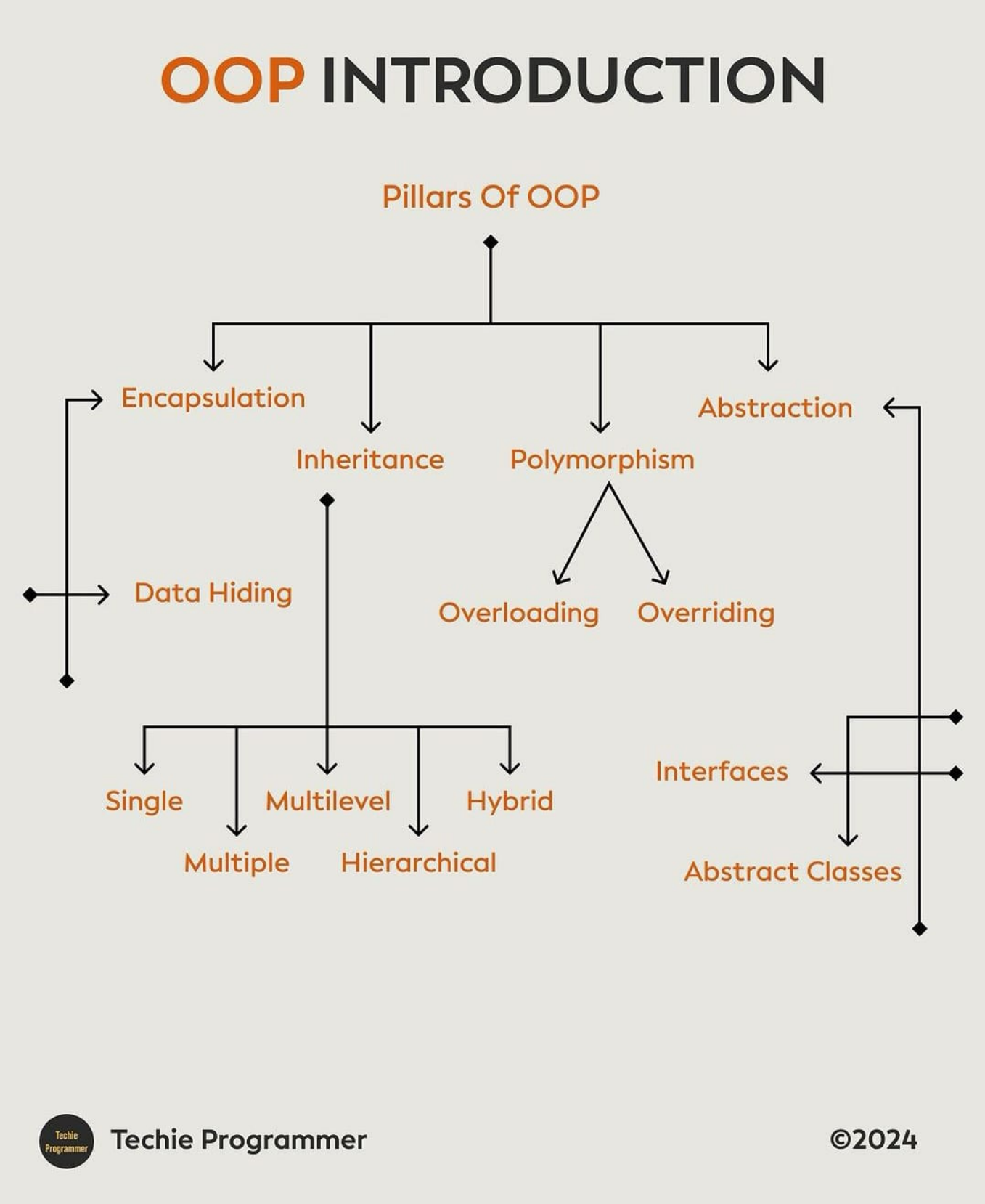
## OOPs in Java




## Diff btw Run-compile time

In programming, **runtime** and **compile-time** refer to different phases in the lifecycle of a program. Understanding the differences between them is crucial for debugging, optimizing, and writing effective code. Here's a breakdown of what each term means and how they differ:

## Compile-Time

- **Definition**: Compile-time refers to the period during which the source code is translated into executable code by a compiler. This is the phase where the program is checked for syntax errors, type checking, and other static code analysis.
- **Key Aspects**:
  1. **Syntax Checking**: The compiler checks the code for syntax errors. If the syntax is incorrect, the program will not compile.
  2. **Type Checking**: The compiler ensures that variable types are consistent and correct according to the language's rules.
  3. **Static Analysis**: The compiler may perform optimizations, in-lining, and other static analysis techniques to improve performance and catch potential issues.
  4. **Error Detection**: Errors that can be detected without executing the program, such as undeclared variables, type mismatches, or missing semicolons, are flagged at compile-time.
- **Example**:

```
int x = "Hello";  // Compile-time error: incompatible types
```

  This code will fail to compile because you're trying to assign a `String` to an `int`.
- **Compile-Time Errors**: Errors that are detected at this stage are known as compile-time errors, which must be fixed before the program can be successfully compiled.

## Runtime

- **Definition**: Runtime refers to the period when the program is executed, after it has been compiled. This is the phase where the compiled code is run, and the program performs its intended tasks.
- **Key Aspects**:
  1. **Execution of Code**: The code that was compiled is now executed. The program interacts with resources such as memory, file systems, and networks.
  2. **Dynamic Behavior**: Any dynamic aspects of the program, like user input, network communication, or file handling, occur at runtime.
  3. **Error Detection**: Errors that occur during the execution of the program, such as division by zero, null pointer exceptions, or out-of-bounds array access, are detected at runtime.
  4. **Memory Management**: Dynamic memory allocation and garbage collection, if applicable, happen at runtime.
- **Example**:

```
int[] arr = new int[5];
System.out.println(arr[10]);  // Runtime error: ArrayIndexOutOfBoundsException
```

  This code will compile successfully, but it will throw an `ArrayIndexOutOfBoundsException` when run because it attempts to access an index that doesn't exist.
- **Runtime Errors**: Errors that occur during the execution of the program are known as runtime errors. These errors might be due to unexpected user input, resource unavailability, or logic errors in the code.

## Key Differences

1. **Timing**:
   - **Compile-Time**: Occurs before the program runs, during the compilation process.
   - **Runtime**: Occurs while the program is running.
2. **Error Types**:
   - **Compile-Time**: Syntax errors, type mismatches, and other issues that prevent the program from being compiled.
   - **Runtime**: Errors that occur when the program is executing, such as logic errors, exceptions, and resource-related issues.
3. **Code Checking**:
   - **Compile-Time**: The compiler checks for errors and performs optimizations.
   - **Runtime**: The program is actually executed, and errors related to the program's logic or environment may arise.
4. **Performance**:
   - **Compile-Time**: Affects the time it takes to compile the program; optimizations can improve runtime performance.
   - **Runtime**: Affects the program's execution time and efficiency.

## Summary

- **Compile-Time**: The stage where the program is checked and translated into executable code. Issues found here prevent the program from being compiled.
- **Runtime**: The stage where the program is executed. Issues found here occur while the program is running and can lead to crashes or unexpected behavior.

Understanding the distinction between compile-time and runtime is essential for effective debugging and program development.

# Class And Object Creation

**Here the Objects are in stack memory and their properties are in heap memory
Whenever class(student) called it will be automatically memory in heap for the properties in it.

## 1. Class as a Blueprint

- **Class**: A class in Java is a blueprint or template for creating objects. It defines the attributes (properties) and methods (behaviors) that the objects created from this class will have.
- **Attributes**: These are the properties or characteristics that objects of the class will have. For example, in a `Car` class, attributes might include `color`, `model`, and `year`.
- **Methods**: These are the actions or behaviors that objects of the class can perform. For example, a `Car` class might have methods like `startEngine()`, `stopEngine()`, and `displayDetails()`.

  **Analogy**: Think of a class as an architectural blueprint for a house. The blueprint itself is not a house, but it contains all the information needed to build one.

```
public class Car {
    String color;  // Attribute
    String model;  // Attribute
    int year;      // Attribute
```

```
    // Constructor to initialize a Car object
    public Car(String color, String model, int year) {
        this.color = color;
        this.model = model;
        this.year = year;
    }

    // Method to display car details
    public void displayDetails() {
        System.out.println("Car Model: " + model);
        System.out.println("Color: " + color);
        System.out.println("Year: " + year);
    }
}
```

## 2. Object as an Entity

- **Object**: An object is a concrete instance of a class, created using the `new` keyword. When you create an object, you bring the blueprint (class) to life. The object occupies space in memory and contains specific values for the attributes defined by the class.

  **Analogy**: Continuing the house analogy, if the class is the blueprint, then an object is an actual house built according to that blueprint. The house has specific colors, dimensions, and features.

The `new` keyword in Java is used to create a new object of a class. Here's a step-by-step explanation of how the `new` keyword works to create an object:

### 1. Memory Allocation

- When you use the `new` keyword, Java allocates memory on the heap to store the object. The amount of memory allocated depends on the size of the attributes (fields) defined in the class.
- For example, if you create a new `Car` object, memory is allocated for its `color`, `model`, and `year` attributes, along with any other instance variables.

### 2. Constructor Call

- After memory is allocated, the constructor of the class is called. The constructor is a special method that initializes the object. It sets up the initial state of the object by assigning values to its fields.
- If you pass parameters to the `new` keyword (e.g., `new Car("Red", "Toyota Corolla", 2020)`), these parameters are passed to the constructor to initialize the object with specific values.
- If no constructor is explicitly defined, Java provides a default constructor that initializes fields with default values (e.g., `0` for integers, `null` for objects, etc.).

### 3. Object Initialization

- Inside the constructor, the fields of the object are initialized with the values passed as arguments, or with default values if no arguments are provided.
- For example, in the `Car` class, the `color`, `model`, and `year` attributes are set based on the values passed to the constructor.

### 4. Returning the Reference

- After the object is created and initialized, the `new` keyword returns a reference to the object. This reference is usually stored in a variable.
- This reference is what you use to interact with the object (e.g., calling methods, accessing fields).

## 5. Garbage Collection

- The object created with `new` stays in memory as long as there are references pointing to it. When there are no more references to the object, the Java garbage collector automatically frees the memory used by the object, making it available for new objects.

### Example in Action

Consider the following example:

```
public class Car {
    String color;
    String model;
    int year;

    // Constructor to initialize the object
    public Car(String color, String model, int year) {
        this.color = color;
        this.model = model;
        this.year = year;
    }
}

// In your main method:
Car car1 = new Car("Red", "Toyota Corolla", 2020);
```

### Step-by-Step Breakdown:

1. **Memory Allocation**:
   - When `new Car("Red", "Toyota Corolla", 2020)` is executed, Java allocates memory on the heap to hold the `Car` object.
2. **Constructor Call**:
   - The `Car` constructor is called with the parameters `"Red"`, `"Toyota Corolla"`, and `2020`.
3. **Object Initialization**:
   - Inside the constructor, the fields `color`, `model`, and `year` of the `Car` object are set to `"Red"`, `"Toyota Corolla"`, and `2020`, respectively.
4. **Returning the Reference**:
   - The `new` keyword returns a reference to the newly created `Car` object. This reference is stored in the variable `car1`.

### Summary:

- The `new` keyword is essential in object-oriented programming as it allows you to create instances of classes.
- It performs the crucial steps of allocating memory, initializing the object via the constructor, and then returning a reference to the object for further use in your program.

Without the `new` keyword, you wouldn't be able to create new objects in Java, and therefore, you wouldn't be able to utilize the object-oriented capabilities of the language.

```
Car car1 = new Car("Red", "Toyota Corolla", 2020);  // Object creation
```

- Here, `new Car("Red", "Toyota Corolla", 2020)` creates an object of the `Car` class. This object is a specific car with the color "Red", model "Toyota Corolla", and year 2020.

## 3. Instance as a Reference

- **Instance**: An instance is the reference to the object created from the class. It is the variable that points to the memory location where the object is stored. When you declare a variable and assign it the result of `new`, that variable becomes an instance of the class.

  **Analogy**: If the object is the house, the instance is the address or name you give to that house, so you can find it later. For example, `car1` is the instance that refers to the specific `Car` object you created.

  ```
  Car car1 = new Car("Red", "Toyota Corolla", 2020);  // car1 is the instance
  ```

  - In this line, `car1` is the instance of the `Car` class, pointing to the specific `Car` object with the attributes you provided.

## How They Relate to Each Other

- **Class as a Blueprint**:
  - The class defines what an object will look like and how it will behave. It's a static template that does not represent any specific object but can be used to create many objects.
- **Object as an Entity**:
  - The object is the actual representation of the class in memory, with real values for the attributes. It is a concrete manifestation of the class.
- **Instance as a Reference**:
  - The instance is the name or reference you use to interact with the object. It allows you to access and manipulate the object in your code.

### Example Together

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car car1 = new Car("Red", "Toyota Corolla", 2020);

        // Using the instance to access the object's method
        car1.displayDetails();
    }
}
```

- **Class (`Car`)**: Defines what every car object will look like (attributes like `color`, `model`, `year`) and how it will behave (methods like `displayDetails()`).
- **Object (`new Car(...)`)**: The actual car with the color "Red", model "Toyota Corolla", and year 2020 is created in memory.
- **Instance (`car1`)**: The reference to this specific car object, which you use to call methods like `displayDetails()` and access its attributes.

This clear connection between the class, object, and instance is fundamental to understanding how object-oriented programming (OOP) works in Java.

## What is an "Instance"?

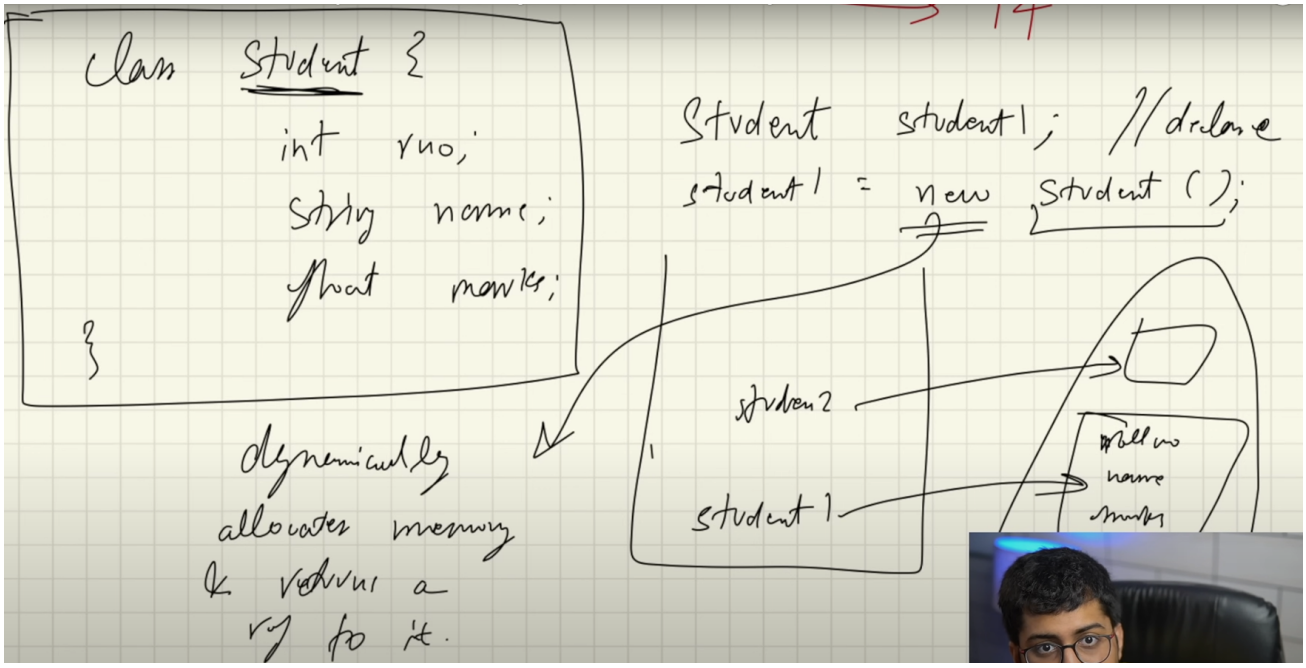
- **Instance**: In object-oriented programming, an instance refers to a specific object created from a class. A class is like a blueprint, and an instance is an actual object built using that blueprint.

### Example with the `Stack` Class:

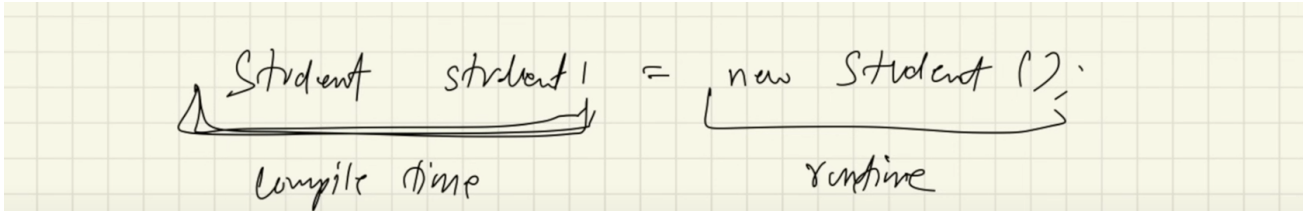
- **Class (`Stack`)**: The `Stack` class in Java is a blueprint that defines the properties (like the underlying data structure) and behaviors (like `push()`, `pop()`, `peek()`) that any stack should have.
- **Instance (`new Stack<>()`)**: When you write `new Stack<>()`, you're using the `Stack` class to create a specific object that behaves like a stack. This object is an "instance" of the `Stack` class.

### Memory Allocation:

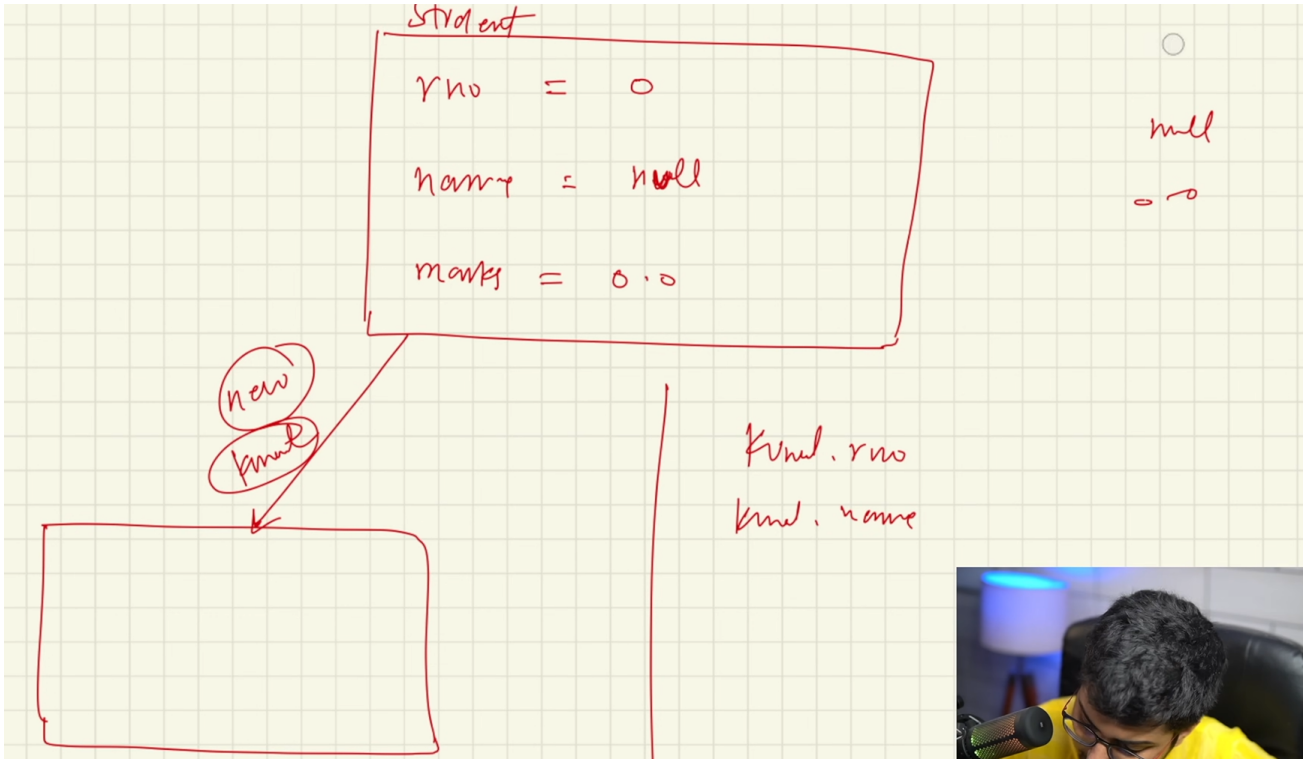
- **Memory Allocation**: When you create a new instance using `new Stack<>()`, Java allocates space in memory to store this stack. The stack's data (like the integers you push onto it) will be stored in this memory space.

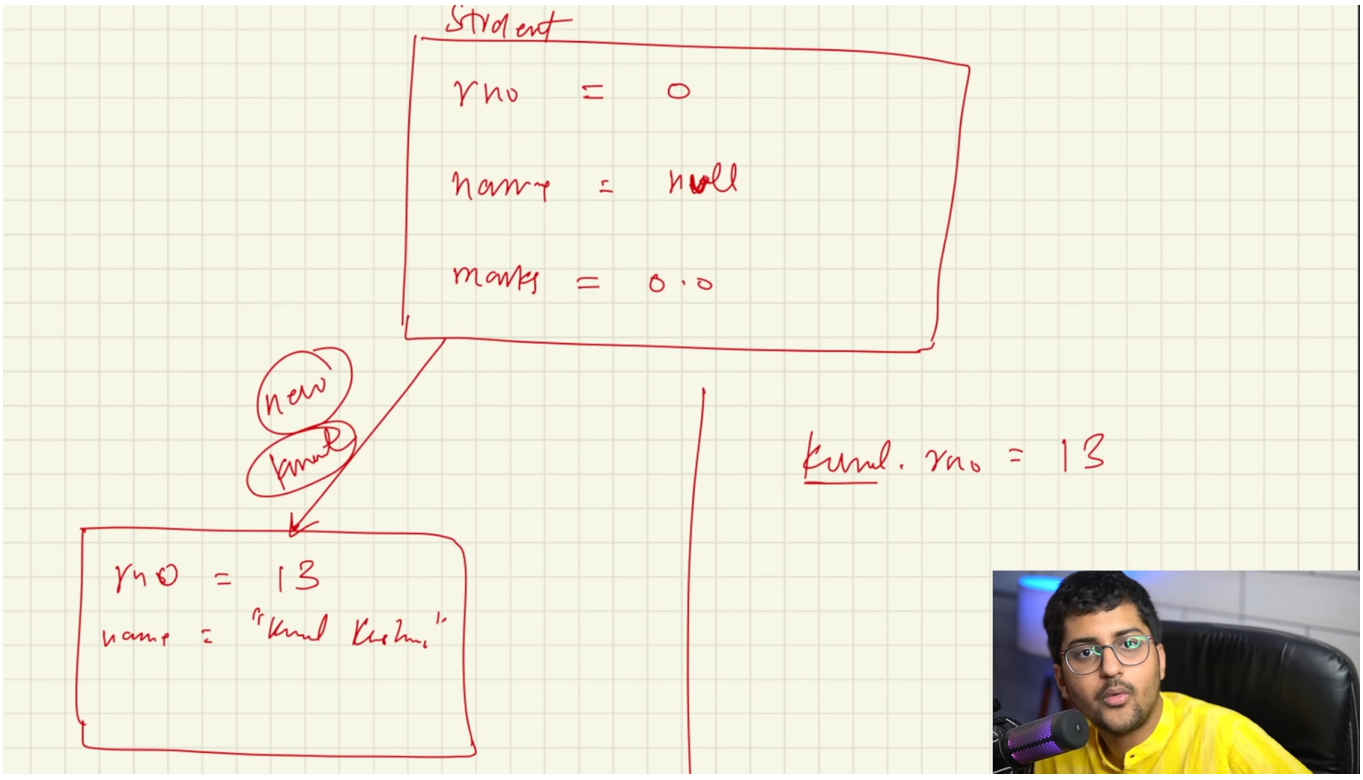



**Object will be created in compile time and Properties are created in runtime(Here only it will allocate memory)**



* If there is no Value assign to the properties it will allocate the default values with respect to the datatype.




** By Passing parameters through the constructor.we can assign values to the properties of the object;

Student student1 = new Student();

compile time — runtime

Constructor is special function, that runs when you create an object and it allocates some variables.

Student Kunal =

new Student (13, "Kunalkushwaha", 84.3);

special type of function in the class.

→ bind these arguments with the object.

```java
class Student {
    int rno;
    String name;
    float marks = 90;

    // we need a way to add the values of the above
    // properties object by object

    // we need one word to access every object

    Student () {
        this.rno = 13;
        this.name = "Kunal Kushwaha";
        this.marks = 88.5f;
    }

}
```

## Creating a object using another object

```java
Student random = new Student(kunal);
```

```java
Student  (Student other) {
    this.name = other.name;
    this.rno = other.rno;
    this.marks = other.marks;
}
```

## Constructor overloading

- If you create object in constructor with parameter inside it .then it will automatically call the second constructor named as student with parameters.

- create normally without it student constructor which does not have parameters.

```
Student () {
    this.rno = 13;
    this.name = "Kunal Kushwaha";
    this.marks = 88.5f;
}


// Student arpit = new Student(17, "Arpit", 89.7f);
// here, this will be replaced with arpit
Student (int rno, String name, float marks) {
    this.rno = rno;
    this.name = name;
    this.marks = marks;
}
```

## Class Structure

```
public static void main(String args) {
}
static class lens{
    String company;
    String Focus;
    Double Price;
    String Colour;
    boolean isprime;
    lens(String company,String Focus ,Double Price,String colour,boolean isprime){
        this.company=company;
        this.Focus=Focus;
        this.Price=Price;
        this.Colour=Colour;
        this.isprime=isprime;
    }

}
```

## Object Creation

```
public static void main(String[] args) {
    lens L1=new lens("Sony","30MM",3234.99,"RED",true);
    lens L2=new lens("Canon","40MM",40.00,"BLACK",true);
    lens L3=new lens("Gopro","10MM",3500.00,"BLACK",false);
```

```
}
```

### final Code

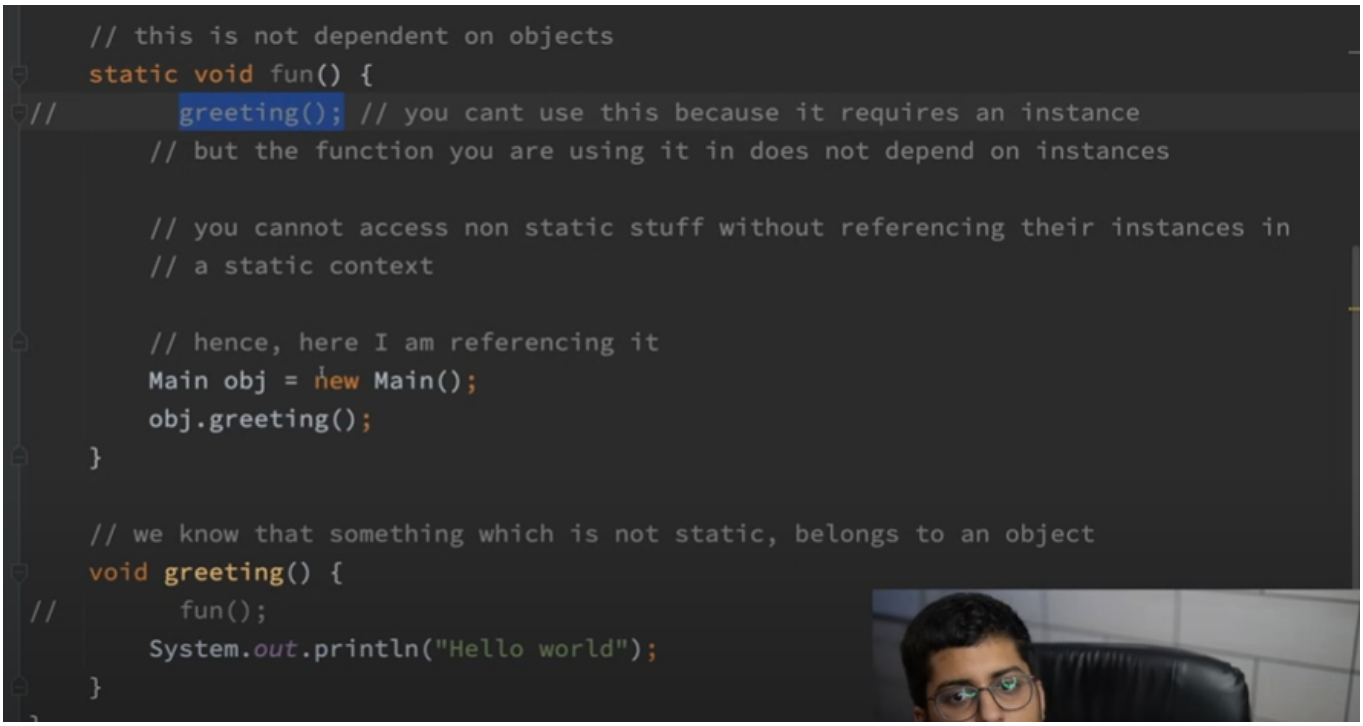
```
public static void main(String[] args) {
    lens L1=new lens("Sony","30MM",3234.99,"RED",true);
    lens L2=new lens("Canon","40MM",40.00,"BLACK",true);
    lens L3=new lens("Gopro","10MM",3500.00,"BLACK",false);
    System.out.println("Lens 1->"+L1.company+" FocalLength-> "+L1.Focus+",  Price->
"+L1.Price+",  Colour-> "+L1.Colour+",  isPrime-> "+L1.isprime);
    System.out.println("Lens 2->"+L2.company+" FocalLength-> "+L2.Focus+",  Price->
"+L2.Price+",  Colour-> "+L2.Colour+",  isPrime-> "+L2.isprime);
    System.out.println("Lens 3->"+L3.company+" FocalLength-> "+L3.Focus+",  Price->
"+L3.Price+",  Colour-> "+L3.Colour+",  isPrime-> "+L3.isprime);

}
static class lens{
    String company;
    String Focus;
    Double Price;
    String Colour;
    boolean isprime;
    lens(String company,String Focus ,Double Price,String colour,boolean isprime){
        this.company=company;
        this.Focus=Focus;
        this.Price=Price;
        this.Colour=Colour;
        this.isprime=isprime;
    }
}
```

```
Lens 1->Sony FocalLength-> 30MM,  Price-> 3234.99,  Colour-> null,  isPrime-> true
Lens 2->Canon FocalLength-> 40MM,  Price-> 40.0,  Colour-> null,  isPrime-> true
Lens 3->Gopro FocalLength-> 10MM,  Price-> 3500.0,  Colour-> null,  isPrime-> false
```

# Static keyword

The `static` keyword in Java is used for memory management primarily. It can be applied to variables, methods, blocks, and nested classes. Here's an in-depth look at its usage and implications:

Static only access static data ,Not an Non-static Data(you cant use this because it requires an instance),but the function you are using it in does not depend on instances




## Static Variables

A static variable is shared among all instances of a class. It's associated with the class itself rather than any particular instance of the class..

```
class Example {
    static int counter = 0; // Static variable

    Example() {
        counter++;
    }

    void display() {
        System.out.println("Counter: " + counter);
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj1 = new Example();
        Example obj2 = new Example();

        obj1.display(); // Output: Counter: 2
        obj2.display(); // Output: Counter: 2
    }
}
```

## Static Methods

***OVERRIDDING IS DEPEND ON OBJECT. BUT STATIC METHOD DOES NOT DEPEND ON OBJECT. SO STATIC CANNOT BE OVERRIDE

Static methods belong to the class rather than any instance of the class. They can be called without creating an instance of the class. Static methods can only access other static variables and static methods directly.

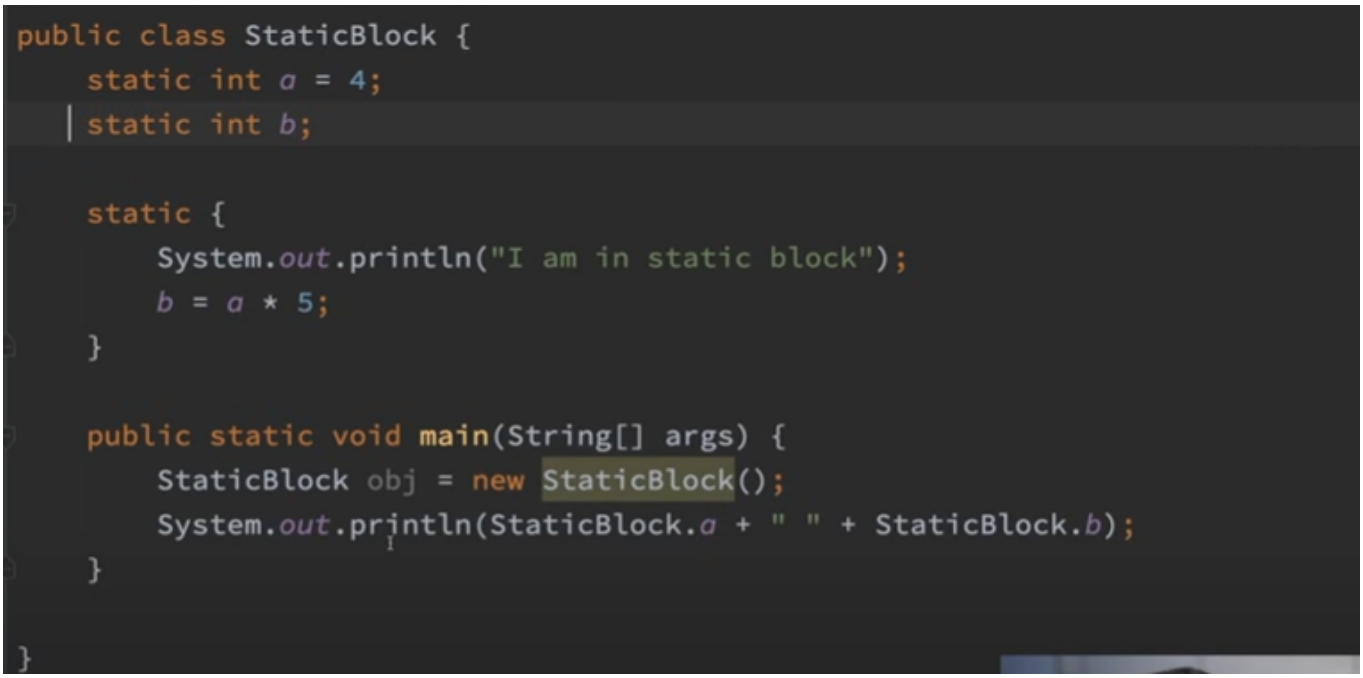
```java
class Example {
    static int counter = 0;

    static void incrementCounter() { // Static method
        counter++;
    }

    static void displayCounter() {
        System.out.println("Counter: " + counter);
    }
}

public class Main {
    public static void main(String[] args) {
        Example.incrementCounter();
        Example.incrementCounter();
        Example.displayCounter(); // Output: Counter: 2
    }
}
```

## Static Blocks

Static blocks are used for static initializations of a class. This code inside static blocks is executed once, when the class is first loaded into memory.

```java
class Example {
    static int counter;

    static { // Static block
        counter = 10;
        System.out.println("Static block executed");
    }

    static void displayCounter() {
        System.out.println("Counter: " + counter);
    }
}

public class Main {
    public static void main(String[] args) {
        Example.displayCounter(); // Output: Static block executed, Counter: 10
    }
}
```

// Static Block will only run once, when the first obj is create i.e. when the class is loaded

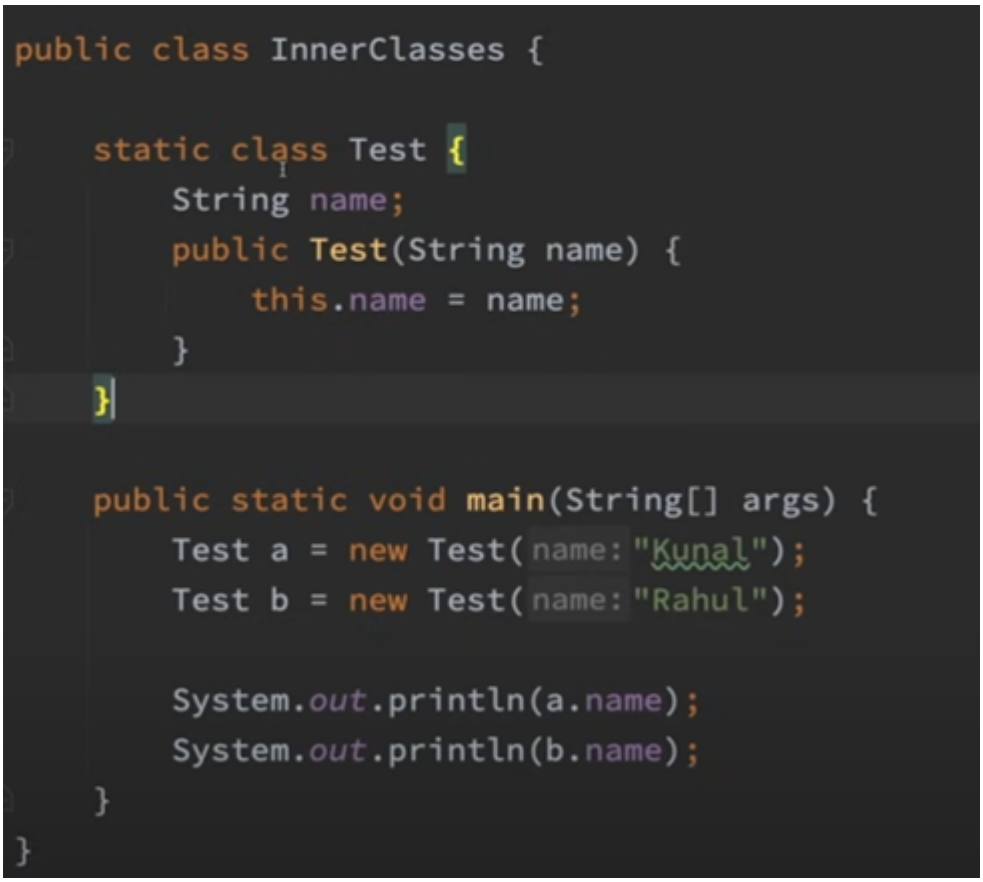


```
public class StaticBlock {
    static int a = 4;
    static int b;

    static {
        System.out.println("I am in static block");
        b = a * 5;
    }

    public static void main(String[] args) {
        StaticBlock obj = new StaticBlock();
        System.out.println(StaticBlock.a + " " + StaticBlock.b);
    }

}
```

## Static Classes (Nested Classes)

A static nested class is a static class defined inside another class. It can access static members of the outer class but cannot directly access non-static members.

```
class OuterClass {
    static int outerStaticVar = 10;

    static class StaticNestedClass {
        void display() {
            System.out.println("Outer static variable: " + outerStaticVar);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        nestedObject.display(); // Output: Outer static variable: 10
    }
}
```




### Summary of Static Keyword Usage:

| Feature | Description |
|---|---|
| **Static Variables** | Variables shared among all instances of a class. |
| **Static Methods** | Methods that belong to the class rather than any instance, accessible without creating an instance. |
| **Static Blocks** | Blocks used for static initialization of a class, executed once when the class is first loaded. |
| **Static Classes** | Nested classes marked as static, which can access the outer class's static members but not non-static members. |

By using the `static` keyword, Java provides a mechanism to create class-level attributes and methods that can be accessed without needing to instantiate objects, improving memory management and efficiency in certain scenarios.
***By Static we can inherit but cannot be override
override depends on objects. but static does not depend object.so static cannot be override

# Singleton Class

```java
public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();

        Singleton obj2 = Singleton.getInstance();

        Singleton obj3 = Singleton.getInstance();

        // all 3 ref variables are pointing to just one object
    }
}
```

```java
public class Singleton {
    private Singleton () {

    }

    private static Singleton instance;

    public static Singleton getInstance() {
        // check whether 1 obj only is created or not
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }

}
```

A **singleton class** in Java is a design pattern that restricts the instantiation of a class to one single instance. This is useful when exactly one object is needed to coordinate actions across the system, such as in cases where a single point of control or a shared resource is required.

## Key Characteristics of a Singleton Class:

1. **Single Instance**: Only one instance of the class is created, and this instance is reused across the application.
2. **Global Access**: The single instance is globally accessible, meaning that it can be accessed from any part of the program.
3. **Controlled Instantiation**: The class controls the creation of the instance, typically using a private constructor to prevent direct instantiation from outside the class.

## Implementation of a Singleton Class

Here's a simple example of how a singleton class can be implemented in Java:

```
public class Singleton {
    // Step 1: Create a private static variable to hold the single instance of the class.
    private static Singleton instance;

    // Step 2: Make the constructor private to prevent instantiation from other classes.
    private Singleton() {
        // Private constructor to prevent instantiation
    }

    // Step 3: Provide a public static method that returns the single instance of the class.
    public static Singleton getInstance() {
        if (instance == null) {
            // Lazy initialization: instance is created only when it is needed.
            instance = new Singleton();
        }
        return instance;
    }

    // Other methods of the singleton class
    public void showMessage() {
        System.out.println("Hello from Singleton!");
    }
}
```

## How It Works:

1. **Private Static Instance**:
   - `private static Singleton instance;` is a static variable that holds the single instance of the class. Since it's static, it's shared across all instances of the class (but here, there will only be one).
2. **Private Constructor**:
   - `private Singleton() {}` is the constructor that is marked private, so no other classes can instantiate the `Singleton` class directly using the `new` keyword.
3. **Public Static Method**:
   - `public static Singleton getInstance() {}` is a public static method that returns the single instance of the class. If the instance doesn't exist yet ( `instance == null` ), it creates it. This ensures that only one instance is created.
4. **Lazy Initialization**:
   - The instance is created only when it is needed, known as lazy initialization. This can save resources if the instance is never used.

## Example Usage

```
public class Main {
    public static void main(String[] args) {
        // Get the only object available
        Singleton singleton = Singleton.getInstance();

        // Show a message
```

```
        singleton.showMessage(); // Output: Hello from Singleton!
    }
}
```

## Key Points to Consider

1. **Thread Safety**: In a multithreaded environment, you must ensure that the singleton instance is created only once, even when multiple threads try to access it simultaneously. This can be achieved using synchronization.
2. **Eager Initialization**: An alternative to lazy initialization is eager initialization, where the instance is created at the time of class loading, like this:

```
private static final Singleton instance = new Singleton();

public static Singleton getInstance() {
    return instance;
}
```

3. **Reflection**: Singleton can be broken using reflection because private constructors can be accessed using reflection. This can be prevented by throwing an exception if the constructor is already called once.
4. **Serialization**: When dealing with serialization, the singleton pattern can be broken if the class does not implement the `readResolve` method. Implementing `readResolve` can prevent a new instance from being created during deserialization.

## When to Use Singleton

- **Resource Management**: When you need to manage a shared resource, like a database connection pool.
- **Logging**: Implementing a logging class that is used throughout the application.
- **Configuration Settings**: Managing configuration settings or global state in an application.

The Singleton pattern is a widely used design pattern, but it should be used carefully to avoid issues like unnecessary tight coupling and difficulties in testing.

# Final Keyword

The `final` keyword in Java is a modifier that can be applied to variables, methods, and classes, and it imposes specific restrictions on how they can be used. Here's a detailed explanation of how the `final` keyword works in each context:

## 1. Final Variables

When a variable is declared as `final`, it means that the variable can only be assigned once. After its initial assignment, the value of a `final` variable cannot be changed.

### Example:

```
public class Example {
    public static void main(String[] args) {
        final int x = 10;
        x = 20;  // Compile-time error: Cannot assign a value to final variable 'x'
    }
}
```

### Key Points:

- **Primitive Types**: For primitive types (like `int`, `float`, `char`), the value itself cannot be changed after assignment.
- **Reference Types**: For reference types (like objects or arrays), the reference cannot be changed to point to a different object, but the object's internal state can still be modified.

```
final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");  // This is allowed
sb = new StringBuilder("New");  // Compile-time error: Cannot assign a value to final variable 'sb'
```

## 2. Final Methods

When a method is declared as `final`, it means that the method cannot be overridden by subclasses. This is useful when you want to prevent altering the behavior of a method in derived classes.

### Example:

```
class Parent {
    public final void display() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    public void display() {  // Compile-time error: Cannot override the final method from Parent
        System.out.println("Trying to override.");
    }
}
```

### Key Points:

- **Prevent Overriding**: `final` methods cannot be overridden in subclasses, ensuring that the implementation remains unchanged.

## 3. Final Classes

When a class is declared as `final`, it means that the class cannot be subclassed. No other class can extend a `final` class. This is typically used to prevent inheritance for security reasons or to ensure that the class's behavior cannot be altered through subclassing.

### Example:

```
final class FinalClass {
    public void display() {
        System.out.println("This is a final class.");
    }
}

class SubClass extends FinalClass {  // Compile-time error: Cannot subclass the final class
```

```
'FinalClass'
}
```

### Key Points:

- **No Inheritance**: `final` classes cannot be extended, which makes them the last class in their inheritance hierarchy.
- **Immutable Classes**: Many immutable classes in Java, such as `String`, are declared as `final` to prevent their behavior from being altered.

## 4. Final Parameters

The `final` keyword can also be applied to method parameters, meaning that the parameter cannot be reassigned within the method.

### Example:

```
public void method(final int param) {
    param = 10;  // Compile-time error: Cannot assign a value to final parameter 'param'
}
```

### Key Points:

- **Immutable Parameters**: Using `final` with method parameters ensures that the parameter value cannot be changed within the method, which can help avoid unintended side effects.

## Summary

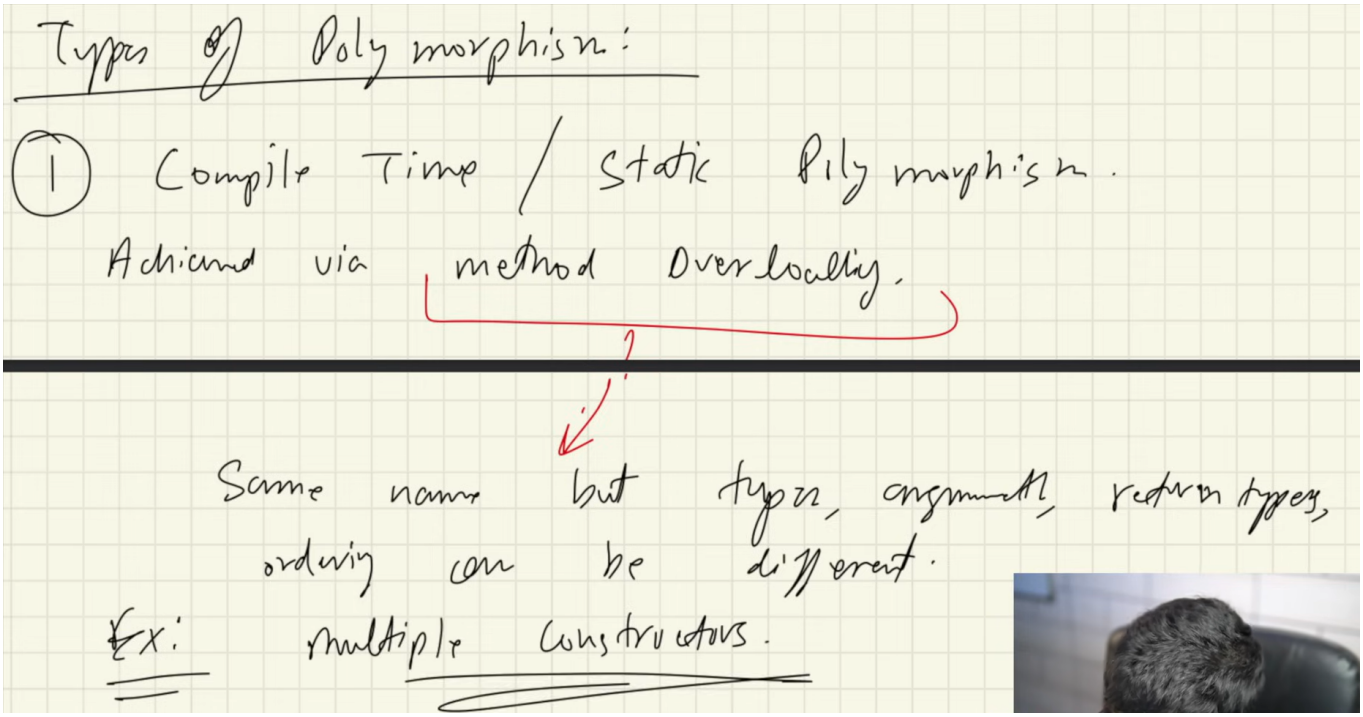
- **Final Variables**: Can only be assigned once; after that, their value (for primitives) or reference (for objects) cannot be changed.
- **Final Methods**: Cannot be overridden by subclasses, preserving the method's original behavior.
- **Final Classes**: Cannot be subclassed, preventing inheritance.
- **Final Parameters**: Cannot be reassigned within the method, ensuring the original value remains unchanged during execution.

The `final` keyword is a powerful tool for ensuring immutability, enforcing design constraints, and improving code safety and readability in Java.

# Polymorphism

**Polymorphism is one of the fundamental concepts of object-oriented programming (OOP). It allows objects to be treated as instances of their parent class rather than their actual class. Polymorphism enables a single function, operator, or object to behave differently in different contexts.**
***There are two main types of polymorphism in OOP:**
**1.compile-time polymorphism(Achieved by defining multiple methods with the same name but different parameters. The decision about which method to invoke is made at compile time.) and**
**2.run-time polymorphism.(Achieved by defining a method in the subclass with the same signature as a method in the superclass. The decision about which method to invoke is made at run time, based on the actual object type.)**

# Function Overloading (Compile-Time Polymorphism)&&static Polymorphism.




Function overloading, also known as compile-time polymorphism, occurs when multiple functions in the same scope have the same name but different parameters (number, type, or both). The appropriate function is selected by the compiler at compile time based on the arguments passed to the function.

## Characteristics:

- **Same method name**: Multiple methods with the same name.
- **Different parameter lists**: Methods must have different parameters, either in type or number.
- **Resolved at compile time**: The compiler determines which method to call based on the method signature (method name and parameter list).

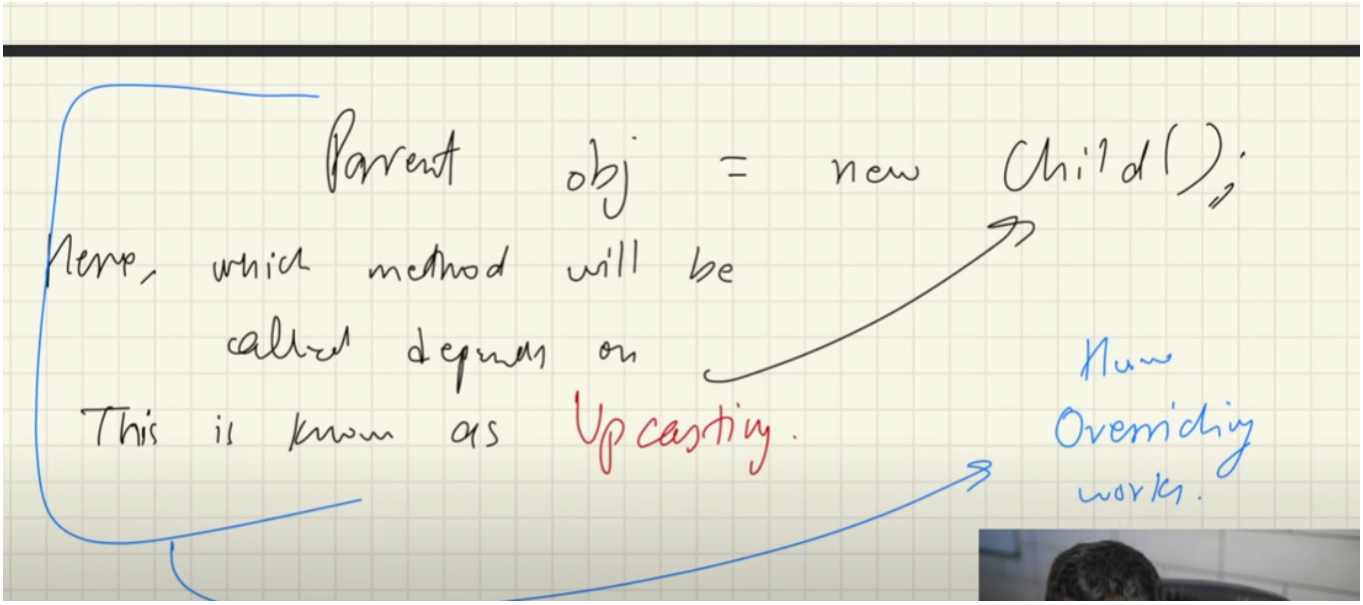
```
class MathOperations {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }
    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
    // Overloaded method to add two strings
    public String add(String a, String b) {
        return a + b;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        MathOperations mathOps = new MathOperations();
        // Calling different overloaded methods
        System.out.println(mathOps.add(5, 10));           // Calls add(int, int)
        System.out.println(mathOps.add(5, 10, 15));       // Calls add(int, int, int)
        System.out.println(mathOps.add(5.5, 10.5));       // Calls add(double, double)
        System.out.println(mathOps.add("Hello, ", "World!")); // Calls add(String, String)
    }
}
```

## Function Overriding (Run-Time Polymorphism)

***OVERRIDDING IS DEPEND ON OBJECT
Function overriding is an example of run-time polymorphism in Java. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass must have the same name, return type, and parameters as the method in the superclass.

### Characteristics:




- **Same method signature**: The method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- **Resolved at run time**: The JVM determines which method to call at run time based on the object type.
- **Inheritance**: Requires a superclass and a subclass relationship.

```java
class Animal {
    // Method to be overridden
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
```

```
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Upcasting
        Animal myCat = new Cat();  // Upcasting

        // Calls the overridden methods
        myDog.makeSound(); // Outputs: Dog barks
        myCat.makeSound(); // Outputs: Cat meows
    }
}
```

## Dynamic method dispatch

Dynamic method dispatch, also known as runtime polymorphism, is a core concept in object-oriented programming, particularly in languages like Java. It allows a program to determine at runtime which method implementation to execute, based on the actual object's type rather than the reference type.

### How Dynamic Method Dispatch Works

1. **Inheritance and Overriding**:
   - In Java, a class can inherit from another class (superclass) and override its methods in a subclass.
   - Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.
2. **Method Signature**:
   - The method signature (name, parameters, and return type) in the subclass must match the method signature in the superclass for overriding to occur.
3. **Polymorphism**:
   - Polymorphism allows a subclass object to be referenced by a superclass variable.
   - At runtime, the actual method that gets invoked is determined by the type of the object, not the type of the reference.

### Example of Dynamic Method Dispatch

Consider the following example to understand dynamic method dispatch:

```
// Superclass
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
```

```
}

// Subclass 1
class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }
}

// Subclass 2
class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        // Reference of type Animal but object of type Dog
        Animal myAnimal = new Dog();
        myAnimal.makeSound();  // Output: Dog barks

        // Reference of type Animal but object of type Cat
        myAnimal = new Cat();
        myAnimal.makeSound();  // Output: Cat meows
    }
}
```

## Detailed Explanation

1. **Class Definitions**:
   - `Animal` is a superclass with a method `makeSound`.
   - `Dog` and `Cat` are subclasses of `Animal` that override the `makeSound` method.
2. **Reference Type vs. Object Type**:
   - In the `main` method, `myAnimal` is a reference of type `Animal`.
   - `myAnimal` can refer to objects of type `Dog`, `Cat`, or `Animal`.
3. **Method Invocation**:
   - **First Assignment**: `myAnimal = new Dog();`
     - Here, `myAnimal` is assigned an object of type `Dog`. Despite the reference being of type `Animal`, the actual object is a `Dog`.
     - When `myAnimal.makeSound()` is called, Java's dynamic method dispatch mechanism determines that `makeSound` should invoke the `Dog`'s version of `makeSound`, resulting in "Dog barks".
   - **Second Assignment**: `myAnimal = new Cat();`
     - Now, `myAnimal` is assigned an object of type `Cat`.
     - When `myAnimal.makeSound()` is called again, Java determines that the `Cat`'s version of `makeSound` should be executed, resulting in "Cat meows".

## Why Dynamic Method Dispatch is Useful

1. **Flexibility**:

- It allows a program to decide at runtime which method to invoke, based on the actual object type. This enables the creation of more flexible and reusable code.

2. **Code Maintainability**:
   - It supports a clean separation of interface and implementation. Changes in the subclass implementations do not require changes in the code that uses the superclass references.
3. **Extensibility**:
   - New classes can be introduced without modifying existing code that uses superclass references. This adheres to the Open/Closed Principle in object-oriented design.

### Summary

Dynamic method dispatch is a powerful feature of object-oriented programming that enables a program to determine the method to invoke at runtime based on the actual object type. It enhances flexibility, maintainability, and extensibility of code, making it a fundamental concept in designing polymorphic behavior in applications.

## Superclass and Subclass

- **Superclass**: The superclass is the parent class from which other classes inherit. In this case, `Animal` is the superclass.

```
class Animal {
    // Method to be overridden
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
```

- **Subclass**: The subclass is the child class that inherits from the superclass and can provide a specific implementation of the methods defined in the superclass. In this case, `Dog` and `Cat` are subclasses of `Animal`.

```
class Dog extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    // Overriding the makeSound method
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}
```

## Explanation of Overriding

When the method `makeSound` is defined in the `Animal` class, it is the method that will be overridden by the subclasses. The `Dog` and `Cat` classes provide their specific implementations of the `makeSound` method.

### Example Usage

```
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();  // Upcasting: Dog object is treated as an Animal
        Animal myCat = new Cat();  // Upcasting: Cat object is treated as an Animal
        // Calls the overridden methods
        myDog.makeSound(); // Outputs: Dog barks
        myCat.makeSound(); // Outputs: Cat meows
    }
}
```

- `Animal myDog = new Dog();`: Here, a `Dog` object is created but referenced as an `Animal`. This is an example of upcasting, where a subclass object is treated as an instance of its superclass.
- `Animal myCat = new Cat();`: Similarly, a `Cat` object is created but referenced as an `Animal`.

When `myDog.makeSound()` and `myCat.makeSound()` are called, the JVM determines at runtime which version of the `makeSound` method to execute based on the actual object type (`Dog` or `Cat`), not the reference type (`Animal`). This is run-time polymorphism (function overriding) in action.
***Overriding can be avoid using final keyword

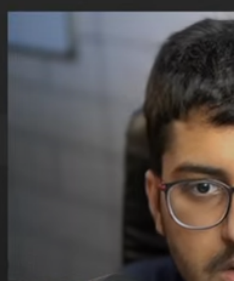
# Inheritance

**Inheritance in Java is a mechanism where one class acquires the properties (fields) and behaviors (methods) of another class. The class that inherits the properties is known as the subclass (or child class, or derived class), and the class from which properties are inherited is known as the superclass (or parent class, or base class). There are several types of inheritance in Java:**
**.***Inheritance can be avoid using final keyword(if the class or method to be final)

```
package com.kunal.properties.inheritance;

public class Main {
    public static void main(String[] args) {
//        Box box1 = new Box(4.6, 7.9, 9.9);
//        Box box2 = new Box(box1);
//        System.out.println(box1.l + " " + box1.w + " " + box1.h);

        BoxWeight box3 = new BoxWeight();
        BoxWeight box4 = new BoxWeight( l: 2,  h: 3,  w: 4,  weight: 8);
        System.out.println(box3.h + " " + box3.weight);
    }
}
```

```java
package com.kunal.properties.inheritance;

public class BoxWeight extends Box{
    double weight;

    public BoxWeight() {
        this.weight = -1;
    }

    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w); // what is this? call the parent class constructor
        // used to initialise values present in parent class
        this.weight = weight;
    }
}
```

```java
package com.kunal.properties.inheritance;

public class Box {
    double l;
    double h;
    double w;

    Box () {
        this.h = -1;
        this.l = -1;
        this.w = -1;
    }

    // cube
    Box (double side) {
        this.w = side;
        this.l = side;
        this.h = side;
    }

    Box(double l, double h, double w) {
        this.l = l;
```



```
Box box5 = new BoxWeight(l:2, h:3, w:4, weight:8);
System.out.println(box5.w);

// there are many variables in both parent and child classes
// you are given access to variables that are in the ref type i.e. BoxWeight
// hence, you should have access to weight variable
// this also means, that the ones you are trying to access should be initialised
// but here, when the obj itself is of type parent class, how will you call the cons
// this is why error
BoxWeight box6 = new Box(l:2, h:3, w:4);
System.out.println(box6);
```

```
Box(double l, double h, double w) {
    this.l = l;
    this.h = h;
    this.w = w;
}

Box(Box old) {
    this.h = old.h;
    this.l = old.l;
    this.w = old.w;
}

public void information() {
    System.out.println("Running the box");
}
```

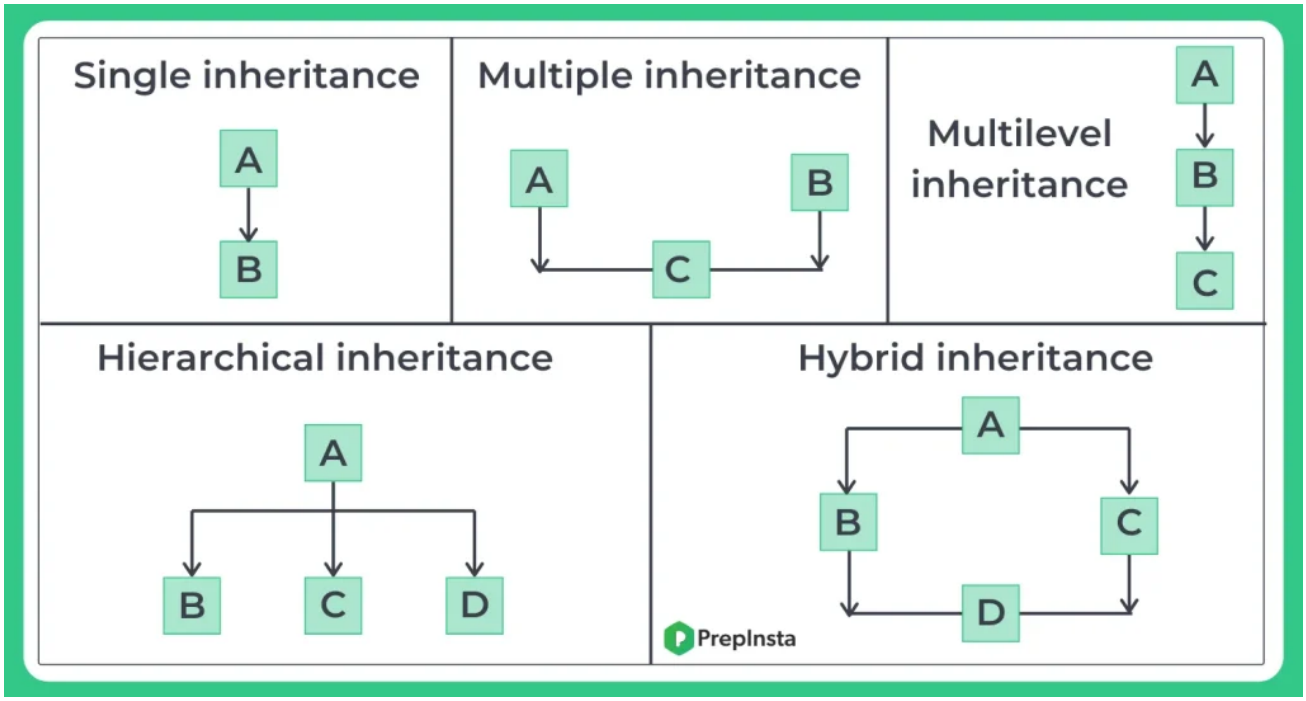
**Super keyword**

```
public BoxPrice(double side, double weight, double cost) {
    super(side, weight);
    this.cost = cost;
}
```

```
BoxWeight(double side, double weight) {
    super(side);
    this.weight = weight;
}
```

```
Box (double side) {
    // super(); Object class
    this.w = side;
    this.l = side;
    this.h = side;
}
```

- **Single Level Inheritance**
- **Multi-Level Inheritance**
- **Multiple Inheritance
- **Hierarchical Inheritance**
- **Hybrid Inheritance




## Single-level Inheritance

**Single level inheritance is when a class inherits from one superclass directly. In this case, there is only one level of inheritance.

```java
// Superclass
class Shape {
    public void area() {
        System.out.println("Displays area");
    }
}

// Subclass
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println(0.5 * l * h);
    }
}

// Main class to test inheritance
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the subclass
        Triangle t = new Triangle();

        // Calling method from the superclass
        t.area(); // Outputs: Displays area

        // Calling method from the subclass
        t.area(10, 5); // Outputs: 25.0
    }
}
```

## Explanation

### Superclass (Shape)

The `Shape` class is the superclass. It has one method `area()` which prints "Displays area".

```java
class Shape {
    public void area() {
        System.out.println("Displays area");
    }
}
```

### Subclass (Triangle)

The `Triangle` class is the subclass that extends the `Shape` class. It inherits the `area()` method from `Shape` and also has an overloaded method `area(int l, int h)` that calculates the area of a triangle.

```java
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println(0.5 * l * h);
```

```
    }
}
```

### Main Class (OOPS)

The `OOPS` class contains the `main` method to test the inheritance.

```
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the subclass
        Triangle t = new Triangle();

        // Calling method from the superclass
        t.area(); // Outputs: Displays area

        // Calling method from the subclass
        t.area(10, 5); // Outputs: 25.0
    }
}
```

## Key Points

- **Superclass (`Shape`)**: Contains the method `area()`.
- **Subclass (`Triangle`)**: Inherits the method `area()` from `Shape` and defines an additional method `area(int l, int h)`.
- **Inheritance**: The `Triangle` class inherits from the `Shape` class, allowing it to use the `area()` method from `Shape` as well as its own `area(int l, int h)` method.
- **Method Overloading**: The `Triangle` class overloads the `area` method, providing a different implementation for calculating the area of a triangle.

## Single-Level Inheritance Benefits

- **Code Reusability**: The subclass reuses the code of the superclass.
- **Method Overloading**: Subclasses can have methods with the same name as those in the superclass but with different parameters.

# Multi-Level Inheritance

Multi-level inheritance is a type of inheritance where a class is derived from another class, which is also derived from another class, forming a chain of inheritance. In other words, there is more than one level of inheritance.

In multi-level inheritance, a class (child class) inherits from another class (parent class), which itself is a subclass of another class (grandparent class).

## Example of Multi-Level Inheritance in Java

Let's extend the previous example to illustrate multi-level inheritance.

```
// Grandparent class (superclass)
class Shape {
    public void area() {
        System.out.println("Displays area");
```

```
    }
}

// Parent class (subclass of Shape)
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println(0.5 * l * h);
    }
}

// Child class (subclass of Triangle)
class EquilateralTriangle extends Triangle {
    @Override
    public void area(int l, int h) {
        System.out.println((l * h) / 2);
    }
}

// Main class to test inheritance
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the child class
        EquilateralTriangle eqTriangle = new EquilateralTriangle();

        // Calling method from the grandparent class
        eqTriangle.area(); // Outputs: Displays area

        // Calling overridden method from the child class
        eqTriangle.area(10, 5); // Outputs: 25
    }
}
```

## Explanation

### Grandparent Class (Shape)

The `Shape` class is the top-level superclass. It has one method `area()` which prints "Displays area".

```
class Shape {
    public void area() {
        System.out.println("Displays area");
    }
}
```

### Parent Class (Triangle)

The `Triangle` class extends the `Shape` class, inheriting its `area()` method. It also defines an overloaded method `area(int l, int h)` that calculates the area of a triangle using the formula `0.5 * l * h`.

```
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println(0.5 * l * h);
```

```
    }
}
```

### Child Class (EquilateralTriangle)

The `EquilateralTriangle` class extends the `Triangle` class, inheriting its `area(int l, int h)` method and overriding it to provide a specific implementation for calculating the area of an equilateral triangle using the formula `(l * h) / 2`.

```
class EquilateralTriangle extends Triangle {
    @Override
    public void area(int l, int h) {
        System.out.println((l * h) / 2);
    }
}
```

### Main Class (OOPS)

The `OOPS` class contains the `main` method to test the multi-level inheritance.

```
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the child class
        EquilateralTriangle eqTriangle = new EquilateralTriangle();

        // Calling method from the grandparent class
        eqTriangle.area(); // Outputs: Displays area

        // Calling overridden method from the child class
        eqTriangle.area(10, 5); // Outputs: 25
    }
}
```

## Key Points

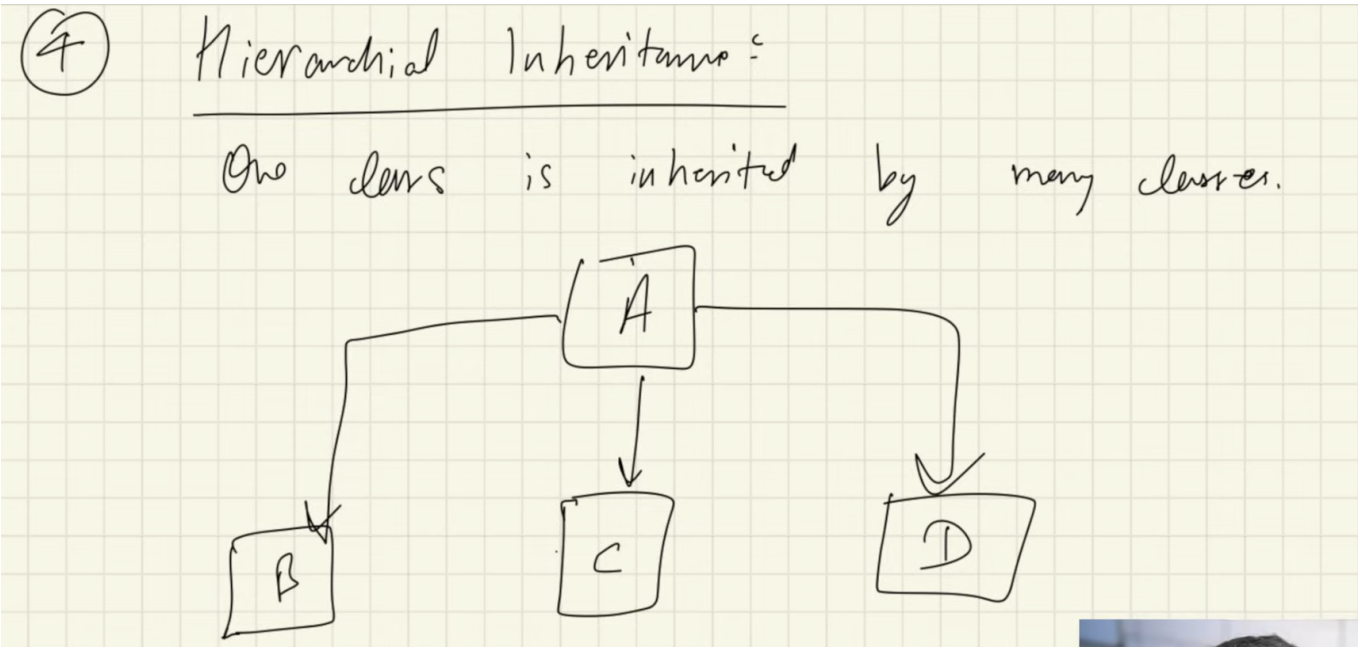
- **Grandparent Class (`Shape`)**: Contains the method `area()`.
- **Parent Class (`Triangle`)**: Inherits from `Shape` and defines an additional method `area(int l, int h)`.
- **Child Class (`EquilateralTriangle`)**: Inherits from `Triangle` and overrides the method `area(int l, int h)` to provide a specific implementation.
- **Multi-Level Inheritance**: `EquilateralTriangle` inherits from `Triangle`, which in turn inherits from `Shape`, forming a chain of inheritance.
- **Method Overriding**: The `EquilateralTriangle` class overrides the `area(int l, int h)` method to provide its specific calculation for the area.

## Benefits of Multi-Level Inheritance

- **Code Reusability**: Allows for the reuse of code across multiple levels of inheritance.
- **Enhanced Functionality**: Subclasses can enhance or modify the behavior of methods inherited from their parent classes.

# Hierarchical Inheritance

Hierarchical inheritance is a type of inheritance where multiple classes inherit from a single superclass. This means that a single parent class can have multiple child classes, each inheriting the properties and methods of the parent class.




## Example of Hierarchical Inheritance in Java

Let's extend your example to illustrate hierarchical inheritance.

```
// Superclass
class Shape {
    public void area() {
        System.out.println("Displays area");
    }
}

// Subclass 1
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println((0.5) * l * h);
    }
}

// Subclass 2
class Circle extends Shape {
    public void area(int r) {
        System.out.println((3.14) * r * r);
    }
}

// Main class to test inheritance
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the Triangle subclass
        Triangle triangle = new Triangle();
        // Create an object of the Circle subclass
        Circle circle = new Circle();
```

```
        // Calling method from the superclass using Triangle object
        triangle.area(); // Outputs: Displays area

        // Calling method from the Triangle subclass
        triangle.area(10, 5); // Outputs: 25.0

        // Calling method from the superclass using Circle object
        circle.area(); // Outputs: Displays area

        // Calling method from the Circle subclass
        circle.area(7); // Outputs: 153.86
    }
}
```

## Explanation

### Superclass (Shape)

The `Shape` class is the superclass. It has one method `area()` which prints "Displays area".

```
class Shape {
    public void area() {
        System.out.println("Displays area");
    }
}
```

### Subclass 1 (Triangle)

The `Triangle` class extends the `Shape` class, inheriting its `area()` method. It also defines an overloaded method `area(int l, int h)` that calculates the area of a triangle using the formula `0.5 * l * h`.

```
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println((0.5) * l * h);
    }
}
```

### Subclass 2 (Circle)

The `Circle` class extends the `Shape` class, inheriting its `area()` method. It also defines an overloaded method `area(int r)` that calculates the area of a circle using the formula `π * r^2`.

```
class Circle extends Shape {
    public void area(int r) {
        System.out.println((3.14) * r * r);
    }
}
```

### Main Class (OOPS)

The `OOPS` class contains the `main` method to test hierarchical inheritance.

```
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the Triangle subclass
        Triangle triangle = new Triangle();
        // Create an object of the Circle subclass
        Circle circle = new Circle();

        // Calling method from the superclass using Triangle object
        triangle.area(); // Outputs: Displays area

        // Calling method from the Triangle subclass
        triangle.area(10, 5); // Outputs: 25.0

        // Calling method from the superclass using Circle object
        circle.area(); // Outputs: Displays area

        // Calling method from the Circle subclass
        circle.area(7); // Outputs: 153.86
    }
}
```

## Key Points

- **Superclass (`Shape`)**: Contains the method `area()`.
- **Subclass 1 (`Triangle`)**: Inherits from `Shape` and defines an additional method `area(int l, int h)` to calculate the area of a triangle.
- **Subclass 2 (`Circle`)**: Inherits from `Shape` and defines an additional method `area(int r)` to calculate the area of a circle.
- **Hierarchical Inheritance**: Both `Triangle` and `Circle` classes inherit from the `Shape` class, forming a hierarchy where multiple subclasses share a common parent class.
- **Method Overloading**: Each subclass can have its own implementation of the `area` method, specific to its shape.

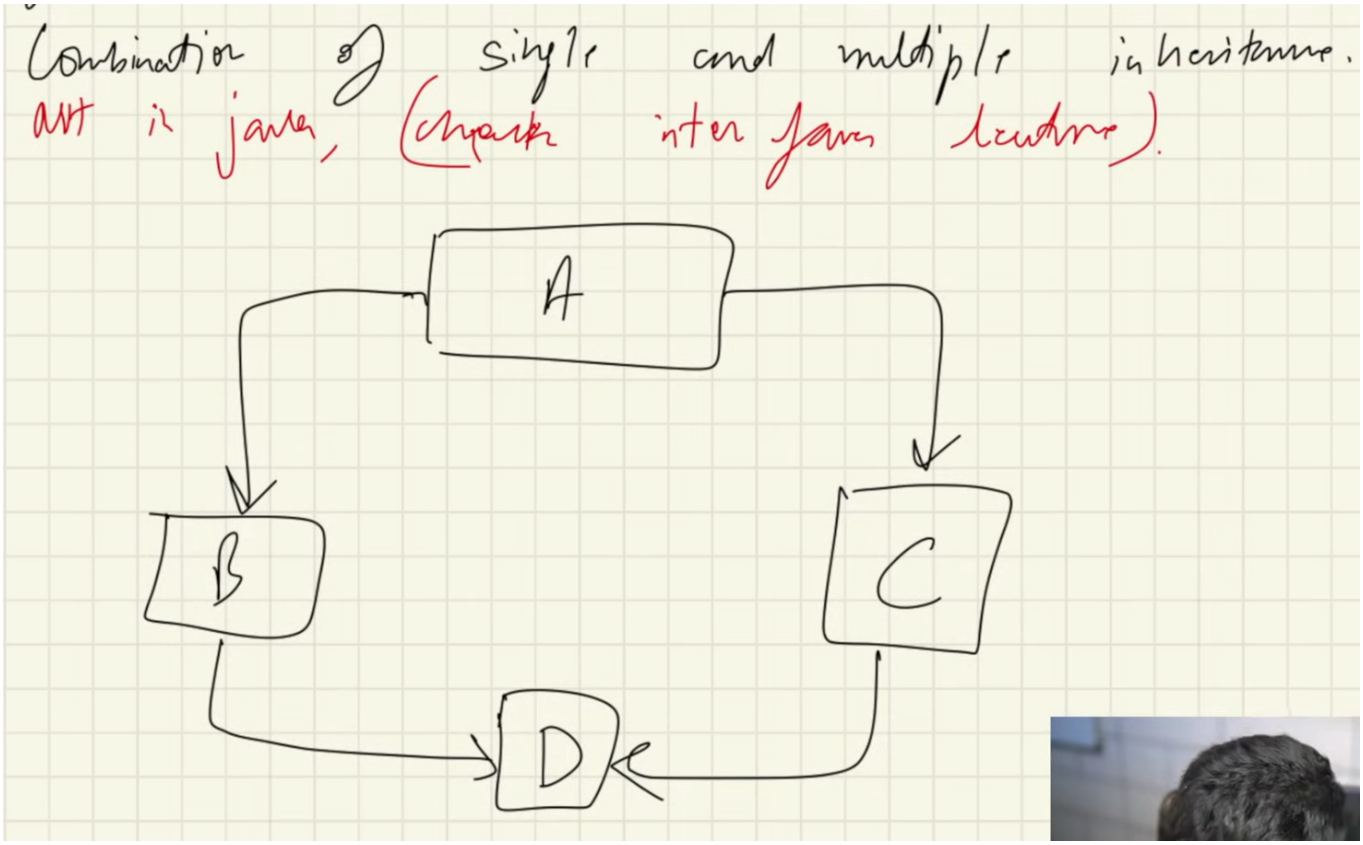
## Benefits of Hierarchical Inheritance

- **Code Reusability**: The common code in the superclass is reused by multiple subclasses.
- **Organized Structure**: Helps in organizing classes in a structured and hierarchical manner.
- **Extensibility**: Easy to extend the code by adding new subclasses without modifying the existing superclass.

# Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance, such as single inheritance, multiple inheritance (through interfaces in Java), hierarchical inheritance, and multi-level inheritance. It allows for a more

complex inheritance structure and enables the modeling of complex relationships in object-oriented programming.




## Example of Hybrid Inheritance in Java

Let's create an example that combines single inheritance, multiple inheritance through interfaces, and hierarchical inheritance.

```
// Superclass
class Shape {
    public void display() {
        System.out.println("Displaying shape");
    }
}

// Interface for multiple inheritance
interface Colorful {
    void color();
}

// Interface for multiple inheritance
interface Textured {
    void texture();
}

// Subclass 1
class Triangle extends Shape {
    public void area(int l, int h) {
        System.out.println(0.5 * l * h);
    }
}
```

```
// Subclass 2
class Circle extends Shape {
    public void area(int r) {
        System.out.println(3.14 * r * r);
    }
}

// Hybrid inheritance combining hierarchical and multiple inheritance
class ColoredTexturedTriangle extends Triangle implements Colorful, Textured {
    @Override
    public void color() {
        System.out.println("Triangle is colored.");
    }

    @Override
    public void texture() {
        System.out.println("Triangle has a texture.");
    }
}

// Main class to test inheritance
public class OOPS {
    public static void main(String args[]) {
        // Create an object of the hybrid subclass
        ColoredTexturedTriangle ctt = new ColoredTexturedTriangle();

        // Calling method from the superclass (Shape)
        ctt.display(); // Outputs: Displaying shape

        // Calling method from the Triangle subclass
        ctt.area(10, 5); // Outputs: 25.0

        // Calling methods from the interfaces
        ctt.color(); // Outputs: Triangle is colored.
        ctt.texture(); // Outputs: Triangle has a texture.
    }
}
```

1. **Single Inheritance**:
   - `Triangle` extends `Shape`: The `Triangle` class inherits the `display()` method from the `Shape` class.
   - This allows `Triangle` to reuse the code in `Shape` and add its own functionality, specifically the `area(int l, int h)` method.
2. **Multiple Inheritance through Interfaces**:
   - `ColoredTexturedTriangle` implements `Colorful` and `Textured`: The `ColoredTexturedTriangle` class implements two interfaces, `Colorful` and `Textured`, which define the `color()` and `texture()` methods, respectively.
   - This means `ColoredTexturedTriangle` must provide implementations for both `color()` and `texture()` methods, allowing it to gain behavior from multiple sources.
3. **Multi-level Inheritance**:
   - `ColoredTexturedTriangle` extends `Triangle`: The `ColoredTexturedTriangle` class is a subclass of `Triangle`. This makes `ColoredTexturedTriangle` a part of the hierarchy where `Shape` is the grandparent class, `Triangle` is the parent class, and `ColoredTexturedTriangle` is the child class.

- `ColoredTexturedTriangle` inherits the `area(int l, int h)` method from `Triangle` and the `display()` method from `Shape`.

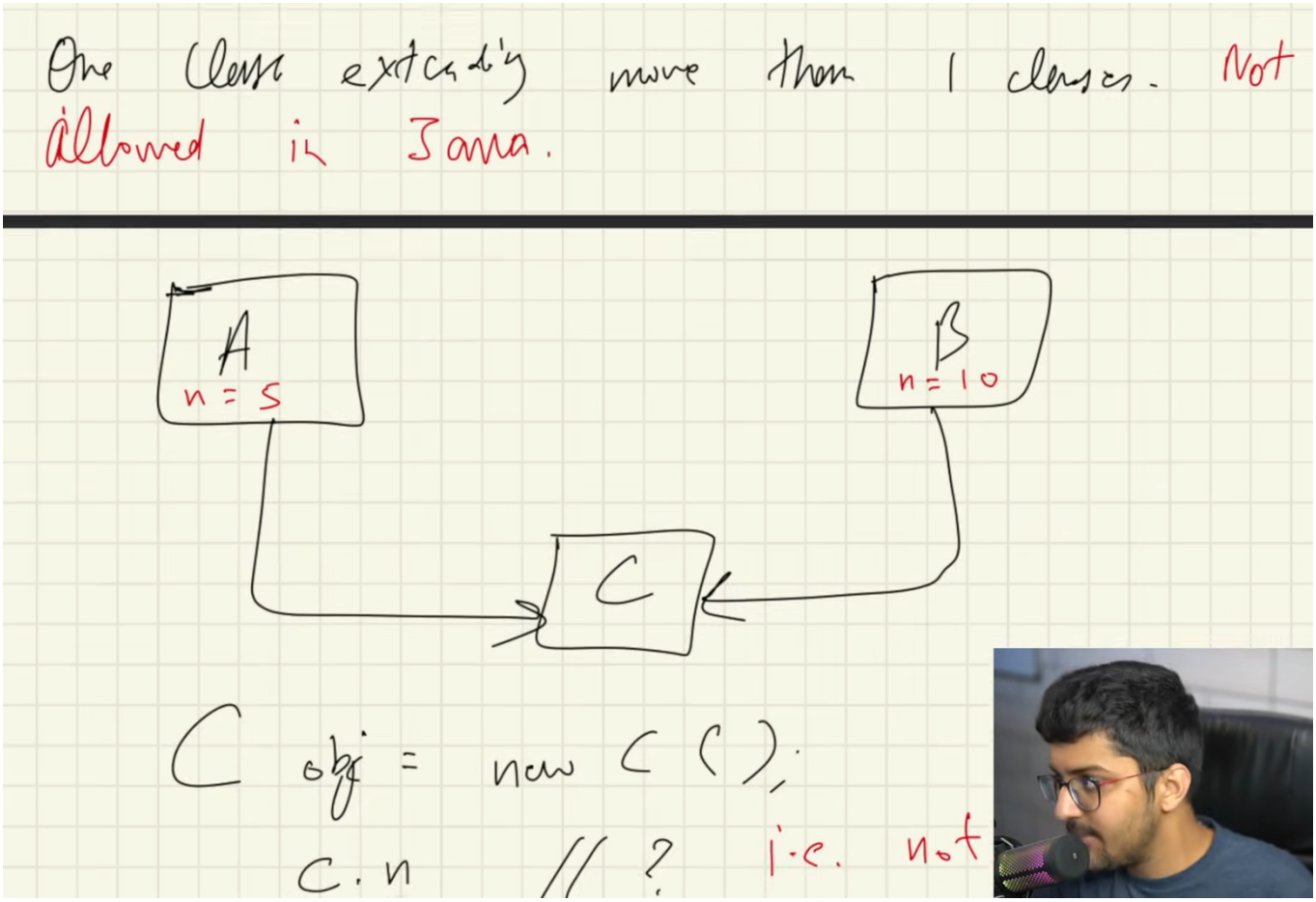
### Combined Hybrid Inheritance

By combining these forms of inheritance:

- `ColoredTexturedTriangle` inherits the `display()` method from `Shape` via `Triangle`.
- `ColoredTexturedTriangle` inherits and can use the `area(int l, int h)` method from `Triangle`.
- `ColoredTexturedTriangle` implements additional functionality specified by the `Colorful` and `Textured` interfaces.

## Multiple Inheritance

**Multiple Inheritance** is a feature in object-oriented programming where a class can inherit characteristics and behaviors from more than one parent class. This means a subclass can have multiple super classes and can inherit properties and methods from all of them. However, this can lead to complexity and potential issues, such as the "diamond problem."




### Multiple Inheritance in Java

Java does not support multiple inheritance with classes to avoid the complexity and ambiguity it can create. Instead, Java supports multiple inheritance through interfaces. This allows a class to implement multiple interfaces and thus inherit the behavior defined in those interfaces.
Example in Java (using interfaces)

```java
// Interface 1
interface Colorful {
    void color();
}

// Interface 2
interface Textured {
    void texture();
}

// Class implementing multiple interfaces
class ColoredTexturedShape implements Colorful, Textured {
    public void color() {
        System.out.println("Shape is colored.");
    }

    public void texture() {
        System.out.println("Shape has a texture.");
    }
}

public class Main {
    public static void main(String args[]) {
        ColoredTexturedShape cts = new ColoredTexturedShape();
        cts.color(); // From Colorful
        cts.texture(); // From Textured
    }
}
```

## Explanation

1. **Interface Definitions**:
   - `Colorful` interface defines a method `color()`.
   - `Textured` interface defines a method `texture()`.
2. **Class Implementing Interfaces**:
   - `ColoredTexturedShape` class implements both `Colorful` and `Textured` interfaces.
   - This means `ColoredTexturedShape` must provide implementations for both `color()` and `texture()` methods.
3. **Implementing Methods**:
   - `ColoredTexturedShape` provides specific implementations for the `color()` and `texture()` methods.
4. **Main Method**:
   - In the `main` method, an instance of `ColoredTexturedShape` is created.
   - The `color()` and `texture()` methods are called on this instance, demonstrating the inherited behavior from both interfaces.

## Key Points

- **Combining Behaviors**: Multiple inheritance through interfaces allows a class to combine behaviors from multiple sources, which can be useful for creating flexible and reusable code.
- **Avoiding Complexity**: By using interfaces, Java avoids the complexity and ambiguity associated with multiple inheritance of classes, such as the diamond problem.

- **Flexibility**: Interfaces provide a way to specify that a class must implement certain methods, without dictating how those methods should be implemented, allowing for greater flexibility.

## Diamond Problem

The diamond problem is an issue that arises in multiple inheritance when a class inherits from two classes that have a common ancestor. This can lead to ambiguity about which inherited properties or methods to use. Java avoids this problem by not supporting multiple inheritance with classes and instead using interfaces, which do not have the same ambiguity because interfaces do not have implemented methods that could conflict.

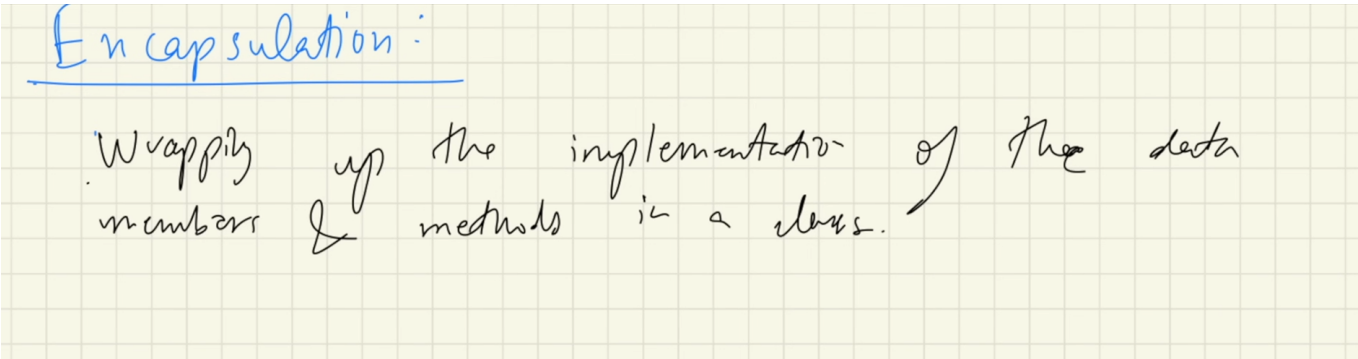
## Summary

- **Multiple Inheritance**: Involves a class inheriting from multiple parent classes.
- **Java's Approach**: Java supports multiple inheritance through interfaces, allowing a class to implement multiple interfaces and inherit behavior from them.
- **Example**: The `ColoredTexturedShape` class implements both `Colorful` and `Textured` interfaces, inheriting the behavior specified by these interfaces.
- **Avoiding Diamond Problem**: By using interfaces, Java avoids the complexity and ambiguity associated with multiple inheritance of classes.

# Encapsulation

https://www.w3schools.com/java/java_encapsulation.asp

Encapsulation in Java is a fundamental principle of object-oriented programming that involves bundling data (variables) and methods (functions) that operate on the data into a single unit, often a class. The key idea behind encapsulation is to restrict direct access to some of an object's components, usually the internal state (variables), and enforce access through methods (getters and setters) that maintain the object's integrity and behavior.




## Key Points of Encapsulation

- **Data Hiding**: Encapsulated data is not directly accessible from outside the class, which prevents unintended modification and ensures that the object's state remains consistent.
- **Access Control**: Access to the encapsulated data is typically provided through public methods (getters and setters), which can enforce validation rules, calculations, or any other logic before allowing changes to the data. This helps in maintaining the internal consistency of the object.
- **Modularity**: Encapsulation promotes modularity by compartmentalizing the implementation details of a class. This allows you to change the internal implementation without affecting other parts of the program that use the class, as long as the public interface (methods) remains unchanged.
- **Security**: By hiding implementation details and providing controlled access through methods, encapsulation enhances security by reducing the risk of unintended data modification and ensuring that data interactions

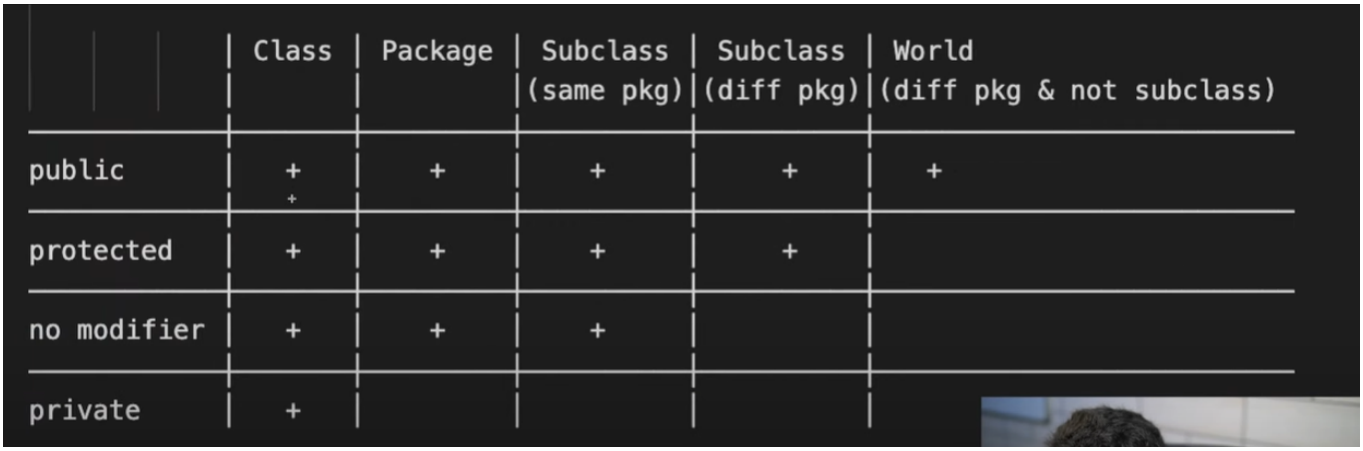
adhere to expected behaviors.

## Relation with Packages

In Java, a **package** is a namespace that organizes classes and interfaces. Packages are used to group related classes together, which helps in managing large codebases.

- **Encapsulation and Packages**: By placing classes in packages, you can control the scope of access to classes and their members. Classes within the same package can access each other's package-private members (those without any explicit access modifier), but classes outside the package cannot.

## Access Modifiers



| | Class | Package | Subclass (same pkg) | Subclass (diff pkg) | World (diff pkg & not subclass) |
|---|---|---|---|---|---|
| public | + | + | + | + | + |
| protected | + | + | + | + | |
| no modifier | + | + | + | | |
| private | + | | | | |

| Feature | Public | Default (Package-Private) | Protected | Private |
|---|---|---|---|---|
| **Access Level** | Accessible from any other class. | Accessible only within the same package. | Accessible within the same package and subclasses. | Accessible only within the same class. |
| **Keyword** | `public` | No keyword | `protected` | `private` |
| **Access from the Same Class** | Yes | Yes | Yes | Yes |
| **Access from Same Package** | Yes | Yes | Yes | No |
| **Access from Subclass (Same Package)** | Yes | Yes | Yes | No |
| **Access from Subclass (Different Package)** | Yes | No | Yes | No |
| **Access from Non-Subclass (Different Package)** | Yes | No | No | No |

### Public Example

```java
package com.example.shapes;

public class Shape {
    // Public variable
    public String color;

    // Public method
    public void display() {
        System.out.println("Displaying shape");
    }
}

class Triangle extends Shape {
    public void setColor(String color) {
        this.color = color;
    }
}
```

### Default (Package-Private) Example

```java
package com.example.shapes;

class Shape {
    // Default (Package-Private) variable
    String color;

    // Default (Package-Private) method
    void display() {
        System.out.println("Displaying shape");
    }
}

class Triangle extends Shape {
    void setColor(String color) {
        this.color = color;
    }
}
```

### Protected Example

```java
package com.example.shapes;

public class Shape {
    // Protected variable
    protected String color;

    // Protected method
    protected void display() {
        System.out.println("Displaying shape");
    }
}

class Triangle extends Shape {
```

```
    void setColor(String color) {
        this.color = color;
    }

    void show() {
        display(); // Accessing protected method from superclass
    }
}

// In a different package
package com.anotherpackage;

import com.example.shapes.Shape;

class AnotherPackageTriangle extends Shape
```

In Java, the `protected` access modifier allows a member (field, method, or constructor) to be accessed in the following ways:

1. **Within the same package**: Any class within the same package can access the `protected` member.
2. **In subclasses**: Subclasses can access the `protected` member, even if they are in different packages.

To clarify, here's an example:

### Example:

**Package:** `com.example.base`

**BaseClass.java**

```
package com.example.base;

public class BaseClass {
    protected void protectedMethod() {
        System.out.println("Protected method in BaseClass");
    }
}
```

**Package:** `com.example.subclass`

**SubClass.java**

```
package com.example.subclass;

import com.example.base.BaseClass;

public class SubClass extends BaseClass {
    public void accessProtectedMethod() {
        protectedMethod(); // This is allowed
    }

    public static void main(String[] args) {
        SubClass subClass = new SubClass();
        subClass.accessProtectedMethod();
```

```
    }
}
```

In this example:

- The `protectedMethod` in `BaseClass` is declared as `protected`.
- `SubClass`, which is in a different package (`com.example.subclass`), extends `BaseClass`.
- `SubClass` can access `protectedMethod` because it is a subclass of `BaseClass`, even though they are in different packages.

**Important Points:**

- A `protected` member is not accessible to non-subclass classes in different packages.
- Only subclasses (direct or indirect) can access the `protected` member in different packages.

**Package:** `com.example.other`

**OtherClass.java**

```
package com.example.other;

import com.example.base.BaseClass;

public class OtherClass {
    public void accessProtectedMethod() {
        BaseClass base = new BaseClass();
        // base.protectedMethod(); // This will cause a compile-time error
    }
}
```

In this example:

- `OtherClass` is neither in the same package as `BaseClass` nor a subclass of `BaseClass`.
- Therefore, it cannot access the `protectedMethod`, and attempting to do so will result in a compile-time error.

`private` Example

```
class Shape {
    // Private variable
    private String color;

    // Private method
    private void display() {
        System.out.println("Displaying shape");
    }
}

class Triangle extends Shape {
    void setColor(String color) {
        // Cannot access private variable 'color' directly
        // this.color = color; // Error: 'color' has private access in 'Shape'
    }

    void show() {
        // Cannot access private method 'display' directly
```

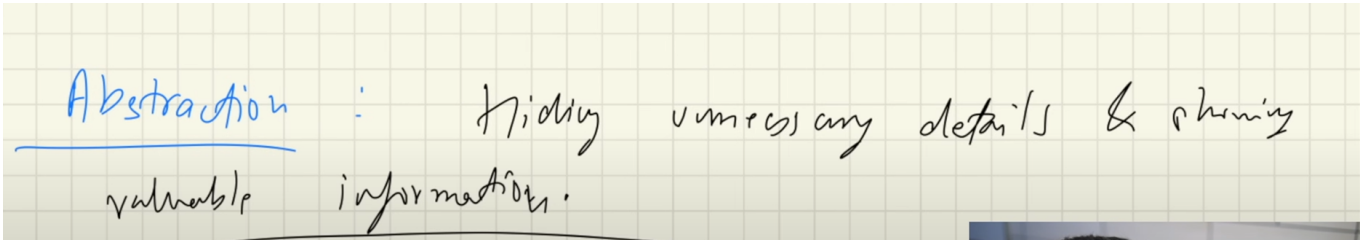
```
        // display(); // Error: 'display()' has private access in 'Shape'
    }
}
```

In practice, encapsulation is achieved by declaring the class's variables as private and providing public methods (getters and setters) to access and modify these variables. This approach not only improves code maintainability and readability but also facilitates effective teamwork by defining clear boundaries for class interactions. Overall, encapsulation is a cornerstone of building robust and scalable object-oriented systems in Java and other programming languages.

# Abstraction

**Abstraction** in object-oriented programming is a process of hiding the implementation details and showing only the essential features of an object. It helps in reducing complexity by allowing the programmer to focus on interactions at a higher level rather than getting bogged down with details.

Think of a car: you interact with the car through the steering wheel, pedals, and buttons (interface), without needing to know the intricate details of how the engine works or how the fuel is converted into motion (implementation details).




In Java, abstraction is achieved using:

1. **Abstract Classes**
2. **Interfaces**

## Example Using Abstract Class

### Abstract Class Example

```
// Abstract class representing a Shape
abstract class Shape {
    // Abstract method to calculate area
    public abstract double calculateArea();

    // Concrete method to display information
    public void display() {
        System.out.println("Displaying shape");
    }
}

// Concrete class Circle extending Shape
class Circle extends Shape {
    private double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
```

```
    }

    // Implementing abstract method to calculate area
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Concrete class Rectangle extending Shape
class Rectangle extends Shape {
    private double length;
    private double width;

    // Constructor
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementing abstract method to calculate area
    @Override
    public double calculateArea() {
        return length * width;
    }
}

// Main class to test abstraction
public class Main {
    public static void main(String[] args) {
        // Creating objects of Circle and Rectangle
        Circle circle = new Circle(5.0);
        Rectangle rectangle = new Rectangle(3.0, 4.0);

        // Displaying shapes
        circle.display();
        rectangle.display();

        // Calculating and displaying area
        System.out.println("Area of Circle: " + circle.calculateArea());
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());
    }
}
```

## Explanation

- **Abstract Class** `Shape`:
  - Defines an abstract method `calculateArea()` that must be implemented by any subclass.
  - Contains a concrete method `display()` that provides a common implementation for all subclasses.
- **Concrete Classes** `Circle` **and** `Rectangle`:
  - Extend the abstract class `Shape`.
  - Provide specific implementations of the abstract method `calculateArea()`.
- **Main Class** `Main`:

- Creates instances of `Circle` and `Rectangle`.
- Calls the `display()` method (inherited from `Shape`) and the `calculateArea()` method (specific to each subclass).

## Example Using Interfaces

### Interface Example

```
// Interface representing a Shape
interface Shape {
    double calculateArea();  // Abstract method
    void display();          // Abstract method
}

// Concrete class Circle implementing Shape interface
class Circle implements Shape {
    private double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }

    // Implementing calculateArea method
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }

    // Implementing display method
    @Override
    public void display() {
        System.out.println("Displaying Circle");
    }
}

// Concrete class Rectangle implementing Shape interface
class Rectangle implements Shape {
    private double length;
    private double width;

    // Constructor
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Implementing calculateArea method
    @Override
    public double calculateArea() {
        return length * width;
    }

    // Implementing display method
    @Override
    public void display() {
```

```
        System.out.println("Displaying Rectangle");
    }
}

// Main class to test abstraction
public class Main {
    public static void main(String[] args) {
        // Creating objects of Circle and Rectangle
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(3.0, 4.0);

        // Displaying shapes
        circle.display();
        rectangle.display();

        // Calculating and displaying area
        System.out.println("Area of Circle: " + circle.calculateArea());
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());
    }
}
```

## Explanation

- **Interface** `Shape`:
  - Declares two abstract methods: `calculateArea()` and `display()`.
  - Any class that implements the `Shape` interface must provide concrete implementations of these methods.
- **Concrete Classes** `Circle` **and** `Rectangle`:
  - Implement the `Shape` interface.
  - Provide specific implementations of `calculateArea()` and `display()` methods.
- **Main Class** `Main`:
  - Creates instances of `Circle` and `Rectangle`.
  - Calls the `display()` and `calculateArea()` methods on each instance.

## Key Differences Between Abstract Classes and Interfaces

| Feature | Abstract Class | Interface |
|---|---|---|
| **Methods** | Can have abstract and concrete methods | Can only have abstract methods (Java 8 allows default and static methods) |
| **Fields** | Can have instance variables (fields) | Cannot have instance variables (can have constants) |
| **Inheritance** | Can extend one class | Can implement multiple interfaces |
| **Use Case** | Used when classes share a common base class and behavior | Used to define a contract that multiple classes can implement |

## Summary

Abstraction in Java is a core principle of object-oriented programming that focuses on simplifying complex systems by hiding the implementation details and exposing only the essential features. It allows developers to work with high-level concepts and interactions without needing to understand the intricate workings beneath. This is achieved through

abstract classes and interfaces, where abstract classes can contain both abstract methods (without implementation) and concrete methods (with implementation), and interfaces define a contract with abstract methods that must be implemented by any class that adheres to the interface. By using abstraction, developers can create more modular, maintainable, and understandable code, as they can concentrate on what an object does rather than how it does it.