If you have any questions or suggestions regarding the notes, please feel free to reach out to me on

Sanjay Yadav Phone: 8310206130

https://www.linkedin.com/in/yadav-sanjay/

https://www.youtube.com/@GeekySanjay

https://github.com/geeky-sanjay/

==========

Roadmap: Java Basics to Advanced - Part 1

==========

OOPs Fundamentals from Java Perspective:

- Sub-topics
  - Classes
  - Object
  - Constructor etc.
  - 4 pillars
    - Inheritance
    - Polymorphism
    - Abstraction
    - Encapsulation

Basic Overview of Java:

- Procedural vs OOPs
- What is Java and what makes it Platform Independent
- JDK vs JRE vs JVM
- Installation
- Setting Classpath Environment Variables
- Writing First Java Program

==========

Effortless Installation: Setting up IntelliJ IDEA on Windows!

Visit

https://youtu.be/2fm5ddR8fJ8

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

==========

==========

Going One Level Deep

==========

- **Understanding Classes, different types and Objects**
- **Abstract Classes**
- Inner Classes etc.
- **Understanding Variables**
  - **Static Variables**
  - **Final Variables**
  - **Primitive Variables**
  - **Object references**
  - **Cover Big decimal vs Double**
  - **Understanding about String**
    - String Pool
    - String Immutability
- **Access Specifiers**
- **Type Casting**
  - Implicit Type Casting
  - Explicit Type Casting

- **Understanding Method and Different Types**
  - Cover Return Type
  - Cover static method
  - Method parameters
  - Pass by value vs pass by reference
  - Overloading etc.
- **How does Memory Management Happens in Java**
  - Understand about heap and stack memory
- **Garbage Collector**
- **Understand Constructor**
  - Private Constructor
  - Default Constructor
  - Parameterized Constructor
  - Constructor vs Method
- **Files and Directories in Java**
  - Read and Write from File using Scanners
- **Understanding Package and import**
- **Understand POJOs etc.....**

OOPs Fundamentals from Java Perspective

- Overview of OOPS
- Objects and Classes
- Pillar of OOPs
1 Data Abstraction
2. Encapsulation
3 Inheritance
- Types of Inheritance: Single, Multiple etc.
4 Polymorphism
- Static Polymorphism
- Dynamic Polymorphism
- Relationship
- Is-a Relationship
- Has-a Relationship

OOPS OVERVIEW ——————————————————————————————————————>

- OOPS mean Object Oriented Programming
- Here Object means real word entity like Car, Bike, ATM etc.
- Procedural Programming Vs OOPS

S.No.	Procedural Programming	OOPS
I.	Program is divided into parts called Functions	Program is divided into Objects
2.	Does not provide proper way to hide data, gives importance to function and data moves freely.	Objects provide data hiding, gives important to data.
3.	Overloading is not possible	Overloading is possible
4.	Inheritance is not possible	Inheritance is possible
5.	Code reusability does not present	Code reusability is present
6.	Examples: Pascal, C, FORTRAN etc.	Examples:Java, C\#, Python, C++ etc.

Real-world entities -> Real-world entities in terms of objects mean representing things or concepts from the real world as objects in computer programming. For example, if you're building a program for a school, real-world entities could be students, teachers, and classrooms. Each of these entities becomes an object in the code, with specific characteristics (attributes) and actions (behaviors) that mimic their real-world counterparts. This approach makes it easier to design software that mirrors the structure and functionality of the actual system or scenario you're working with.

In any given subject, there are essentially two components: properties (attributes) and

behaviors (functionalities that can perform some actions), collectively referred to as entities.

1. DATA ABSTRACTION

- It hides the internal implementation and shows only essential functionality to the user.
- It can be achieved through Interface and abstract classes

Example:

- Car - we only showed the BRAKE pedal, and if we press it, Car speed will reduce. But HOW? That is ABSTRACTED to us.
- Cell Phone - how a call is made that is Abstracted to us.

Advantages of abstraction:

- Increase security and confidentiality

DEMO OF DATA ABSTRACTION
Abstraction is like simplifying things for someone. Imagine a car with lots of features, but when you give it to someone, you only show them the basics like the brake, ignition, and speed controls. This helps make it easier for them, and it also keeps the car safe. We achieve this simplification using tools like interfaces and abstract classes, which allow us to show only what's necessary (like pushing a button to start the car) without revealing all the complicated inner workings. This way, we protect the car from being messed up accidentally, keeping things simple and secure.

2. DATA ENCAPSULATION

- Encapsulation bundles the data and the code working on that data in a single unit
- Also known as DATA-HIDING.
- **Steps to achieve encapsulation:**
  - Declare Variable of a Class as private.
  - Provide Public Getters and Setters to modify and view the value of the variables.
  - Advantages of encapsulation
- Loosely coupled code
- Better access control and security

DEMO OF DATA ENCAPSULATION
It's about keeping information hidden. Imagine putting all the details about something, like its characteristics and actions, in one container (class). This way, we have control over who gets to see what by using access modifiers like "private." We can decide which parts are accessible and only show certain information using getter methods like "getName" When changes are made, we can observe who's doing what through setter methods, and we can also limit access for specific users. by using object of class with help of accessor(.) we can access this getter, setters. This means we have complete control over the data. We don't

reveal everything to the user; we only show what we want. This practice is known as data hiding.

3. INHERITANCE

- Capability of a Class to inherit properties from their Parent Class,
- It can inherit both functions and variables, so we do not have to write them again in child classes.
- Can be achieved using extends keyword or through interface.
- **Types of Inheritance:**
  - Single Inheritance
  - Multilevel Inheritance.
  - Hierarchical Inheritance.
  - Multiple Inheritance - through interface we can resolve the diamond problem

Advantages of Inheritance

- Code reusability
- We can achieve Polymorphism using Inheritance.

DEMO OF TYPES OF INHERITANCE

Inheritance is the ability to acquire all the public and protected methods and data members from a parent class by using the "extends" keyword in a child class. It is allowed from parent to child not child to parent means we can not get any property and method of child in parent using inheritance.

a child class does not have direct access to private members (data and methods) of its parent class. Private members are intentionally kept hidden from outside classes to maintain encapsulation and data integrity.

However, child classes do inherit the public and protected members of the parent class. Public members are accessible by any class, including the child class, while protected members are accessible only by the parent class and its descendants (including the child class). It's important to note that inheritance is a mechanism that allows child classes to reuse and extend the behaviour of a parent class without directly accessing its private members.

So, in short, a child class generally cannot access the private members of its parent class. Access is limited to public and protected members, promoting encapsulation and maintaining the integrity of the parent class's implementation details.

Type of inheritance -> Inheritance in object-oriented programming can be categorised into several types based on how classes inherit and share their characteristics. The main types of inheritance are:

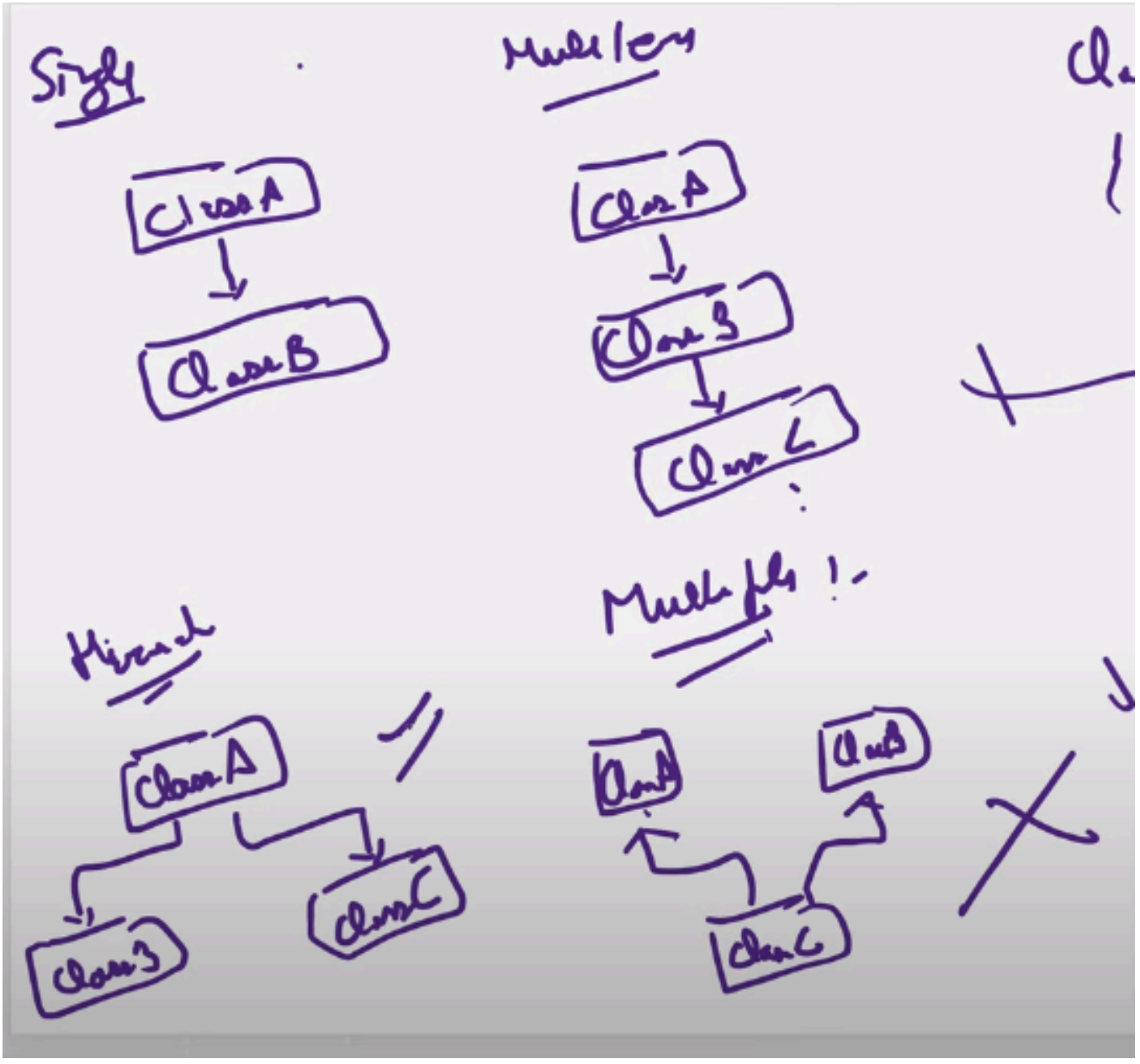
Single Inheritance: A class inherits from only one base class.
Example: class ChildClass extends ParentClass

Multilevel Inheritance:
A class inherits from another class, and then another class inherits from it.
Example: class GrandparentClass -> class ParentClass extends GrandparentClass -> class ChildClass extends ParentClass

Hierarchical Inheritance:
Multiple classes inherit from a single base or parent class.
Example: class ParentClass -> class Child1Class extends ParentClass -> class Child2Class extends ParentClass

Multiple Inheritance:
A class inherits from more than one base class.
Not supported by java languages due to complexity and ambiguity issues.
Example: class ChildClass extends ParentClass1, ParentClass2

As we can see below **1st** Single inheritance, **2nd** Multilabel inheritance, **3rd** Hierarchical Inheritance are allowed but **4th** Multilevel Inheritance is **not allowed in java.**



4. POLYMORPHISM

- Poly means Many and Morphism means Form.
- The same method behaves differently in different situations.

  Example:
  - A person can be a father, husband, employee etc.
  - Water can be liquid, solid and Gas etc.

- **Types of Polymorphism**

  - Compile Time / Static Polymorphism / Method Overioading
  - Run Time / Dynamic Polymorphism / Method Overriding

DEMO OF TYPES OF POLYMORPHISM

Poly means **many**, and **morphism** means **forms**. So, when something has many different forms or behaviours, it's called polymorphism. For example, consider a man who can be a father, a husband, and an employee. Although he's one person, he takes on different roles in different situations. At home, he's a father to his children, a husband to his wife, and in the office, he's an employee for the company. This versatility in roles is an example of polymorphism.

In programming, there are two types of polymorphism:

1. Compile-time / Static Polymorphism / Method Overloading:

This occurs when multiple methods in the same class have the same name but different parameters.

The compiler decides which method to invoke based on the method signature during compilation.

- **signature** -> "method signature" refers to the unique combination of a method's name and its parameter types. The method signature helps the compiler differentiate between different overloaded methods in the same class.

  A method signature includes the following components:

  Method Name:

  This is the name given to the method and serves as its identifier.

  Parameter Types:

  It includes the types and order of parameters that the method accepts.

  For example, consider two methods in the same class:

```
public class Example {
```

```
    // Method with the name "add" and two int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method with the name "add" and two double parameters
    public double add(double a, double b) {
        return a + b;
   }
}
```

In this example, there are two methods named "add," but their method signatures differ based on the parameter types. The first add method accepts two int parameters, and the second add method accepts two double parameters.

During compilation, when the compiler encounters a method call, it uses the method signature to determine which specific method to invoke based on the arguments provided. This is why it's crucial for the method signatures to be distinct, allowing the compiler to resolve method calls accurately.

Note: The method signature does not include access modifiers (such as public or private) and return types. Additionally, methods with the same number of parameters, having the same data types but different parameter names, are considered to have the same signature. During compilation, the differentiation is based solely on data types, not on parameter names.

2. Run-time / Dynamic Polymorphism / Method Overriding:
This occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
The decision on which method to execute is made at runtime, based on the actual type of the object.

So, polymorphism in programming allows for flexibility and adaptability by enabling entities to exhibit different behaviours in various contexts.

Note: it's important to note that data members (fields or attributes) cannot be overridden. Unlike methods, which can be redefined in a subclass to provide a specific implementation, data members maintain their original definition in the subclass.

For example, consider a base class Parent with a data member int x:

```
public class Parent {
    int x = 10;
```

```
}
```

If a subclass Child extends Parent:

```
public class Child extends Parent {
   // Data member x from the Parent class cannot be overridden.
   // Child has its own copy of x, and modifying it won't affect Parent's x.
   int x = 20;
}
```

In this case, Child has its own x data member, and it does not override the x from the Parent class. Each class has its own distinct data members.

OBJECTS RELATIONSHIPS ——————————————————————————————————>

In object-oriented programming, the terms "**isa**" and "**hasa**" refer to two types of relationships between objects:

*** Is-a Relationship**

- **Achieved through Inheritance**
- **Example:DOG is-a Animal**
- **Inheritance form an is-a relation between its parent child classes.**
- **Has-a Relationship**
  - Whenever an Object is used in Other Class, it's called HAS-A relationship
  - Relationship could be one-one, one-many, many to many.
  - **Example:**
    - School has Students,
    - Bike has Engine,
    - School has Classes.
  - **Association relationship between 2 different Objects.**
    - Aggregation Boch Objects. Can survive individually, means ending of one object will not end gather object.
    - Composition: Ending of One object will end another object.

Is-a Relationship (Inheritance):
"Is-a" relationship represents inheritance or specialisation. It signifies that one class is a specialised version of another.
Example: If there is a class "Car" and a class "Sedan" that extends "Car," it can be said that "Sedan is a Car." This implies that "Sedan" inherits attributes and behaviours from the more general "Car" class.

Has-a Relationship (Composition or Aggregation):

"Has-a" relationship represents composition or aggregation. means it has an object of another class. It signifies that one class has another class as a part or member.
Example: If there is a class "Car" and another class "Engine," it can be said that "Car has an Engine." In this case, the "Engine" class is part of the "Car" class, and there is a composition relationship between them.
In summary:

"Is-a" relationship is about inheritance and specialisation.
"**Has-a" relationship** is about composition or aggregation, indicating that one class has another as a part.

Let's use an analogy with classes for courses and students:

In a **one-to-one relationship**, a student is associated with exactly one course object.
In a **one-to-many relationship**, a student is linked to multiple course objects.
In a **many-to-many relationship**, a student can have many course objects, and conversely, a course class can be associated with multiple student instances.
These relationships help describe how classes and objects interact, allowing for a more flexible and accurate representation of connections in a system.

Association: relationship between 2 different Objects -> Association refers to the relationship between two distinct objects, emphasising the connection and interaction between individual instances rather than the classes to which they belong. In this context, we are discussing the association between specific object instances, highlighting their connection in a system.

Aggregation -> Aggregation is a type of association where one class represents a part of another class. Represents a looser association(loosely coupled), and the part class can exist independently. is good for code structure
Symbol: Aggregation is typically represented by a hollow diamond on the side of the whole class.
Example: If we have a class "Library" and a class "Book," an aggregation relationship could be established because a library has books, but books can exist independently of the library.

Composition -> Composition is a stronger form of association, where one class is composed of another class. Implies a stronger relationship, and the part class cannot exist independently and is tightly bound (tightyly coupled) to the whole class's lifecycle. it is bad for code structure.
Symbol: Composition is represented by a filled diamond on the side of the whole class.
Example: If we have a class "Car" and a class "Engine," a composition relationship could be established because the engine is an integral part of the car, and it cannot exist separately.

==========

Java Program Execution Journey: From Source Code to Bytecode and Beyond

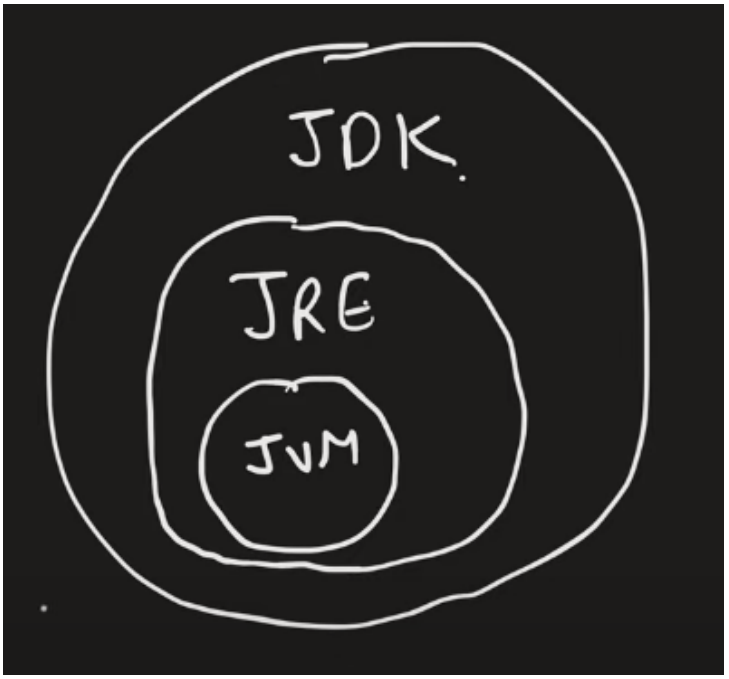
==========

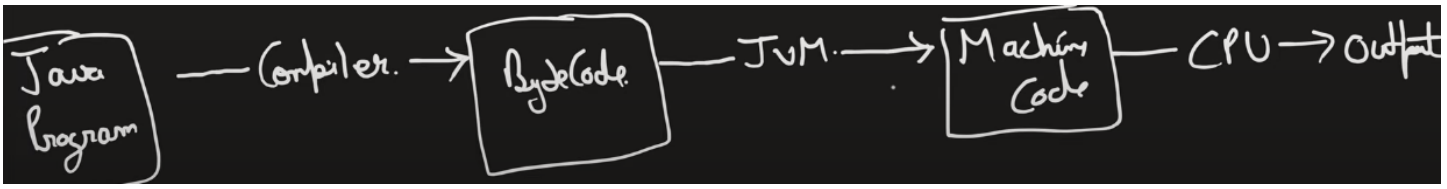
What is for JAVA?

Java is a widely-used, high-level programming language renowned for its platform independence and strong support for object-oriented programming (OOP) principles. Initially developed by Sun Microsystems (now owned by Oracle Corporation), Java was introduced in 1995 and has since gained immense popularity globally.

WORA -> WORA stands for "Write Once, Run Anywhere" in Java. It refers to Java's platform independence, where Java programs can be written once and then run on any device or platform that has a Java Virtual Machine (JVM) installed, without the need for recompilation. This feature allows Java developers to create software that can be deployed across different operating systems and hardware platforms with minimal effort.

It has 3 main component which makes it platform independent



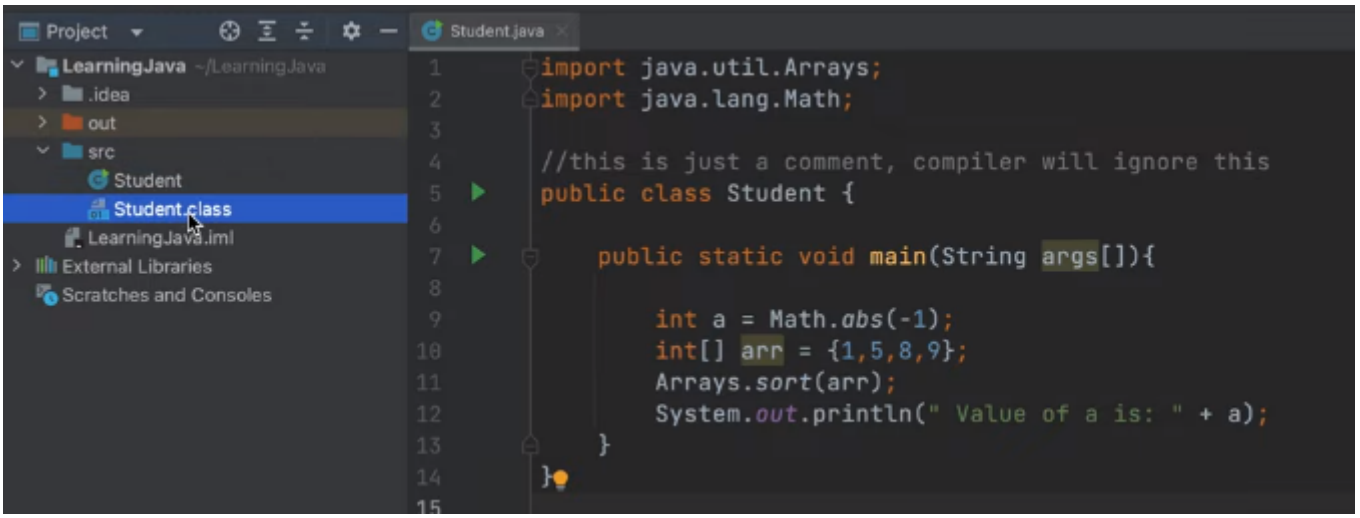


In Java, the process of executing a program involves several steps, including compilation, bytecode generation, and execution on the Java Virtual Machine (JVM). Here's an overview of the process:

Writing Java Code:
Developers write Java source code using a text editor or an integrated development environment (IDE).

Compilation:
The Java source code (.java files) is compiled by the Java Compiler (javac) into intermediate code called bytecode. The bytecode is a set of instructions specific to the Java Virtual Machine (JVM) rather than machine-specific code.
javac YourProgram.java -> **write in intelije terminal**

Bytecode Generation: above command executes the program and performs compilation
The compilation process generates bytecode (YourProgram.class files), **which is a platform-independent** representation of the program.



Java Virtual Machine (JVM):
The JVM is a software-based virtual machine that interprets and executes Java bytecode. The JVM provides a runtime environment for Java programs to run on different platforms without modification. **It is platform dependent, meaning the window has its own jvm and mac also.**

Bytecode Execution:
The JVM loads the generated bytecode and executes it.
During execution, the JVM translates bytecode into machine code using its **Just-In-Time (JIT) compiler**. The translated machine code is specific to the underlying hardware architecture.

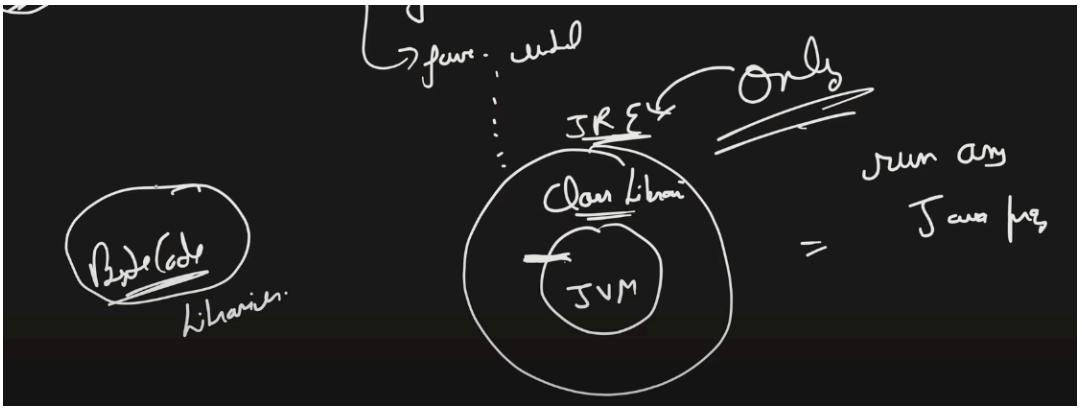
Execution on CPU:
The translated machine code is executed by the Central Processing Unit (CPU) of the computer.

Output:

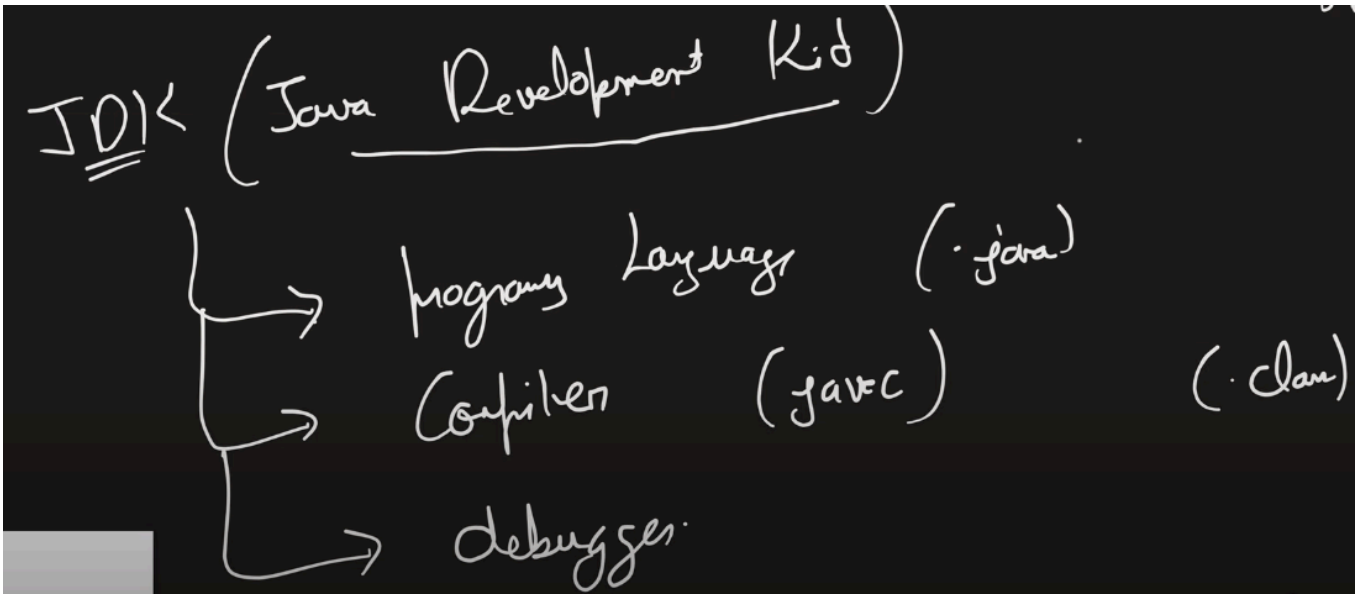
The program produces output based on its logic and functionality.

In summary, Java source code is compiled into bytecode, which is then executed by the JVM. The JVM, in turn, translates bytecode into machine code for the specific CPU architecture, allowing Java programs to run on various platforms. This compilation and execution process contributes to Java's "write once, run anywhere" principle, enabling portability across different systems.

JRE (Java RunTime Environment) ->
JRE stands for Java Runtime Environment. It is a software package that provides the necessary runtime support for executing Java applications and running Java applets. The JRE includes the Java Virtual Machine (JVM), class libraries, and other supporting files required for the execution of Java programs. In simple terms, JRE is what allows Java applications to run on a computer by providing the runtime environment necessary for their execution. **Simply it is prewritten set of libraries like java.Math, java.util etc with JVM**



JDK -> ==========
JDK stands for Java Development Kit. It is a software development kit used for developing Java applications and applets. The JDK includes the Java Runtime Environment (JRE), as well as additional tools and libraries needed for Java development. Developers use the JDK to write, compile, and debug Java code. In summary, while the JRE is focused on providing the runtime environment for executing Java programs, the JDK goes a step further by including tools and resources for the actual development of Java applications.



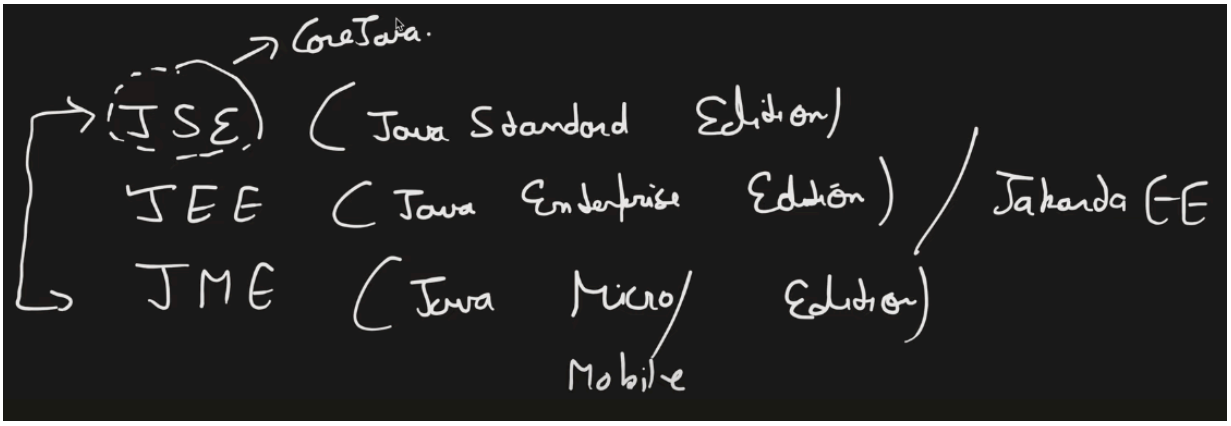
JDK = JRE + (Plateform + Compilor + Debuger

JDK stands for Java Development Kit. It includes the Java Runtime Environment (JRE) along with additional tools such as a compiler and debugger, making it a comprehensive package for Java development.

Note: While the Java Virtual Machine (JVM), Java Runtime Environment (JRE), and Java Development Kit (JDK) are platform-dependent components, the beauty of Java lies in its platform independence. The compiled bytecode (.class files) generated by the JDK can be executed on any platform with the help of the corresponding JVM. This platform independence is achieved through the JVM's ability to translate bytecode into machine code that is understandable by the specific environment in which it is running, allowing Java programs to run seamlessly across different operating systems.

To run java in system we required JDK which involved all required library and run time environment we can download java from below link
https://www.java.com/en/download/
java -version is a command to check which version java is installed.

Type of java -> ==========
JSE, JEE, and JME refer to different editions of the Java platform, each tailored for specific application domains:



JSE (Java Standard Edition):

- JSE is the core Java platform that provides the fundamental APIs (Application Programming Interfaces) and libraries.
- It includes the basic components required for developing desktop applications, command-line tools, and applets.
- JSE forms the foundation for general-purpose Java development.

JEE (Java Enterprise Edition):

- JEE is an extension of JSE, designed for developing enterprise-level, scalable, and distributed applications.
- It includes additional APIs and features for building web applications, enterprise-level services, and distributed systems.
- JEE provides tools and specifications for developing large-scale, robust, and secure applications in a server-side environment.

JME (Java Micro Edition):

- JME is a compact version of the Java platform, specifically designed for resource-constrained environments, such as mobile devices and embedded systems.
- It is tailored to run on devices with limited processing power, memory, and storage.
- JME includes a subset of the JSE APIs to accommodate the constraints of smaller devices.

Note about Java Program ->

==========

- Naming convention dictates that the filename of a class should match the class name itself.
- Each class file typically contains only **one public class**.
- Within a class, you can include variables (data members), methods, constructors, and even nested classes or inner classes.
- After a change in the program it is always required to compile using **javac Filname.java** first then run jvm because jvm only reads compiled code and runs **java Filename**.
- **The main method functions as the program's entry point**. After converting the program into bytecode, the JVM directly invokes the main method. This is why it is declared as public because it is called from anywhere like from outside of the package and it is static; there's no need to create an object. Additionally, it is declared as void

because it does not return any value. Here we have argos which are used if a user wants to pass any argument.
public static voide main(String args[])


```java
public class Employee {
 public static void main(String angs[]) {
 //this is a comment Line, compiler witl iqnore this
 int a = -10;
 System.out.println("this is my first progran and output of a is: " + a );
 }}
```

To compile write: **javac Employee.java**
To run in jvm and see output: **java Employee**

Variables -> ==========

Topics which i have covered:
- what is variable
- How to declare variable
- Naming convention of a variable
- Types of Variable:
  - Primitive Type:
    - boolean Type
      - boolean
    - Numeric Type
      - Integral Type
        - byte
        - char
        - short
        - int
        - long
      - Floating Type
        - float
        - double
  - Reference Type:
    - Classes
    - Interfaces
    - Array
    - String
    - Enum

- Kind of Variables:
  - Local
  - Instance
  - Static/class variable
  - Method Parameters
  - Constructor Parameters

- Types of Conversion:
  - Widening/ Automatic Conversion
  - Narrowing/ DownCasting / Explicit Conversion
  - Promotion during expression
  - Explicit Casting during Expression

- Wrapper Classes
  - Boxing/Auto-boxing and Unboxing

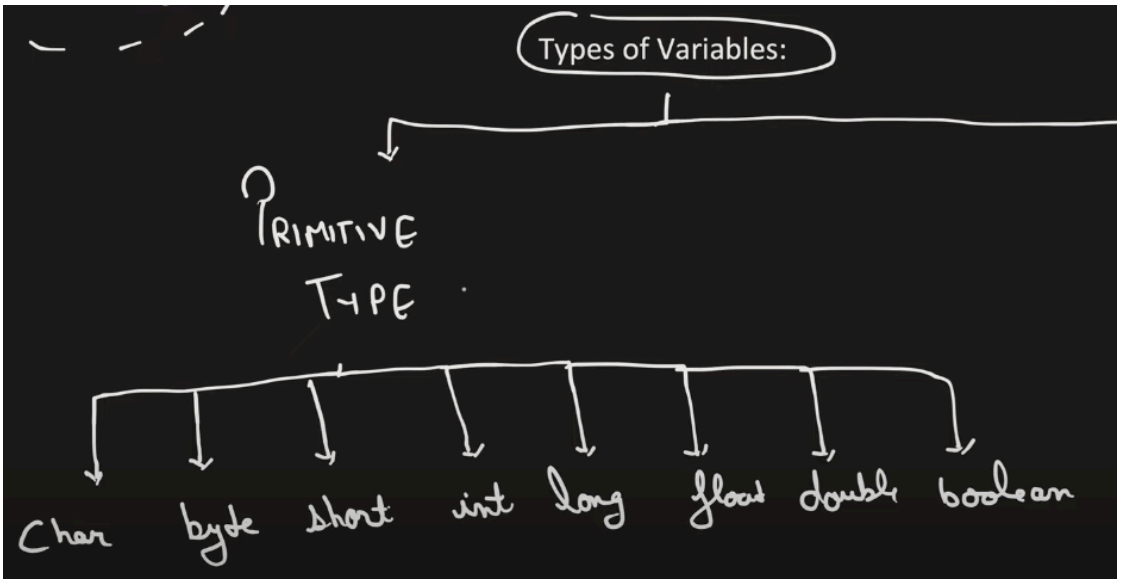
- final variables

- static final variable (similar to constant in C) only 1 copy and immutable

1. What is Variable:

```
Variable is a container which holds a Value
[Datatype VariableName = value; ]
int var = 32;
```

1. Variable name is Case-sensitive
2. Variable name can be **any legal identifier** means can contain **Unicode letters** and **Digits**.
3. Variable name can start is $, _ and letter.
4. Variable name can not be JAVA reserved Keywords like "new", "class", "while", "for", "interface", "int", "float" etc.
5. Variable should be all small if it contains only 1 word else camel case should be followed
6. For Constant, Variable name should be define in CAPITAL LETTERS

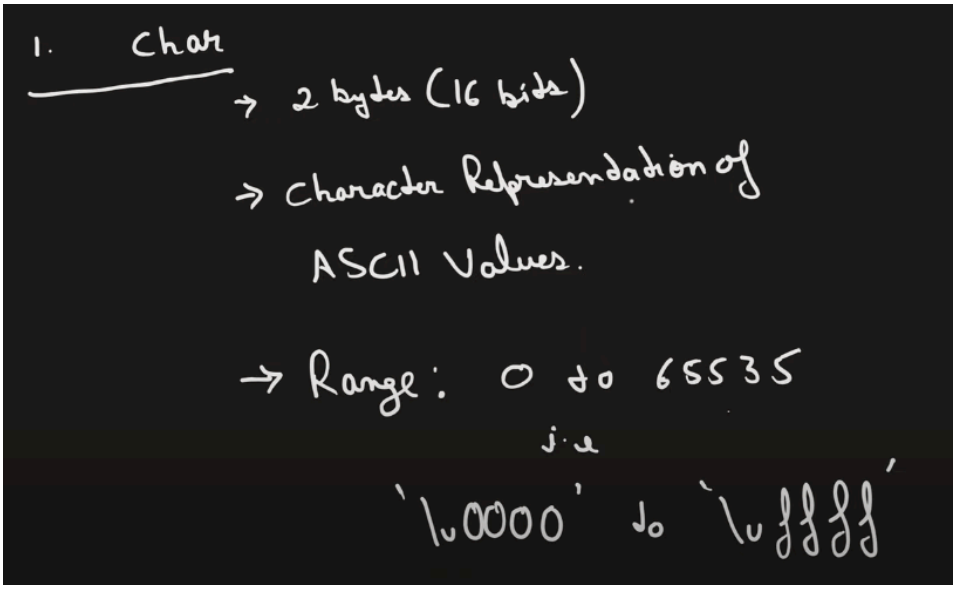


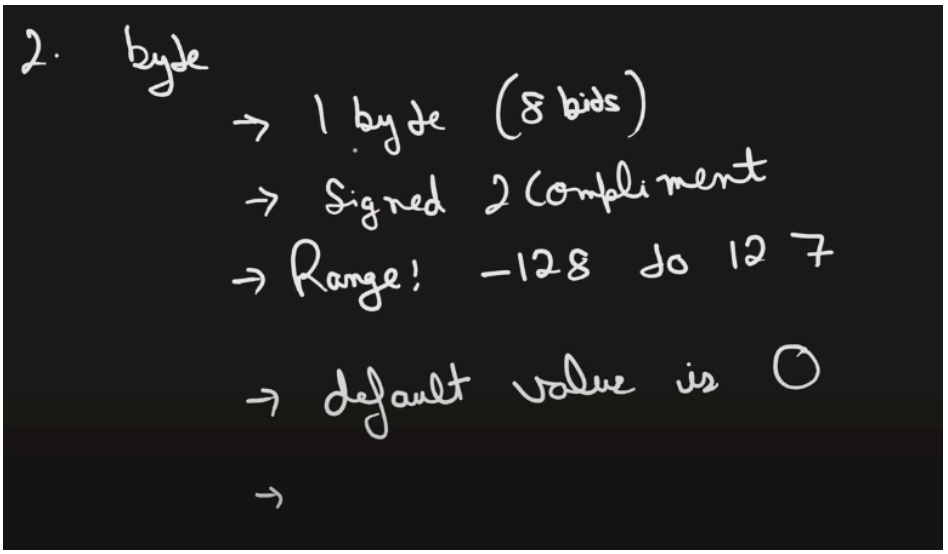
Note ->

1. default value is only works for class variables not for local*Variables like in methods. if we will not define any value to class variable it will intilized default value like 0 for int but in method it will give compile time error and insist to assign any value.

2. Char, byte, short, int, long are all integral types, meaning they support both positive and negative values, and these are considered signed variables.

3. If you require **precise** and arbitrary-precision representation of decimal numbers, especially when dealing with very large or very small values, you may consider using a data type like **BigDecimal** in Java or similar in other programming languages. BigDecimal provides arbitrary precision and **avoids some of the limitations inherent in floating-point representation**.





3. short
→ 2 bytes (16 bits)
→ Signed 2 Compliment
→ Range :- -32768 do 32767
→ default value is 0

int
→ 4 bytes (32 bits)
→ Signed 2 Compliment
→ Range :- -2^{31} do $2^{31}-1$
→ default value is 0

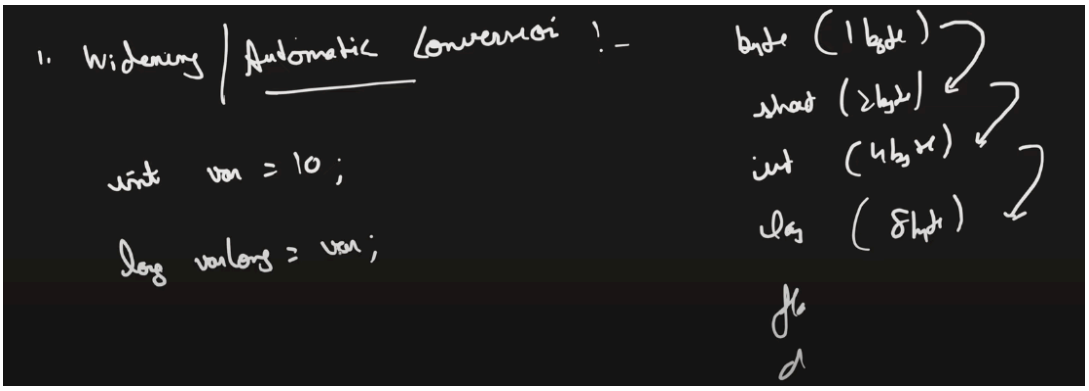
long
→ 8 bytes (64 bits)
→ Signed 2 Compliment
→ Range: -2^{63} do $2^{63}-1$ ← long = 500L;
→ Default Value is 0.

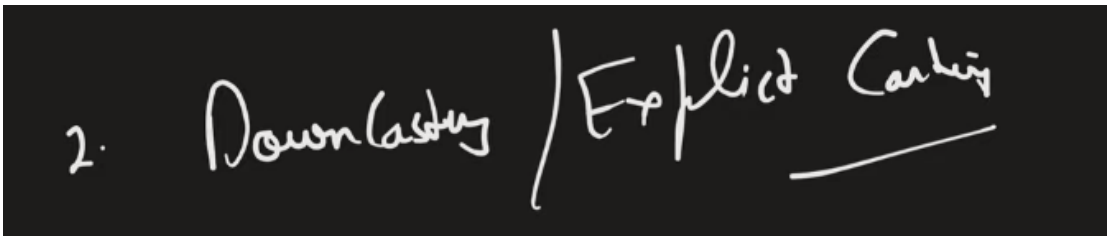
6. float
→ 32 bit IEEE 754 value

7. double
→ 64 bit IEEE 754 value

boolean :- → 1 bit
→ True / false.

Types of conversion ->





Downcasting means whenever we will assign height type to lower type it will happen and i t will give us a compile time error to solve that we have to do it manually it is called explicit casting. We will do it manually but if the value overflows it will return minus or random values. like int store in byte here byte value is -128 to 127 and if we will try to add 128 it will output -127.

```
public class Student {
 public static void main(String args[]) {
   int integerVariable = 10;
   byte byteVariable = (byte) integerVariable
   System.out.println(byteVariable);
 }
}
```

Member variables -> Directly declared within a class using an access modifier. They become accessible after creating an object and are referred to as instance and member variables. Each time a new object is instantiated, a new copy of this data member is created in all objects.

Local variables -> Defined within a method and are only accessible within that method.

Class variables -> Denoted by the addition of the static keyword. They are termed class variables because they belong to the class and can be directly accessed using the class name. These variables cannot be used in non-static methods and are not accessible using an object. Accessing them via the class name also yields only one copy, making them class or static variables.

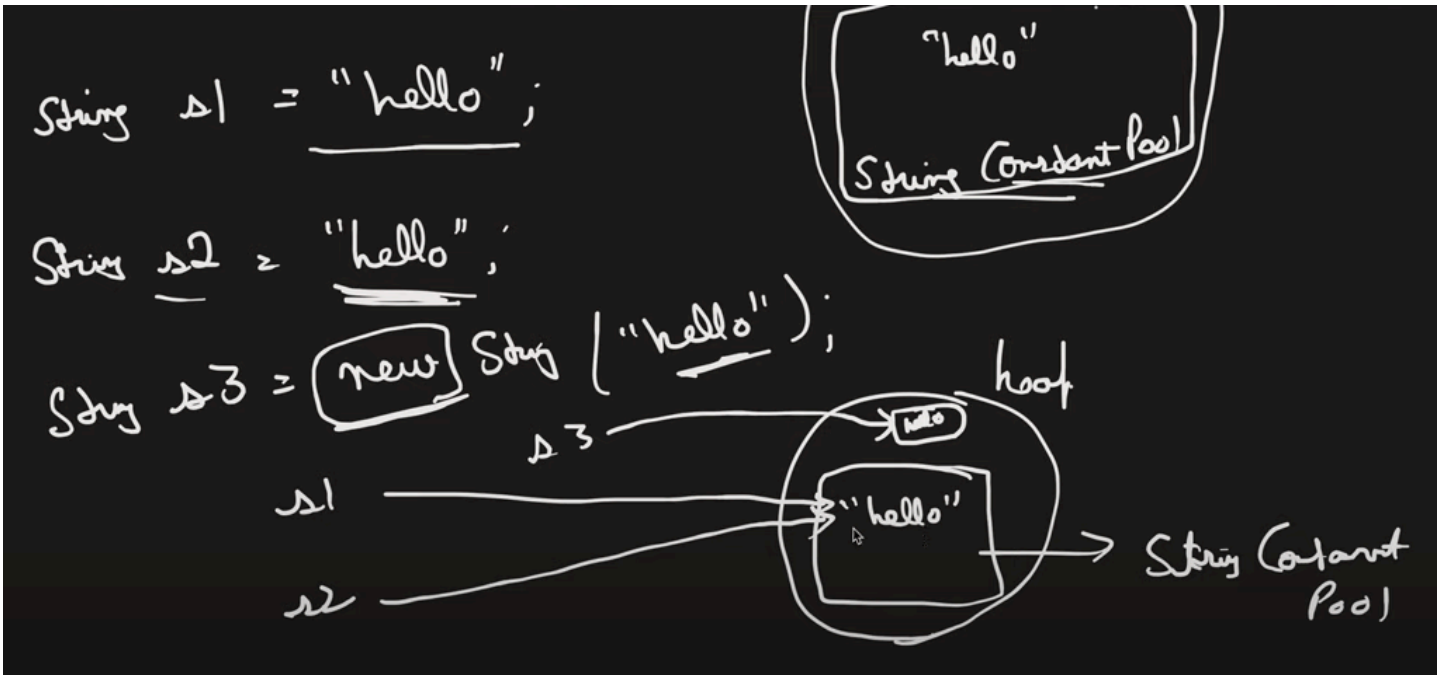
Reference Data Type ->

reference data types are types that store references (memory addresses) to objects rather than the actual data itself. Here are some examples of reference data types in Java:

Class: Classes in Java are reference data types that define blueprints for objects. Objects are instances of classes, and each object created from a class has its own set of attributes and behaviours defined by that class.

```
class MyClass {
 // Class definition
}

MyClass myObject = new MyClass(); // Creating an instance of the class
```

String: strings are instances of the String class immutable means it can not be change after creation whenever we reassign different value it will create new reference in heap . They serve as a widely used means to represent sequences of characters, offering a range of methods for string manipulation. Strings have a constant pool in the heap memory. If a string literal (enclosed in double quotes) is already present in the pool, creating a new string with the same literal will result in a reference to the existing string. However, if a string is created using the new String("hello") syntax, a new memory space will be allocated in the heap instead of referencing the existing "hello" literal.



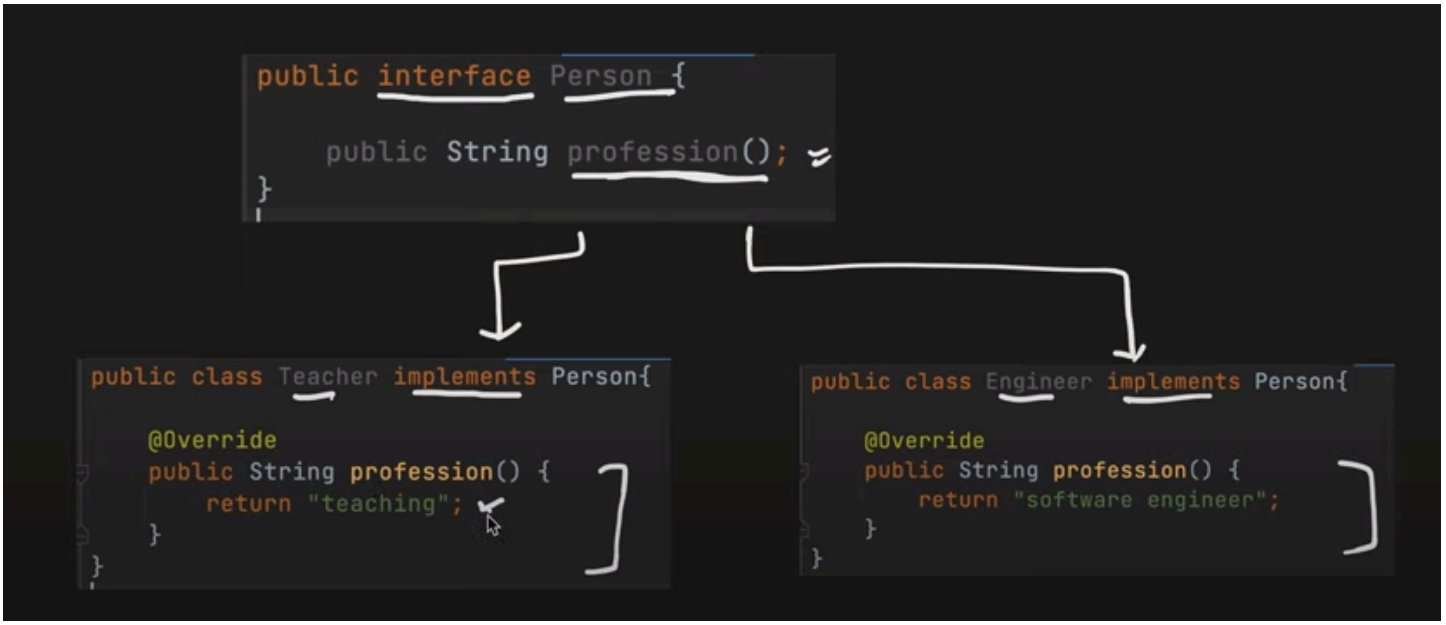
String myString = "Hello, Java!"; // Creating a String object

```
public class student {
 public static void main(String args[]) {
   String s1 = "hello";
   String s2 = "hello";
   System.out.println(s1==s2):
 }
}
```

result will be true because of same reference

Interface: Interfaces in Java are reference data types that define a set of abstract methods. Classes implement interfaces, providing concrete implementations for the methods defined in the interface. We can not create object of interface.

```
interface MyInterface {
 void myMethod();
}
class MyClass implements MyInterface {
 public void myMethod() {
     // Implementation of the method
 }
}
```



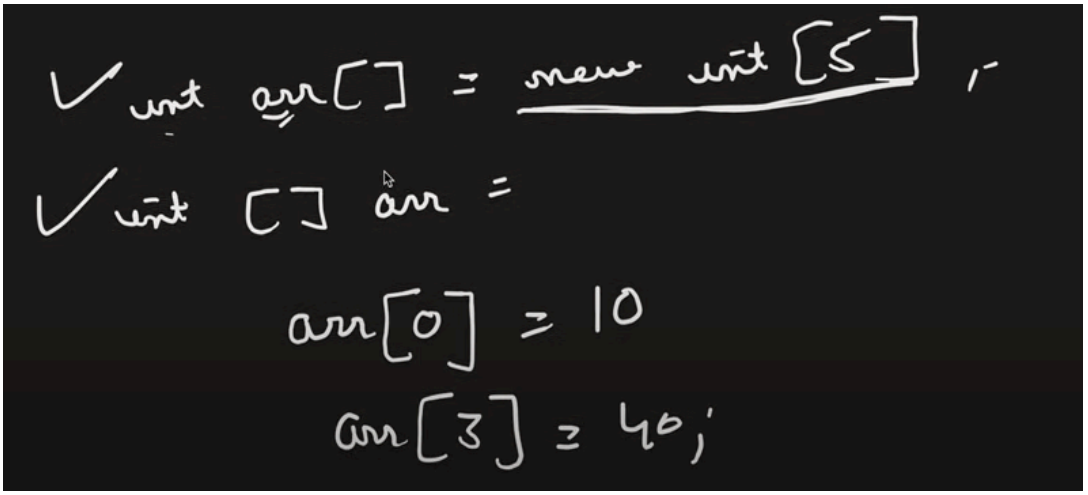

```
public class Student {
 public static void main(String args[]) {
 Person softwareEngineer = new Engineer();
 Person teacher = new Teacher();
 Teacher teacher1 = new Teacher();
 Engineer softwareEngineer1 = new Engineer();
 }
}
```

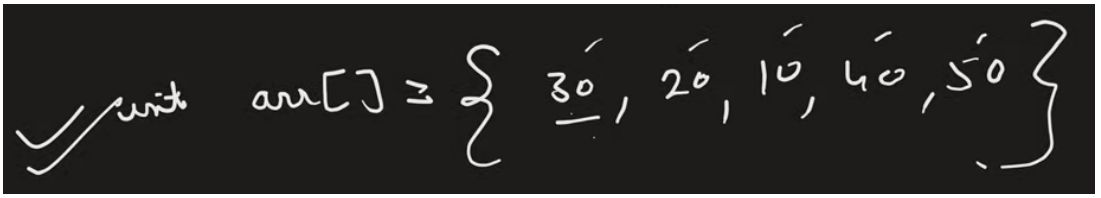
Array:

Arrays in Java are reference data types used to store a fixed-size sequential collection of elements of the same type.

Elements in an array can be accessed using an index.

int[] myArray = {1, 2, 3, 4, 5}; // Creating an array of integers

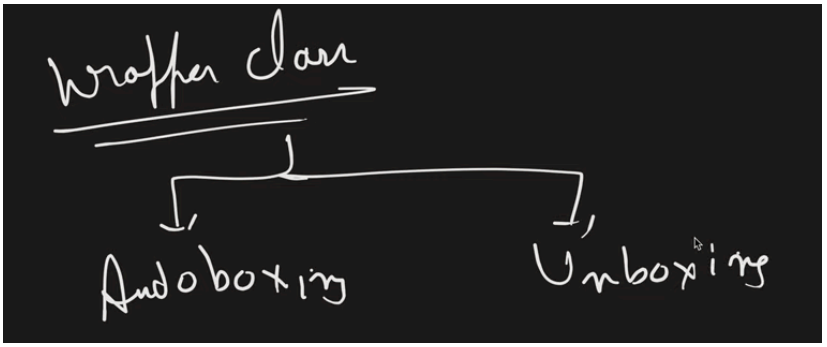




Wrapper Class -> There are 8 primitive types in Java, and for each of them, there is a corresponding reference type or wrapper class. Wrapper classes serve to treat primitive data types as objects, encapsulating their values and offering utility methods for diverse operations. For instance, the Integer class wraps the int primitive type, Character wraps char, and so forth.

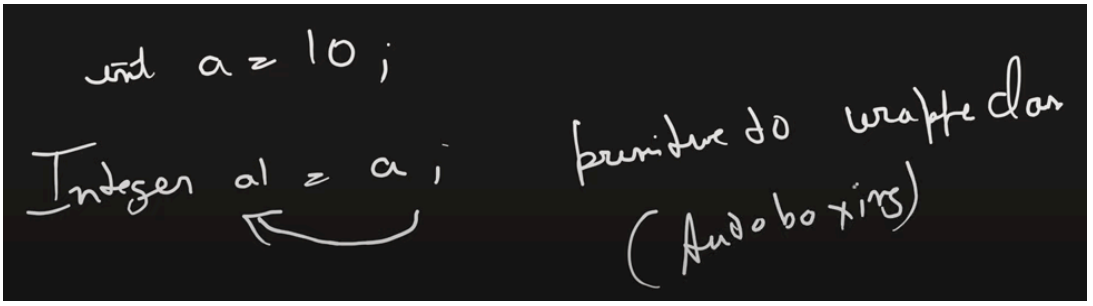
Wrapper classes are essential because they enable primitive data types to function as reference types. This becomes particularly crucial in scenarios like collections, where only reference types can be used, and primitive data types need to be wrapped in these classes to be utilised effectively within such contexts.

int - Wrapper class: **Integer**
char - Wrapper class: **Character**
short - Wrapper class: **Short**
byte - Wrapper class: **Byte**
long - Wrapper class: **Long**
float - Wrapper class: **Float**
double - Wrapper class: **Double**
boolean - Wrapper class: **Boolean**



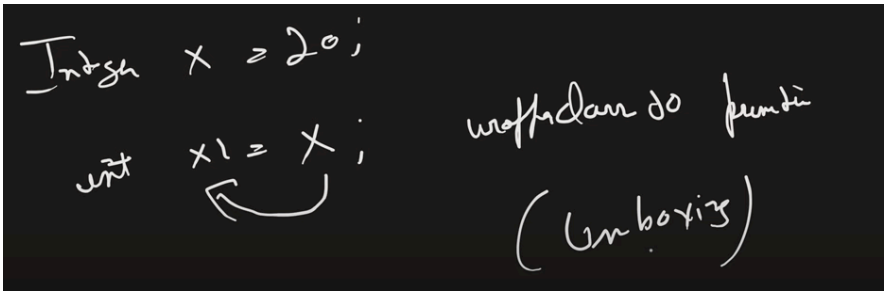
Autoboxing and unboxing are concepts in Java related to the automatic conversion between primitive data types and their corresponding wrapper classes. These features were introduced to simplify the handling of primitive types when working with classes that expect objects.

Autoboxing: Autoboxing is the process of automatically converting a primitive type to its corresponding wrapper class object.
For example, when you assign a primitive value to an object of the corresponding wrapper class, the Java compiler automatically converts the primitive type to the wrapper class.



int primitiveInt = 42;
Integer wrappedInt = primitiveInt; // Autoboxing
In this example, the int primitive is automatically converted (boxed) into an Integer object.

Unboxing: Unboxing is the process of automatically converting a wrapper class object to its corresponding primitive type.
For example, when you assign a wrapper class object to a primitive type, the Java compiler automatically extracts the primitive value from the wrapper object.



Integer wrappedInt = 42;
int primitiveInt = wrappedInt; // Unboxing
In this example, the Integer object is automatically converted (unboxed) to an int primitive.

Static final variable -> A static final constant variable in Java represents a variable that is both static and unmodifiable (constant). Here's a breakdown of each part:

Static: The static keyword in Java indicates that the variable belongs to the class rather than to instances of the class. This means there is only one copy of the variable shared among all instances of the class, and it can be accessed using the class name.

Final: The final keyword denotes that the variable's value cannot be changed after it is initialized. Once assigned a value, a final variable cannot be reassigned, making it a constant. Together, static final is commonly used to define class-level constants that remain unchanged throughout the program's execution. Here's an example:

```
public class Constants {
 // Static final constant variable
 public static final int MAX_VALUE = 100;
 public static void main(String[] args) {
     System.out.println(Constants.MAX_VALUE); // Accessing the constant using the
                                              class name
 }
}
```

In this example, MAX_VALUE is a static final constant variable, and it can be accessed using the class name Constants. Once assigned, its value cannot be modified throughout the program.

==========

Methods

==========

- What is Method
- Method declaration
    - Return type
    - Access Specifiers
    - Method Name and its convention
    - Parameters
    - Method Body
- Types of Method:
    - System Defined Method,
    - User Defined Method,
    - Overloaded Method,
    - Overridden Method,
    - Static Method,
    - Final Method,
    - Abstract Method
- Variable Arguments in Method

What is Method:

- Method is Used to perform certain tasks.
- It's a Collection of instructions that perform some specific task.
- It can be used to bring the code readability and re-usability.

```
public class Calculation {
 public int sum(int val1, int val2){
```

```
 int total = val1 + val2;
 //doing some logging stuff
 System.out.println("addition of val1 and val2 is:" + total);
 return total;
 }

 public int getPrice0fPen(){
 int capPrice = 2;
 int penBodyPrice = 5;
 int totalPenPrice = sum(caPrice penBodvPrice);
 }
}
```

Access Specifiers:

- **Public**: can be accessed through any class in any package.
- **Private**: can be accessed by methods only in the same class.
- **Protected**: can be access by other classes in same Package or other subclasses in different package.
- **Default**: if we do not mention anything, then the Default access specifier is used by Java. It can be only accessed by classes in the same package.

Return Type:

- Method do not return anything use "void"
- Use Class Name or primitive data types as return type of the method.

Method Name:

- It should be verb (some kind of action)
- Should start with small letter and follow camel case in case of multiple words

Parameters:

- It's a list of variables that will be used in the method.
- Parameter list can be blank too.

Method Body:

- Method body gets finished when you call "return" in mid.
- Get finished when you reach the end.
- You can also stop the method by "return" even for void return type.

System Defined Method:

Methods which are already defined and ready to use in Java like Math.sqrt()

User Defined Method:
Methods which a programmer creates based upon the program necessity.

Overloaded Method:
More than one method with the same name is created in the same class.

Overridden Method:
Subclass has the same method as the parent class.

Static Method:
- These methods are associated with the Class.
- Can be called just with class name.
- Static methods can not access Non Static Instance variables and methods.
- Static method can not be overridden.

When to declare method Static:
- Methods which do not modify the state of the object can be declared Static.
- Utility Method which do not use any instance variable and compute only on arguments.

Final Method:
- Final method can not be overridden in java.
- Why? the final method means its implementation can not be changed. If child class can not change its implementation then no use of overridden.

Abstract Method:
- It is defined only in Abstract classes.
- Only method declaration is done.
- Its implementation is done in Child classes.

VARIABLE ARGUMENTS (VARARGS):
- Variable number of inputs in the parameter.
- Only one variable argument can be present in the method.
- It should be the last argument in the list.

```
public static int sum(int ...variable) {
 int output = 0;
 for(int var : variable) {
   output = output + var;
 }
}
```

```
Calculation calculationObj = new Calculation();
calculationObj.sum(...variable:3);
calculationObj.sum(...variable:3,8,9,10);
calculationObj.sum(...variable:3,8,9,10,3,3,2,2,2,2,2,2);
```

==========

Getting Started with GitHub:

A Beginner's Guide to Collaborative Coding

Visit

https://youtu.be/i-a8xd4foLg

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

==========

==========

Java Memory Management and Garbage Collection

==========

- Type of Memory (Stack and Heap)
- What kind of data is Stored in Stack and Heap with Example
- Types of References
  - Strong Reference
  - Weak Reference
  - Soft Reference
- Heap Memory Structure
  - Young Generation
    - Eden
    - Survivor (S0 and S1)
  - Old Generation
  - Metaspace
- How Garbage Collector work & clean up the Heap memory with Example
- Types of Garbage Collector
  - Single GC
  - Parallel GC
  - CMS (Concurrent Mark and Sweep)
  - G1 GC

2 Types of Memory:

- STACK
- HEAP

Both Stack and Heap are created by JVM and stored in RAM.

Stack Memory:

- Store Temporary Variables and separate memory block for methods.

- Store Primitive data types -

- Store Reference of the heap objects

- Strong reference
- Weak reference
  - Soft reference

- Each thread has its own Stack memory.

- Variables within a SCOPE is only visible and as soon as any variable goes out of the SCOPE, it get deleted from the Stack (in LIFO order)

- When Stack memory goes full, its throws "java.lang.StackOverflowError"

Store Reference of the heap objects

Strong Reference:

- A strong reference is the standard reference that we use in most programming languages.
- When an object has at least one strong reference pointing to it, the garbage collector will not reclaim the memory occupied by that object.
- As long as there is a strong reference to an object, it will not be eligible for garbage collection.

Object obj = new Object(); // Strong reference

In this example, obj is a strong reference to the newly created object.

Weak Reference:

- A weak reference is a type of reference that does not prevent the referenced object from being garbage collected.
- If an object has only weak references pointing to it, it is considered eligible for garbage collection.
- Weak references are useful for scenarios where you want to have access to an object but don't want to prevent it from being collected if there are no strong references.

WeakReference<Object> weakRef = new WeakReference<>(new Object());

In this example, weakRef is a weak reference to the object. If there are no strong references to the object, it can be garbage collected.

Soft Reference:

- A soft reference, like a weak reference, does not prevent the referenced object from being garbage collected.
- However, a soft reference is only cleared when the garbage collector determines that it is necessary to free up memory.
- Soft references are commonly used in scenarios where you want to keep an object in memory unless memory pressure occurs.

SoftReference<Object> softRef = new SoftReference<>(new Object());

In this example, softRef is a soft reference to the object. It will be cleared only when the garbage collector decides that there is a need to free up memory.

In summary, strong references prevent objects from being garbage collected, weak references allow objects to be collected if there are no strong references, and soft references provide a middle ground by allowing collection under memory pressure.
In Java memory management, specifically in the Java Virtual Machine (JVM), there are commonly referred to as two primary memory spaces: the **heap** and the **Metaspace**.
Heap: The heap is a region of memory where Java objects are allocated and managed by the garbage collector.

- It is divided into two main areas: the Young Generation (Eden, S0, S1) and the Old Generation (Tenured Generation).
- The heap is where the actual objects, including their instance variables, are stored.

Heap Memory:

- Store Objects.
- There is no order of allocating the memory.
- Garbage Collector is used to delete the unreferenced objects from the heap.
  - Mark and Sweep Algorithm
  - Types of GC:
    - Single GC
    - Parallel GC
    - CMS (concurrent Mark Sweep)
    - G1
- Heap memory is shared with all the threads.
- Heap also contains the String Pool.
- When Heap memory goes full, its throws "java.lang.OutofMemoryError"
- Heap memory is further divided into:
  - Young Generation (minor GC happens here)
    - Eden
    - Survivor
- Old/Tenured Generation (major GC happens here)
- Permanent Generation

Garbage Collector -> The JVM automatically triggers the garbage collector as needed, often periodically when there is available memory space or quickly if the heap memory is filling rapidly. The exact timing depends on the JVM's internal mechanisms. The garbage collector is responsible for eliminating unreferenced objects from the heap, ensuring efficient memory management.

In the context of garbage collection in the Java Virtual Machine (JVM), the heap is generally divided into three main areas:

Young Generation:

- This is where new objects are initially created.
- It consists of three spaces: Eden and two survivor spaces (often denoted as S0 and S1).
- Objects are first allocated in the Eden space.

Eden: The Eden space is where newly created objects reside initially.
When garbage collection occurs, unreferenced objects in Eden are collected.

Survivor Spaces (S0 and S1):

- Survivor spaces are used to store objects that survive one or more garbage collection cycles.
- Objects that survive the garbage collection in Eden are moved to one of the survivor spaces.
- The survivor spaces are used alternatively for each garbage collection cycle.

Old Generation (Tenured Generation):

- Objects that have survived multiple garbage collection cycles in the young generation are eventually promoted to the old generation.
- Long-lived objects reside in the old generation.
- Garbage collection in the old generation is less frequent compared to the young generation.

In summary, the heap in the JVM typically has a Young Generation (Eden, S0, S1) for newly created objects and an Old Generation for longer-lived objects. The survivor spaces help manage the transition of objects from the young to the old generation.

The garbage collector process involves using a "**mark and sweep**" algorithm. In the young stack memory, It has 2 phase

Marking Phase:

- First, it marks unreferenced objects in both Eden and space 0.
- Objects that are no longer needed are identified for removal.

Sweeping Phase:

- Then, it deletes the marked unreferenced objects, freeing up space.
- The remaining objects in space 0 have their ages incremented (from 0 to 1).

Moving to the Next Space:
- New objects are created in Eden.
- When the garbage collector returns, it marks objects in both Eden and space 0 again.
- Unreferenced objects are removed, and the surviving objects move to the next space, space 1.
- Objects coming from space 0 have their ages marked as 2, and those from Eden are marked as 1.
- Eden and space 0 are then freed up for new objects.

This process helps manage memory efficiently by identifying and removing objects that are no longer in use, making room for new objects in a controlled manner.

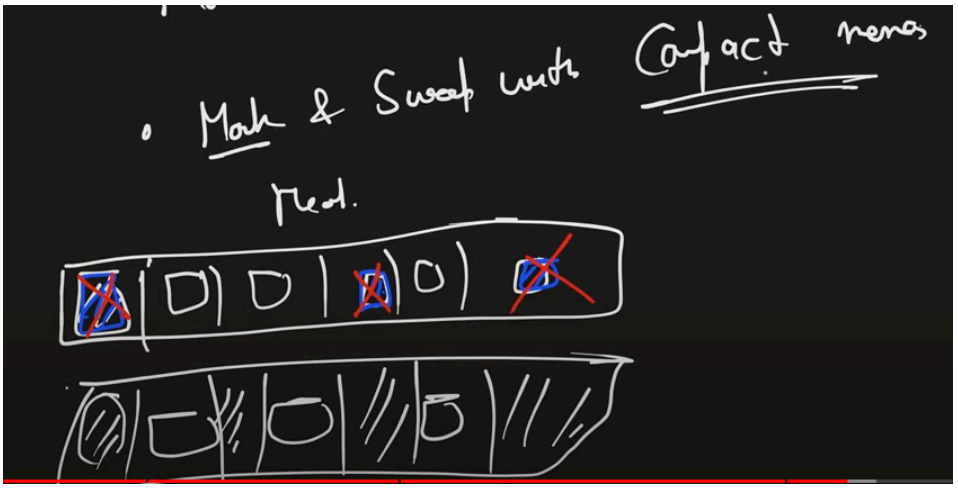
In Java, the term "Metaspace" refers to the memory area where metadata related to class structures and methods is stored. Metaspace is introduced as a replacement for the traditional "Permanent Generation" (PermGen) in Java's memory model.
Here are some key points about Metaspace:

Metadata Storage:
- Metaspace is used to store metadata related to class definitions, method information, and other runtime-level information.
- In contrast to PermGen, Metaspace dynamically adjusts its size based on the application's needs.

Dynamic Size: Unlike the fixed-size PermGen, Metaspace can grow or shrink dynamically, which helps prevent issues related to memory constraints.

Preventing PermGen Out-of-Memory Errors: One of the reasons Metaspace was introduced was to address Out-of-Memory errors related to the fixed-size PermGen in older JVM versions before java 8. It was part of the heap.

Mark & sweep with compact Algo -> Mark and sweep is the same as above After marking and sweeping, there may be gaps or fragmentation in the heap due to the removal of unreachable objects. The compact phase aims to reduce fragmentation by compacting the remaining live objects, moving them closer together in memory.



There are several different types of Garbage Collector ->

Serial Garbage Collector (SerialGC): It's a simple and single-threaded garbage collector suitable for small applications or environments with low memory requirements.
Usage: Often used for client-side applications or on machines with small heaps.

Parallel Garbage Collector (ParallelGC) (Default in java 8): Also known as the throughput collector, it uses multiple threads to perform garbage collection in the Young Generation. It is designed to improve throughput by utilising multiple processors.
Usage: Suited for applications where high throughput is a priority and pauses are acceptable.

Concurrent Mark-Sweep Garbage Collector (CMS GC): Also known as the CMS collector, it focuses on minimising pause times by doing most of the garbage collection work concurrently with the application threads.
Usage: Suitable for applications with a large heap and where low-latency is a critical requirement.

G1 Garbage Collector (G1 GC): The Garbage-First collector is designed to provide high throughput and low-latency by dividing the heap into regions and using a combination of parallel and concurrent approaches for garbage collection.
Usage: Suitable for large heaps, especially in server environments, with the goal of minimising both short and long-term garbage collection pauses.

Z Garbage Collector (ZGC): Introduced in JDK 11, ZGC is designed to provide low-latency performance benefits by minimising pause times, even for very large heaps.
Usage: Suitable for applications where low-latency is critical, and there is a need to scale to very large heap sizes.

Shenandoah Garbage Collector (Shenandoah GC): Also introduced in JDK 11, Shenandoah is designed to provide low-latency garbage collection for applications with large heaps.
Usage: Suited for applications that require low-latency and can benefit from concurrent garbage collection on very large heaps.

==========

Mastering GitHub and Git: A Comprehensive Tutorial for All Skill Levels

Visit

https://youtu.be/AkqOtUig5p4

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

==========

==========

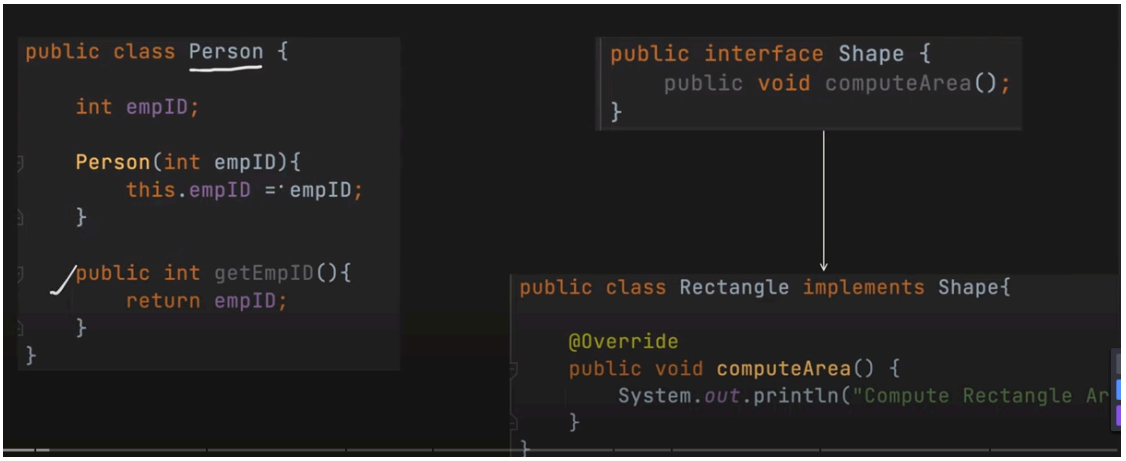
Type of Classes

==========

- **Concrete Class**
- **Abstract Class**
- **SuperClass and SubClass**
- **Object Class**
- **Nested Class**
  - Inner Class (Non Static Nested Class)
  - Anonymous Inner Class
  - Member Inner Class
  - Local Inner Class
  - Static Nested Class / Static Class
- **Generic Class**
- **POJO Class**
- **Enum Class**
- **Final Class**
- **Singleton Class**
- **Immutable Class**
- **Wrapper Class**

Concrete Class ->
concrete classes are classes that can be instantiated, meaning you can create objects of those classes. These classes can have fields, methods, and can provide a complete implementation for all the methods declared in their interfaces or inherited from their parent classes.

Concrete Class:
- These are those classes where we can **create an instance using the NEW keyword**.
- All the methods in this class have implementations.
- It can also be your child class from interface or extend abstract class.
- A class access modifier can be "public" or "package private" (no explicit modifier defined)



We can observe that the left side represents a concrete class, while the right side represents an interface, which is not concrete and cannot be instantiated on its own.
Abstract Class ->
An abstract class in Java cannot be directly used to create objects, but we can use its reference in other classes. We can use this reference as a type or to call static methods. If a subclass extends an abstract class, it needs to either implement all the abstract methods from the parent class or itself be declared as abstract. It's not mandatory for an abstract class to have at least one abstract method. Abstract classes are often used to prevent the creation of instances and to provide common functionality that can be shared among subclasses

Abstract Class (0 to 100% Abstraction):
Show only important features to users and hide its internal implementation.
2 ways to achieve abstraction:
- Class is declared as abstracted through keyword "abstract"
- It can have both abstract(method without body) and non abstract methods.
- We can not create an instance of this class.
- We parent has some features which all child classes have in common, then this can be used.
- Constructors can be created inside them. And with super keyword from child classes we

can access them.

```
public abstract class Car {
 int mileage;
  Car(int mileage){
   this mileage = mileage;
  }
  public abstract void pressBreak();
  public abstract void pressClutch();
  public int getNumberOfWheels(){
  return 4;
 }
}
```

Superclass ->

- A superclass, also known as a parent class or base class, is a class that is extended or inherited by another class.
- It provides a common set of attributes and methods that can be shared by its subclasses.
- Superclasses are more general and abstract, defining common behaviour that can be inherited by more specialised classes.
- If it is not extended by any class it extends the default object class. In Java all classes have 1 object super class and all classes of subclasses of object class.

Subclass ->

- A subclass, also known as a child class or derived class, is a class that extends or inherits from another class (the superclass).
- It inherits the attributes and methods of the superclass and can provide additional or more specialised functionality.
- Subclasses are more specific and concrete, representing a more specialised type of object.

Super and Subclass:

- A class that is derived from another class is called a Subclass.
- And from the class through which Subclass is derived its called Superclass.
- In Java, in the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- Object is the topmost class in Java.

It has some common methods like clone(), toString(), equals(), notify(), wait() etc...

```
public class ObjectTest {
```

```
  public static void main (String[] args){
  ObjectTest obj = new ObjectTest();
  Object obj1 = new Person ( empID: 1);
  Object obj2 = new Audi ( mileage: 10);
  System.out.println(obj1.getClass());
  System.out.println(obj2.getClass());
 }
}
```

Nested Class -> In Java, a nested class is a class that is defined within another class. There are several types of nested classes in Java

Nested Class:

Class within another class is called Nested Class.

When to use?

If you know that, a class(A) will be used by only one another class(B), then instead of created a new file(A.java) for it, we can create a nested class inside B class itself.
And Also help us to group logically related classes in onè file.

Scope:

Its Scope is the same as of its Outer class.

It is of 2 types:

- Static Nested Class
- Non Static Nested Class
  - Member Inner Class
  - Local Inner Class
  - Anonymous inner Class

Static Nested Class:

- A static nested class is defined as a static member of its enclosing class.
- It can be instantiated without creating an instance of the outer class.
- It cannot directly access non-static members of the outer class.
- It can be private, public, protected or package-private (default, no explicit declaration)

```
class OuterClass {
  int instanceVariable = 10;
  static int classVariable = 20;
  static class NestedClass {
    public void print(){
      System.out.println(classVariable + instanceVariable)
```

```
    }
  }
}
```

The instance variable mentioned above is non-static, and attempting to access it within a static method results in a compilation error, indicated by a red.

```
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass.NestedClass nestedObj = new OuterClass.NestedClass();
        nestedObj.print();
    }
```

We can apply various modifiers to inner classes and create objects by accessing them through the main class, as shown earlier. However, if we declare the inner class as private, we won't be able to access or create its objects directly outside of the classes.

None Static Nested Class ->

Inner Class or Non Static Nested Class:

- It has access to all the instance variable and method of outer class.
- Its object can be initiated after initiating the object of the outer class.

1. Member Inner Class:
   - its can be private, public, protected, default

```
class OuterClass {

    int instanceVariable = 10;
    static int classVariable = 20;

    class InnerClass{

        public void print(){
            System.out.println(classVariable + instanceVariable);
        }
    }
}
```

```
public class ObjectTest {

    public static void main(String args[]) {

        OuterClass outerClassObj = new OuterClass();

        OuterClass.InnerClass innerClassObj = outerClassObj.new InnerClass();
        innerClassObj.print();

    }
}
```

Note -> The distinction between static and non-static inner classes lies in how you create objects outside the class. For a non-static inner class, you need to create an object of the main class first and then create an object of the inner class using that main class object. On the other hand, for a static inner class, you can create its object directly without needing an instance of the main class by using the class name.

Local Inner Class -> a local inner class is a class that is defined within a block of code, such as a method, constructor, or even a statement block. The scope of a local inner class is limited to the block in which it is defined. Local inner classes have access to the members of the enclosing class and can also access local variables of the method or block in which they are declared.

a local inner class has access to both instance variables and static variables of the enclosing class. However, it's important to note that if the local inner class is within a static method, it cannot access non-static (instance) variables directly. This is because static methods are associated with the class itself and not with any particular instance.

2. Local Inner Class:

- These are those classes which are defined in any block like for loop, while loop block, If condition block, method etc.
- It can not be declared as private, protected, or public. Only default(not defined explicit) access modifier is used.
- It can not be initiated outside of this block.

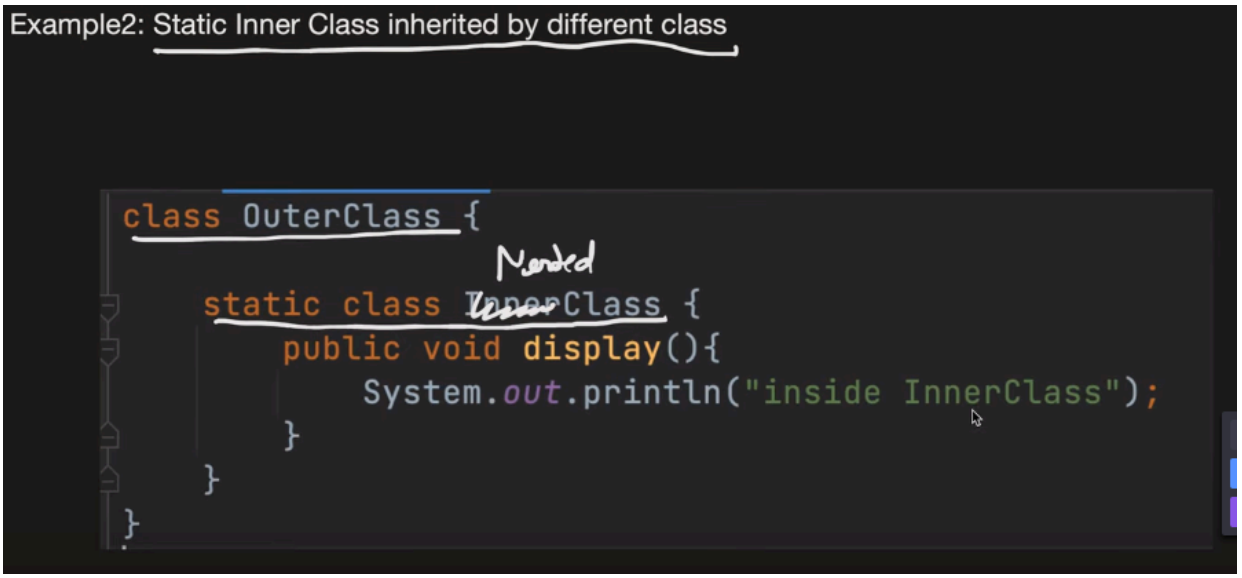
```
class OuterClass {
   int instanceVariable = 1;
   static int classVariable = 2;
   public void display(){
     int methodLocalVariable = 3;
     class LocalInnerClass{
       int localInnerVariable = 4;
       public void print(){
           System.out.println(instanceVariable + classvariable + methodLocalVariable
                              + localLInnerVariable);
       }
     }

     LocalInnerclass localObj = new LocalInnerClass();
     LocalObj.print();
   }
}
```

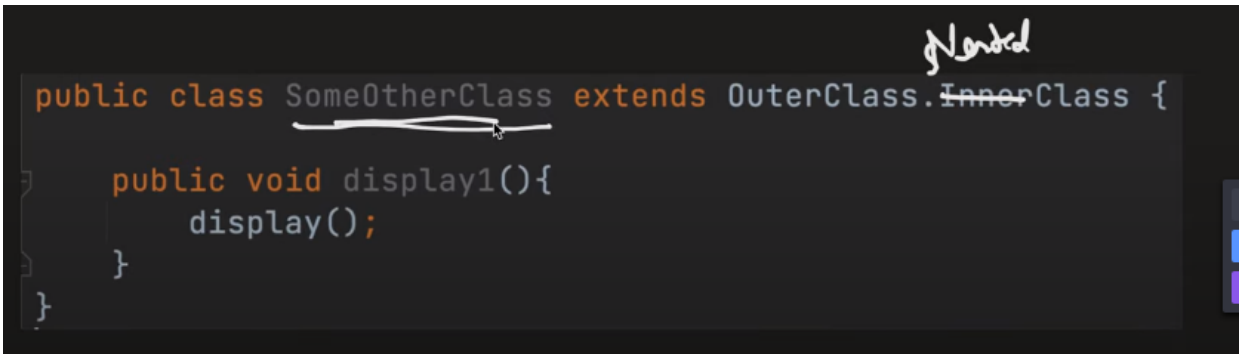
Inheritance in Nested Classes

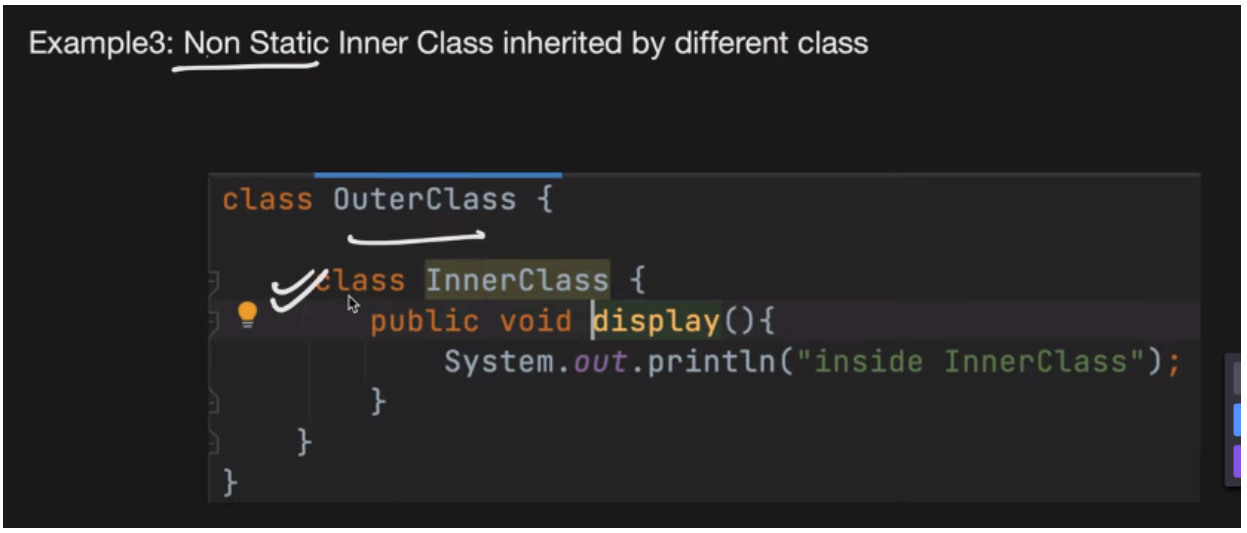
Example: one inner class inherit another inner class in the same outer class.

```
class OuterClass {
   int instanceVariable = 1;
   static int classVariable = 2;
   class InnerClass1 {
       int innerClass1 = 3;
   }
   class InnerClass2 extends InnerClass1{
       int innerClass2 = 4;
       void display(){
       System.out.println(innerClass1 + innerClass2 + instanceVariable +
       classVariable);
     }
   }
}
```

```
public class ObjectTest {
 public static void main (String args[]) {
   OuterClass outerClassObj new OuterClass();
   OuterClass.InnerClass2 innerClass2Obj = outerClassObj.new InnerClass2(),
   innerClass2Obj.display();
 }
}
```








```
public class SometherClass extends OuterClass.InnerClass {
  SomeOtherClass(){
    new OuterClass().super();
    //as you know, when child class constructor invoked, it first invoked the constructor of parent.
    //but here the parent is Inner class, so it can only be accessed by the object of outerclass only.
  }
  public void display(){
    display();
  }
}
```

Anonymous Inner Class: An anonymous inner class, a subtype of local inner classes, lacks a specific name. Its purpose is often for brief, one-time implementations of interfaces or extensions of classes. If there's a need to implement a method from an abstract class without creating a new class, an anonymous class serves this purpose efficiently.

Anonymous Inner Class:

An inner class without a name called Anonymous class.

Why its used:

- When we want to override the behaviour of the method without even creating any subclass.

```
public abstract class Car {
```

```
  public abstract void pressBreak();
}
```

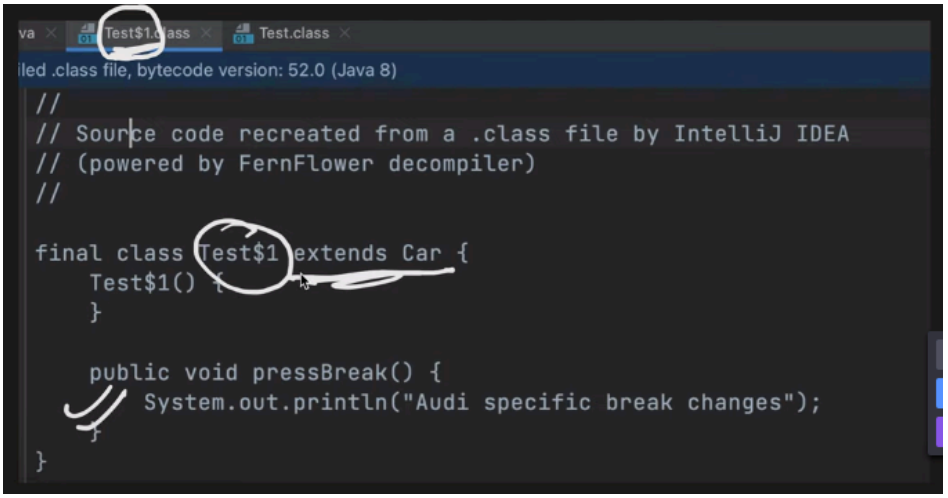
```
public class Test {
 public static void main (String args[]) {
   Car audiCarObj = new Car() {
     @Override
     public void pressBreak() {
       //my audi specific implementation here
       System.out.println("Audi specific break changes");
     }
   }
 }
}
```

2 things happened behind the scene:

- Subclass is created, name decided by the compiler
- Creates an object of subclass and assign its reference to object "audiCarObj"

Similarly for the interface also, it works in the same way...

As we can see in the below screen for **Anonymous class** compiler created a Test$1.class



==========

Generic Class ->

==========

- What is Generic Class and what is the need of it?
- How to define the Generic Class
- Inheritance with Generic Classes
  - Non Generic Subclass
  - Generic Subclass
- Multiple Type Parameters in Generic Class
- Generic Methods
- What is Raw Type
- Bounded Generics

    - Upper Bound
    - Multi Bound
- Wildcards
  - UpperBound
  - LowerBound
  - Unbounded
- Type Erasure

What is Generic Class and what is the need of it? -> A generic class in Java is a class that can work with different types (classes and interfaces). It allows you to define a class with placeholders for the data types of its fields, methods, or parameters. These placeholders, known as type parameters, are specified when the generic class is instantiated.

The primary goal of generic classes is to create classes, methods, or interfaces that can work with any data type without sacrificing type safety. Before the introduction of generics in Java 5, collections like ArrayList or HashMap could only store objects of type Object, leading to potential runtime errors and the need for explicit type casting.

```java
public class Print {

    Object value;

    public Object getPrintValue(){
        return value;
    }

    public void setPrintValue(Object value){
        this.value = value;
    }
}
```

```java
public class Main {

    public static void main(String args[]) {

        Print printObj1 = new Print();
        printObj1.setPrintValue(1);
        Object printValue = printObj1.getPrintValue();
        //we can not use printValue directly, we have to typecast it, else it will be compile time erro
        if((int)printValue == 1){
            //do something
        }
    }
}
```

We observe that since all classes extend the Object class, an object reference can be used to hold a reference to any class. However, this approach has its drawbacks. Initially, we need to

perform type casting, which is safeguarded by compile-time errors. However, during runtime, errors may occur if unexpected data is sent, leading to a loss of type control.

Use Generic in the correct way -> In the screenshot below, we've replaced Object with <T> in our class. This <T> serves as a placeholder for the type that we pass from the main class on the second screen. This gives us control over the type, allowing us to catch type-related errors. Additionally, we can use this class with multiple different types by specifying a different type when creating an object. We can pass as <T> anything apart from the primitives class.

Make use of Generic Classes....

```
public class Print<T> {
 T value;
 public T getPrintValue(){
   return value;
 }

 public void setPrintValue(T value){
   this.value = value;
 }
}
```

```
public class Main {

    public static void main(String args[]) {

        Print<Integer> printObj1 = new Print<Integer>();
        printObj1.setPrintValue(1);
        Integer printValue = printObj1.getPrintValue();
        if(printValue == 1){
            //do something
        }
    }
}
```

1. Non Generic Subclass

```
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value = value;
    }
}
```

```
public class ColorPrint extends Print<String>{

}
```

Generic Child class

```
public class Print<T> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value = value;
    }
}
```

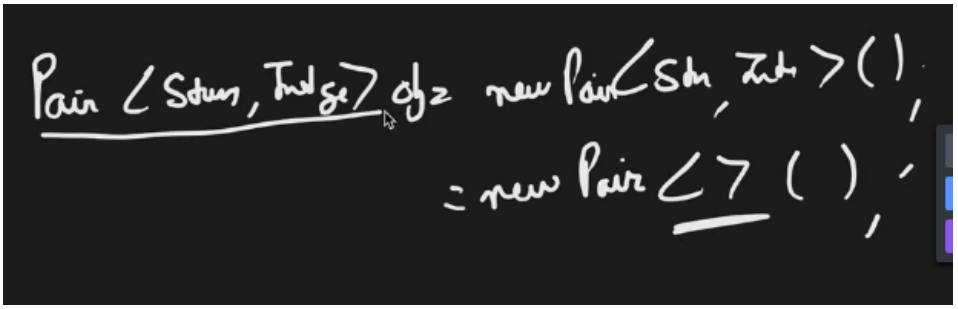
```
public class ColorPrint<T> extends Print<T>{

}
```

```
public class Main {

    public static void main(String args[]) {

        ColorPrint<String> colorPrintObj = new ColorPrint<>();
        colorPrintObj.setPrintValue("2");
    }
}
```

Multiple generic type

```java
public class Pair<K,V> {

    private K key;
    private V value;

    public void put(K key, V value){
        this.key = key;
        this.value = value;
    }
}
```

```java
public class Main {

    public static void main(String args[]) {

        Pair<String, Integer> pairObj = new Pair<>();
        pairObj.put("hello", 1243);

    }
}
```



Both ways we can pass types.

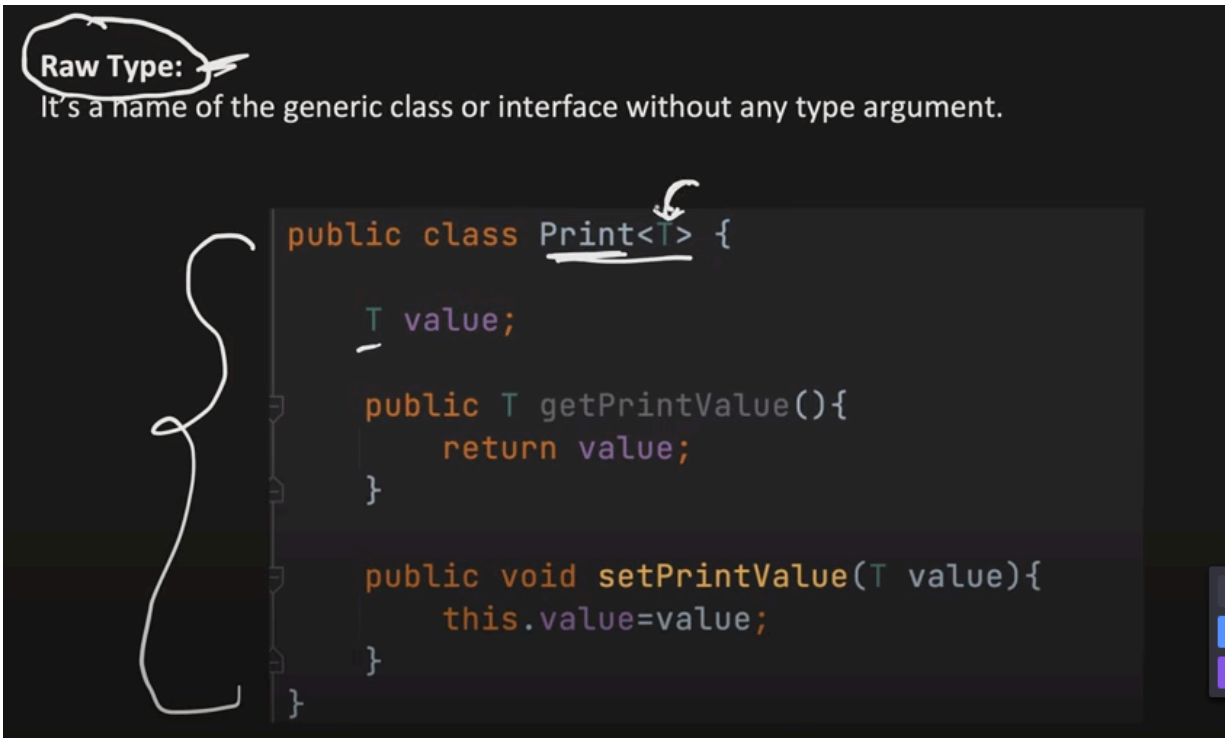
Generic Method -> a generic method is a method that can work with different types. It allows you to define a method with type parameters, making it versatile and capable of handling various data types. The type parameters are specified in angle brackets (<>) before the return type of the method.

Generic Method:

What if we only wants to make Method Generic, not the complete Class, we can write Generic Methods too.

- Type Parameter should be before the return type of the method declaration.
- Type Parameter scope is limited to method only.

```java
public class GenericMethod {

    public <K,V> void printValue(Pair<K,V> pair1, Pair<K,V> pair2 ){

        if(pair2.getKey().equals(pair2.getKey())){
            //do something
        }
    }
}
```




```java
public class Main {

    public static void main(String args[]) {

        Print<String> parametrizedTypePrintObject = new Print<>();

        //internally it passes Object as parametrized type.
        Print rawTypePrintObject = new Print();
         rawTypePrintObject.setPrintValue(1);
         rawTypePrintObject.setPrintValue("hello");
    }
}
```

In the second example, we notice that we are not explicitly passing a generic type to Print. This is referred to as a raw type, where Java internally treats it as an Object type. However, this approach raises concerns about type safety. Java employs this mechanism for backward compatibility.

Bounded Generics:
It can be used at generic class and method

- Upper Bound (<T extends Number>) means T can be of type Number or its Subclass only. Here superclass (in this example Number) we can have interface too.
- Multi Bound

Bounded Generics:

It can be used at generic class and method
- Upper Bound (<T extends Number>) means T can be of type Number or its Subclass only. Here superclass (in this example Number) we can have an interface too.
- Multi Bound

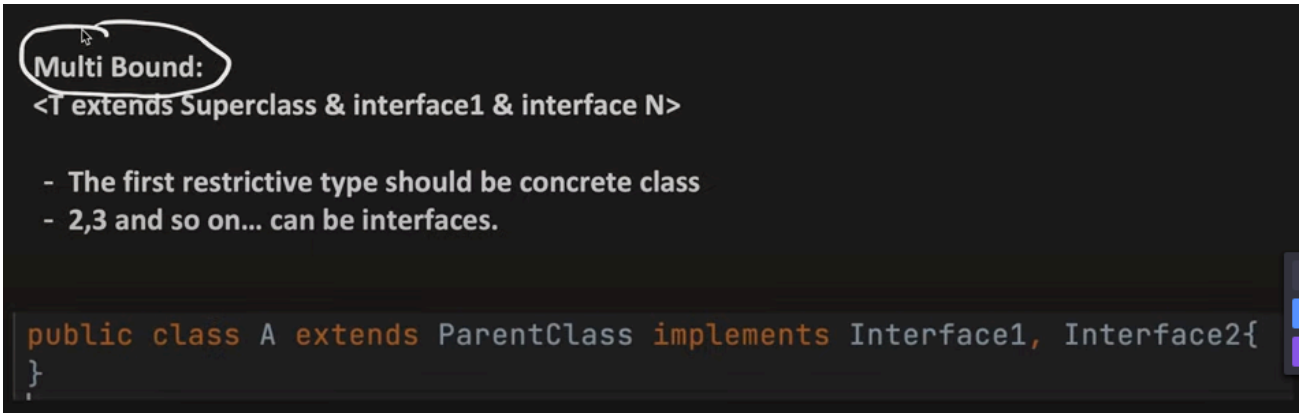
It is used to restrict the user to send specific types like numbers only now he is allowed to define type integer, double only not string.

As below we can pass only child class of Number and Number it self

```
public class Print<T extends Number> {

    T value;

    public T getPrintValue(){
        return value;
    }

    public void setPrintValue(T value){
        this.value=value;
    }
}
```

```
public class Main {

    public static void main(String args[]) {

        Print<Integer> parametrizedTypePrintObject = new Print<Integer>();
    }
}
```

```
public class Main {

    public static void main(String args[]) {

        Print<String> parametrizedTypePrintObject = new Print<>();
    }
}
```




```
public class Print<T extends ParentClass & Interface1 & Interface2> {
   T value;
   public T getPrintValue() { return value; }
   public void setPrintValue(T value) { this.value = value; }
}
```

Like above ParentClass is a concrete class rest is interfaces and it is required to implement all interfaces in a concrete class.

Wildcards -> a wildcard is a special character (?) that represents an unknown type. Wildcards are often used in situations where the exact type doesn't matter, and you want more flexibility in working with different types.

Wildcards:

- Upper bounded wildcard : <? extends UpperBoundClassName> i.e. class Name and below
- Lower bounded wildcard: <? Super LowerBoundClassName> i.e. class Name and above
- Unbounded wildcard <?> only you can read

There are two types of wildcards: the upper-bounded wildcard (<? extends T>) and the lower-bounded wildcard (<? super T>).

Upper-bounded wildcard (<? extends T>):

- Denoted by <? extends T>, where T is a specific type or a wildcard itself.
- It allows the wildcard to represent any type that is a subtype of T.
- It provides a way to specify that the unknown type must be of a certain type or a subtype of that type.

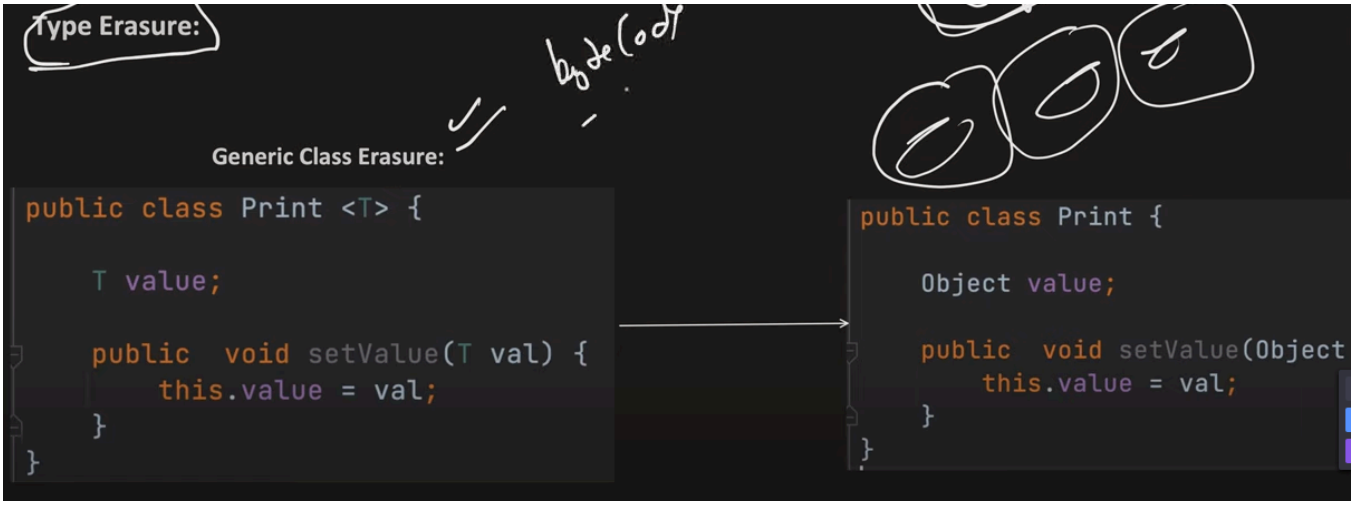
```
public void processList(List<? extends Number> list) {
 // Can read elements from the list (upper-bounded)
 for (Number number : list) {
     System.out.println(number);
 }
}
```

Lower-bounded wildcard (<? super T>):

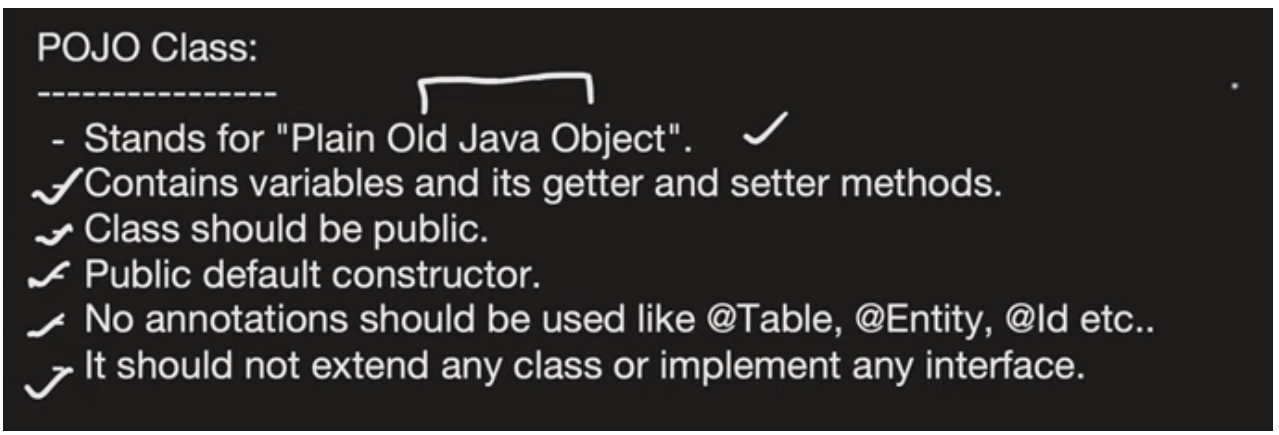
- Denoted by <? super T>, where T is a specific type or a wildcard itself.
- It allows the wildcard to represent any type that is a supertype of T. It means given type and acentor of that class.
- It provides a way to specify that the unknown type must be of a certain type or a supertype of that type.

```
public void addElement(List<? super Integer> list) {
 // Can add elements to the list (lower-bounded)
 list.add(42);
}
```

Unbounded wildcard (<?>):
an unbounded wildcard is represented by the question mark (?) without any type constraints. The unbounded wildcard can be used when you want to work with generic types but it doesn't have a strong preference about the actual type. It will behave the same as the object type.
Type erasure -> It is a concept where, during the conversion of generic types to bytecode, the specific type information is erased and replaced with Object. This is done for backward compatibility reasons.



Pojo Classes ->



ENUM -> an enum (short for enumeration) is a special data type that represents a fixed set of constants. making it easier to work with constants in code. It is also a class but You declare it using the keyword "enum" instead of "class." Enums can have properties, a private constructor preventing direct instantiation, and they also have **methods like values,ordinal,valueOf, name.** Enums are useful for improving code readability and organisation.

ENUM Class:

- It has a collection of CONSTANTS (variables which values can not be changed)
- Its CONSTANTS are static and final implicitly (we do not have to write it).
- It can not extend any class, as it internally extends java.lang.Enum class
- It can implement interfaces.
- It can have variables, constructor, and methods.
- It can not be initiated (as its constructor will be private only, even you give default, in bytecode it make it private)
- No other class can extend Enum class
- It can have an abstract method, and all the constants should implement that abstract method.

Why is it not possible to extend an ENUM class in Java?
The reason an Enum class cannot extend another class is because, internally (implicitly), it already extends the java.lang.Enum class. In Java, a class is limited to extending only one class, and since Enums inherently extend **java.lang.Enum**, they cannot further extend another class. This restriction is in line with Java's single inheritance principle, where a class can have only one direct superclass.

Normal ENUM Class

```
public enum EnumSample{

    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;
}
```

Note: Add the semicolon in end rest separate bia comma.

```
public class Main {
 public static void main(String args[]) {
   /*Common functions which is used
   - values()
   - Ordinal()
   - value0f()
   - name()
   */

   //1. usage of Values() and ordinal()
   for (EnumSample sample : EnumSample.values()){
     System.out.println(sample.ordinat());
   }

   //2. usage of Valueûf() and name()
   EnumSample enumVariable = EnumSample.valueOf( name:"FRIDAY");
   System.out.println(enumVariable.name());
 }
}
```

Enum Methods: Enums in Java come with default methods inherited from the java.lang.Enum class.

values():

- Returns an array containing all the constants of the enum.
- Example: EnumSample.values() returns an array with all the enum constants.

ordinal():

- Retrieves the position of an enum constant within its enum type.
- If not explicitly assigned, ordinal values are assigned internally starting from 0.
- Example: EnumSample.SUNDAY.ordinal() returns 0, EnumSample.MONDAY.ordinal() returns 1, and so on.

valueOf():

- Returns the enum constant with the specified name.
- Example: EnumSample.valueOf("FRIDAY") returns the EnumSample object for the constant "FRIDAY."

name():

- Obtains the name of the enum constant.
- Example: enumVariable.name() returns the name of the enum constant, such as "FRIDAY" for the EnumSample variable.

These methods provide convenient ways to work with enums, retrieve their values, and perform operations based on their positions or names within the enum type.

Enum with custom values:

```
public public enum EnumSample {
   MONDAY(val: 101, comment: "1st day of the week"),
   TUESDAY( val: 102, comment: "2nd day of the week"),
   WEDNESDAY (val: 103, comment: "3rd day of the week"),
   THURSDAY( val: 104, comment: "4th day of the week"),
   FRIDAY( val: 105, comment: "5th day of the week"),
   SATURDAY( val: 106, comment: "its 1st WeekOff"),
   SUNDAY ( val: 107, comment: "its 2nd WeekOff"):
   private int val;
   private String comment;

   EnumSample(int val, String comment) {
     this.val = val;
     this.comment = comment;
   }
   public int getVal {
     return val;
   }
}
```

Keep in mind whenever we will define an field or any method in an enum it belongs to every enum as we can see above because every Enum behaves like a separate instance of object. like EnumSample constructor val, comment filed. Whenever we want to define a class level method we will use a static keyword so it will not belong to any specific enum. We can call the above method like below. directly using Enum name and it will return match enum object.

```
public class Main {
 public static void main (String args[]) {
   EnumSample sampleVar = EnumSample.getEnumFromValue(107);
   System.out.println(sampleVar.getComment());
 }
}
```

Method Override by constructor -> Overriding Methods in Enum Constructors:
In enums, each method is associated with every individual constant. If there's a need to override a method, it can be directly accomplished within the enum itself. For example, if we want to customise the behaviour of a method for a specific enum constant like MONDAY, we can directly override it within the enum declaration.

```java
public enum EnumSample{

    MONDAY{

        @Override
        public void dummyMethod(){
            System.out.println("Monday dummy Method");
        }
    },
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    public void dummyMethod(){
        System.out.println("default dummy Method");
    }
}
```

```java
public class Main {
 public static void main (String args[]) {
   EnumSample fridayEnumSample = EnumSample.FRIDAY;
   fridayEnumSample.dummyMethod();
   EnumSample mondayEnumSample = EnumSample.MONDAY;
   mondayEnumSample.dummyMethod();
 }
}
```

Enum with Abstract -> you can use abstract methods in enums to provide a common interface that each enum constant must implement. This allows you to have different behaviour for each constant while ensuring they all adhere to the same contract. Here's a simple example:

```java
public enum EnumSample{

    MONDAY {
        public void dummyMethod(){
            System.out.println("in Monday");
        }
    },
    TUESDAY{
        public void dummyMethod(){
            System.out.println("in Tuesday");
        }
    },
    SUNDAY{
        public void dummyMethod(){
            System.out.println("in Sunday");
        }
    };

    public abstract void dummyMethod();
}
```

```
public class Main {
 public static void main (String args[]) {
   EnumSample mondayEnumSample = EnumSample.MONDAY;
   mondayEnumSample.dummyMethod();
 }
}
```

Enum implement interface:

```
public interface MyInterface {
  public String toLowerCase();
}
```

```
public enum EnumSample implements MyInterfacef {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY;
  @Override
  public String toLowerCase() {
    return this.name().toLowerCase();
  }
}
```

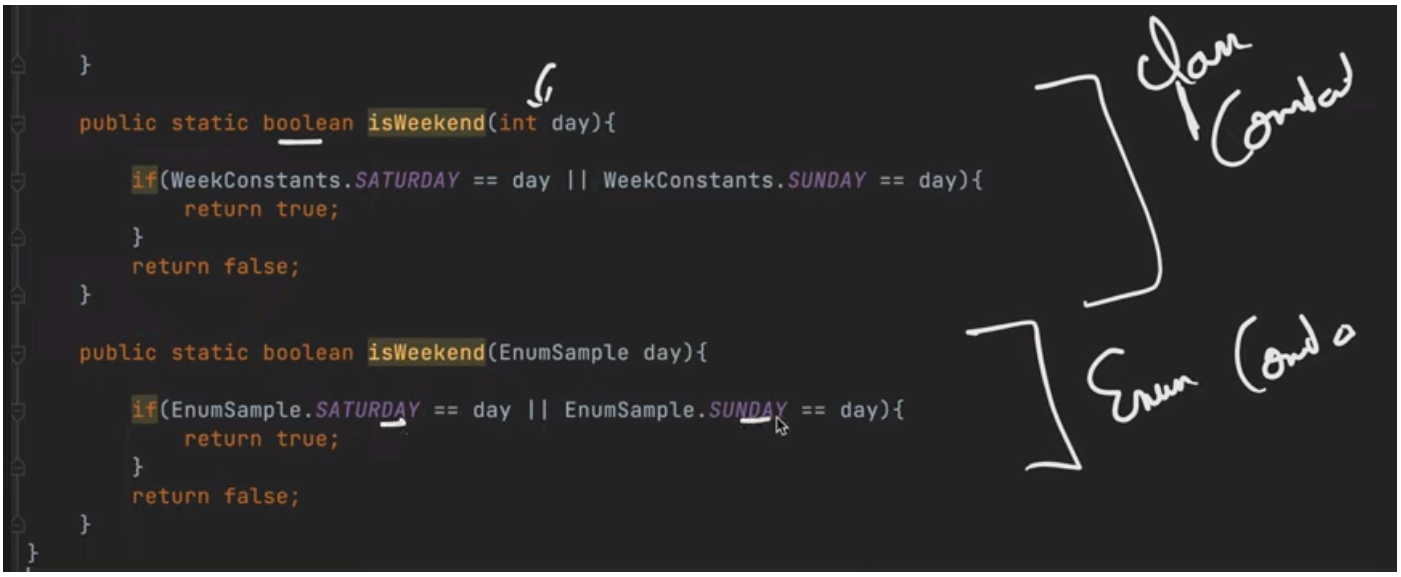
What is the benefit of ENUM class when we can create a constant through "static" and "final" keywords?

Below, we have created two classes—one using the "class" keyword and another using the "enum" keyword. Both classes perform the same task, but using "enum" is a more optimised approach. Enums provide safety in terms of values. In the main function, you can observe that for the class with static final values, any value can be passed, even incorrect ones like 1000. This may lead to unexpected errors. On the other hand, if we use an enum, we can only pass the desired values, ensuring that the parameter only accepts values defined in the enum. This brings several benefits, such as improved readability and compile-time error checking, as it restricts input to valid enum values.

```java
public class WeekConstants {

    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;

}
```

```java
public enum EnumSample{
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;
}
```

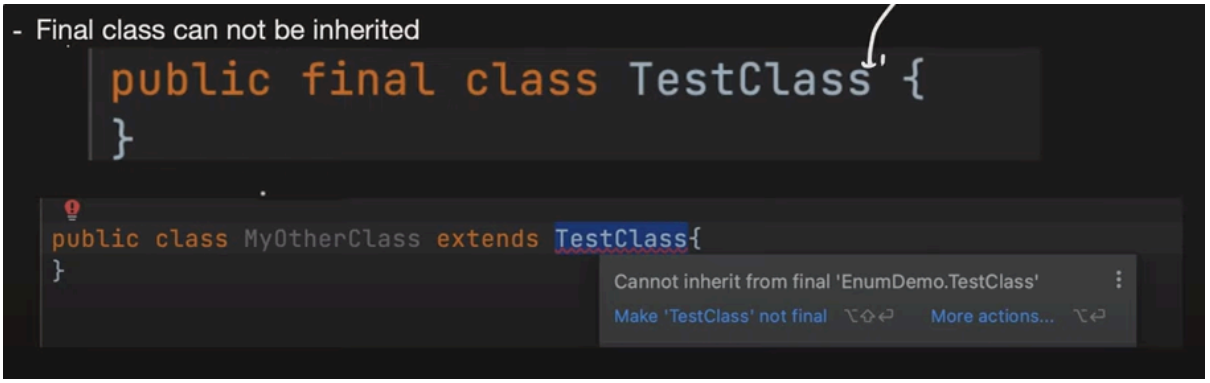
```java
public class Main {
 public static void main(String args[]) {
   isWeekend( day: 2); //wednesday, so it will return false
   isweekend( day: 6); //sunday, so it will return value
   isweekend( day: 100): //this value 1s not expected, but still we are able to send this in parameter
   //better readability and full control on what value we can pass an parameter
   isWeekend(EnumSample.WEDNESDAY); // return false
   isWeekend(EnumSample.SUNDAY): // return true
 }
}
```



Final Class -> A final class in Java is a class that cannot be subclassed or extended. Once a class is declared as final using the final keyword, it cannot have any subclasses. This design decision is made to prevent further modification or extension of the class, ensuring that its implementation remains unchanged.

```java
final class MyFinalClass {
 // Class members and methods
}
```

In this example, MyFinalClass is declared as final, meaning that no other class can extend it. If you try to create a subclass for a final class, the compiler will generate an error.



The main reasons for declaring a class as final include:
Security: Preventing sensitive parts of the code from being altered by subclassing.
Performance: The compiler and runtime can make certain optimizations knowing that the class won't be subclassed.
Design Intent: Indicating that the class design is complete and should not be extended to avoid unexpected issues.

It's important to note that methods within a final class can still be overridden unless they are explicitly declared as final or belong to a final class.

—----------

Singleton Class : This class objective is to create only 1 and Object.

Different Ways of creating Singleton Class:

- **Eager Initialisation**
- **Lazy Initialisation**
- **Synchronised Method**
- **Double Checked Locking**
- **Bill Pugh Singleton Solution**
- **Enum Singleton Solution**

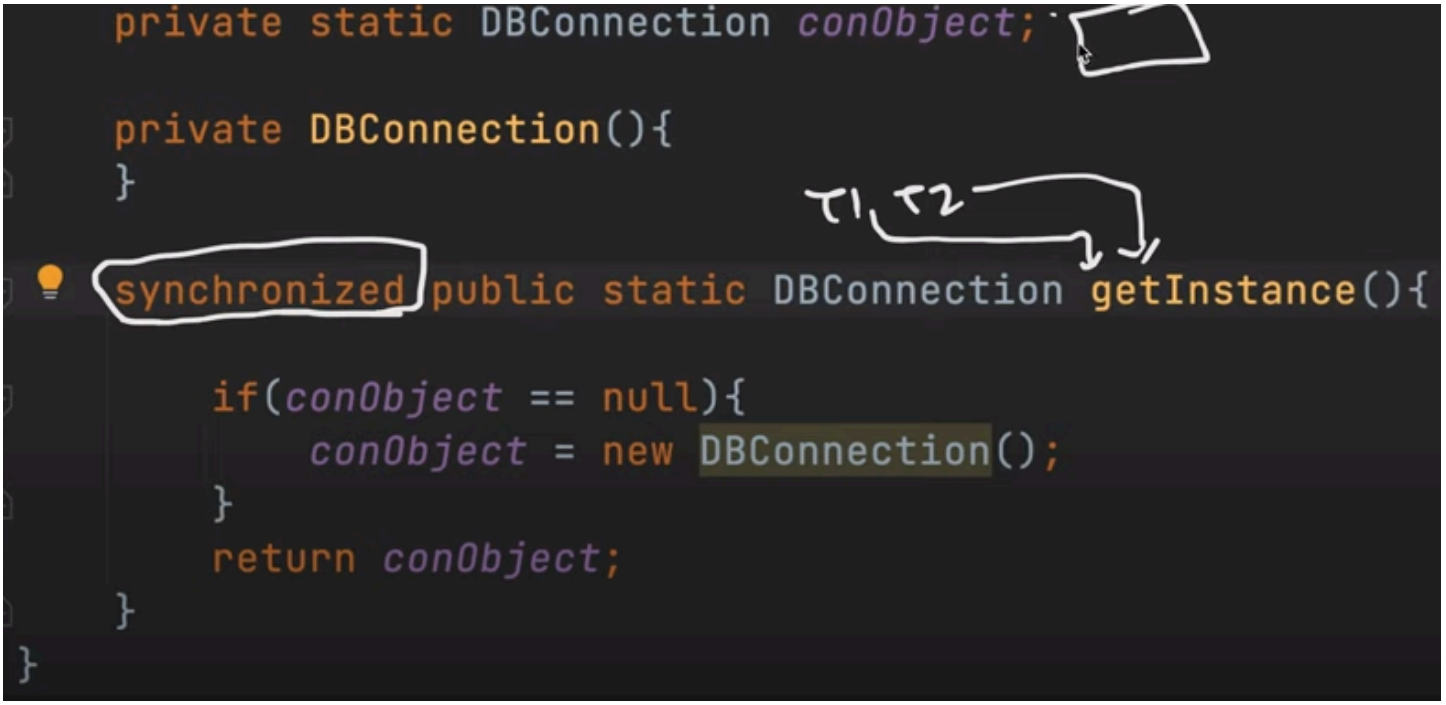
Eager Initialisation -> The issue with this method is that we cannot pass parameters to the constructor, and an object is automatically created when the class is loaded. Due to this behaviour, it is not an optimised approach.

```
public class DBConnection {
 private static DBConnection conObject = new DBConnection();
 private DBConnection(){}
 public static DBConnection getInstance(){
   return conObject;
 }
}
```

Lazy Initialisation -> In this way, we avoid making an object when the class loads. Instead, we declare and set it up in the constructor with an if condition. This ensures that the object is only created when necessary. However, a drawback is that it's not thread-safe; if two threads come at the same time, it might create more than one object.

```
private static DBConnection conObject;
private DBConnection(){}
public static DBConnection getInstance(){
  if(conObject == null){
    conObject = new DBConnection();
  }
  return conObject;
}
```

Synchronised Method -> This addresses the multithreading problem by making it thread-safe, but the trade-off is that it will run sequentially, losing the benefits of parallel processing and potentially slowing down the execution.



Double Checked Locking -> Double Checked Locking helps in avoiding the performance overhead of locking every time by ensuring that only the first call to getInstance results in locking and instantiation. Subsequent calls do not lock if the instance is already created, making it more efficient in a multithreaded environment. However, it's important to note that this pattern should be used with caution and is often unnecessary with modern JVMs and safer alternatives like static initialization.

```
private static volatile DatabaseConnection conObject;
private DatabaseConnection() {
}

public static DatabaseConnection getInstance() {
  if(conObject == null){
    synchronized (DatabaseConnection.class){
      if(conObject == null) {
        conObject = new DatabaseConnection();
      }
    }
  }
  return conObject
}
```

.

volatile -> The volatile keyword is used as a modifier for instance variables of a class. When a variable is declared as volatile, it indicates that the variable's value may be changed by multiple threads simultaneously. It ensures that any thread reading the variable sees the most recent modification made by any other thread.

The volatile keyword provides atomicity and visibility guarantees for the variable. Specifically:

Let's consider the scenario of **double-checked locking**. When one thread enters the synchronised block, it creates an object, but this object is initially stored in the L1 cache. Subsequently, another thread enters the synchronised block, checks if the object is created in memory, but the first thread's object resides in its L1 cache, not in main memory. Consequently, the second thread also perceives the object as null and creates an additional object in its L2 cache. To address this issue and ensure that threads see the most recent value directly from memory instead of checking in cache, we use the volatile keyword, which helps synchronise the value across all threads and resolves this problem.

Bill Pugh Singleton Solution -> The Bill Pugh Singleton Solution, also called the Initialization-on-demand holder idiom, is a method to create a thread-safe singleton in Java. It uses a special inner class to manage the creation of the singleton instance. The key benefit is that it allows the singleton to be created only when needed, without requiring complex synchronisation. This approach is seen as efficient and safe for use in multithreaded environments.

```
public class DatabaseConnection {
 private DatabaseConnection() {}
 private static class DBConnectionHelper {
   private static final DatabaseConnection INSTANCE_OBJECT = new
DatabaseConnection();
 }
```

```
  public static DatabaseConnection getInstance() {
    return DBConnectionHelper.INSTANCE_OBJECT;
  }
}
```

In this case, we're employing eager loading, but with a twist. We introduce an additional subclass responsible for creating the object inside it. Unlike eager loading, this subclass isn't created during the initial loading phase. It's made public but designed in a way that prevents external access. Instead, the object is generated on demand, following a lazy-loading strategy, and we avoid the intricacies associated with complex synchronisation.

Here's an example of the Bill Pugh Singleton Solution:

```
public class BillPughSingleton {

  private BillPughSingleton() {
      // private constructor to prevent instantiation
  }

  private static class SingletonHelper {
      private static final BillPughSingleton INSTANCE = new BillPughSingleton();
  }

  public static BillPughSingleton getInstance() {
      return SingletonHelper.INSTANCE;
  }
}
```

Explanation of the code:

The BillPughSingleton class has a private constructor to prevent external instantiation.

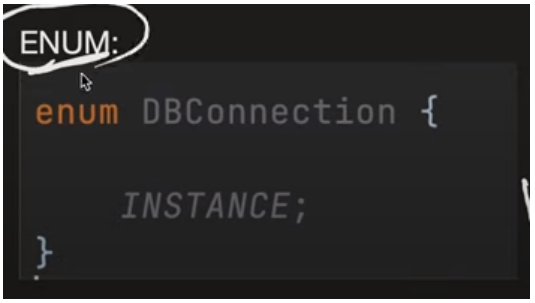
The SingletonHelper is a static inner class that contains the static final instance of the singleton. The JVM guarantees that the initialization of static variables in a class is thread-safe.

The getInstance method provides access to the singleton instance. When this method is called for the first time, the SingletonHelper class is loaded, and the singleton instance is created.

This approach ensures that the singleton instance is created lazily and is thread-safe without the need for explicit synchronisation. It takes advantage of the fact that class initialization in Java is inherently thread-safe, and the singleton instance is created only when the getInstance method is invoked for the first time.

This Bill Pugh Singleton Solution is considered a modern and effective way to implement a thread-safe singleton in Java.

Enum Singleton Solution -> An enum singleton is a design pattern where a singleton is implemented using an enumeration. Enums in Java provide a convenient and thread-safe way to implement a singleton because enum values are guaranteed to be instantiated only once in a Java program.



Here's an example of an enum singleton:

```
public enum EnumSingleton {
 INSTANCE;

 // Other methods and fields can be added here

 public void doSomething() {
     // Perform singleton-related operations
 }
}
```

Explanation of the code:

The EnumSingleton is an enum type with a single value INSTANCE.
The singleton instance is automatically created by the Java runtime when the enum type is loaded. It is guaranteed to be created only once.
Additional methods and fields can be added within the enum to provide functionality specific to the singleton.

Enum singletons provide a concise and thread-safe way to implement singletons in Java. They are inherently resistant to reflection attacks and serialization issues, making them a robust choice for creating singleton instances. The singleton instance can be accessed using EnumSingleton.INSTANCE throughout the code.

IMMUTABLE CLASS ->

- We can not change the value of an object once it is created.
- Declare class as 'final' so that it can not be extended.
- All class members should be private. So that direct access can be avoided.
- And class members are initialised only once using a constructor.
- There should not be any setter methods, which are generally used to change the value.
- Just getter methods. And returns Copy of the member variable.
- Example: String, Wrapper Classes etc.

```
final class MyImmutableClass {
 private final String name;
 private final List<Object> petNameList;
 MyImmutableClass(String name, List<Object> petNameList){
   this.name = name;|
   this.petNameList = petNameList;
 }

 public String getName(){
   return name;
 }

 public List<Object> getPetNameList(){
 //this is required,- because making list final,
 // means you can not now point it to new list, but still can add, delete values in
it
 //so that's why we send the copy of it.
   return new ArrayList<>(petNameList);
 }
}
```

```
public class Main {
 public static void main (String args[]) {
   List<Object> petNames = new ArrayList<>();
   petNames.add ("sj"):
   petNames.add("pj");

   MyImmutableClass obj = new MyImmutableClass( name: "myName", petNames);
   obj.getPetNameList().add("hello");
   System.out.println(obj.getPetNameList());
 }
}
```

In the mentioned class, if our goal is to create an entirely immutable class without setter methods, and our fields are primitive types, we can directly return them since they are private

and final. This ensures that their values cannot be altered. However, if we are dealing with a list, the final keyword prevents reassignment to a new list, but the elements within the list can still be modified. To safeguard against this, we should return a new copy of the list using the new keyword. This approach guarantees that the same reference is not returned, and a fresh memory allocation is provided, preventing unintended changes through other reference variables.

==========

Interface in Depth

==========

What is Interface?

- How to define Interface
- Why we need Interface
- Diamond Problem and Solution
- Method in Interface
- Fields in Interface
- Interface Implementation and Rules
- Nested Interface
    - Interface within Interface
    - Interface within a Class
- Abstract Class Vs Interface
- Java8 and Java9 Interface features
    - Default Method
    - Static Method
    - Functional Interface
    - Private Method

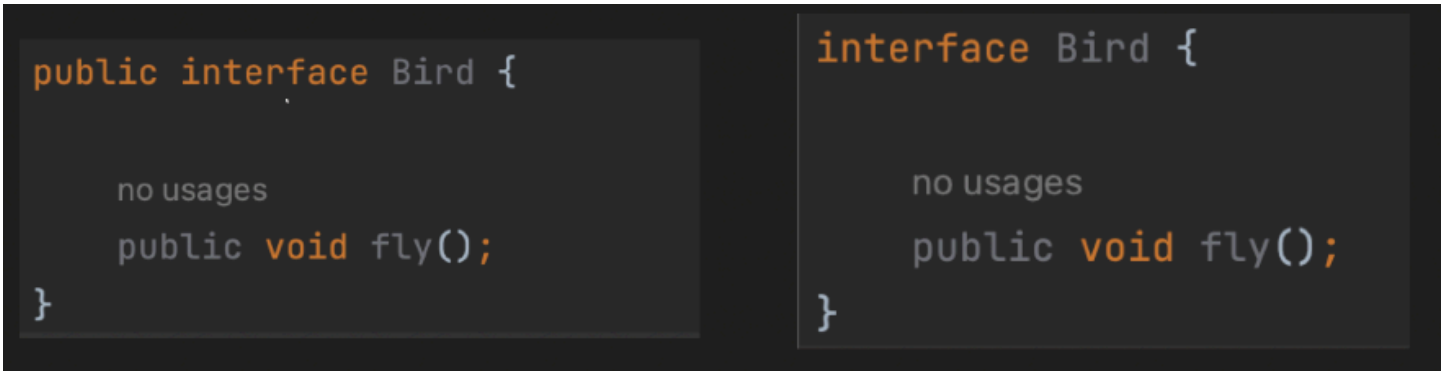
What is Interface -> an interface is a collection of abstract methods (methods without a body) and constant variables. It is a way to achieve abstraction and support multiple inheritance in Java. An interface allows you to define a set of methods that must be implemented by any class that implements the interface. It provides a way to define a contract for classes that implement it, ensuring that they provide specific behaviours

Interface is something which helps 2 systems to interact with each other, without one system has to know the details of the other Or in simple terms I can say, it helps to achieve ABSTRACTION.

How to define the interface?

Interface declaration consist of
- Modifiers
- "interface" keyword
- Interface Name
- Comma separated list of parent interfaces
- Body

Only Public and Default Modifiers are allowed (Protected and private are not allowed)



```
public interface Bird {

    no usages
    public void fly();
}
```

```
interface Bird {

    no usages
    public void fly();
}
```

Comma separated list of parent interfaces (it can extend from Class) Example:

```
public interface NonFlyingBirds extends Bird, LivingThingst{
 no usages
 public void canRun();
}
```

Why do we need Interface?

1. **Abstraction:** Using interface, we can achieve full Abstraction means, we can define WHAT class must do, but not HOW it will do

```
public interface Bird {
  no usages
  public void fly();
}


public class Eagle implements Bird {
  no usages
  @Override
  public void fly() {
   //the complex process of flying take place here
  }
}
```

2. Polymorphism:

- Interface can be used as a Data Type.
- we **can not create the object** of an interface, but it can **hold the reference** of all the classes which implements it. And at runtime, it decide which method need to be invoked.

```java
public class Main {
 public static void main(String args[]){
   Bird birdObject1 = new Eagle():
   Bird birdObject2 = new Hen();
   birdObject1.fly();
   birdObject2.fly();
 }
}
```

```java
public class Eagle implements Bird{

    2 usages
    @Override
    public void fly() {
        System.out.println("Eagle Fly Implementation");
    }
}
```

```java
public class Hen implements Bird{

    2 usages
    @Override
    public void fly() {
        System.out.println("Hen Fly Implementation");
    }
}
```

3. Multiple Inheritance: In Java Multiple inheritance is possible only through Interface

```java
public interface LandAnimal {

    1 usage  1 implementation
    public boolean canBreathe();
}
```

```java
public interface WaterAnimal {

    1 usage
    public boolean canBreathe();
}
```

```java
public class Crocodile implements LandAnimal, WaterAnimal{

    2 usages
    @Override
    public boolean canBreathe() {
        return true;
    }
}
```

```java
public class Main {
 public static void main(String args[]){
   Crocodile obj = new Crocodile();
   obj.canBreathe()
 }
}
```

Methods in Interface:

- All methods are implicit public only.
- Method can not be declared as final.

```
public interface Bird {
 1 usage 1 implementation
 void fly();
 no usages
 public void hasBeak();
}
```

Fields in Interface:

- Fields are public, static and final implicitly (CONSTANTS)
- You can not make fields private or protected.

```
public interface Bird {
 no usages
 int MAX_HEIGHT_IN_FEET = 2000;
}
          ==
public interface Bird {
 no usages
 public static final int MAX_HEIGHT_IN_FEET = 2000;
}
```

Access Specifiers in Overriding: When you implement an interface in a class and override its methods, you can't make those methods more restrictive in terms of access. For example, if an interface method is declared as public, your implementation in the class cannot be private or protected.
Obligation to Override: If a class implements an interface, it has to provide implementations (override) for all the methods declared in that interface. You can't leave any method unimplemented.
Abstract Classes vs. Interfaces: Classes marked as abstract are not forced to provide implementations for all methods they declare. However, if a class implements an interface, it must give concrete implementations for all methods from that interface.
Multiple Interfaces: A class is allowed to implement (adopt) multiple interfaces. This means it can inherit behaviours from different sources by implementing multiple interfaces.

```java
public interface Bird {
 2 usages 2 implementations
 public void fly();
}
```

```java
public class Eagle implements Bird {
 2 usages
 @Override
 public void fly() {
   System.out.println("Eagle Fly
Implementation");
  }
}
```

The access modifier for methods declared in an interface is always public and cannot be changed.

```java
public class Eagle implements Bird {
 1 usage
 @Override
 protected void fly() {
   //do something
 }
}
```

As shown below, we must either implement all the methods in a concrete class. If we only implement some of the interface methods, the concrete class also becomes abstract. Any other concrete class that extends this class must provide implementations for the remaining methods.

Exp. of Abstract class implementation of Interface

```java
public interface Bird {
 no usages 1 implementation
 public void canFly();
 no usages 1 implementation
 public void noOfLegs();
}
```

```java
public abstract class Eagle implements Bird
{
 no usages
 @Override
 public void canFly() {
   // Implementation goes here
 }
 no usages
 public abstract void beakLength();
}
```

```java
public class WhiteEagle extends Eagle{
 no usages
 @Override
 public void noOflegs() {
   //implement interface method
```

```
 }

 no usages
 @Override
 public void beakLength() {
   //implementing abstract class method
 }
}
```

Nested Interface:

- Nested Interface declared within another Interface.
- Nested Interface declared within a Class.

Generally it is used to group logical related interfaces. And Nested interface

Rules:

- A nested interface declared within an interface must be public.
- A nested interface declared within a class can have any access modifier.
- When you implement the outer interface, inner interface implementation is not required and vice versa.

```
public interface Bird {

 no usages 1 implementation
 public void canFly();

 no usages
 public interface NonFlyingBird{
   no usages
   public void canRun();
 }
}
```

```java
public class Eagle Implements Bird,
 Bird. NonFlyingBird {
 1 usage
 @Override
 public void canRun() {
 }
 no usages
 @override
 public void canfly(){
 }
}
```

```java
public class Main {
 public static void main(String args[]) {
   Bird.NonFlyingBird obj = new Eagle();
   obj.canRun();
 }
}
```

Nested Interface declared within a Class.

```java
public class Bird {
  protected interface NonFlyingßird {
      no usages
      public void canRun();
  }
}


public class Eagle implements Bird.NonFlyingBird {
  no usages
  @Override
  public void canRun() {
  }
}
```

Interface Vs Abstract Class:

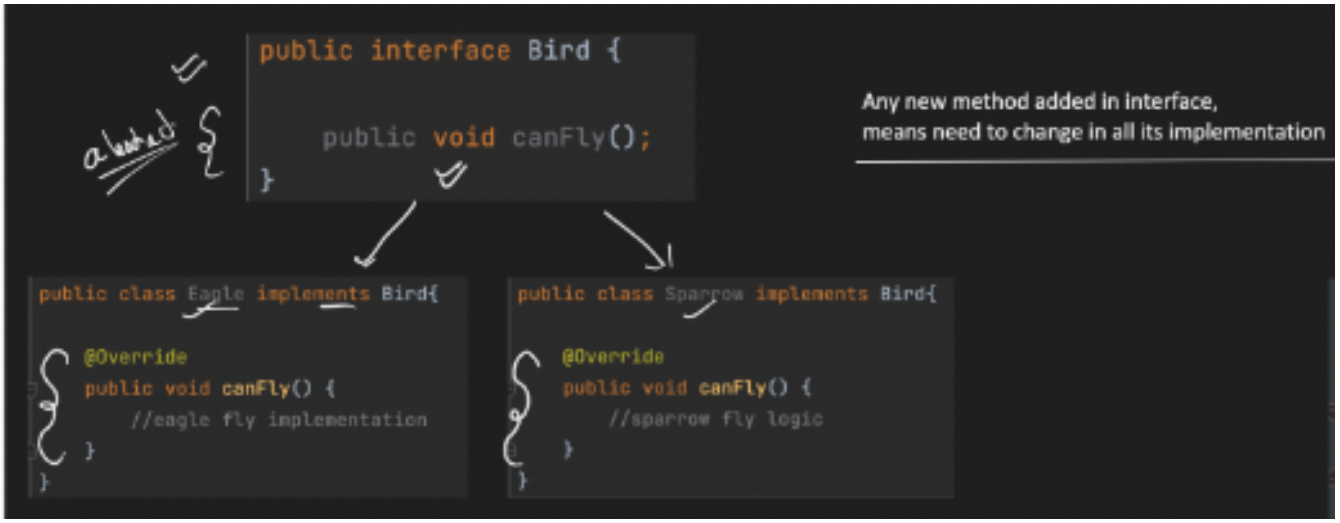
S.No.	Abstract Class	Interface
1.	Keyword used here is "abstract"	Keyword used here is "interface"
2.	Child Classes need to use keyword "extends"	Child Classes need to use keyword "implements"
3.	It can have both abstract and non abstract method	It can have only Abstract method (from Java8 onwards it can have default ,static and private method too, where we can provide implementation)
4.	It can Extend from Another class and multiple interfaces	It can only extends from other interfaces
5.	Variables can be static, non static, final , non final etc.	Variable are by default CONSTANTS
6.	Variables and Methods can be private, protected, public, default.	Variable and Methods are by default public (in Java9 , private method is supported)
7.	Multiple Inheritance is not Supported	Multiple Inheritance supported with this in Java
8.	It can provide the implementation of the interface	It can not provide implementation of any other interface or abstract class.
9.	It can have Constructor	It can not have Constructor
10.	To declare the method abstract, we have to use "abstract" keyword and it can be protected, public, default.	No need for any keyword to make method abstract. And be default its public.

==========

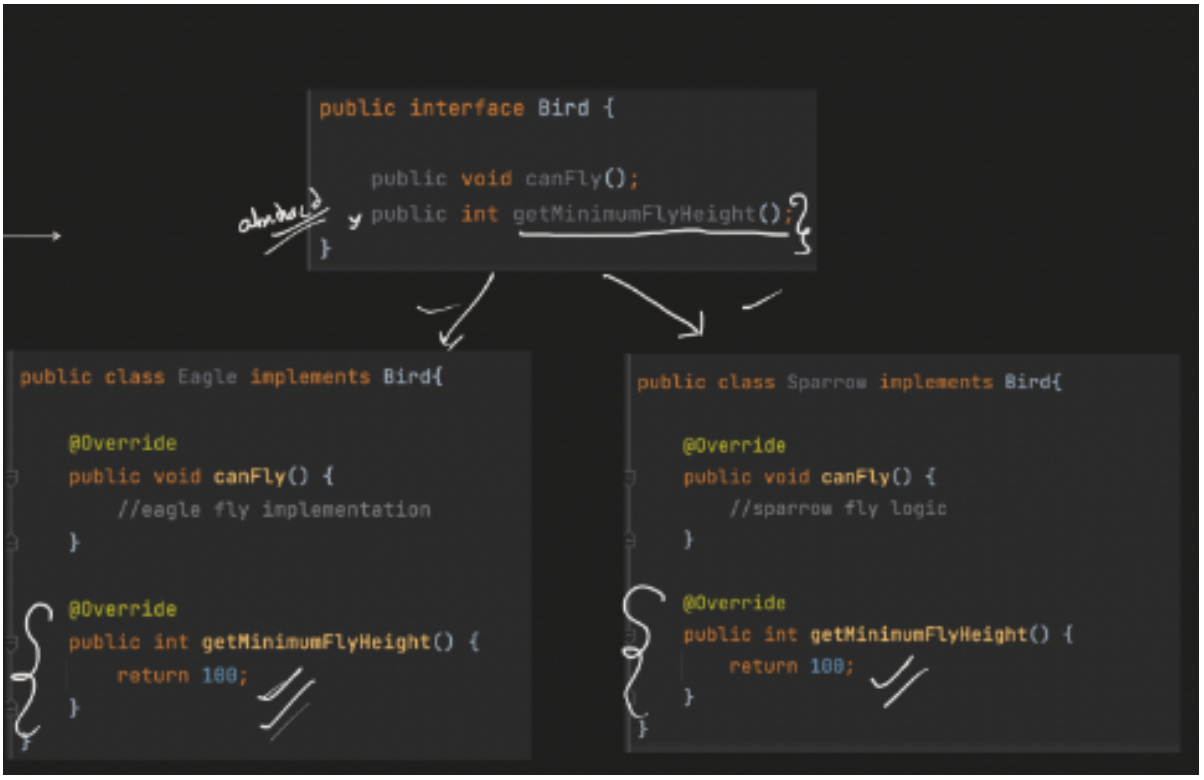
JAVA 8 and 9 Features

==========

1. **Default Method(Java8):** Before Java8, interface can have only Abstract method. And all child classes has to provide abstract method implementation.



The problem with interfaces is that if we add a new method, we have to modify the implementation in all classes that use this interface, even if the logic for that method is the same across all implementations.

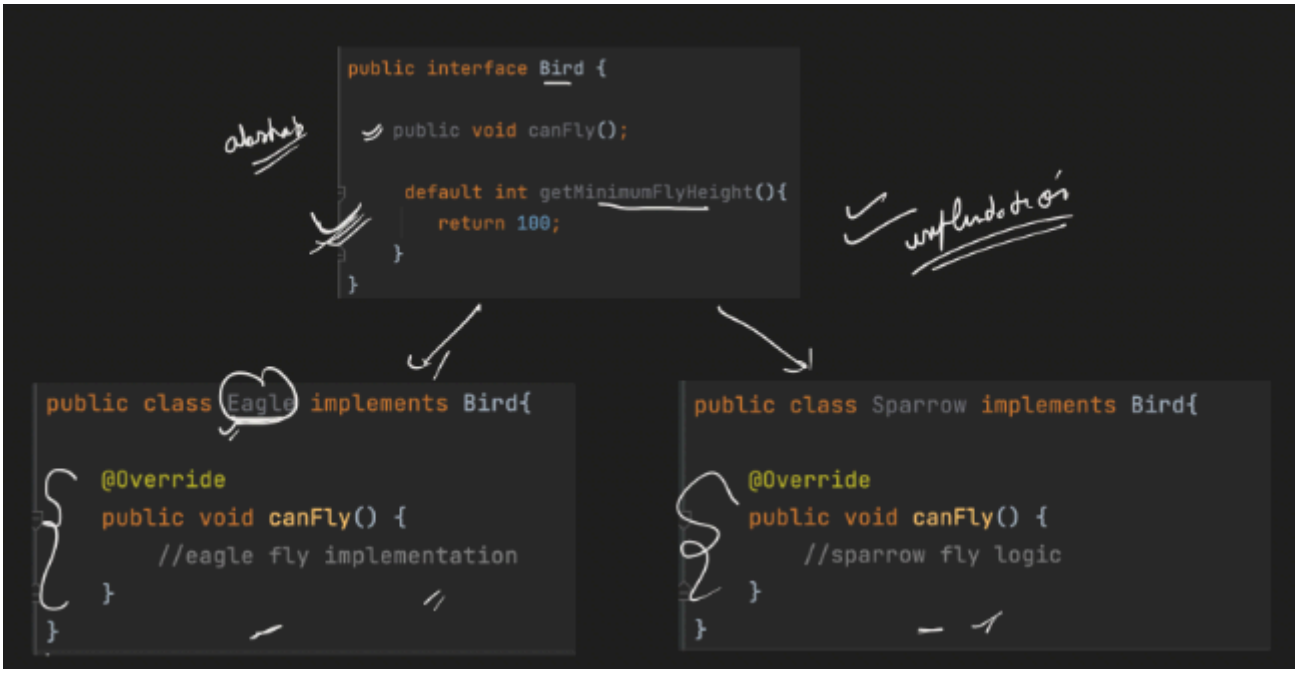


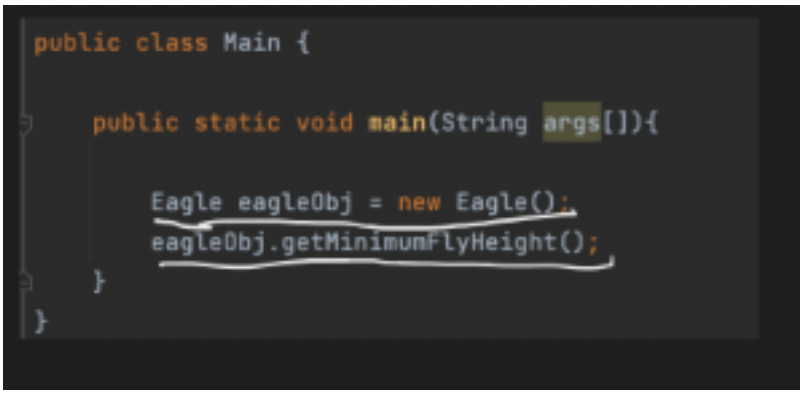
To address the above implementation in all interface issues, we can introduce a default method in the interface. This way, we provide a default implementation in the interface itself.

If necessary, classes or interfaces that implement this interface can choose to override this default method

Why Default method was introduced: Default methods in interfaces were introduced in Java 8 to address the issue of multiple inheritance and provide a way to extend interfaces without breaking existing implementations. Here's a brief explanation:To add functionality in the existing Legacy Interface we need to use the Default method. Example steam() method in Collection.

Use Default Method:

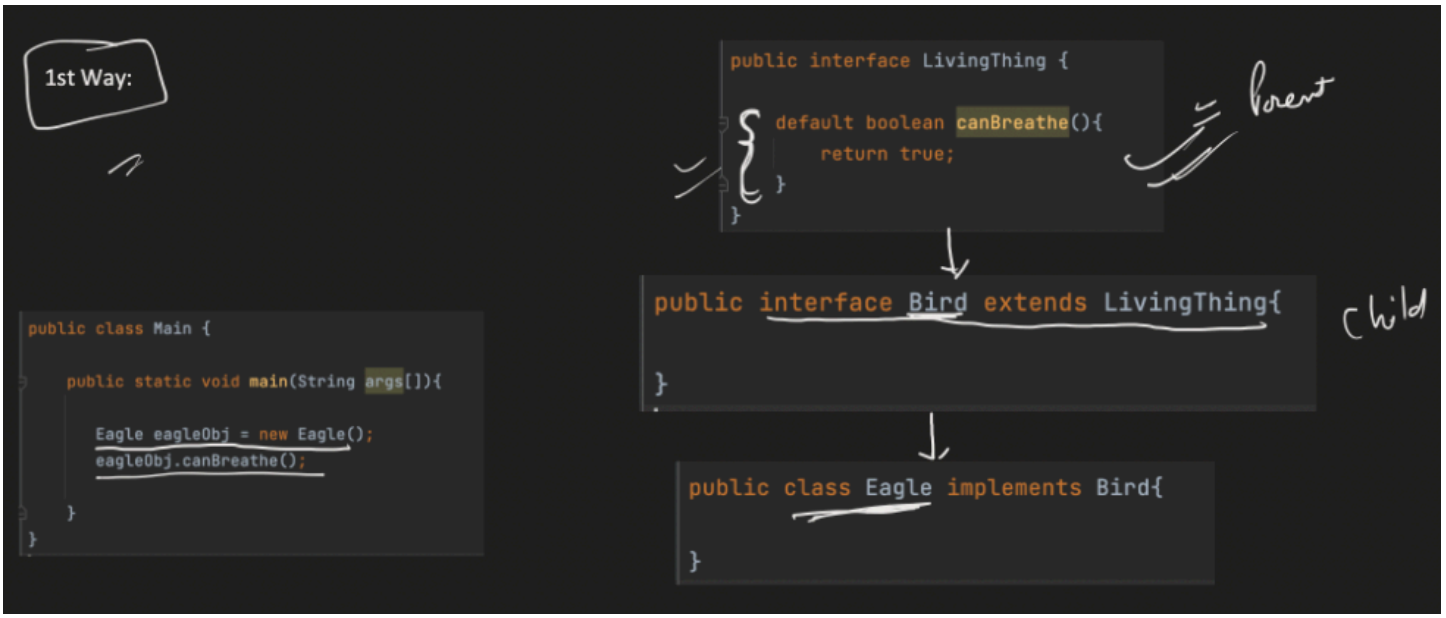




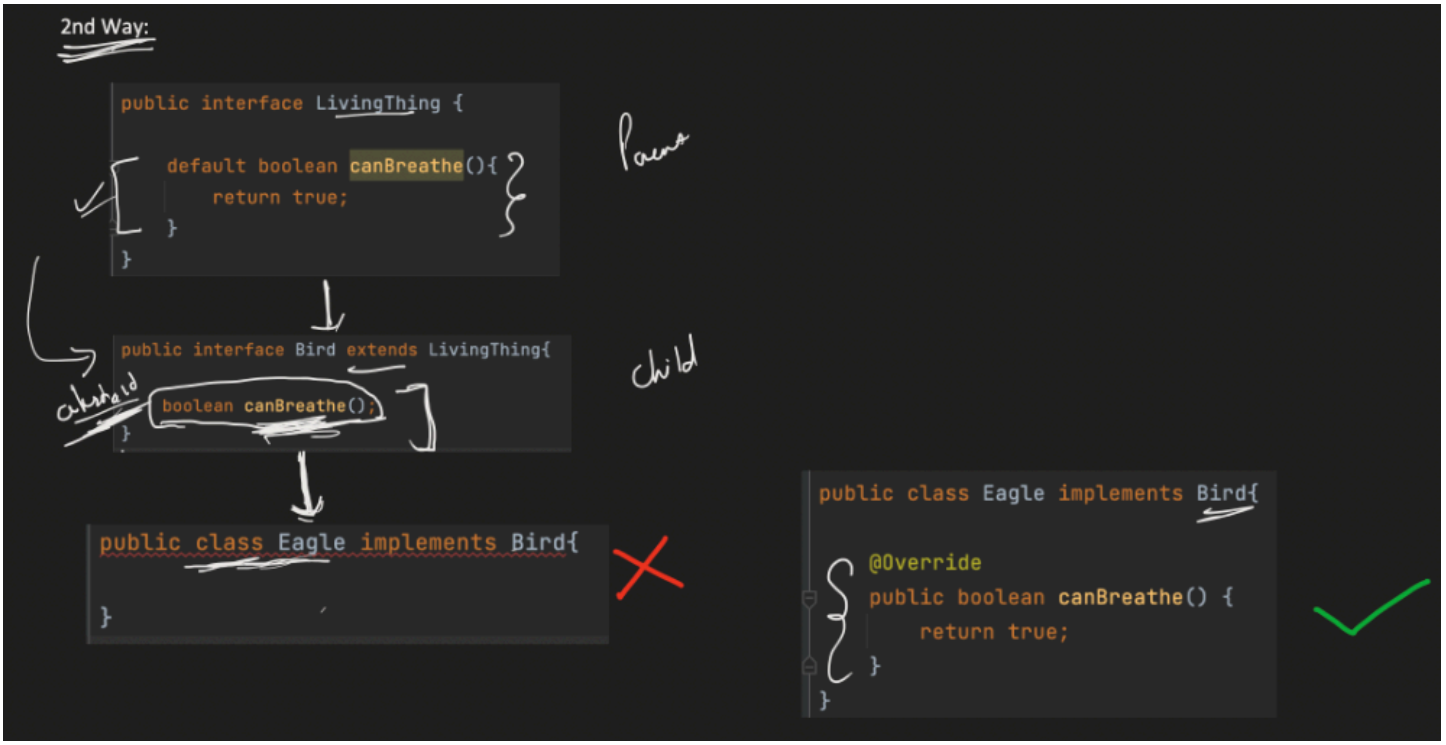
Default and Multiple Inheritance, how to handle

Default and Multiple Inheritance in Java can be managed by implementing multiple interfaces. However, when two or more interfaces contain a default method with the same signature, it creates confusion for the compiler. To resolve this, you need to provide a specific implementation in the class or interface that extends or implements these conflicting interfaces. If the default method signatures are different across interfaces, no implementation is required for the extending class or interface.

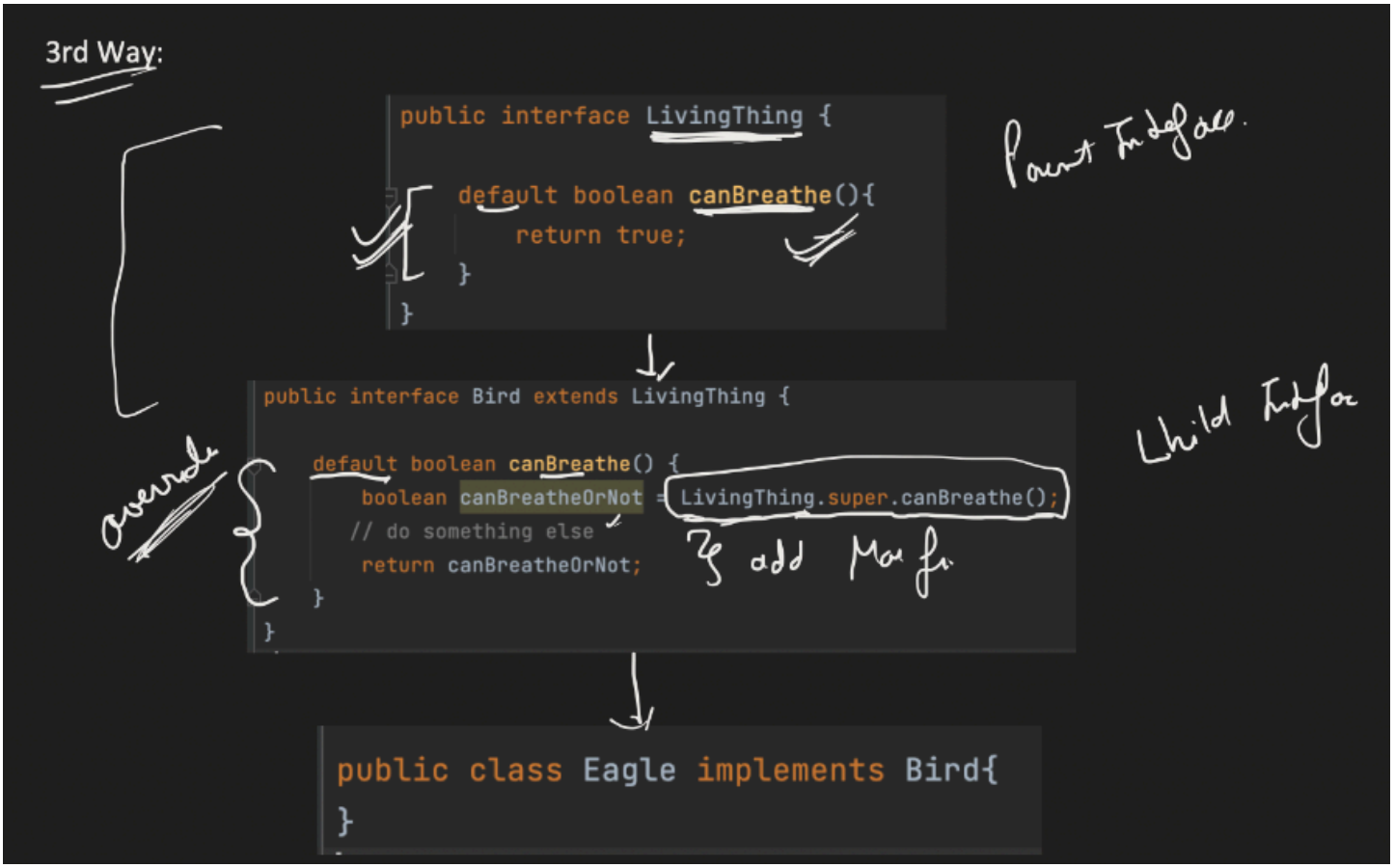
How to extend interface, that contains Default Method



In the second approach, we initially incorporated a default method, and subsequently, in the next interface that extends the LivingThing interface, a corresponding abstract method was introduced without an implementation. This implies that the second Bird interface overrides the default method. Consequently, any class extending this interface is obligated to implement the method, as the implemntation has been overridden by the Bird interface.

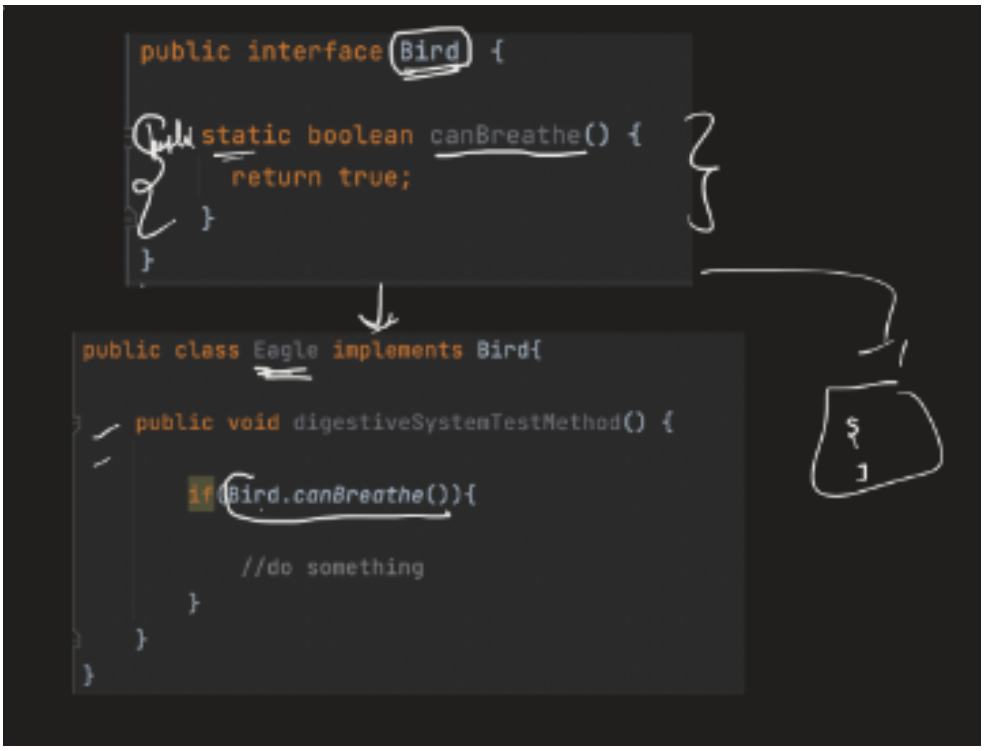


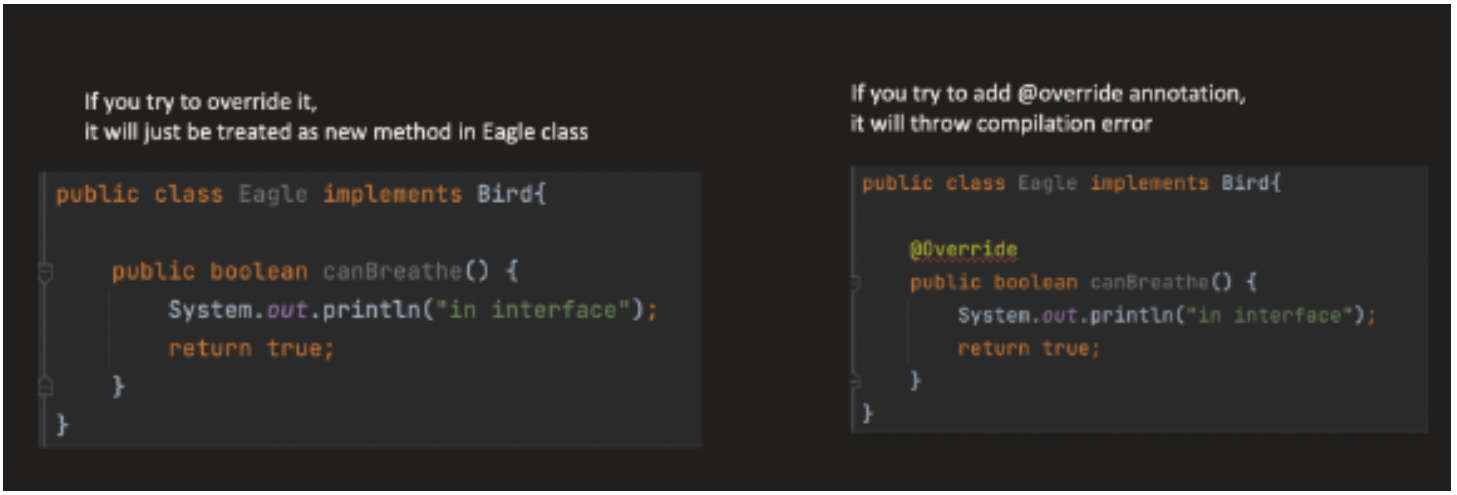
As demonstrated below, it is possible to override the default method implementation of a parent interface in its child interface. If there is a desire to utilize the default method from the parent interface, it can be accessed using the "super" keyword. Additionally, one can choose to implement this method further in the child interface.



2. Public Static Method (Java8):

- We can provide the implementation of the method in the interface.
- But it can not be overridden by classes which implement the interface.
- We can access it using the Interface name itself.
- It's by default public.





3. Private Method and Private Static method Java9:

- We can provide the implementation of the method but as a private access modifier in the interface.
- It brings more readability of the code. For example if multiple default methods share some code, that can help.
- It can be defined as static and non-static.
- From the Static method, we can call only private static interface methods.
- Private static methods can be called from both static and non-static methods.
- Private interface methods can not be abstract. Means we have to provide the definition.
- It can be used inside of the particular interface only.

```java
public interface Bird {
 void canFly(); //this is equivalent to public abstract void canFly();

 public default void minimumFlyingHeight() {
   myStaticPublicMethod(); //calling static method
   myPrivateMethod(); //calling private method
   myPrivateStaticMethod(); //calling private static method
 }

 static void myStaticPublicMethod(){
   myPrivateStaticMethod(); //from static we can other static method only
 }
 private void myPrivateMethod(){
   // private method implementation
 }
 private static void myPrivateStaticMethod(){
   // private static method implementation
 }
}
```

==========

Designing Class and Schema Diagrams for Splitwise: A Step-by-Step Guide

Visit

https://youtu.be/NGsenhTCMoA

Like | Comment | Subscribe

==========| **@GeekySanjay** |==========

==========

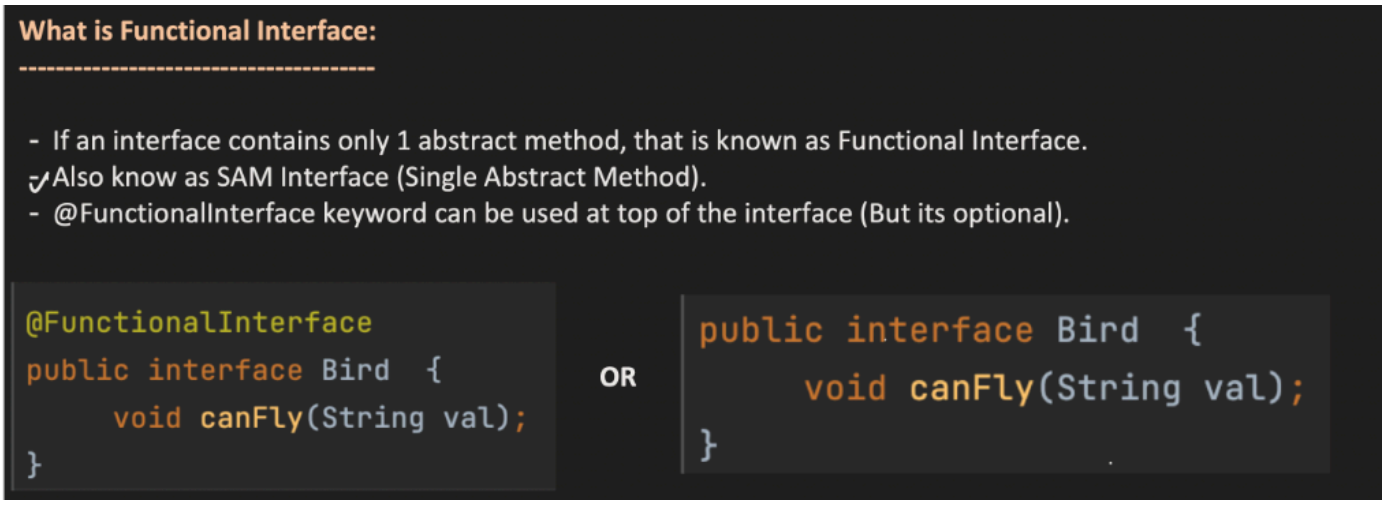
==========

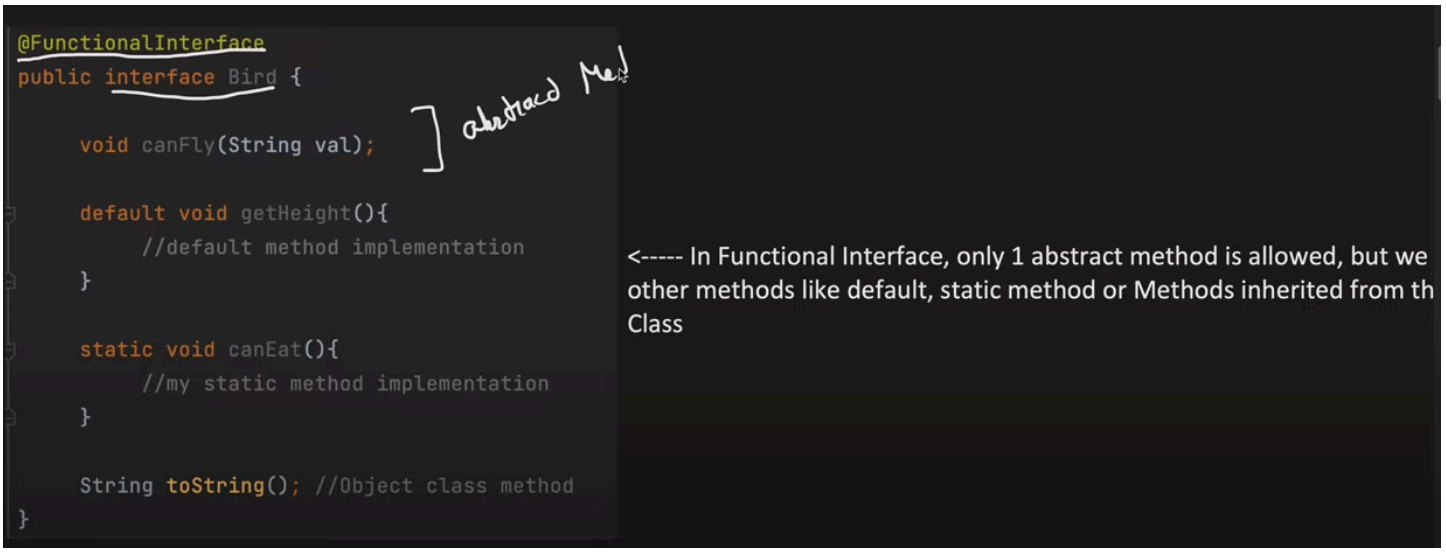
Functional Interface and Lambda Expression

==========

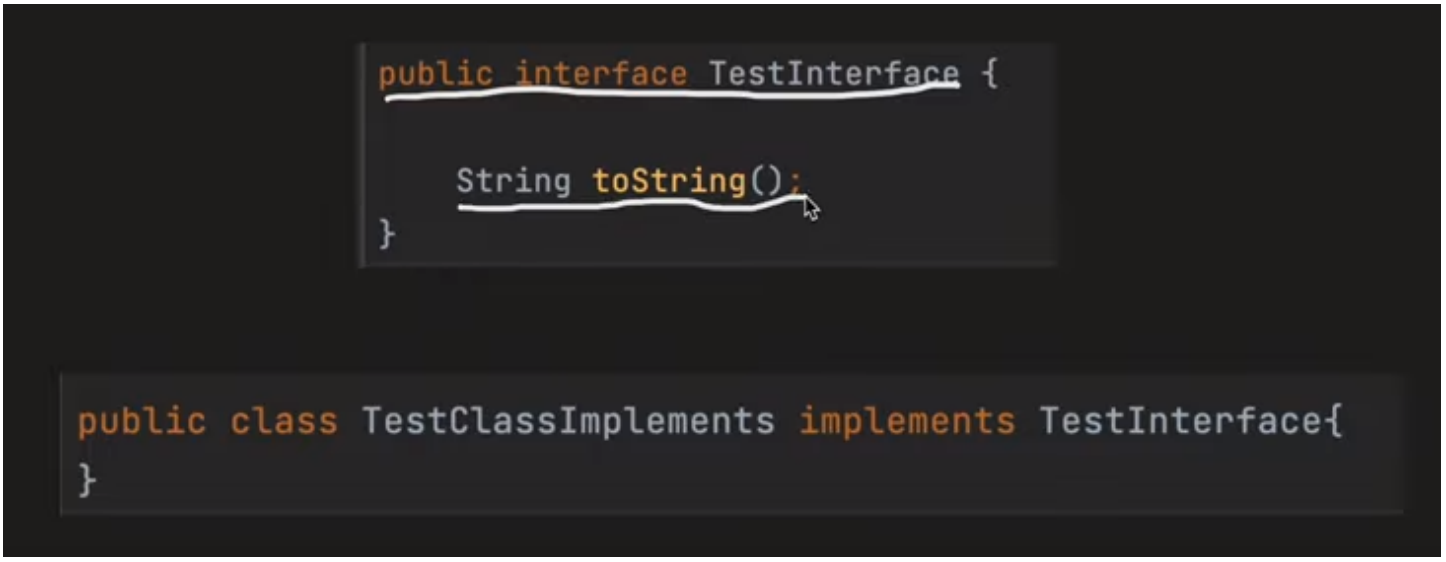
Functional Interface -> a functional interface is an interface that contains only one abstract method. However, it can have multiple default or static methods. The key characteristic is that it should have a single abstract method for it to be considered a functional interface.

- What is Functional Interface?
- What is Lambda Expression?
- How to use Functional Interface with Lambda expression
- Advantage of Functional Interface?
- Types of Functional Interface?
  - Consumer
  - Supplier
  - Function
  - Predicate
- How to handle use case when Functional Interface extends from other Interface(or Functional Interface)?





In the provided Bird interface, we included one abstract method and String toString(). However, there's a mistake saying that String toString() is abstract. It's not abstract; it's inherited from the fundamental Object class, making the Bird interface still a functional interface. In the Test Interface below, there's only one String toString() method. Since every class automatically extends Object and this method is already implem-ented in the Object class, there's no requirement to implement this method again in other classes that utilise this interface.



What is Lambda Expression -> A lambda expression in Java is a concise way to express an anonymous function—a function without a name. It provides a way to write more readable and compact code, especially when working with functional interfaces.

Lambda expression is a way to implement the Functional Interface

The basic syntax of a lambda expression is as follows:
(parameters) -> expression

Parameters: The input parameters for the function. If there are no parameters, you still need to use empty parentheses ().

Arrow ->: It separates the parameters from the body of the lambda expression.

Expression: The body of the function, which is executed when the lambda expression is invoked. If the body is a single expression, you can skip the curly braces {} and the return keyword. If there are multiple statements, you should use curly braces and the return keyword.

Here's a simple example of a lambda expression:

// Without Lambda Expression

```
Runnable runnableWithoutLambda = new Runnable() {
 @Override
 public void run() {
     System.out.println("Hello, World!");
 }
};
```

// With Lambda Expression

Runnable runnableWithLambda = () -> System.out.println("Hello, World!");
In the above example, the lambda expression () -> System.out.println("Hello, World!") represents a Runnable that prints "Hello, World!" when its run method is called.

Lambda expressions are often used when working with functional interfaces, which are interfaces with a single abstract method. They make the code more concise and expressive, particularly in scenarios involving stream processing, event handling, and functional programming constructs introduced in Java 8 and later versions.

How it relates to a functional interface -> Below, we have three types of implementations for a functional interface: first, a simple one with a class; second, using an anonymous class; and third, with a lambda expression. If we have a functional component to implement, creating a complete class or an anonymous class involves a lot of code writing for just one method. We define the class, specify the type of method, and give it a name. To simplify this, we can directly use the required code using a lambda expression. For example, a lambda expression for a Bird interface implementation would look like this:

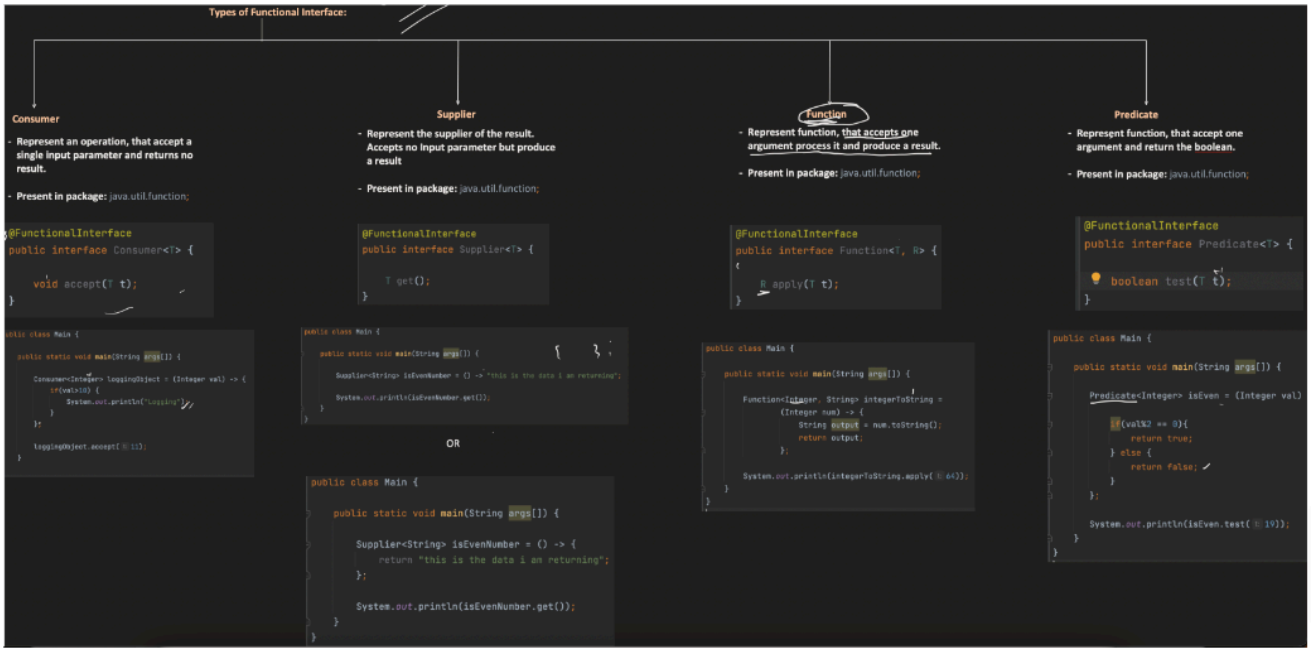
Bird eagleObject = (String value) -> System.out.println("This is a lambda expression");

The “->” is the lambda operator, and anything after it is the lambda expression. If there are multiple parameters or no parameters, we use (); if there is a single parameter, we can avoid it. Similarly, if there's only one line of code after the lambda operator, we can skip using {}, but if there are multiple lines, we need to include them.

Type of functional interface -> Starting from Java 8, the introduction of lambda expressions and the @FunctionalInterface annotation made it easier to work with functional interfaces. Lambda expressions provide a concise way to implement the abstract method of a functional interface.

Some commonly used functional interfaces in Java, available in the java.util.function package, include:
Consumer: Represents an operation that accepts a single input and returns no result.
Supplier: Represents a supplier of results.
Predicate: Represents a predicate (boolean-valued function) of one argument.
Function: Represents a function that accepts one argument and produces a result.
UnaryOperator: Represents an operation on a single operand that produces a result of the same type as its operand.
BinaryOperator: Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

The **@FunctionalInterface** annotation is optional, but it is a good practice to use it because it ensures that the interface has only one abstract method. If you try to declare more than one abstract method in an interface marked with @FunctionalInterface, the compiler will generate an error.



Consumer ---------->
- Represent an operation, that accept a single input parameter and returns no result.
- Present in package: java.util.function;

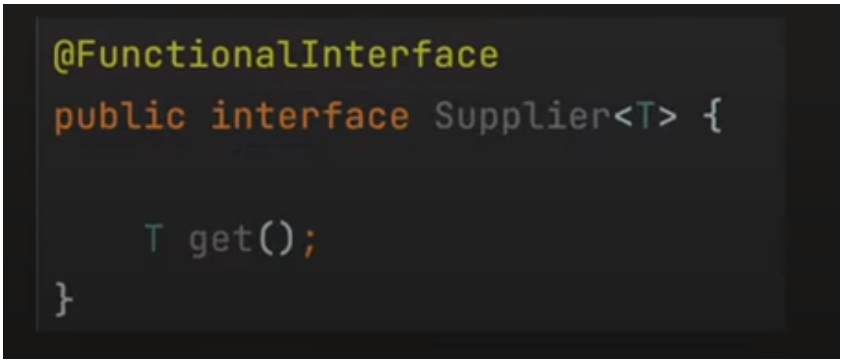
```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
}
```

```
public class Main {
 public static void main(String args[]) {
   Consumer<Integer> loggingObject = (Integer val) -> {
     if(val>10) {
       System.out.println("Logging");
     }:
   }
 }
}
```

Supplier ---------->

- Represent the supplier of the result. Accepts no Input parameter but produce a result

- Present in package: java.util.function;




```
public class Main {
 public static void main(String args[]) {
   Supplier<String> isEvenNumber = () -> "this is the data i am"
   System.out.println(isEvenNumber.get());
 }
}
```

OR

```
public class Main {
 public static void main (String args[]) {
   Supplier<String> isEvenNumber = () -> {
     return "this is the data i am returning":
   };
   System.out.println(isEvenNumber.get());
 }
}
```

Function ---------->

- Represent function, that accepts one argument process it and produce a result.
- Present in package: java.util.function;

```java
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
}
```

```java
public class Main {

    public static void main(String args[]) {

        Function<Integer, String> integerToString =
                (Integer num) -> {
                    String output = num.toString();
                    return output;
                };

        System.out.println(integerToString.apply( t: 64));
    }
}
```

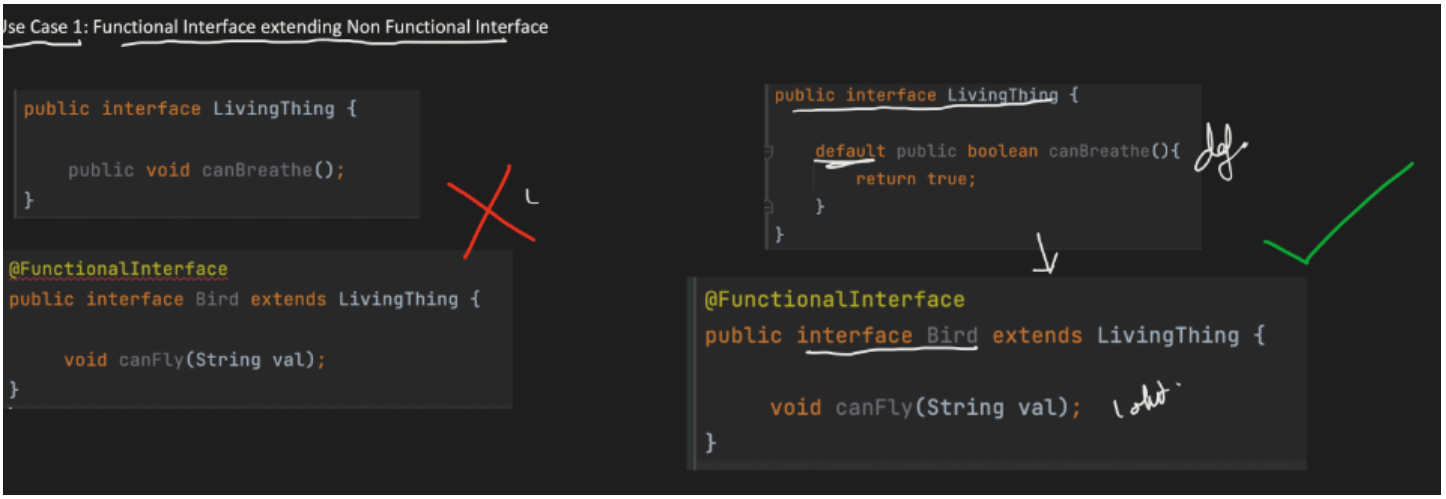
Predicate type

- Represent function, that accept one argument and return the boolean.
- Present in package: java.util.function;

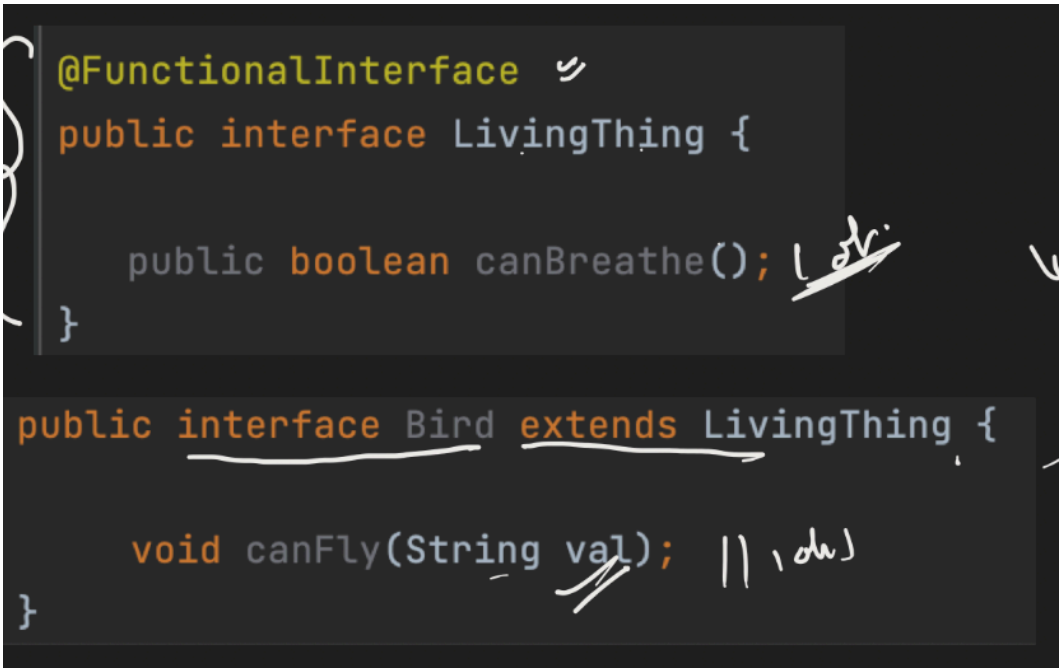
```java
@FunctionalInterface
 public interface Predicate<T> {
 boolean test(T t);
}
```

```java
public class Main {

    public static void main(String args[]) {

        Predicate<Integer> isEven = (Integer val) -> {

            if(val%2 == 0){
                return true;
            } else {
                return false;
            }
        };
```

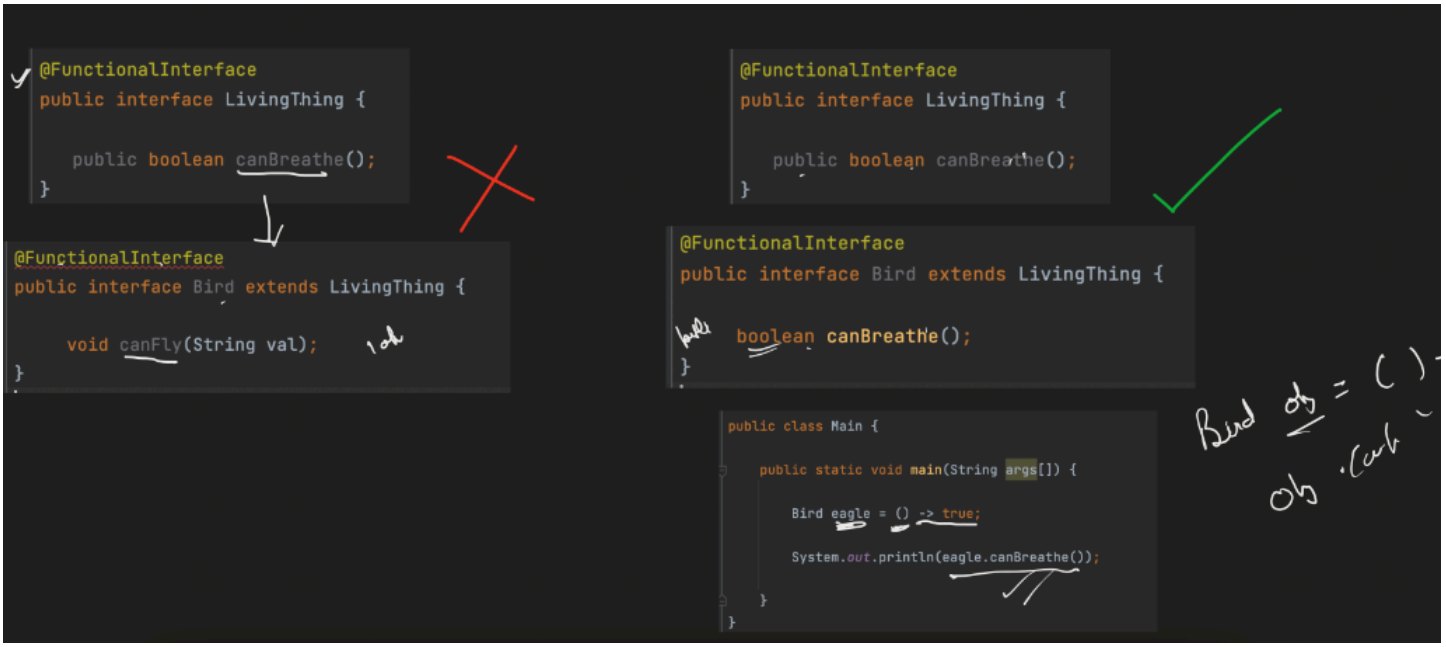
Handle use case when Functional Interface extends from othor Interface ->As seen below, if a functional interface extends a non-functional interface and their method signatures are different, the compiler will generate an error if the @FunctionalInterface annotation is present. This is because the inheritance introduces two abstract methods in the functional interface. However, in the second example, if the first non-functional interface includes a default method, there is no complaint because a functional interface can have one abstract method along with static and default methods.



Use Case 2: Interface extending Functional Interface -> It will work fine because interface can have any number of abstract methods.



Use Case: Functional Interface extending another Functional Interface -> If a functional interface extends another functional interface with a different method signature, the compiler will produce an error. However, if both interfaces have the same method signature, no error will be generated, as illustrated in the second example.



@GeekySanjay

Thank You

==========

Contact Me

If you have any questions or suggestions regarding the video,
please feel free to reach out to me
on WhatsApp: **Sanjay Yadav Phone: 8310206130**
https://www.linkedin.com/in/yadav-sanjay | https://www.youtube.com/@GeekySanjay

==========