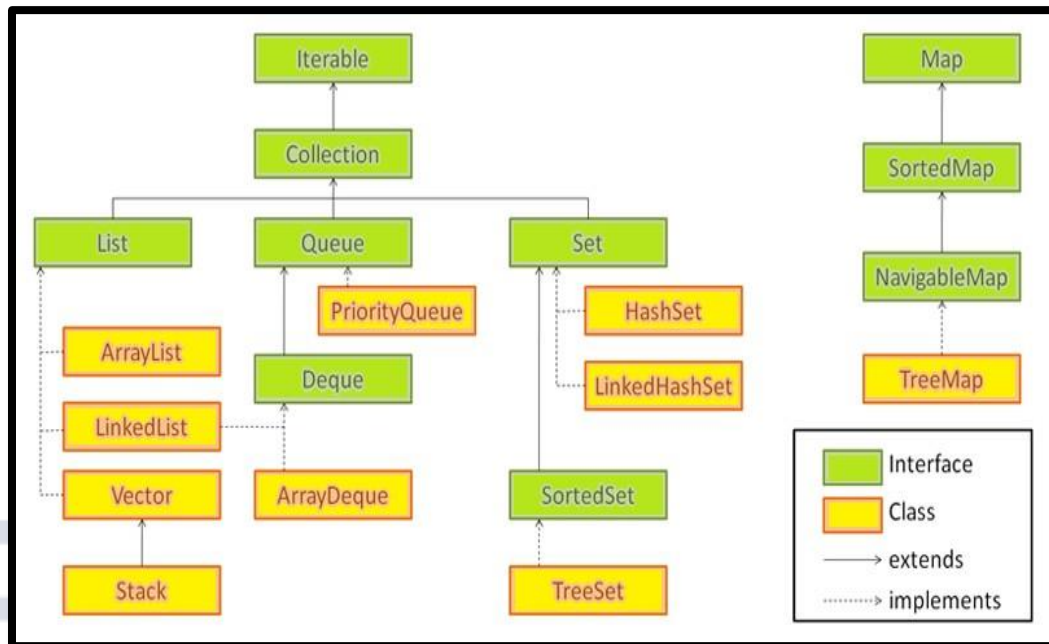
- **Collections Interfaces :**

  - This is the root of the collection hierarchy.

  - A collection represents a group of objects known as its elements.

  - List, Queue and Set are all sub interfaces of collection interface.

  - It **extends Iterable** interface.

  - Iterable interface has only one method called iterator().

- **List Interface :**

  - **List Interface** represents an ordered or sequential collection of objects.

  - The classes which implement the List interface are called as Lists.

  - Elements can be inserted or removed from a specific position.

  - List interface extends Collection interface.

  - List interface is an ordered collection in which duplicate elements are also allowed.

- **Queue Interface:**

  - The **Queue Interface** extends Collection interface.

  - Queue is a data structure in which elements are added from one end and elements are deleted from another end.

  - It follows FIFO pattern i.e. First in First out except priority queue.

  - In a priority queue, elements are assigned priorities

- **Deque Interface:**

  - The Deque Interface is the short name for **"Double Ended Queue".**

  - The Deque interface defines the methods needed to insert, retrieve and remove the elements from both ends.

  - It follows both (queue) FIFO & (stack) LIFO pattern.

  - Deque can have duplicate elements.

- **Set Interface:**

  - The **Set interface** defines a set.

  - The set is a linear collection of objects with no duplicates.

  - The Set interface extends Collection interface.

  - Methods are inherited from collection interface as it does not have it's own methods.

- **SortedSet Interface:**

  - The **SortedSet interface** extends Set interface.

  - SortedSet is a set in which elements are placed according to supplied comparator.

  - This Comparator is supplied while creating a SortedSet.

  - If you don't supply comparator, elements will be placed in ascending order.

- Unlike others, it doesn't inherit from Collection interface.

- Map is an object of key-value pairs where each key is associated with a value.

- A map can not have duplicate keys but can have duplicate values.

- Each key-value pairs of the map are stored as Map.

- The common implementations of Map interface are HashMap, LinkedHashMap and TreeMap.

- **SortedMap Interface:**

    - It provides sorting of keys stored in a map.

    - It extends the Map interface.

    - The SortedMap interface includes all the methods of the Map interface.

    - In order to use the functionalities of the SortedMap interface, we need to use the class TreeMap that implements it.

- **NavigableMap Interface :**

    - It is considered as a type of SortedMap.

    - In order to use the functionalities of the NavigableMap interface, we need to use the TreeMap class that implements Navigable Map.

    - NavigableMap extends the SortedMap interface.

- **ArrayList Class:**

    - ArrayList is same like normal array but it can grow and shrink dynamically to hold any number of elements.

    - ArrayList class implements List interface and extends AbstractList.

    - It also implements 3 interfaces like RandomAccess, Cloneable and Serializable.

    - In Arraylist first element will be placed at index 0 and last element at index n-1.

    - ArrayList can have duplicate elements

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 32:

```java
import java.util.ArrayList;
public class arrayList {
public static void main(String[] args)
{
 ArrayList list = new ArrayList();
list.add("FIRST");
list.add("SECOND");
list.add("THIRD");
list.add("FOURTH");
list.add("FIFTH");
for (int i = 0; i < list.size(); i++) {
System.out.println(list.get(i));
}
System.out.println("Size of array " + list.size());} }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

- **Vector Class:**

    - The **Vector Class** is also dynamically growable and shrinkable collection of objects like an ArrayList class.

    - Vector class is preferred over ArrayList class for multi threaded application.

    - All methods of Vector class are synchronized so that only one thread can execute them at any given time.

    - The main feature of Vector class is that it is thread safety.

- **LinkedList Class:**

  - Elements in the LinkedList are called as nodes.

  - Each node consist of 3 parts i.e. Reference To Previous Element, Value Of The Element and Reference To Next Element.

  - Elements can be inserted at both the ends and also in the middle of the LinkedList.

  - The LinkedList can be used as stack. It has the methods pop() and push() which make it to function as Stack.

- **PriorityQueue Class:**

  - Elements in the PriorityQueue are arranged according to supplied Comparator, if not than elements will be placed in their natural order.

  - The PriorityQueue is a special type of queue because it is not a First in First out (FIFO) as in the normal queues.

  - PriorityQueue class extends AbstractQueue class which in turn implements Queue interface.

- **ArrayDeque Class:**

  - It does not have any restrictions on capacity. It expands automatically as we add more elements.

  - ArrayDeque can be used as a stack (LIFO) as well as a queue (FIFO)

  - The ArrayDeque class extends AbstractCollection class and implements Deque interface. It also implements Cloneable and Serializable marker interfaces.

- **HashSet Class:**

  - The **HashSet class** in Java is an implementation of Set interface. HashSet is a collection of objects which contains only unique elements.

  - The HashSet internally uses HashMap to store the objects.

- Duplicates are not allowed in HashSet.

- HashSet class extends AbstractSet class and implements Set interface. It also implements Cloneable and Serializable marker interfaces.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 33:

```java
import java.util.HashSet;
import java.util.Iterator;
public class HashSetCollection
{
public static void main(String[] args)
{
HashSet cities = new HashSet();
// Below line of code will remove element from HashSet

cities.add("New Delhi");
cities.add("Mumbai");
cities.add("Chennai");
cities.add("Kolkata");

//Below line of code will remove element from HashSet

cities.remove("New Delhi");
Iterator iterator = cities.iterator();
while(iterator.hasNext()) {
String nameOfCity=(String) iterator.next();
System.out.println(nameOfCity);
}}}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

- **LinkedHashSet Class:**

  - LinkedHashSet internally uses LinkedHashMap to store it's elements just like HashSet.

  - LinkedHashSet maintains insertion order.

  - LinkedHashSet is recommended over HashSet to maintain insertion order.

  - The LinkedHashSet class extends HashSet class and implements Set interface. It also implements Cloneable and Serializable marker interfaces.

- **TreeSet Class:**

  - Elements in TreeSet are sorted according to supplied Comparator, if not than elements will be placed in ascending order.

  - TreeSet is not synchronized.

  - TreeSet internally uses TreeMap to store it's elements just like HashSet and LinkedHashSet

  - The TreeSet class in java is a direct implementation of NavigableSet interface which in turn extends SortedSet interface

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 34:

```java
import java.util.TreeSet;
public class TreeSetJavaCollection
{
public static void main(String[] args)
{
TreeSet treeSet = new TreeSet();

//Adding elements to treeSet
```

```
treeSet.add("A");
treeSet.add("Z");
treeSet.add("N");
treeSet.add("K");
treeSet.add("B");
treeSet.add("D");
treeSet.add("Y");
System.out.println(treeSet);
} }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

- **TreeMap class:**

  - TreeMap provides an efficient way to store key/value pairs in sorted ascending order.

  - Java TreeMap contains only unique elements keys duplication are not possible.

  - Java TreeMap cannot have a null key but can have multiple null values.

  - Java TreeMap is non synchronized.

  - It implements the NavigableMap interface and extends AbstractMap class.

File Input/Output

File

- The **File** class is the class that provides access to the file system to JVM.

- Syntax to declare an object of the File class is as follows:

File newFile = new File(including name of the file);

- A pathname could be either relative or absolute path.

- A relative path must be interpreted in terms of information taken from some other pathname or current path of the application, which is typically the path where JVM is running the application and program.

- An absolute pathname is complete, including drive and directory in which no other information is required to locate the file.

- By default, the classes in the java.io package always resolve relative pathnames against the current user directory

Example:

```java
import java.io.File;
import java.io.IOException;
public class FileExample {
public static void main(String[] args) throws IOException {
File newFile = new File("ListOfCourses.txt");
if (newFile.createNewFile()) {
System.out.println("File created.");
} else {
System.out.println("File exists.");
} } }
```

Output:

File created.

- These program will create the file, **ListOfCourses.txt** under the current project directory, because we have given only the filename as a parameter of the **File** class, which will be treated as a relative path by JVM.

- The next line of code is checking the file presence of file with the same name in the system using the method **createNewFile()** that returns true or false.

- Based on the return of we are printing a text message on console; so, in our case, the method returns true and create a file.

- Hence the cursor goes in to the if section and prints the message **"File created"**

Stream

- Stream is the sequence of data flowing from source to destination. Here, source is called input and destination is called output. Input and Output streams support many data types such as character, string, and object. There are two types of streams - input and output and both implement byte and character type of streams to read and write the data

- **Byte** Byte Streams are used to perform input and output of 8-bit bytes. Classes such as **FileInputStream** and **FileOutputStream** support byte streams.

- **Character** Character Streams are used to perform input and output of 16-bit Unicode characters. Classes such as **FileReader** and **FileWriter** support character streams

Input Stream & File Reader

- Input is the process to read the data from any source such as file, device, socket, and console.

- When a third-party application of human interact with system, the system will expect some input to process further steps.

- Input Stream helps us to read the bytes coming from external sources.

- **InputStream** is an abstract class and super class of all the input classes

- **FileReader** is the class that helps to read the characters or text from file using the default buffer size.

- This only reads the stream of characters from file and makes a connection with the file.

FileInputStream

- **FileInputStream** is a class that helps us to obtain input byte from file and build a connection between file stored in the storage system and application.

- The syntax of **FileInputStream** constructor for opening connection to a file, the parameter object or name is the file stored in the file system:

  FileInputStream(File fileObject) or
  FileInputStream(String fileName)

- Output stream writes data as an output into an array or file or any output device.

- This is an abstract class, and its sub-classes get implemented to generate the output.

- Class that is part of output streams are FileOutputStream

- Let's discuss output streams in detail

FileOutputStream

- FileOutputStream is used to create a file in the filesystem and write data into that file.

- If you try to write data into an existing file using FileOutputStream and that named file doesn't exist in the file system, then FileOutputStream will create a file with given name and add the data into it.

- Let's see the syntax for creating

    OutputStream f = new FileOutputStream("path of the file. Or object of the File class")

- To create a file using FileOutputStream, we need to pass an argument that will call the constructor and create the file. The argument can be an absolute or relative path of the file, or an object of File class

Example:

```
import java.io.FileOutputStream;import java.io.IOException;
import java.io.OutputStream;
public class OutputStreamExample {
public static void main(String args[]) {
try {
OutputStream oStream = new FileOutputStream("Output.txt");
oStream.write(70);
oStream.write(65);
oStream.write(78);
oStream.close();
System.out.println("Successfully created and written the file.");
} catch (IOException e) {
System.out.print("Exception");
}}}
```

Successfully created and written the file.

File   Edit   Format   View   Help

FAN
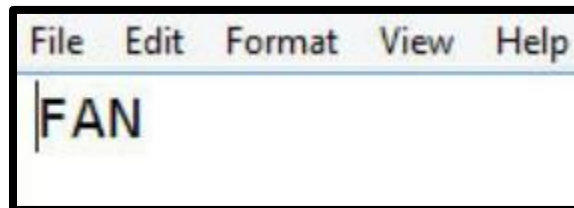
- The preceding program will successfully create a file **Output.txt** in the current directory and write the Unicode characters for numbers 70, 65, and 78 (FAN).

- The above figure shows the content of the **Output.txt** file

InputStreamReader

- The InputStreamReader, a sub-class of the Reader class is a bridge from byte streams to character streams.

- This reads bytes and decodes those bytes into characters using specific charsets.

- For example:

  - **InputStreamReader (InputStream in)** is the declaration of **InputStreamReader** with default charset.

  - **InputStreamReader (InputStream in, String charset)** is the declaration of **InputStreamReader** with given String type charset.

- Writer class is an abstract class that helps to write the streams of characters. There is no direct implementation of the **Writer** class, but we use its sub-classes to perform the write operation on file. The methods and must be implemented by subclasses of the **Writer** class

```java
import  java.io.BufferedReader;
import  java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
public class BufferReaderExample {
public static void main(String[] args) {
try {
FileInputStream file = new FileInputStream("ListOfCourses.txt");
BufferedReader reader = new BufferedReader(new InputStreamReader(file));
String line;
while((line = reader.readLine()) != null){
System.out.println(line);
} }
catch (IOException e) {
e.printStackTrace();
} } }
```

Output:

```
Java
Python
Android
IoT
```

BufferedReader

- The **BufferedReader** class is a subclass of the **Reader** class that reads text or buffering characters or character input stream and provides efficient reading of characters, lines or arrays.

- This can be used with **FileReader** or **InputStreamReader** or other readers classes to read the byte of inputs from file or stream and return serialized and deserialized data.

- Here is the syntax to declare **BufferedReader** by specifying size:
  BufferedReader bReader = new BufferedReader(readerObject)

- Syntax to declare **BufferedReader** by specifying size:
  BufferedReader bReader = new BufferedReader(readerObject, int size)

Example:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class FileAndBufferedReader
{
public static void main(String[] args) throws IOException
{
FileReader fReader = new FileReader("SampleFile.txt");
BufferedReader bReader = new BufferedReader(fReader);
int records;
while((records = bReader.read()) != -1) {
System.out.print((char)records);
}
bReader.close();
fReader.close();
} }
```

Output:

```
EduBridge is the one stop destination for all your career needs
Invest in YOUR future. Accelerate YOUR career.
GET TRAINED. GET HIRED. OUR GUARANTEE.
```

- These Java program shows the implementation of the **FileReader** class to open connection with a file **SampleFile.txt** stored in the file system.

- Here, the **BufferedReader** class helps to read the characters from a specified file object.

- After completion of the read operation, we close the objects and remove the references with the help of the close() method at the end of the program.

Serialization and Deserialization

- Serialization is the process of transforming an object of Java into a stream of bytes.

- When we serialize an object, it means we are actually converting its state to a stream of bytes, so that the stream of bytes can be later converted into a copy of the Java object

- We can make a Java object serializable by implementing the java.io.Serializable interface or its sub interface, Serialization can be implemented to persist the state of any object, and travel or pass that object over the network so that the other system or application can consume or use that stream of bytes.

- This also helps us to store the state of object into a file or database as permanent storage.

- Stream of bytes are platformindependent, which means we can serialize and deserialize on any machine or platform, and that can be read and understood by any system.

- Mainly, we use the approach of serialization when we have some data or file, which we want to transfer to other systems/applications or communicate with external applications.

- Serializable saves or persists only non-static member of the object; static members of an object will not get serialized.

- If we try to serialize an object that doesn't support serialization, then the system will throw **NotSerializableException** for that.

- Deserialization is the opposite of serialization, where we transform a stream of bytes into a Java object.

- **ObjectOutputStream** and **ObjectInputStream** are the IO classes that have the **writeObject** and **readObject** methods, which help us to write and read the state of an object for its class.

- The **writeObject()** method is responsible for writing the state of the object for its class and **readObject()** method is responsible for reading from the stream and restoring the fields of classes

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 35:

```java
import java.io.*;
class Link implements Serializable {
    private String commons;
    private int favorites;
    public Link(String commons, int favorites) {
        this.commons = commons;
        this.favorites = favorites;
    }
    public void printLink() {
        System.out.println("Link : " + this.commons);
    }
}
public class Serialization {
    public static void serializeLink(Link inputLink, String fileName) throws IOException {

        FileOutputStream file = new FileOutputStream(fileName);
        ObjectOutputStream out = new ObjectOutputStream(file);
        out.writeObject(inputLink);
        out.close();
        file.close();

    }
    public static Link deserializeLink(String fileName) throws IOException, ClassNotFoundException {

        FileInputStream file = new FileInputStream(fileName);
        ObjectInputStream on = new ObjectInputStream(file);
        return (Link) on.readObject();

    }
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Link randomLink = new Link("My first link", 5);
```

```
    final String filename = "example.bin";
    System.out.println("Prior to Serialization : ");
    randomLink.printLink();
    serializeLink(randomLink, filename);
    Link linkFromFile = deserializeLink(filename);
    System.out.println("Following Serialization : ");
    linkFromFile.printLink();
  }
}
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Multithreading

Threads in JAVA

- Thread is the smallest execution unit of a process and a process may have many threads that are executing at the same time.

- Thread has its own execution path within the process and shares the memory of the process with other threads, which are running in the same process.

- Thread doesn't allocate any memory, but it uses the memory allocated by its process; this helps faster and efficient communication between threads within the same process.

Process and Thread

- In parallel programming, there are two ways to achieve the concurrent execution of a same program or multiple tasks at the same time from an application or program:

  - Processor

  - Thread

- There is a huge difference between the processor and thread.

- A process always allocates separate memory in the system for the execution and has a self-contained execution environment and executes with the same allocated memory.

- The Java virtual machine runs as a single process and executes java program within the same process, but if needed, we can create additional processes using **start()** of the **ProcessBuilder** object.

- **For example:**
  Process p = new ProcessBuilder("customProcess","args").start();

Thread

- A thread never allocates its own memory; it always uses the memory allocated by the processor and executes within a process.

- Threads are very lightweight and consume fewer resources for execution.

- A simple Java program always starts with a main thread, and a thread can also create further threads to lessen the execution of the program

- A thread can communicate with other threads running in the same process using the thread methods such as wait(), notify(), and yield()

```
Process
    Thread 1
    Thread 2
    Thread 3
```
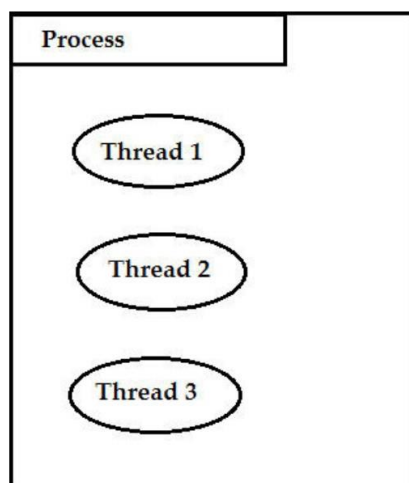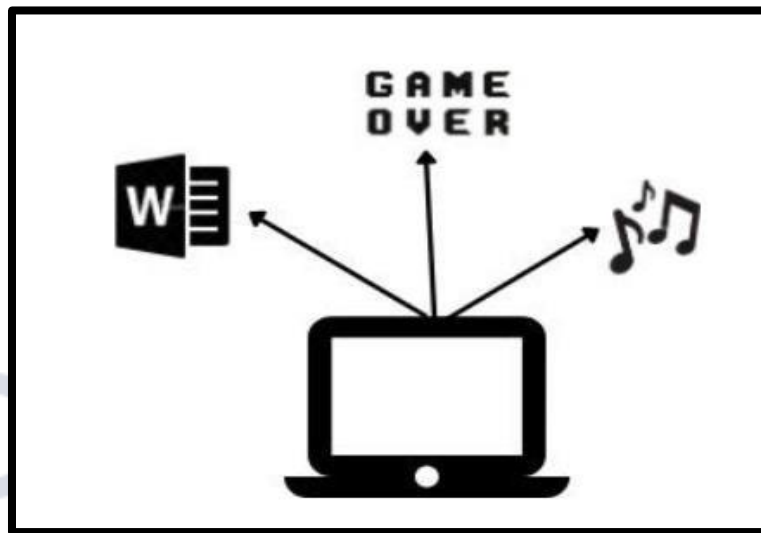
Multitasking

- Multitasking is the term use to refer when a machine or person is doing or running multiple works or tasks at the same time.
- **For example,**
  when we work on a computer, at the same time, there are multiple programs and applications running such as playing music, writing on a word document, browsing Internet, and many others. So, we can say that our computer is multitasking.
  There is no multitasking where multiple tasks are being executed on the CPU. But it is all about utilizing the CPU's ideal time and allowing other processes or threads to execute, when one program is waiting for input or other resources.

Introduction to Multithreading

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
- Each thread defines a separate path of execution.
- This means that a single program can perform two or more tasks simultaneously.
- **For example:**

- One thread is performing typing action on a file at the same time another thread is checking and resolving grammatical mistakes at the same time.

Thread Lifecycle
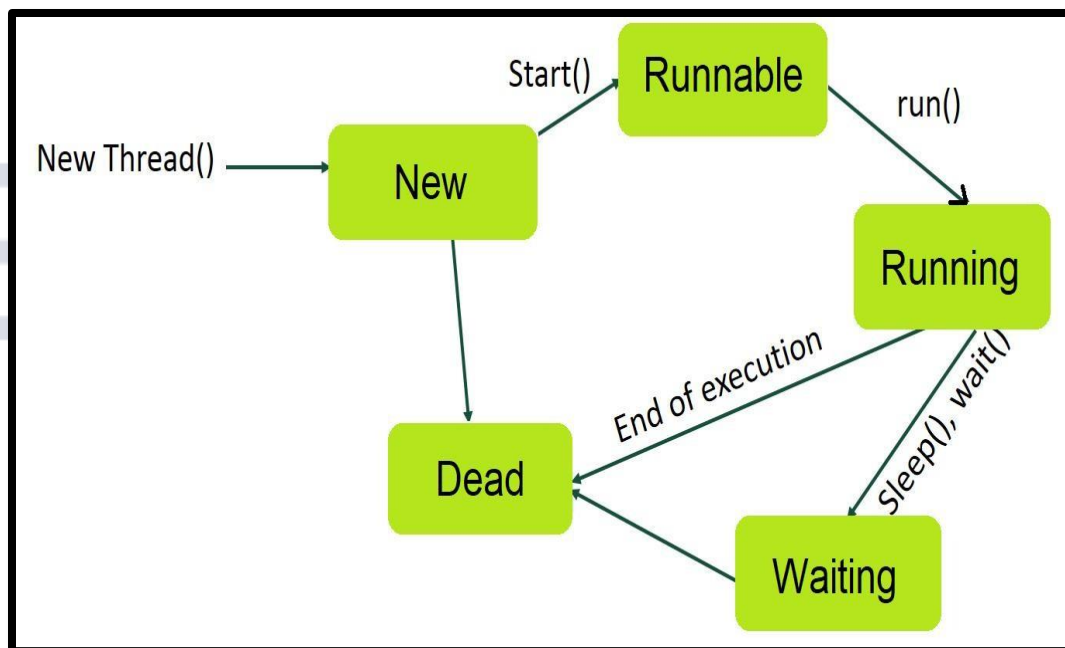
- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.
- **New :** A thread begins its life cycle in the new state. It remains in this state until the program starts the start() method.
- **Runnable :** After a newly born thread is started by invocating start() method on new thread, the thread becomes runnable.
- **Running :** A thread is in running state if the thread scheduler has selected it.
- **Waiting :** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
- **Terminated :** A thread enter the terminated state when it complete its task.

Thread Priorities

Every thread has a priority in Java programming; that thread priority helps the JRE to prioritize the execution of thread. A thread with high priority always gets executed first. A new thread has its priority initially set equal to the priority of the creating thread and is a daemon thread if and only if the creating thread is a daemon. There are 10 logical ranges of thread priority; the logical value is an integer number from 1 to 10 and can be set to any thread object using the **setPriority(int)** method. There are three static variables defined in the **Thread** class for priority. Default priority of a Java thread is

The maximum priority that a thread can have. The logical thread value is 10. The minimum priority that a thread can have. The logical thread value is between 1 to 4. The default priority that is assigned to a thread. The logical thread value is between 5 to 9. To identify and get the priority of an executing thread, we may check with the help of the **getPriority()** method of Thread

Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

- Every thread has a priority that determine the order in which each threads are scheduled for execution.
- In java thread priority ranges between 1 to 10

  - MIN PRIORITY (a constant of 1)

  - MAX PRIORITY (a constant of 10)

- By default every thread is given a

  - NORM PRIORITY(a constant of 5).

Creating a thread in Java

- Java defines two ways for creating a thread.
  - By implementing the **Runnable interface**.
  - By extending the **Thread class**.
- When we implement the runnable interface, we must create object of **Thread** class and pass the object of our thread class as a parameter of the **Thread** class.
- And to start the **Thread** class, we will call the start method on object of the **Thread** class, and not on the actual **Thread** class, which we created.
- A class should implement the Runnable interface and override its **run()** method. The run method is the entry point of a thread and starts execution from there when the **start()** method executed on that thread.

- Let create a class that implements the runnable interface as shown in syntax.

  - **Syntax** : public void run()

- Now as a second step, instantiate a thread object as shown in syntax.

- Thread(Runnable threadObj, String threadName);

- Where,

  - **threadObj** is an instance of a class that implements the Runnable interface.

  - **threadName** is the name given to the new thread.

- Once a Thread object is created, we can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method

  - void start()

Example:

```java
class RunnableThread implements Runnable
{
    public void run()
    {
        System.out.println("Implementing the Runnable Interface, thread started running..");
    }
}

public class Main
{
    public static void main(String[] args) {
        RunnableThread t1 = new RunnableThread();
        Thread t2 = new Thread(t1);
        t2.start();
    }
}
```

Output

```
Implementing the Runnable Interface, thread started running..



...Program finished with exit code 0
Press ENTER to exit console.
```

- Here is an example that creates a new thread using runnable interface
- To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.
- If we are implementing Runnable interface in class, then we need to explicitly create a Thread class object and need to pass the Runnable interface implemented class object as a argument in its constructor.

Extending the thread class :

- This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override **run()** method which is the entry point of new thread.

- This approach provides more flexibility in handling multiple threads.

- We will need to override **run( )** method available in Thread class.

     Syntax :   public void run( )

- Once Thread object is created, we can start it by calling **start()** method, which executes a call to run( ) method.

- Following is a simple syntax

     Syntax :   void start( )

Example:

```java
class ExtendingThread extends Thread
{
    public void run()
    {
        System.out.println("Extending thread class, thread started running..");
    }
}

public class Main
{
    public static void main(String[] args) {
        ExtendingThread t1 = new ExtendingThread();
        t1.start();
    }
}
```

Output

```
Extending thread class, thread started running..


...Program finished with exit code 0
Press ENTER to exit console.
```

- Here is an example that creates a new thread by extending the **Thread** class.
- In this case also, we must override the **run()** and then use the **start()** method to run the thread.
- Also, when we create ExtendingThread class object, Thread class constructor will also be invoked, as it is the super class, hence ExtendingThread class object acts as Thread class object

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 36:

package Hello.World;

```java
public class Hello_World{


        public static void main(String[] args) {

                MultiThraed thread1 = new MultiThraed("Thread1");
                thread1.start();
                MultiThraed thread2 = new MultiThraed("Thread2");
                thread2.start();


        }
}

class MultiThraed implements Runnable {
        Thread Cooking;
        private String IoT;
        MultiThraed(String name) {
        IoT = name;
        }
        @Override
        public void run() {
         System.out.println("Thread running" + IoT);
         for (int i = 0; i < 4; i++) {
         System.out.println(i);
         System.out.println(IoT);
          try {
           Thread.sleep(1000);
          }
          catch (InterruptedException e) {
```

```
        System.out.println("Thread is interrupted");
       }
      }
     }
     public void start() {
      System.out.println("Thread started");
      if (Cooking == null) {
       Cooking = new Thread(this, IoT);
       Cooking.start();
      }


     }
     }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

Thread Synchronization

- Synchronization is a keyword in the Java programming language that facilitates the programmer to control threads that are sharing data.

- This is the best approach in Java technology to provide a mechanism to treat the data carefully.

- Synchronized keywords are used to declare a method or create a method or block of statement thread safe, that means only one thread can access that block of code at a time and other threads will be in a waiting state until the first thread completes the execution.

- This is done when we want to get the updated data after evaluation or while modifying the object so that other threads get only the updated value after the successful modification of the object.

- If we do not lock the code block, then other threads may perform the operation on the same object, which causes ambiguity and corrupts the actual data

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 37:

```java
package coreJava; class
SynchronizedEx { synchronized
void show(String p){ try {
System.out.println("*****");
System.out.print(p);
Thread.sleep(1000);
System.out.println("#####");
}
catch(InterruptedException e){
e.printStackTrace();
} } }
class MyThread implements Runnable {
Thread c; String
m;
SynchronizedEx r;
MyThread(SynchronizedEx w, String k) {
r=w;m=k;
c=new Thread(this);
}
public void run(){
r.show(m);
} }
public class test {
public static void main(String s[])throws
InterruptedException {
SynchronizedEx d=new SynchronizedEx();
MyThread t1=new MyThread(d,"Hello");
MyThread t2=new MyThread(d,"JAVA");
t1.c.start();
t2.c.start();
```

```
t1.c.join();
t2.c.join();
} }
```

Post completion of the activity trainer will share the feedback/ suggestions on the work done by the participants.

DeadLock

- Deadlock in Java is a part of multithreading.

- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.

- Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
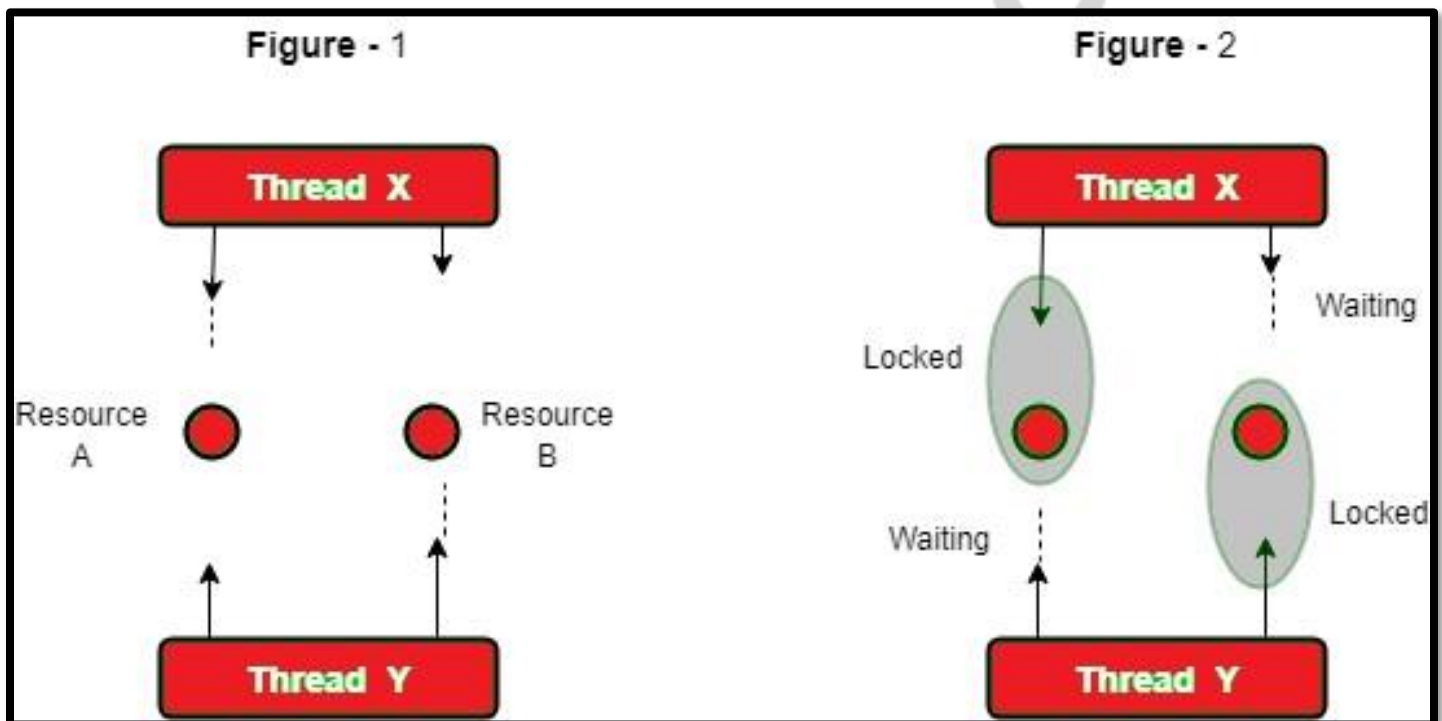
The above figure shows the scenario of Deadlock in Java:

- Deadlock in Java is a condition when two or more threads try to access the same resources at the same time.

- Then these threads can never access the resource and eventually go into the waiting state forever.

- So, the deadlock condition arises when there are more than two threads and two or more than two resources.

- Basically, a deadlock occurs when multiple threads request for the same resource but they are received in a different order.

- Eventually, they get stuck for an infinite period of time and cause a deadlock.

Avoid DeadLock

- Though it is not possible to completely get rid of the deadlock problem in java still we can take precautions to avoid such deadlock conditions. These preventive measures are as follows:

    1. By avoiding Nested Locks

    2. By avoiding unnecessary locks

    3. By using Thread join

1. **Avoid Nested Locks**
   - Using nested locks can be the main cause of the occurrence of deadlocks in Java, We can avoid the use of nested locks to prevent the deadlocks in Java.
   - Nested locks mean we try to provide access to resources to multiple threads. If we have already assigned one lock to a thread then we should avoid giving it to the another thread

2. **Avoid Unnecessary Locks**
   - We should also avoid giving locks to members or threads which do not need it. We should only provide the lock to the important threads and avoid using unnecessary locks.
   - If we provide an unnecessary lock to a thread that does not really need it, then it may cause a condition of deadlock.

3. **Use Thread Joins**
   - The condition of deadlock usually occurs when one thread is waiting for another thread to complete its execution and occupy that resource.
   - In this situation, we can use the Thread.join() method and give it a maximum time which a thread approximately takes to finish the execution. This can help us from removing the risk of deadlock in Java.

Memory Management

Memory Management

- Heap is the memory that gets created within the RAM for random access, where classes, objects and members created.

- Garbage collection is the backbone of JVM for dealing with memory and allocating and reclaiming the memory from heap.

- Heap and sack are the random memory types and heap is major memory, which is being use by Java programming.

- Heap memory is further classified into the three logical parts called generations - new, old, and permanent are the three generations of the heap memory.

- All three generations of memory are managed and handled by minor, major, and full garbage collection that helps to reclaim the unused memory from system by deleting the unreferenced objects from heap.

Garbage collection

- **Garbage collection** is the process of memory management that finds and removes or cleans the unused heap memory allocated to the objects and keeps the memory reusable and available for new resources.

- Garbage collection executes automatically and removes the allocated memory for only those objects and members that do not hold any references and no more in use, or in general terms, we can say reclaims the unused heap memory

- When we execute a Java program in a machine, it requires the memory

- Java virtual memory provides various garbage collectors that help the system or operating system to improve the performance and efficiency of the memory

- There are two methods **Runtime.gc()** and **System.gc()** that generate and send the request to JVM for garbage collection.

- We cannot force the garbage collectors to reclaim the unused memory, JVM automatically manages all these to run the application or program and a developer need not worry about the memory management in Java.

- Garbage collection can be forcefully executed with the help of any of the following methods:

- **System.gc();**
- **Runtime.getRuntime().gc();**
- **System.runFinalization();**
- **Runtime.getRuntime().runFinalization();**

- **NOTE:** Calling the above methods does not guarantee that Garbage collection will start performing and reclaiming the memory immediately. When all the heap memory is full, it will throw a java.lang.OutOfMemoryError error.

Generations

- Java Heap memory (Hotspot heap structure) is called Generations are the logically divided spaces of heap memory to store the resources and objects based on their life or tenure.

- There are three generations in the heap memory –

    - Young generation (which is also called new generation)

    - Old generation

    - Permanent generation
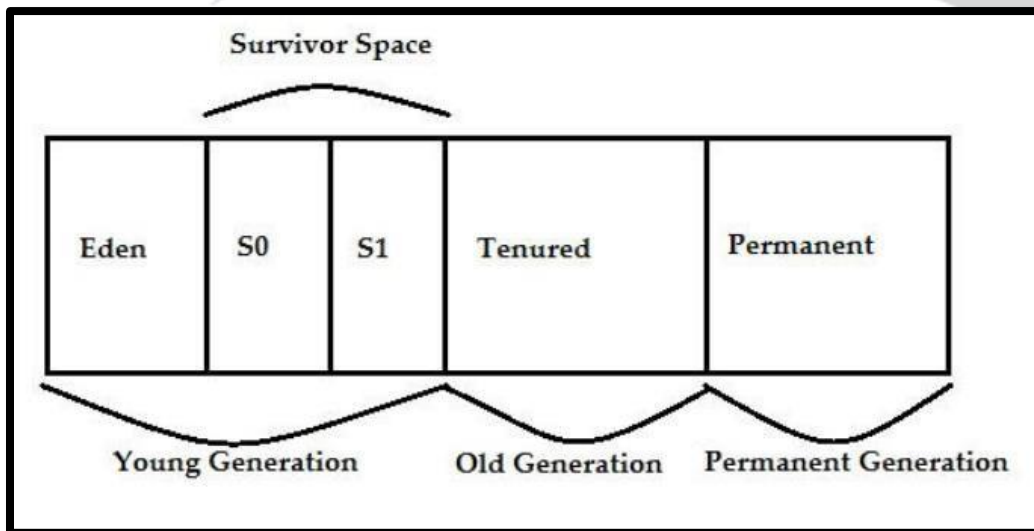
- The above figure shows the division of heap memory and how an object lives in the memory based on the use of object. Here generations – young generation, old generation, and permanent generation are three stacks in which an object lives according to the age of the object.

- First, we have young generation that is split into three parts, an Eden, and two survivor space S0 and S1 that keep moving the object from one part to another by executing the Minor garbage collector.

- Second, we have old generation, also known as tenured generation, that stores the objects, which does not get cleaned by Minor garbage collection in young generation, and if a program or application still using that object, then that object will be moved to old generation of the heap memory. There is a threshold defined on the age of an object in young generation. Once that threshold age met, the object automatically moves from young to old generation. Major garbage collection runs on the old generation to clean the object and reclaim the memory.

- The third one is permanent generation that holds the metadata of classes, objects, and members executed and loaded in the memory of running the program or application. It also holds Java library classes and methods. Full garbage collection runs on the permanent generation and clears the classes from memory when those classes and methods are no more required.

Types of Garbage collection

- There are three garbage collectors that get executed on respective generations of heap memory.

1. **Minor GC:** This runs on the new generation and is responsible for cleaning the dead objects and reclaiming the memory and keeps the memory available for young/new generation .

2. **Major GC:** This runs on old generation of heap memory and responsible for cleaning the old/tenured heap space.

3. **Full GC:** This is cleaning the entire heap memory, including young and tenured spaces.

- These are three garbage collectors that run on their respective generations of heap memory and keeps our application active and running without any memory leakage issue or out of memory issue.

Finalize

- Finalize is the method of garbage collection and is called by garbage collectors on objects to identify and check the references for that before reclaiming the memory.

- The **finalize()** method identifies the objects that do not have any references and pushes them in the garbage collection queue to reclaim the memory.

- The **finalize()** method will not send any live or active object for garbage collection.

- We can explicitly call the **finalize()** method on objects, when we know that we are no longer going to use this object in the program and want to remove from memory.

- But again, it all depends on the GC, whether the memory will be reclaimed or not.

- We cannot guarantee that after calling the finalize method, memory of that object will be free and reclaimed.

- Calling the finalize() method explicitly may impact the performance of your application because of high CPU utilization.

Heap

- The heap size in any machine depends on the maximum address space per process.

- We can control the size of heap memory for any application or program while starting the program.

- Here are the parameters that we use to set the heap size and instruct the JVM to allocate the specified heap memory to an application

    - Xms define minimum heap size of memory
    - Xms define maximum heap size of memory

Build Tool

What is a Build Tool?

- A build tool is a tool that automates everything related to building the software project. Building a software project typically includes one or more of these activities:

- Generating source code (if auto-generated code is used in the project).

- Generating documentation from the source code.

- Compiling source code.

- Packaging compiled code into JAR files or ZIP files.

- Installing the packaged code on a server, in a repository or somewhere else.

Any given software project may have more activities than these needed to build the finished software. Such activities can normally be plugged into a build tool, so these activities can be automated too.

The advantage of automating the build process is that you minimize the risk of humans making errors while building the software manually. Additionally, an automated build tool is typically faster than a human performing the same steps manually.

Installing Maven

- To install Maven on your own system (computer), go to the http://maven.apache.org/download.cgi and follow the instructions there:

1. Set the JAVA_HOME environment variable to point to a valid Java SDK (e.g. Java 8).

2. Download and unzip Maven.

3. Set the M2_HOME environment variable to point to the directory you unzipped Maven to.

4. Set the M2 environment variable to point to M2_HOME/bin (%M2_HOME%\bin on Windows, $M2_HOME/bin on unix).

5. Add M2 to the PATH environment variable (%M2% on Windows, $M2 on unix).

6. Open a command prompt and type 'mvn -version' (without quotes) and press enter.

Build Life Cycle:

Basic maven phases are used as below.

- **clean**: deletes all artifacts and targets which are created already.

- **compile**: used to compile the source code of the project.

- **package**: package is used to convert your project into a jar or war etc.

- **install**: install the package into the local repository for use of another project.

POM(Project Object Model):

- In Maven, POM (Project Object Model) is the fundamental unit of work. It is an XML file which holds the information about the project and configuration details used to build a project by Maven.

- This includes the directories where the source code, test source etc. is located in, what external dependencies (JAR files) your projects has etc.

- The POM file describes what to build, but most often not how to build it. How to build it is up to the Maven build phases and goals. You can insert custom actions (goals) into the Maven build phase if you need to, though.

- Each project has a POM file. The POM file is named pom.xml and should be located in the root directory of your project. A project divided into subprojects will typically have one POM file for the parent project, and one POM file for each subproject.

- Maven reads the pom.xml file, then executes the goal.

- Maven allows to comprehend the complete state of a development effort in the shortest period of time.

- To attain this goal, Maven deals with:

  1. Making the build process easy
  2. Providing a uniform build system
  3. Providing quality project information

- There should be a single POM file for each project.

- All POM files require the project element and three mandatory fields: groupId, artifactId,version.

- Projects notation in repository is groupId:artifactId:version.

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.hello_world</groupId>
<artifactId>example</artifactId>
<name>my pom example</name>
<version>0.7-SNAPSHOT</version>
<packaging>jar</packaging>
<dependencies/>
<build/>
[...]
```

- Root element of POM.xml is project and it has following sub-nodes :

- **project :-** It is the root element of project

- **modelVersion :-** It is the sub element of project. It specifies the modelVersion. It should be set to 4.0.0.

- **groupId :-** It is the sub element of project. It specifies the id for the project group.

- **artifactId :-** It is the sub element of project. It specifies the id for the artifact (project). An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, and WARs.

- **version:-**It is the sub element of project. It specifies the version of the artifact under given group.

Note:-When no packaging is declared, Maven assumes the artifact is the default: jar.

Gradle

- Gradle is a build tool designed specifically to meet the requirements of building Java applications.

- Once it's set up, building your application is as simple as running a single command on the command line or in your IDE.

    - gradlew build

- It includes, compiling, testing, and packaging your application, all with one command.

- For example, once you've compiled your application once, when you try to compile again after not having changed anything, Gradle knows it doesn't need to recompile. This feature, called **incremental build**

- To install Gradle on your own system (computer), go to the https://gradle.org/install/ and follow the instructions there.

gradle init

- Here's what Gradle created when we ran gradle init.
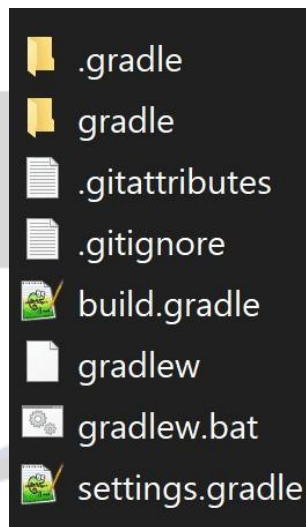
- **settings.gradle** sets up some high-level configuration for the project

- **build.gradle** is the build script configuration file describing your application to Gradle so it can build it

- **gradlew and gradlew.bat** are known as the Gradle wrapper scripts, for Linux/Mac and Windows respectively. These let you build an application without having to download and install Gradle like we did earlier. When the wrapper is executed, it will automatically download Gradle and cache it locally. Normally you always build your application with the wrapper, as it ensures it gets built with the correct version of Gradle.

- **.gitignore** configures Git so that the .gradle and build directories aren't committed into version control. Everything else gets committed though.

- The .gradle directory is a local cache managed by Gradle, and the build directory is where Gradle creates any build outputs such as compiled Java code or jar files.

Gradle

- **Project:** We already know that the **project** is the highest-level Gradle concept. It's a container for everything that Gradle knows about our application.

- **Build script:** Each Gradle project can have a **build script**.

- **Task:** Gradle **tasks** are individual build actions you can run from the command line.

- **Plugin:** The last big concept to wrap your head around is the Gradle **plugin**. When you apply a plugin in your build script, it automatically adds tasks to your project which you can run to achieve some particular outcome

Gradle wrapper

- The **Gradle wrapper** is a script you add to your Gradle project and use to execute your build.

- In an empty directory run gradle init to start the Gradle project setup wizard

```
$ gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4]
```

- Whatever options we choose, the wrapper will get automatically created.

- If we inspect the directory, there are some new files.

```
.
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
└── gradlew.bat
```

This includes:

- **gradle-wrapper.jar** code required for downloading the correct Gradle version when you run the build

- **gradle-wrapper.properties** file to configure the wrapper's properties such as the Gradle version

- **gradlew** a shell script for executing the build on Linux

- **gradlew.bat** a script for executing the build on Windows

- The four Gradle wrapper files described above are automatically created.

Gradle Vs Maven

Gradle	Maven
1. It is an automation build system that is developed in Groovy	1. It is a build and project management tool.
2. Gradle is not driven by any xml file	2. Maven is driven by an xml file
3. Gradle works incrementally and gives a quicker build completion.	3. Maven does not take an incremental approach and is slower in terms of build timings than Gradle.
4. Gradle script is simple, not lengthy and can be understood easily.	4. Maven has the xml file which is descriptive, lengthy

5. Gradle can be customized easily	5. Maven's customization is not easy
6. In Gradle, Java compilation is not a required step.	6. In Maven, compilation is a required step.
7. Gradle is a comparatively modern tool and its users are limited in numbers.	7. Maven is a familiar tool and popular among the Java developers.
8. Numerous dependencies for the project can be added in Gradle without the use of xml.	8. Numerous dependencies can be added to the project by adding them to the xml file (pom), thereby making it more complex and difficult to manage than Gradle.

- The key differences between Gradle and Maven are tabularized below
- Gradle has better performance because of the features listed below

JUNIT

Software testing

- Software testing is meant to avoid software **failure**.

- A **failure** is caused by a **fault** in the code base.

- A **symptom** is an **observable behavior** of the system that enables us to observe a **failure** and possibly find its corresponding **fault**.

- The process of discovering what caused a failure is called **fault identification**.

- The process of ensuring that the failure does not happen again is called **fault correction**, or **fault removal.**

- **Fault identification** and **fault correction** is popularly called **debugging**.

- Software testing, in practice, is about identifying a certain possible system failure and design a test case that **proves that this particular failure is not experienced by the software**.

- "testing can reveal only the presence of faults, never their absence."

- There are many driving sources for software testing:

    - Requirements-driven testing, Structure-driven testing, Statistics-driven testing, Risk-driven testing.

- There are many levels and kinds of software testing:

    - Unit Testing, Integration Testing, Function Testing, Acceptance Testing, Installation Testing.

- At the day-to-day programming level, **unit testing** can easily be integrated in the programming effort by using a **Unit Testing Framework.**

- However, unit testing cannot be applied for higher-level testing purposes such as function testing or acceptance testing, which are system-level testing activities.

- A unit test is a **piece of code** written by a developer that **exercises a very small, specific area of functionality** applied to one of the units of the code being tested. Usually a unit test exercises some particular method in a particular context.

JUNIT

- In Java, the standard unit testing framework is known as **JUnit**.

- Test Cases and Test Results are Java objects.

- JUnit was created by Erich Gamma and Kent Beck, two authors best known for Design Patterns and eXtreme Programming, respectively.

- Using JUnit you can easily and incrementally build a **test suite** that will help you **measure your progress**, **spot unintended side effects**, and focus your development efforts.
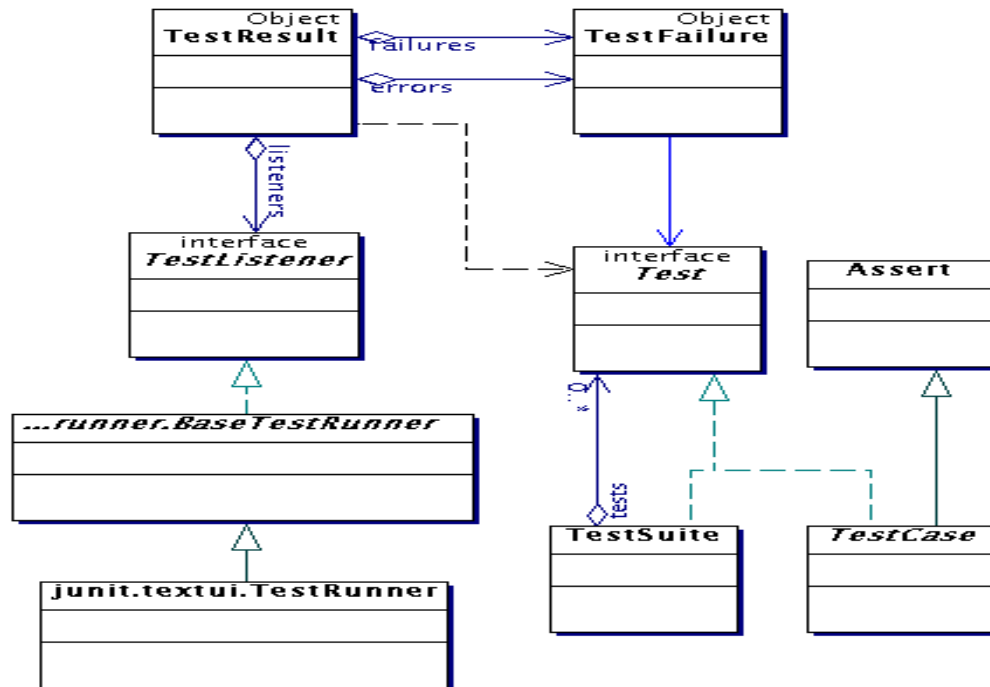
Architecture Overview

- The JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests

- **TestRunner** runs tests and reports **TestResult**s

- You test your class by extending abstract class **TestCase**

- To write test cases, you need to know and understand the **Assert** class

Key JUnit Notions

- **Tested Class** – the class that is being tested.

- **Tested Method** – the method that is tested.

- **Test Case** – the testing of a class's method against some specified conditions.

- **Test Case Class** – a class performing the test cases.

- **Test Case Method** – a Test Case Class's method implementing a test case.

- **Test Suite** – a collection of test cases that can be tested in a single batch.

Writing a TestCase

- To start using JUnit, create a subclass of TestCase, to which you add test methods

- Here's a skeletal test class:

        import junit.framework.TestCase;

        public class TestBowl extends TestCase {

          } //Test my class Bowl

- Name of class is important – should be of the form **Test**MyClass or MyClass**Test**

- This naming convention lets TestRunner automatically find your test classes

Assert methods

- Each assert method has parameters like these:
  message, expected-value, actual-value

- Assert methods dealing with floating point numbers get an additional argument, a tolerance

- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:

    - messages helps documents the tests

    - messages provide additional information when reading failure logs

- List of different types of assertion statements that you can use to test your code

    - assertTrue(String message, Boolean test)

    - assertFalse(String message, Boolean test)

    - assertNull(String message, Object object)

- assertNotNull(String message, Object object)

- assertEquals(String message, Object expected, Object actual) (uses equals method)

- assertSame(String message, Object expected, Object actual) (uses == operator)

- assertNotSame(String message, Object expected, Object actual)

assertTrue(test)	**fails if the boolean test is false**
assertFalse(test)	**fails if the boolean test is true**
assertEquals(expected, actual)	**fails if the values are not equal**
assertSame(expected, actual)	**fails if the values are not the same (by ==)**
assertNotSame(expected, actual)	**fails if the values are the same (by ==)**
assertNull(value)	**fails if the given value is not null**
assertNotNull(value)	**fails if the given value is null**
fail()	**causes current test to immediately fail**

Example:

```java
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    protected void tearDown() { } // no resources to release
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
    }

    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}
```

- Suppose you want to test a class Counter

- public class CounterTest extends junit.framework.TestCase {

    - This is the unit test for the Counter class

- public CounterTest() { } //Default constructor

- protected void setUp()

    - Test fixture creates and initializes instance variables, etc.

- protected void tearDown()

    - Releases any system resources used by the test fixture

- public void testIncrement(), public void testDecrement()

    - These methods contain tests for the Counter methods increment(), decrement(), etc.

    - Note capitalization conventio

- Note that each test begins with a brand new counter

- This means you don't have to worry about the order in which the tests are run

JUnit and Eclipse

- Eclipse comes with both JUnit and a plug-in for creating and working with JUnit tests.

- Eclipse allows you to quickly create test case classes and test suite classes to write your test code in.

- With Eclipse, Test Driven Development (TDD), becomes very easy to organize and implement.

- Eclipse facilitates the testing by generating automatically stubs for testing class methods.

- To add JUnit to an Eclipse project, click:

  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4/5... → Finish**

- To create a test case:

  - right-click a file and
    choose **New → Test Case**

  - or click **File → New → JUnit Test Case**

  - Eclipse can create stubs of method tests for you.
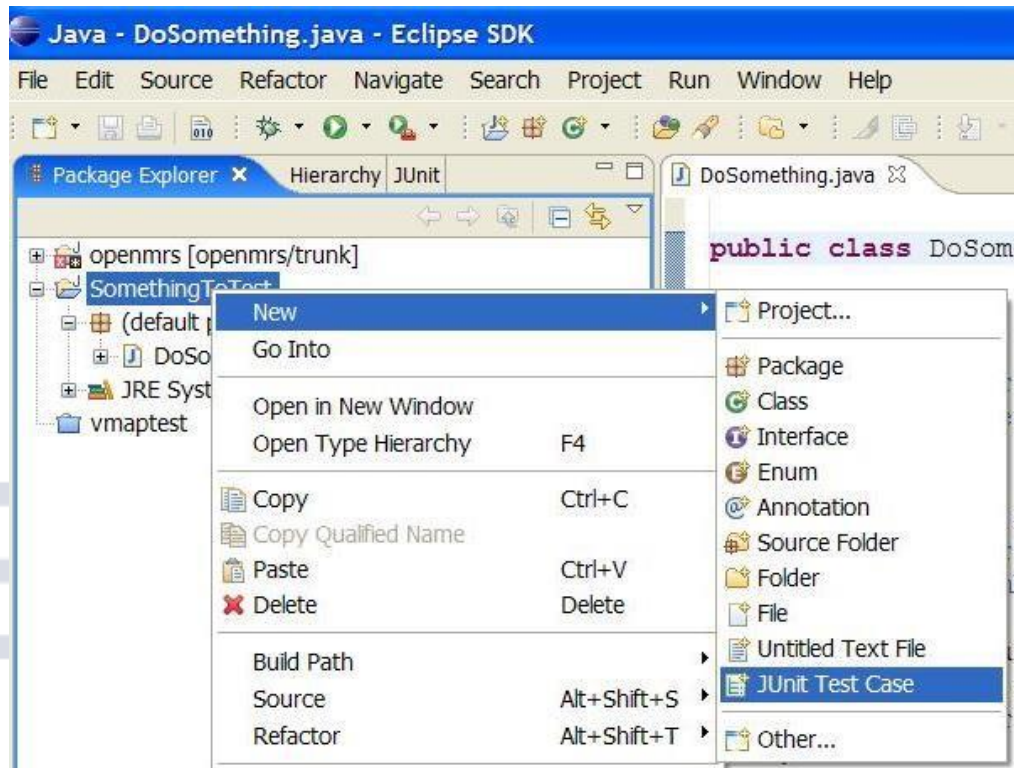
Junit Lifecyle

JUnit provides the following annotations that you can add to methods in your test class to do this:

- **@BeforeAll:** A static method in your test class that is called before all of its tests run.

- **@AfterAll:** A static method in your test class that is called after all of its tests run.

- **@BeforeEach:** A method that is called before each individual test runs.

- **@AfterEach:** A method that is called after each individual test runs.

Lab Activity:

Trainer will ask the participants to refer to the participant's guide and complete the given exercise.

Exercise 38:

```java
import org.junit.jupiter.api.*;

public class LifecycleDemoTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Connect to the database");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Load the schema");
    }

    @AfterEach
    void afterEach() {
        System.out.println("Drop the schema");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("Disconnect from the database");
    }

    @Test
    void testOne() {
        System.out.println("Test One");
    }

    @Test
    void testTwo() {
        System.out.println("Test Two");
    }
```

Test Driver Development (TDD):

Test-Driven Development, also called Test-First Development, is a technique in which you write unit tests before writing the application functionality.

- Tests are non-production code written in the same language as the application.

- Tests return a simple pass or fail, giving the developer immediate feedback
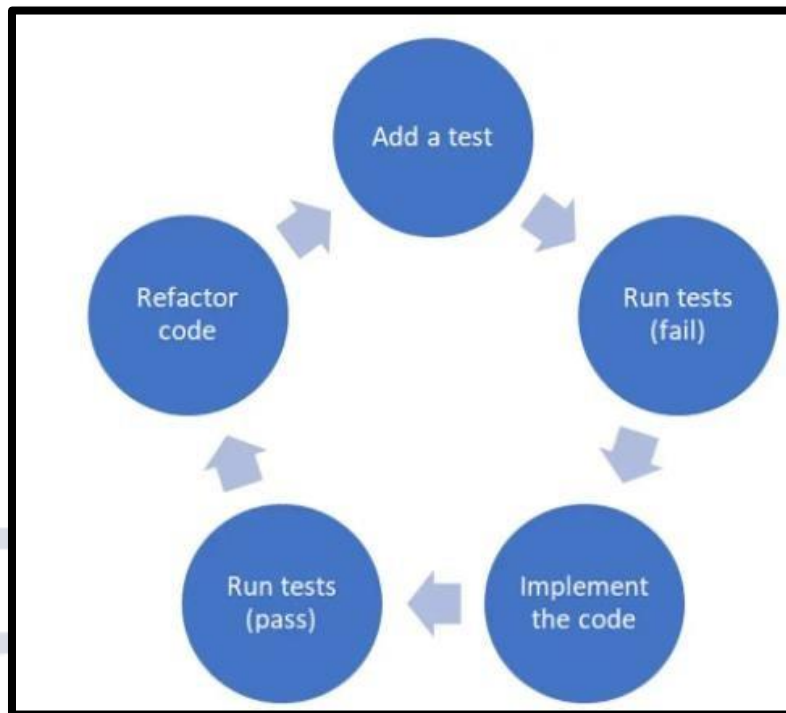
Why TDD?

A significant advantage of TDD is that it enables you to take small steps when writing software.

- TDD is done at Unit level - i.e. testing the internals of a class

- Tests are written for every function

- Mostly written by developers using one of the tool specific to the application

TDD Lifecycle

Test-driven development (TDD) is a software development process that interweaves coding, testing, and design. It is a test-first approach that aims to improve the quality of your applications. Test-driven development is defined by the following lifecycle:

1. Add a test.
2. Run all of your tests and observe the new test failing.
3. Implement the code.
4. Run all of your tests and observe the new test succeeding.
5. Refactor the code.

What is Mockito?

- Mockito is a Java framework allowing the creation of mock objects in automated unit tests

- A mock object is a dummy implementation for an interface or a class in which you define the output of certain method calls.

- If you use Mockito in tests you typically:

  Mock away external dependencies and insert the mocks into the code under test

  Execute the code under test

  Validate that the code executed correctly

How to use Mockito

- Integrate Mockito in your project with Maven:

```xml
<dependencies>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-all</artifactId>
        <version>1.10.19</version>
        <scope>test</scope>
    </dependency>
    ...
</dependencies>
```

- Without Maven: add Mockito JARs on your classpath

Mockito Annotations

- **@Mock** is used for mock creation. It makes the test class more readable.

- The @Mock annotation is always used with @RunWith, a class-level annotation. We will see in detail how both annotations are used to create and use a mock object in the next segment.

```java
@Mock
ToDoService serviceMock;
```

```
@RunWith(MockitoJUnitRunner.class)
public class DemoMock {
.....
}
```

- **@InjectMocks** is used to instantiate the tested object automatically and inject all the @Mock or @Spy annotated field dependencies into it (if applicable).

- InjectMocks annotation is used to mock a class with all its dependencies. This is quite useful to test the behavior completely.

```
@Mock
Map<String, String> Countries;
@InjectMocks
MyDictionary dic = new Continent();
@Test
public void UseInjectMocksAnnotation() {
    Mockito.when(Countries.get("India")).thenReturn("asia");
    assertEquals("asia", dic.getContinent("India"));
}
```

- **@Spy** is used to create a spy instance. We can use it instead spy(Object) method.

- Spy annotation is used to create a real object and spy on that real object. This would help to call all the object methods while still tracking every interaction that is being mocked.

```
@Spy
List<String> myList = new ArrayList<String>();
@Test
public void usingSpyAnnotation() { myList.add("Hello, This
    is LambdaTest"); Mockito.verify(spyList).add("Hello, This
    is LambdaTest"); assertEquals(1, spyList.size());
}
```

- **@Captor** is used to create an argument capto

- Captor annotation is used to create an ArgumentCaptor instance to capture method argument values for further assertions.

```
@Mock
HashMap<String, Integer> MyMap;
@Captor
ArgumentCaptor<String> keyCaptor;
@Captor
ArgumentCaptor<Integer> valueCaptor;
@Test
public void ArgumentCaptorTest()
{
    hashMap.put("A", 10);
    Mockito.verify(MyMap).put(keyCaptor.capture(), valueCaptor.capture());
    assertEquals("A", keyCaptor.getValue());
    assertEquals(new Integer(10), valueCaptor.getValue());
}
```

Functional Constructs

What is a Lambda Expression ?

- The new feature of Java 8 is Lambda Expression. It's a function.

- To put it in other words, the function has no name, return type, or access modifiers (private, public, or protected).

- Anonymous functions are also known as lambda expressions.

- Lambda expressions are available in other programming languages such as Python, Ruby, C#, and others

- The Lambda expression is used to provide a functional interface implementation.

- A function that can be written independently of any class

- It allows you to iterate over a collection, filter it, and retrieve data

- **Syntax:**

  (parameter list) -> { expression body }

- **Examples,**

  1. (int x, int y) -> { return x + y; }

  2. x -> x * x

  3. ( ) -> x

- The first part is an argument list, which might have zero, one, or several arguments and is wrapped by parenthesis.

- The second part is the Arrow-Token (->), which connects the list of arguments to the expression body

- The method body, which contains expressions and statements for lambda expression, is the third part

list of integers with a lambda

- List<Integer> is a parameterized type, parameterized by the type argument <Integer>

- the Arrays.asList method returns a fixed-size list backed by an array; it can take "vararg" arguments

- forEach is a method that takes as input a function and calls the function for each value on the list

- Note the absence of type declarations in the lambda; the Java 8 compiler does type inference

- Java 8 is still statically typed

- Braces are not needed for single-line lambdas (but could be used if desired).

    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> System.out.println(x));

- x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer

Multiline lambda

- Braces are needed to enclose a multiline body in a lambda expression.

    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> { x

      += 2;

      System.out.println(x);

    });

lambda with local variable

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> {

      int y = x * 2;

```
    System.out.println(y);

  });
```

lambda with declared parameter type

- You can, if you wish, specify the parameter type
- The compiler knows the type of intSeq is a list of Integers
- Since the compiler can do type inference, you don't need to specify the type of x.

```
    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach((Integer x -> {

      x += 2;

      System.out.println(x);

    });
```

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function

- It then calls the generated function

- For example,

  x -> System.out.println(x) could be converted into a generated static function

```
    public static void genName(Integer x) {

      System.out.println(x);

    }
```

Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces.

- A functional interface is a Java interface with exactly one non-default method. E.g.,

    public interface Consumer<T> {

       void accept(T t);

    }

- The package java.util.function defines many new useful functional interfaces.

- Examples of existing JDK functional interfaces: Runnable, Comparable<T>, Callable<V>.

Example

- Here is an interface called Consumer with a single method called accept.
- The forEach method iterates through the items in the object Consumer and performs the action accept on each item.
- The lambda expression becomes the body of the function in the interface.
- The signature of the function is defined by the interface.

```
public interface Consumer<T> {

  void accept(T t);

}

void forEach(Consumer<Integer> action {

  for (Integer i:items) {

    action.accept(t);

  }

}

List<Integer> intSeq = Arrrays.asList(1,2,3);

Consumer<Integer> cnsmr = x -> System.out.println(x);

intSeq.forEach(cnsmr);
```

- Here is an interface called Consumer with a single method called accept.
- The forEach method iterates through the items in the object Consumer and performs the action accept on each item.
- The lambda expression becomes the body of the function in the interface.
- The signature of the function is defined by the interface.

Properties of the Generated Method

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface

- The type is the same as that of the functional interface to which the lambda expression is assigned

- The lambda expression becomes the body of the method in the interface

- Any interface with only one nondefault method is considered a functional interface by Java 8.

- So functional interfaces are Java 8's secret sauce for backward compatibility.

Local Variable Capture

- Lambdas can interact with variables defined outside the body of the lambda
- Using these variables is called variable capture
- In example below lambda "captures" the variable var.

```java
public class LVCExample {

  public static void main(String[] args) {

    List<Integer> intSeq = Arrays.asList(1,2,3);

    int var = 10;

    intSeq.forEach(x -> System.out.println(x + var));

  }

}
```

- Note: local variables used inside the body of a lambda must be final or effectively final

Static Variable Capture

- A Java lambda expression can also capture static variables.
- This is not surprising, as static variables are reachable from everywhere in a Java application, provided the static variable is accessible (packaged scoped or public).
- Here is an example class that creates a lambda which references a static variable from inside the lambda body

```java
public class SVCExample {

  private static int var = 10;

  public static void main(String[] args) {

    List<Integer> intSeq = Arrays.asList(1,2,3);

    intSeq.forEach(x -> System.out.println(x + var));

  }

}
```

Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString

arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Conciseness with Method References

- In the case where all your lambda expression does is to call another method with the parameters passed to the lambda, the Java lambda implementation provides a shorter way to express the method call.

  - We can rewrite the statement

    intSeq.forEach(x -> System.out.println(x));

  - more concisely using a method reference

    intSeq.forEach(System.out::println);

- Notice the double colons :: . These signal to the Java compiler that this is a method reference. The method referenced is what comes after the double colons. Whatever class or object that owns the referenced method comes before the double colons.

Default Methods

- Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve backward compatibility with existing published interfaces

- We need default methods because, for example, the Collections interface has no stream() method and we'd like to use existing Collections interface to stream.

Stream API

- One of the major new features in Java 8 is the introduction of the stream functionality – java.util.stream – which contains classes for processing sequences of elements.
- The central API class is the Stream<T>. The following section will demonstrate how streams can be created using the existing data-provider sources.

- The new java.util.stream package provides utilities to support functional-style operations on streams of values.

- A common way to obtain a stream is from a collection:

      Stream<T> stream = collection.stream();

- Streams can be sequential or parallel.

- Streams are useful for selecting values and performing actions on the results.

Stream Operations

- There are many useful operations that can be performed on a stream.
- They are divided into **intermediate operations** (return Stream<T>) and **terminal operations** (return a result of definite type). Intermediate operations allow chaining.
- It's also worth noting that operations on streams don't change the source.

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.

- A terminal operation must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.

Example Intermediate Operations

- Intermediary operation: A method that takes a Stream and returns a Stream.

- They are lazily loaded and will only be executed if you include a terminal operation at the end.

    1. The peek() method

    2. The map() method

    3. The filter() method

- The filter() method allows us to pick a stream of elements that satisfy a predicate.

- To convert elements of a Stream by applying a special function to them and to collect these new elements into a Stream, we can use the map() method

Example Terminal Operations

- Terminal operation: A method that takes a Stream<T> and returns void.

- They are eagerly loaded and will cause the entire pipeline to be executed.

- A terminal operation will "spend" the stream.

  - The forEach() method

  - The count() method

  - The max() method

  - The collect() method

  - The reduce() method

Stream Pipeline

To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: the **source**, **intermediate operation(s)** and a **terminal operation.**

- A stream pipeline has three components:

  1. A source such as a Collection, an array, a generator function, or an IO channel;

  2. Zero or more intermediate operations; and

  3. A terminal operation

Example1

```
int sum = widgets.stream()

        .filter(w -> w.getColor() == RED)

        .mapToInt(w -> w.getWeight())

        .sum();
```

Here, widgets is a Collection<Widget>. We create a stream of Widget objects via Collection.stream(), filter it to produce a stream containing only the red widgets, and then transform it into a stream of int values representing the weight of each red widget. Then this stream is summed to produce a total weight.

Example2

```
List<Integer> list = Arrays.asList(1,2,3);

int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();

System.out.println(sum);
```

Using lambdas and stream to sum the squares of the elements on a list

Here map(x -> x*x) squares each element and then reduce((x,y) -> x + y) reduces all elements into a single number

Good practices by programmer

Commenting
- Commenting is necessary for maintaining any program
- Without commenting, the program is not complete
- Do not assume that the other person who reads this program will understand it clearly
- If commenting is not done at the beginning, it is forgotten.
- Program - a physical file – you need to comment the following at the top.
    - at the top of the program - header comment
    - what the program achieves
    - who coded on what day
- Classes
    - Comment at the beginning of the class about what the class does
    - class Employee
    - // this class is used to manage all information
    - // about employees and retrieve specific data
- Methods within classes
    - what this method/function/procedure does
    - what are the parameters and their purpose
    - what are the return values
- Comment before every loop.
- Comment before every file or database operation
- Comment before every if condition

Naming Conventions
- Readability improves understanding; that further improves maintainability
- If programs are not consistent, maintenance is tough
- If 20 developers work in a project and each one names functions in the way he/she wants, will the client approve the same?
- We must have a document on naming conventions
- Every document must be named properly and it must be stored in a proper folder.
- You need to have a convention for naming
    - Programs
    - Classes
    - Methods
    - Variables
    - Labels in programs
    - Reusable library functions

- There are established methods like Pascal casing, camel casing etc.
- Usually, you will tend to use a mixture of uppercase, lowercase letters, underscore etc.
- The ultimate aim is to achieve consistency across programs

Avoid Hard coding

- Placing a number or quoted text inside active code is hard coding
- This is deadly as the program restricts itself to this hard coded value
- To make a change, you need to change code, recompile and redeploy
- Usual way to avoid hard coding is to use constants at the top of a program; here also, you need to edit the program, but the change is in one place
- Other way is to put all configurable values in a csv or ini or dat file. Every program must read the name value pair from the file and later use them in the code
- Eg. MAX_LENGTH 150, PORT 8097

Modularity of code

- Modularity is important for easy maintenance

- Do not write lengthy methods or procedures

- We can split the modules based on

    – A functional operation as per requirements

    – A piece of code is used in more than one place

    – Logical breakdown of events in the functionality

- Modularity must be decided right at the design level itself

- Modular programs are easy to debug

- Fixes done on modular programs are easy to isolate from other regression effects

Examine the loops

- A loop is - repeat some logic for specific number of times or as long as a condition is true/false

- If this is not checked, it can run forever, infinitely and can bring down the server

- Loop Sample
  - gateway 1 // variables, flags
  - xyz = 10
  - loop starts here
  - ...gateway 2 // variables, flags, counters
  - ..
  - logic
  - ... gateway 3
  - loop ends here
  - gateway 4
- Gateways are the check posts where we must watch the value of the variables, loop counters and loop flags
- Ensure proper resetting of flags and counters and check their values and gateways of the loops
- Never allocate memory inside a loop
- Never instantiate a class inside a loop; if needed, close it within the loop
- Usually companies do not suggest more than 3 levels in the loops

Exceptions

- Issues may be caused by programmer
- Issues may be caused by system
- Most of the languages allow try-catch or on-error-goto exceptions
- Exception is also an error condition
  - we do not know when it will happen

  1. Any file operations – handle exceptions, because
  - file may not exist
  - file is already opened by someone
  - file does not have privilege

2. Any database operation – handle exceptions, because

  – database you may not have rights

  – database is down

  – connections exhausted

3. Memory

  – low memory exception

  – pointers - writing in privileged memory

  – array boundary breach

4. Any external evices

  – your program accesses webcam or printer

Performance

- We need to monitor cpu, memory, network

- Memory is directly related to variables and object – do not declare huge arrays or objects

- Cpu is consumed more when you deal with db or files or devices

- Any database operations, open late and close the connection early.

- Usage of indexes in queries must be examined

- Any column used in where condition of db query must have index on it

- Any memory allocated must be freed – else the program will shut down after some time

- Any object instantiated must be released – else system will be depleted of memory and hence performance will come down

Program Logic

- Usually the logic design will be provided to the developer

- If it is not provided, take 30 minutes and write the logic of the program in English first

- Do not start coding right away. You will make so many assumptions and it will spoil the show

- Get the logic approved by the team lead and then start coding

- Developers usually feel that this takes time; but it reduces the time effectively while coding and reworking

- Since developers are not used to documenting, they feel it is not their job. Hence they miss a lot of finer points

- If you write the logic first, you will get a lot of clarifications at that time itself and hence the code will come out clean

Lab Activity (Homework-Do It Yourself):

participants to complete the exercise given in the SoloLearn App and share the respective certificates with the trainers.