Java Programming Week-03

<Mohan De Zoysa -IIT JAVA CERTIFICATION>

Outline

Object Oriented Programing

OOP

OOP -Object Oriented Programing

"Object-oriented programming is an approach to designing modular, reusable software systems where entities in the problem domain are modeled as objects that have attributes and behaviors." — Robert C. Martin (Uncle Bob)

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods

OOP - Advantages

Object-Oriented Programming (OOP) comes with several advantages that make it a popular and powerful programming paradigm.

Modularity and Reusability: OOP promotes modular design, allowing you to create small, self-contained units (objects) that can be reused in different parts of your program or even in other projects. This makes development faster and more efficient.

Code Organization: OOP helps you structure your code in a more organized and intuitive manner. Objects represent real-world entities, making it easier to understand, maintain, and extend the codebase.

Encapsulation: Encapsulation hides the internal workings of an object and allows you to control access to its data and methods. This improves data security and prevents unintended interference with the object's state

OOP - Advantages continue.

Code Maintenance: With encapsulation and modularity, code maintenance becomes easier. Changes or updates can be made to individual objects without affecting other parts of the program, reducing the risk of introducing bugs.

Flexibility and Extensibility: OOP allows you to add new features or modify existing ones without changing the entire codebase. You can create new classes or modify existing ones to extend functionality.

Inheritance: Inheritance enables code reuse by allowing a new class (subclass) to inherit properties and behaviors from an existing class (superclass). This simplifies the creation of similar objects with shared features.

Polymorphism: Polymorphism enables you to treat objects of different classes in a unified way, which enhances flexibility. This can lead to more adaptable and dynamic code.

OOP – Advantages continue.

Modeling Real-World Entities: OOP closely models real-world entities, making it easier to translate real-world concepts into code. This improves communication between developers and stakeholders.

Collaboration: OOP supports team collaboration. Developers can work on different classes or modules concurrently without worrying about interfering with each other's work.

Software Design: OOP encourages planning and design before implementation, resulting in a more structured and well-thought-out software architecture.

Industry Standard: Many popular programming languages, like Java, C++, and Python, support OOP, making it an industry-standard approach that many developers are familiar with.

OOP -Object Oriented Programing

Class

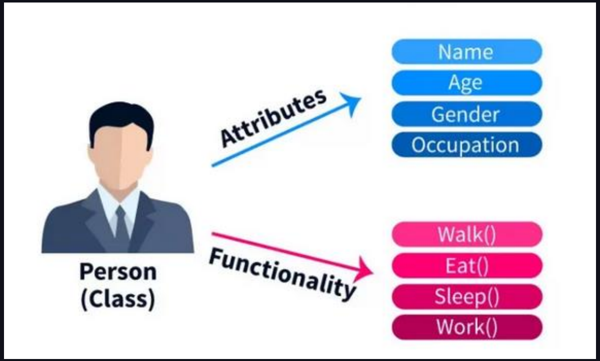
A Java class is a blueprint or template for creating objects. It defines the properties (attributes or fields) and behaviors (methods) that objects of that class will have. A class serves as a prototype from which you can create multiple instances or objects that share the same structure and behavior defined in the class.

Object

An object, on the other hand, is an instance of a class. It represents a specific entity or concept and encapsulates both data (attributes) and the operations (methods) that can be performed on that data. Objects are the fundamental building blocks of object-oriented programming (OOP) and allow you to model real-world entities in your software.

In simpler terms, a class defines the overall characteristics and behaviors that objects will possess, while objects are the actual instances that are created based on the class's blueprint.

Java Class



Syntax

```
class {Class Name}
```

- Class name start with upper case
- file name should be saved as {Class name}.java

```
public class Person {
    String name;
    int age;

}
```

Java Object

Using Objects in Java

```
Syntax {ClassName} {Object Name} = new {ClassName} ;

Person personOne = new Person();
```

To access attributes, and methods use the dot “.” (period) Operator

```
Syntax  {Object Name} .{attribute name} ;
personOne.name =“John”;
personOne.walk();
```

Inside its own class “the” keyword this refers to the current object

```
this.name =“John”;
```

Declaring, Instantiating and Initializing an Object

```
Syntax {ClassName} {Object Name} = new {ClassName} ;
Person personOne = new Person();
```

Declaring an Object - Declarations can appear as part of object creation as you saw above or can appear alone like this

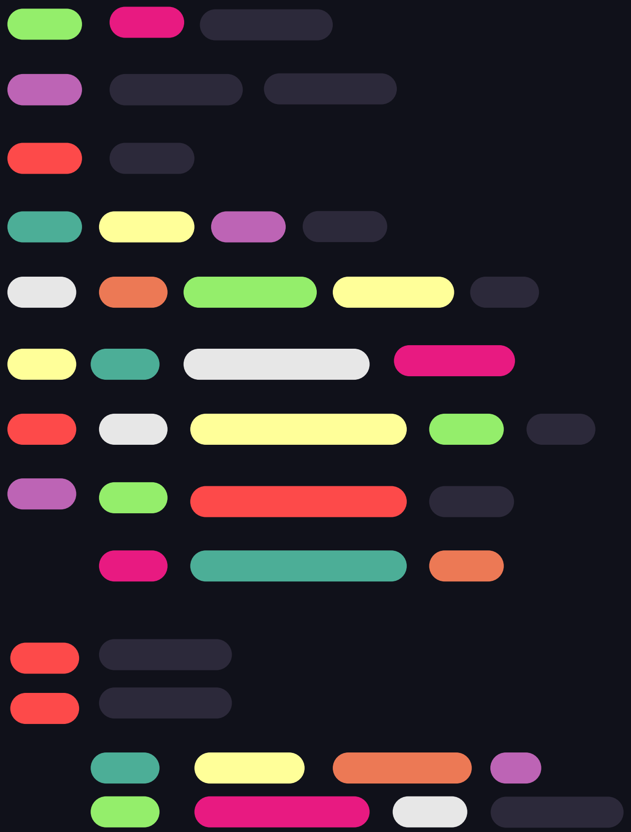
```
Syntax :  {Class Name}  {Object name};
Person person;
```

Instantiating an Object - The new operator instantiates a new object by allocating memory for it. new requires a single argument: a constructor method for the object to be created.

```
person = new Person();
```

Initializing an Object - Classes provide constructor methods to initialize a new object of that type. In a class declaration, constructors can be distinguished from other methods because they have the same name as the class and have no return type.

```
Person();
```



Constructors

< Classes provide constructor methods to initialize a new object constructors are special methods within a class >

** All constructer Name is same as Java Class name

- Default Constructor
- Parameterized Constructor
- Copy Constructor
- Chained Constructor
- Private Constructor

Default Constructors

If a class does not explicitly define any constructors, Java provides a default constructor with no parameters.

It initializes instance variables to their default values (e.g., null for objects, 0 for integers).

```
public class Person {
    // Default constructor
      public Person() {
       // Initialization code (if any)
     }
}
```

```
Usage :  Person person =new Person();
```

Parameterized Constructors

This type of constructor accepts parameters to initialize the instance variables of an object. It allows you to set initial values during object creation.

```
public class Person {
    String name;
    int age;

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Usage : `Person person =new Person("John Doe","Male",24);`

Copy Constructors

A copy constructor creates a new object by copying the attributes of an existing object of the same class. It's used to create a new object that's a duplicate of an existing object.

```
public class Person {
    private String name;
    private int age;

    // Parameterized constructor
    public Person(Person other) {
        this.name = other.name;
        this.age = other.age;
    }
}
          Usage :
          Person other  = new Person();
          Person person = new Person(other);
```

Chained Constructors

A chained constructor is used to call another constructor within the same class. It allows you to reuse code when different constructors need to perform similar tasks.

```
public class Person {
    String name;
    int age;

    // Parameterized constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Chained constructor
    public Person(String name) {
        this(name, 0); // Calls the parameterized constructor with default age
    }
}
```

Private Constructor

A private constructor is not accessible from outside the class. It ensure only one instance of the class is created.

```
public class Person {
    String name;
    int age;
    public static Person person;

    //Get Method
    public static Person getPerson() {
       if(person ==null) {
           person=new Person();
       }
        return person;
    }
    private Person() {
        this.name = "John Doe";
        this.age = 0;
    }

}
```

Inner, Nested class

In object-oriented programming, an inner class or nested class is a class declared entirely within the body of another class or interface

```java
//Inner Class
class Outer {
    private int outerVariable = 20;

    class Inner {
        void display() {
            System.out.println("Inner class: " + outerVariable);
        }
    }
}

//Static Nested Class

class Outer {
    private static int outerVariable = 10;

    static class StaticNested {
        void display() {
            System.out.println("Static nested class: " + outerVariable);
        }
    }
}
```

Static variables and methods

The "static" keyword in Java is used to declare members (variables and methods) that belong to the class itself rather than to individual instances(Objects) of the class. This means that a static member is shared among all instances of the class and can be accessed using the class name, without needing to create an instance of the class.

Static Variables (Class Variables)

- Static variables are shared across all instances of a class.
- They are declared using the static keyword and are associated with the class, not instances.
- They are accessed using the class name (ClassName.variableName).
- Static variables are often used for constants, counters, or shared data.

```
class MyClass {
    static int count = 0; // Static variable

    MyClass() {
        count++; // Increment the count for each instance
    }
}
```

Static variables and methods

Static methods are associated with the class itself, not instances.

- They are declared using the static keyword.
- Static methods can be called using the class name (ClassName.methodName()).
- They can only access static variables and other static methods directly.
- Useful for utility methods or methods that don't require access to instance-specific data.

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

Advantage of Static variables

Advantages of Static Variables:

Shared Data: Static variables are shared among all instances of a class. This means that changes to a static variable are visible to all objects of the class.

Memory Efficiency: Since static variables are shared, they occupy memory only once, regardless of how many instances of the class are created. This can lead to memory savings, especially when dealing with large numbers of instances.

Global Constants: You can use static variables to define global constants that don't need to be tied to any specific instance. This makes your code more readable and maintainable.

Advantage of Static Methods

Advantages of Static Methods:

Utility Methods: Static methods can be used for creating utility functions that don't require access to instance-specific data

Class-Level Functionality: Static methods can provide behavior that is relevant at the class level, not the instance level. For example, the Math class in Java contains static methods for common mathematical operations.

Avoiding Unnecessary Object Creation: In cases where you need to perform a simple operation that doesn't depend on instance variables, using a static method can help avoid unnecessary object creation.

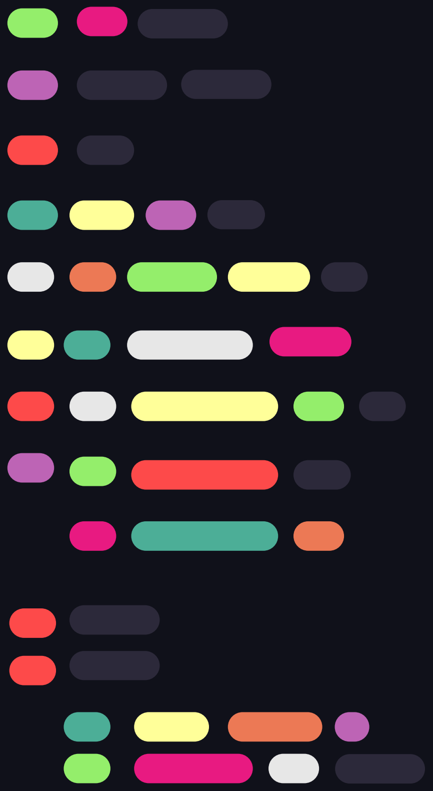
Math.sqrt() or Math.random().
Creating singletons, where you want only one instance of a class to exist.

Encapsulation ***

It refers to the concept of bundling data (attributes or variables) and methods (functions or procedures) that operate on the data into a single unit, known as a class.

It restrict direct access to the internal state (data) of an This helps in achieving data hiding and protecting the object's integrity, as external code cannot directly modify the internal data without going through the designated methods.

In Java, encapsulation is implemented using access modifiers.



```java
public class Person {

 private String name;

 private  int age;

 public void walk(){}

}
```

Access Modifiers

< In Java, access modifiers are keywords used to control the visibility and accessibility of classes, variables, methods, and constructors within your program

They determine which parts of your code can be accessed from other classes and which parts are hidden or restricted>

- Public , private , protected ,default

```java
public class Person {

 private String name;

 private  int age;

 public void walk(){}

}
```

Access Modifiers

< In Java, access modifiers are keywords used to control the visibility and accessibility of classes, variables, methods, and constructors within your program

They determine which parts of your code can be accessed from other classes and which parts are hidden or restricted>

- Public , private , protected ,default

Modifiers in Class

Public and default are supported to be used in a top-level class, however nested class support all for types.

```
public class Person {

}
```

```
class Person {

}
```

Access Modifier Scope

	Private	Deful t	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Pacakge	No	Yes	Yes	Yes
Different Package	No	No	No	Yes

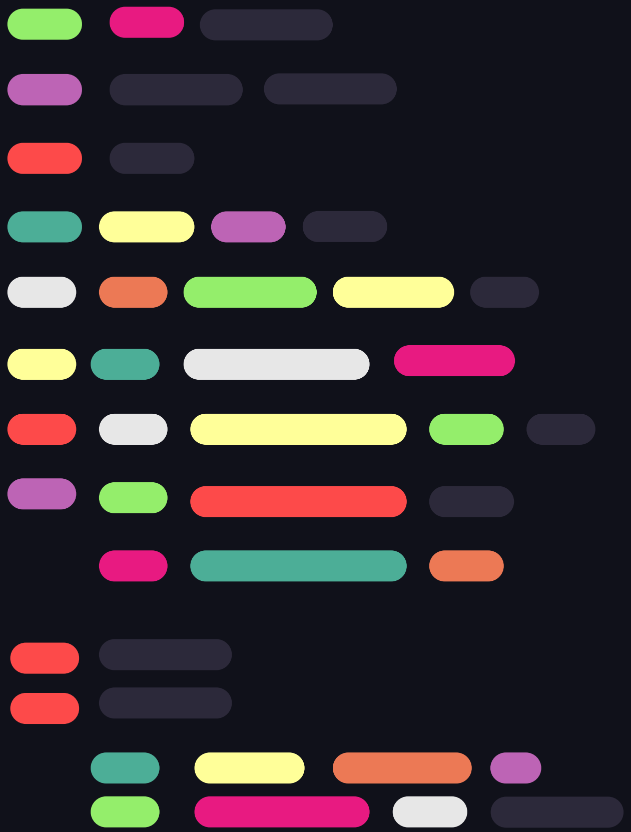
Java Getters & Setters

In Java, getters and setters are methods used to access and modify the private fields (properties) of a class. They are a common practice for encapsulating the fields and providing controlled access to them. Getters are used to retrieve the values of fields, and setters are used to modify the values of fields.

Getters: A getter method returns the value of a private field.
Setters: A setter method sets the value of a private field.

```
public class Person {
    private int age;

    public int getAge() {
        return this.age;
    }
}
```

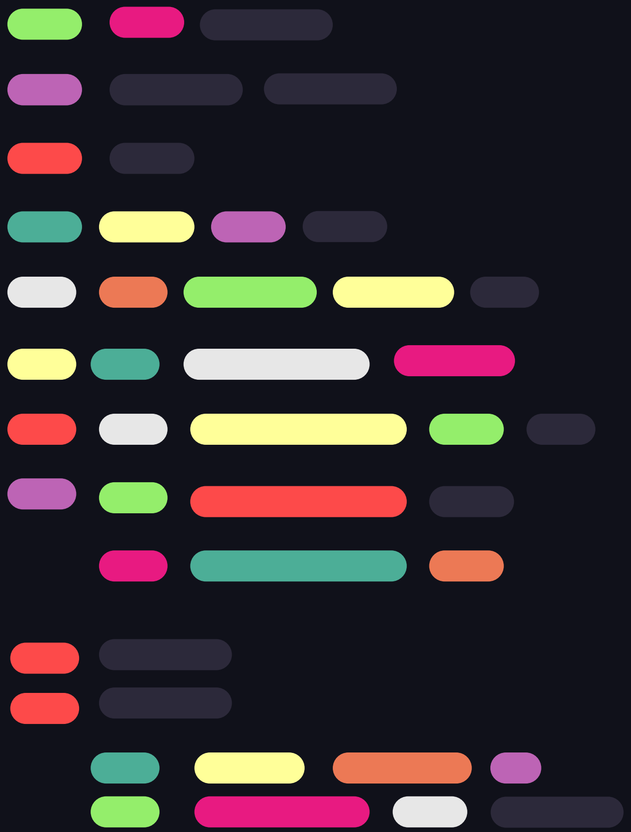
```
public class Person {
    private int age;

    public void setAge(int age) {

        this.age=age;
    }
}
```



Inheritance ***

Inheritance is a core concept in object-oriented programming (OOP) that allows a new class (subclass or derived class) to inherit properties and behaviors (fields and methods) from an existing class (superclass or base class).

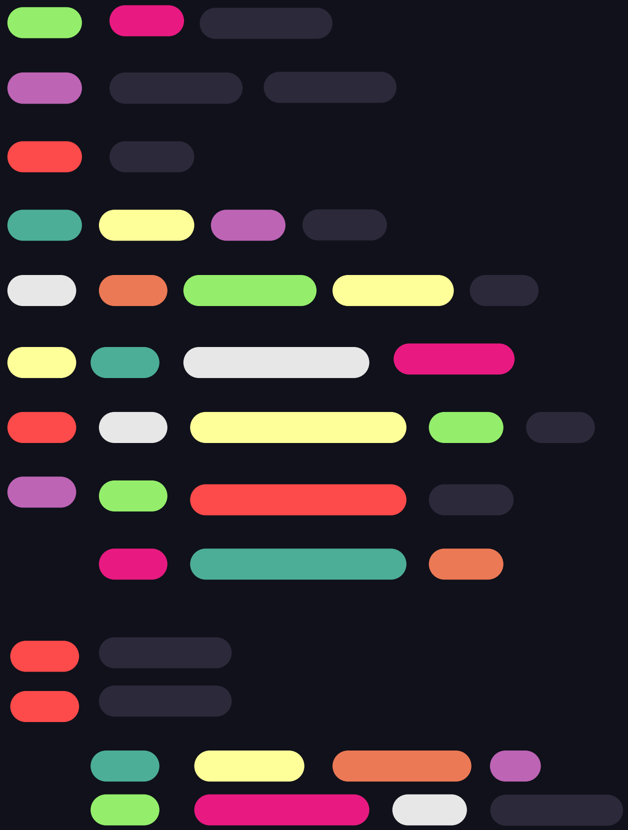
The subclass can then extend or modify these inherited properties and behaviors, while also adding new ones of its own.



Inheritance cont.

In Java, inheritance can take several forms, and understanding these different types of inheritance helps you design your class hierarchies effectively. Here are the main types of inheritance in Java:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance**



Inheritance cont.

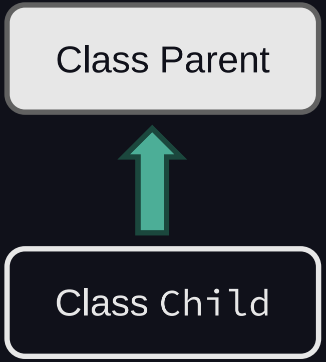
Superclass (Base Class): The existing class from which properties and behaviors are inherited is known as the superclass or base class. It defines common attributes and methods that can be shared among its subclasses.

Subclass (Derived Class): The new class that inherits from the superclass is called the subclass or derived class. It extends the functionality of the superclass by adding its own attributes and methods or overriding the ones inherited from the superclass.

Single Inheritance

This is the most common form of inheritance, where a subclass inherits from a single superclass.

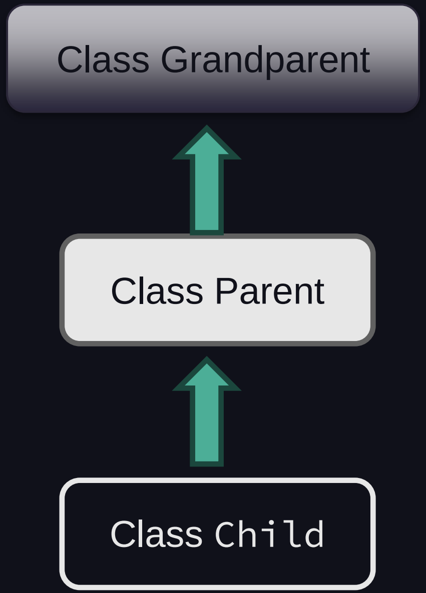
```
class Parent {
    // ...
}

class Child extends Parent {
    // ...
}
```



Multilevel Inheritance

In multilevel inheritance, a subclass inherits from another subclass. This creates a chain of inheritance where each subclass inherits from its immediate superclass. Java supports multilevel inheritance.

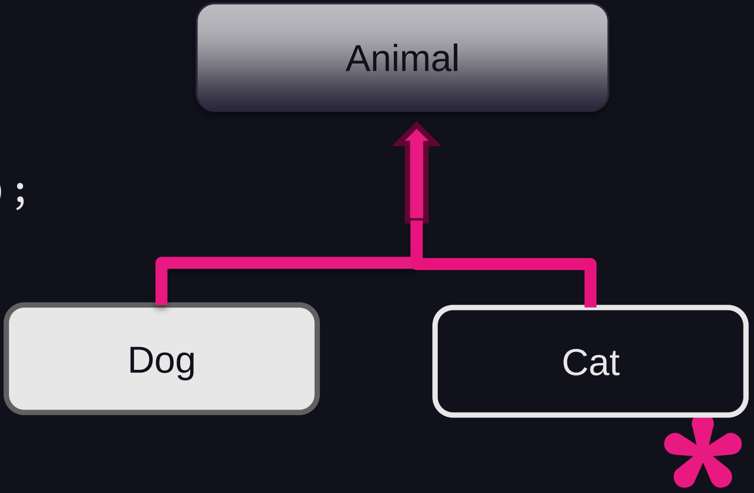
```
class Grandparent {
    // ...
}

class Parent extends Grandparent {
    // ...
}

class Child extends Parent {
    // ...
}
```



Hierarchical Inheritance

Hierarchical inheritance is a concept in object-oriented programming (OOP) where a single class serves as the base class for multiple derived classes. In other words, it's a type of inheritance where a class inherits from a single superclass, but multiple subclasses inherit from that same superclass. Each subclass can have its own additional methods and properties, as well as inherit the properties and methods of the common superclass.

```
class Animal {
    void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}
```



Multiple Inheritance

While Java **does not support multiple inheritance of classes** (a class cannot extend more than one class), it does support multiple inheritance of interfaces. A class can implement multiple interfaces, allowing it to inherit method signatures from each interface. This is a way to achieve some level of multiple inheritance while avoiding the complexities and ambiguities associated with multiple inheritance of classes.

Java Interface

In Java, an interface is a reference type that is like a class, but it is a collection of abstract methods. An interface defines a contract or a set of method signatures that classes implementing the interface must provide concrete implementations for. It allows you to achieve abstraction, multiple inheritance of method signatures, and the definition of common behavior across unrelated classes.

Abstract methods are those types of methods that don't require implementation for its declaration

```
Syntax : interface {interface Name}
```

```
// Define an interface
interface Vehicle {
    void start();
    void stop();
}
```

Interface Implementation

```java
// Define an interface
interface Vehicle {
    void start();
    void stop();
}

// Implement the interface in classes
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car starting...");
    }

    @Override
    public void stop() {
        System.out.println("Car stopping...");
    }
}
```

The @Override annotation is used to indicate that the methods are intended to override the methods from the interface, although it's not required by the language; it's a good practice for clarity.

Multiple Inheritance with Interface

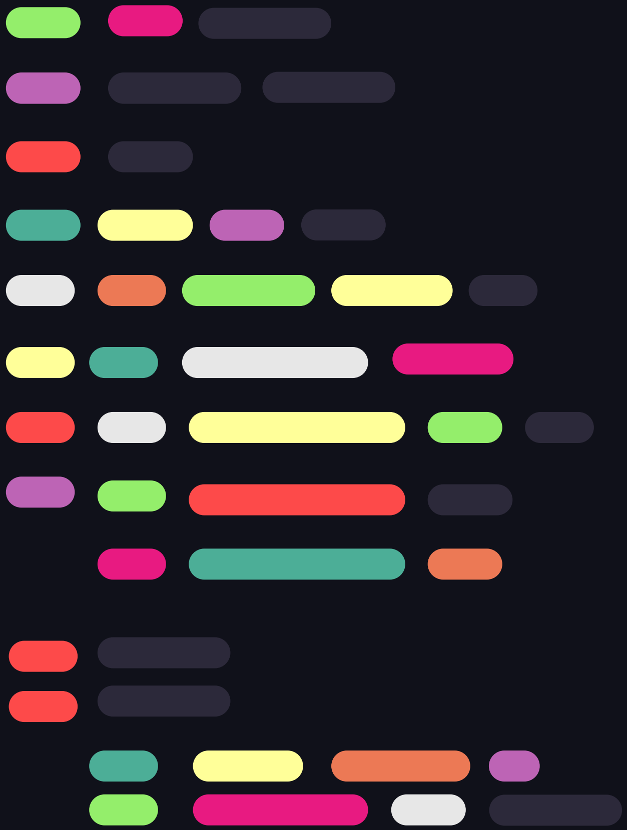
Multiple inheritance is not allowed among classes to avoid complexities and ambiguities that can arise from inheriting from multiple sources. However, Java provides a mechanism to achieve similar functionality through interfaces.

```
class MyClass implements A, B {
    @Override
    public void methodA() {
        System.out.println("Implementation of methodA");
    }

    @Override
    public void methodB() {
        System.out.println("Implementation of methodB");
    }
}
```

```
interface B {
    void methodB();
}

interface A {
    void methodA();
}
```

Access Modifier Scope cont.

	Private	Deful	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package	No	Yes	Yes	Yes
Same Package – Sub Class	No	Yes	Yes	Yes
Different Package	No	No	No	Yes
Diff Package – Sub Class	No	No	Yes	Yes



Polymorphism

Polymorphism is a core concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass or interface. It enables you to write code that can work with different types of objects in a unified way, enhancing code flexibility and reusability. explain more

There are two main types of polymorphism in Java:

- **Compile-time Polymorphism (Method Overloading)**
- **Run-time Polymorphism (Method Overriding)**

Compile-time Polymorphism (Method Overloading)

This occurs when you have multiple methods in the same class with the same name but different parameter lists (different number or types of parameters). The appropriate method to be called is determined at compile-time based on the method signature.

```
class Calculator {

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

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

Run-time Polymorphism (Method Overriding)

This occurs when you have a class that inherits from another class or implements an interface, and it provides a specific implementation for a method that is already declared in the superclass or interface. The method to be called is determined at run-time based on the actual object's type

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}
```