✱ YouTube → Faisal Memon

Type of Programming languages

① **Low-Level Languages:** → These are closest to the computer hardware and are directly understood by the machine

Example
- Machine Language:→ Written in binary code (0s and 1s)

- Assembly Language: Uses symbolic codes (like- MOV, ADD, SUB etc).

  * Used for:- System programming device drivers and embedded systems.

② **High-level Language:** These are easy for humans to understand and write. They require a compiler or interpreter to translate into machine code.

Example C, C++, Java, Python etc.

(3) Procedural Languages:

These are based on a step-by-step Procedure or instructions.

Example: C, Pascal, Fortran.

Used: Scientific, Mathematical and general Purpose Programming.

(4) Functional Languages:

These focus on functions and avoid changing state or data.

Example: Lisp, Haskell, Scala.

Used: Artificial intelligence, Mathematical computation, and data analysis.

(5) Scripting Languages:

These are mainly used for automation and web scripting

Example: JavaScript, Python, PHP, Perl, etc

Used: Web development, automation scripts, and Server management.

(6) Markup and Query Languages

These are used to structure, format, or manage data.

Example
- HTML (Hyper Text Markup Language) - for creating web pages
- SQL (Structured Query Language) for managing databases.
- XML / JSON - for data exchange between systems.

(7) :→ Object Oriented Programming (OOPs)

- ⊙ Everything is represented in the form of objects.
- ⊙ Objects are instance of classes
- ⊙ Class is like a blueprint, Object is a real thing built from that blueprint.

In Simple words:

OOPs helps us design software by thinking in terms of real-world entities – like a car, Student, bank account etc.

OOPs Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism.

* **Class in Java:** A class is a blueprint or template for creating objects. It defines what data (Variables) and functions (method) an object will have.

Example

```
Class car {
    String brand;
    int speed;
    void drive() {
    System.out.println( brand + " is
      driving at " + speed + " km/h");
    }
}
```

Object:

An object is an instance of a class - a real example created using that blueprint.

Example:

```
Car myCar = new Car();
myCar.brand = "Tesla";
myCar.speed = 120;
myCar.drive();
```

Example :- Class and Objects:

```
Class Car {
  // Attributes
  String brand;
  int speed;
  // Method (behavior)
  Void drive () {
    System.out.println(brand + " is
      driving at " + speed + "Km/h");
  }
}
```

```
Public class Main {
    Public static void main (String []
    args) {
     Car Toyota = new Car ();
      Toyota. brand = "Toyota";
      Toyota . speed = 100;

     Car Kia = new Car();
      Kia . brand = "kia";
      Kia . speed = 120;
// Call the drive ( ) method for
    both Objects.
      Toyota . drive();
      Kia . drive ();
    }
}
```

Output :- Toyota is driving at 100 km/h
Kia is driving at 120 km/h.

✱ What is a Constructor in Java?

A Constructor is a special method in Java that is automatically called when an Object of a class is created.

Syntax of a Constructor:

```
Class ClassName {
    ClassName (---) {
    }//-------
}
```

Important Rule:

- ⊙ The Constructor's name must be the same as the class name.
- ⊙ It has no return type - not even void.

✱ Types of Constructors :-

Java में तीन तरह के Constructors होते है :-

(1) Default Constructor:

A constructor with no parameters.

If you do not define any constructor, Java automatically provides a default constructor.

Example:

```
Small
Class Car {
    String brand;
    int speed;
  // Default Constructor
    car() {
        brand = "Unknown";
        Speed = 0;
    }
    Void Show() {
    System.out.println(brand + " is
        driving at " + speed + "km/h");

    }
}
Public class Main {
    Public Static Void main(String[]
        args) {
```

```
        Car c = new Car(); // Default
        Constructor called
            c.show();
        }
    }
```

Output: Unknown is driving at 0km/h.

(2) Parameterized Constructor:

A Constructor that takes parameters to initialize specific values:

* → HOME-WORK: "This is what we used in your Car example"

(3): Copy Constructor (User-Defined):

* Java does not have a built-in copy constructor like c++, but we can make one manually to copy data from an other Object.

Example

```
Class Car {
    String brand;
    String Color;
    int Speed;
// parameterized constructor
    Car (String b, String c, int s) {
        brand = b;
        Color = c;
        Speed = s;
    }
// Copy Constructor
    Car (Car obj) {
        brand = obj. brand;
        Color = obj. color;
        speed = obj. speed;
    }
    Void show ( ) {
        System. Out. println( brand + "
        (" + color + ") speed: " + speed);
        }
    }
```

```
Public class Main {
    public static void main(String [] args){
        Car Toyota = new Car("Toyota",
            "Red", 100);
        Car CopyCar = new Car(Toyota);
        // Copy constructor called.
            Toyota.show();
            CopyCar.show();
    }
}
```

Output: Toyota (Red) speed : 100 → Toyota

Toyota (Red) Speed : 100 → CopyCar

Note: → "this" keyword in Java is a reference variable that refers to the current object (the object which is calling the method or constructor")

Home-work: " Create a class car with instance variables using (this keyword

- brand (String)
- color (" ")
- Speed (int)

✱ Getter and Setter in Java.

Getters and Setters are special methods used to access and modify Private Variables of a class.

They help in encapsulation – hiding data and controlling access to it.

⊙ Why Use Them?

(1) To make class fields private (data hiding)

(2) To provide controlled access to variables

(3) To validate or modify data before setting or returning it.

Example

```
Class Student {
    Private String name;
    Private int age;
  // Getter to read value
   Public String getName() {
      return name;
   }
  // Setter to Set Value.
```

```
Public void setName(String name) {
    this.name = name;
}
Public int getAge() {
    return age;
}
Public void setAge(int Age) {
    if (age > 0) {
    // Validation ---
    this.age = age;
    } else {
        System.out.println("Age must be
            Positive!");
    }
}
}
Public class Main {
    Public static void main(String[] args) {
    Student S1 = new Student();
    S1.setName("Sandy");
    S1.setAge(24);
```

```
        System.out.println("Name : " + s1.getName());
        System.out.println("Age : " + s1.getAge());
    }
}
```

Output: Name : Sandy
Age : 24

Key Points:

Concept	Description.
Getter	Used to read private data (getVariableName())
Setter	Used to write/update private data (setVariableName())
Encapsulation	Achieved by making variables private and using getters/setters.
Validation	Can be added inside setter to control input.
Naming Rule	Always starts with get or set followed by variable name. (capitalized.)

What is a Record in Java:

A Record in Java is a special types of class introduced in Java 14 (Preview) and made stable in Java 16. Used to store immutable data in a compact way.

Example

```
Public record Student(String name,
    int age) { }

public class Main {
    Public static void main(String [] args) {
        Student s1 = new Student("Sandy", 24);
System.out.println(s1.name());
// Getter
System.out.println(s1.age());

System.out.println(s1);
// Auto toStrings()
    }
}
```

Output:

```
Sandy
24
Student[name=Sandy, age 24].
```

* Key Features:

1. Immutable
2. Auto-generated methods
3. Compact syntax
4. Can implement interfaces ("yes")
5. Cannot extend classes
6. No setters.

What is Encapsulation in Java:

Encapsulation means binding data (variables) and methods (function) that operate on that data into a single unit (class) and restricting direct access to the data.

* Key Concept:

- Make variables private not accessible directly.
- Use getter and setter methods to access or modify data safely.

Example

```
class Car {
    Private String brand; // data hidden
    Private int speed;
    // Getter
    public String getBrand () {
        return brand;
    }
    // Setter
    Public void setBrand (String brand) {
        this.brand = brand;
    }
    Public int getSpeed () {
        return speed;
    }
    // setter with validation.
    Public void setSpeed (int speed) {
        if (speed > 0)
            this.speed = speed;
        else
            System.out.println("speed must be
                positive!");
    }
}
```

```
Public class Main {
    Public Static void main(String [] args) {
    Car car = new Car();
        car.SetBrand ("Toyota");
        Car.SetSpeed (120);
    System.Out.println("Brand:" + Car.getBrand());
      System.Out.println("Speed: " + Car.getSpeed());
        }
}
```

Output:

```
Brand : Toyota
Speed : 120
```

- Advantages:

① Data Security

② Data Control

③ Code Flexibility

④ Improved Maintenance

Inheritance in Java

Inheritance is a mechanism in Java by which one class (child/subclass) can acquire properties and behaviors (fields and methods) of another class (Parent/ superclass).

- Purpose:
  - To reuse code
  - To avoid duplication
  - To establish relationship between classes.
  - To support polymorphism.

Example →

```
// Parent Class
class Vehicle {
 String brand = "Generic Vehicle";
 Void start() {
   System.out.println(brand + " is
        starting....");
   }
 }
// Child class
 class Car extends Vahicle {
    int speed = 120;
   void showDetails () {
   System.out.println(brand + "running at" +
      speed + " KM/h");
 } }
```

```
// Main class
Public class Main {
    Public Static Void main (String[] args) {
        Car Car = new Car();
        Car.start();
        car.showDetails();
    }
}
```

Output: Generic Vehicle is starting....
Generic Vehicle running at 120 km/h.

⇒ Types of Inheritance in Java.

Type	Description
① - Single Inheritance	One class inherits another
② Multilevel Inheritance	Class inherits from another class which is already a subclass.
③ ⇒ Hierarchinal In -	Multiple subclasses inherit from one Parent
④ Multiple Inheritance (through interface.)	- One class implements multiple interfaces.

Advantages of Inheritance:-

1. Code Reusability
2. Easy Maintenance
3. Method Overriding
4. Clean Structure

* Rules of Inheritance:

1. Private members of parent are not inherited
2. Constructors are not inherited but can be called using super()
3. Java does not support multiple inheritance with classes (only via interfaces)
4. Order of constructor call, : Parent, Child.

✱ Polymorphism in Java.

Polymorphism means "One name, many forms".

In Java. It allows a single method or object to behave differently based on context or object type.

✱ Types of Polymorphism in Java.

Type	Also called
Compile-Time	Static Polymorphism / Method Overloading
Runtime	Dynamic Polymorphism Method Overriding

(1)- Method Overloading.

```
class Calculator {
    int add (int a, int b) {
        return a+b;
    }
```

```
    double add (double a, double b) {
        return a+b;
    }
}

Public class Main {
    Public static void main(String [] args){
        Calculator calc = new Calculator();
    System.out.println (calc.add (5, 10));
    System.out.println (calc.add (5.5, 10.5);
    }
}
```

Output: 15
16.0

" Same method name, different Parameters "

② Runtime or- Method Overriding

```
Class Vehicle {
    void start () {
        System.out.println("Vehicle is
            starting---);
    }
}
```

```
Class Car extends Vehicle {
    @Override
    Void start() {
    System.out.println("Car is starting---");
    }
}

Public class Main {
    Public static Void main(String []
    args ) {
        Vehicle = new Car(); // Parent
        reference, child Object.
        V.start(); // Calls Car's
        Start() - runtme Poly--
    }
}
```

Output: - Car is starting ---

✱ Abstraction in Java.

Abstraction is the OOP concept of hidding internal implementation details and showing only essential features to the user.

(1) Abstract Class

A class declared with 'abstract' Keyword.
Can have abstract methods (no body) and Concrete method (with body)
Cannot create objects of an abstract class directly

Syntax:

```
abstract class BankAccount {
    String accountHolder;
    double balance;
abstract void deposit(double amount);
abstract void withdraw(double amount);
    Void displayBalance()
{   // Concrete method
    System.out.println("Balance:" +balance);
}
}
```

Key Points:

- Abstract class – Blueprint for sub classes
- Subclass must implement all abstract method
- Can have constructors, fields, and normal methods.
- Can achieve partial abstraction.

(2) Interface:

Pure abstraction: only method signatures
(Java 8+ allows default/staticMethods)
Declared using interface keyword
Classes implement the interface using implements keyword.

Example →

~~Class Robot~~

```
Interface ControlSystem {
    Void Start ();
    Void Stop ();
}
```

```
class car implements ControlSystem {
    @ Override -
    Public Void start ( ) {
     System. out . println ("Car Started");
        }
    @ Override
    Public Void stop ( ) {
     System. out. println("Car Stopped ");
        }
    }
Public class Main {
    Public static Void main(String [ ]
     args ) {
    ControlSystem. c = new Car ( );
            c. Start ( );
            c. Stop ( );
        }
    }
```

Output:

```
Car Started
Car Stopped.
```

* Difference between Abstract Class and Interface:

Feature	Abstract class	Interface
① Inheritance	extends	Implements
② Methods	Abstract + Concrete	Only abstract (before Java 8)
③ Variables	Can have instance Variables	Only constants (Public, Static, Final)
④ Constructor	Yes	No
⑤ Multiple Inheritence	Not allowed	Allowed (using Interfaces)
⑥ Object creation	No	No