JAVA TECHNOLOGIES

SUBJECT CODE : UQ24CA251B

Samyukta D Kumta

Computer Applications

JAVA TECHNOLOGIES

Course Content:

Unit 2: OOP with Java

Introduction to OOP: OOP Concepts, OOP vs. POP, Advantages, Java API's, Classes and Objects, Constructor, Inheritance, Types of Inheritance, Interfaces, Polymorphism, Method Overloading vs. Method Overriding, Exception Handling: Structure, Checked and Unchecked, Custom Exceptions, Finally Block, Threads and Runnable

Experiential Learning: Object-oriented programs in Java using various OOP concepts

JAVA TECHNOLOGIES

Introduction to OOPs concepts

The main purpose of OOPs programming is to implement ideas and solve real-world problems using classes, objects, inheritance, polymorphism, and abstraction.

What is OOPS in Java?

OOPs can be defined as:

- A modular approach where data and functions can be combined into a single unit known as an object.
- It emphasizes data and security and provides the reusability of code. Using OOP, we can resemble our code in the real world.

List of Java OOPs Concepts

The following are the concepts in OOPs :

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Coupling

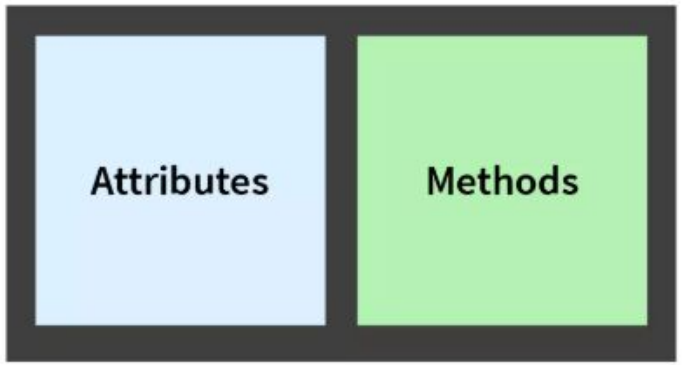
Class in java

What is Class in Java OOPs?

- A Class is a collection of similar types of objects. It is commonly known as the 'blueprint of an object'. It is a template that describes the kinds of state and behavior that the object of its type support.
- Class is a user-defined data type. Also, it doesn't occupy any memory.
- It is composed of name, attributes, and methods. Access Modifiers(public, private, default, protected) are used for having limited or public access to that class so that it can be used by other programs in Java.
- We create individual objects using class.

Class in java

What is Class in Java OOPs?



Class

We can have the following attributes :

- plane_number
- total_seats
- airline_class
- pilot_name

"Plane" class can contain the following methods to use the above attributes

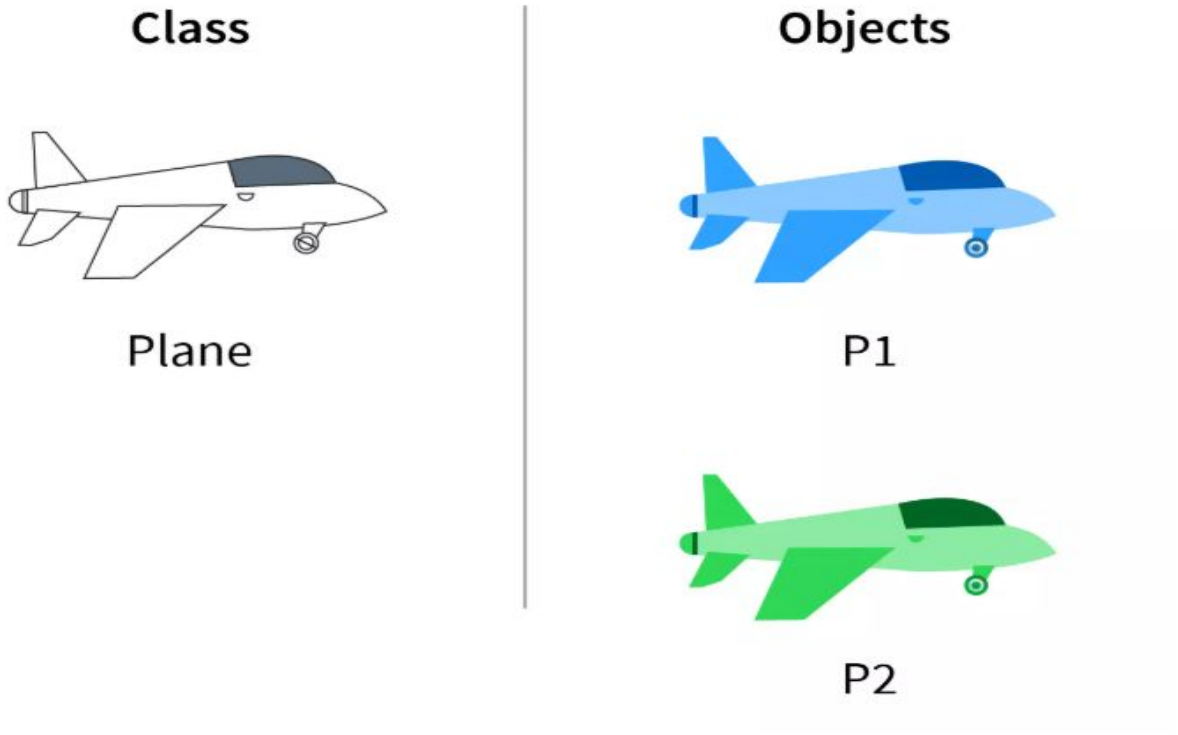
- seatAvailability()- this checks how many seats are available or vacant on that plane.
- selectClass()- lets you choose the class in which you want to travel- eg. business class or economy class.

Object in java

What is an Object in Java?

- Objects are the building blocks of OOP.
- A Java program is mostly a collection of objects talking to other objects by invoking each other's method.
- At a run time, when JVM encounters the "new" keyword, it will use the appropriate class o make an object which is an instance of that class. That object will have its state and access to all of the behavior defined by the class.

Relationship between object and class



The figure shows that the class Plane acts as a template to create objects which contain the attributes of the class(Plane).

Methods in Java Class

Syntax: Method Declaration

```
accessModifier returnType methodName(parameters..)
{
   //logic of the function
}
```

- **methodName:** Represents the identifier that can be used to call the method when required.

- **parameters:** These are the arguments passed into a method necessary for the function's logic. We can pass data to the methods by specifying them within the parentheses if the methods have data.

Methods in Java Class

Access Modifiers

- **Public Access Modifier**: Methods declared with public can be accessed from any other class.
- **Private Access Modifier:** Methods declared with private can only be accessed within the same class.
- **Protected Access Modifier**: Methods declared with protected can be accessed within the same package and by subclasses.
- **Default Access Modifier** (Package-Private): If no access modifier is specified, the method has default (package-private) access. Methods can only be accessed within the same package.

Methods in Java Class

Access Modifiers

```
class Demo {
    public int pubVar = 10;
    protected int proVar = 20;
    int defVar = 30;
    private int priVar = 40;
public void showData() {
        System.out.println("Inside Demo class:");
        System.out.println("Public: " + pubVar);
        System.out.println("Protected: " + proVar);
        System.out.println("Default: " + defVar);
        System.out.println("Private: " + priVar);
    }
}
```

Methods in Java Class

Access Modifiers

```
public class Main {
    public static void main(String[] args) {
        Demo obj = new Demo();
      System.out.println("From Main class:");
      System.out.println("Public: " + obj.pubVar);
      System.out.println("Protected: " + obj.proVar);
      System.out.println("Default: " + obj.defVar);
      System.out.println("Private: " + obj.priVar);
      obj.showData(); // Accessing private through method
    }
}
```

JAVA TECHNOLOGIES

There are 3 ways to initialize object in Java.

- By reference variable
- By method
- By constructor

Object and Class Example: Initialization through reference

```
class Student{
 int id;
 String name;  }
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sunil";
  System.out.println(s1.id+" "+s1.name);
 }  }
```

Object and Class Example: Initialization through method

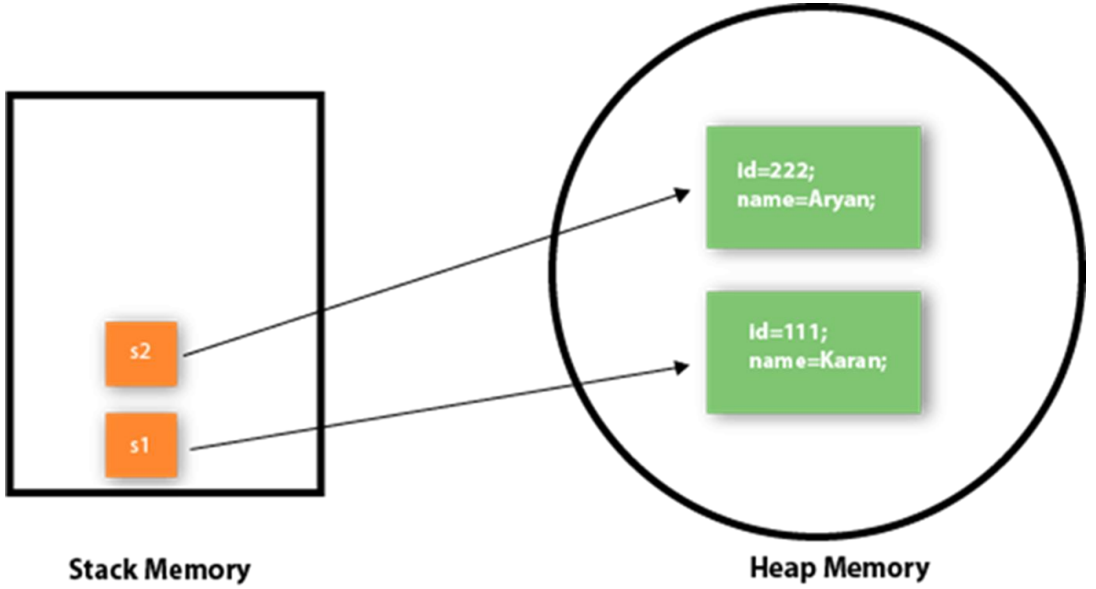
```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;   }
 void displayInformation() {System.out.println(rollno+" "+name);}
}
```

Object and Class Example: Initialization through method

```
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 } }
```

Object and Class Example: Initialization through method

```
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 } }
```



Object and Class Example: Initialization through a constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

1. Default Constructor

```
class student
{
public static void main(String args[])
{
student b=new student( );
}
}
```

1. Default Constructor/ No argument constructor

```
class student
{                       // No argument constructor
student( )
 {
System.out.println(“student Default Constructor");
 }

public static void main(String args[])
{
student b=new student( );
}
}
```

2. Object and Class Example: Initialization through a constructor

```
class Student3
{
int id;
String name;

Student3()
{
Id=2911;
Name="SAM";
}
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
{
Student3 s1=new Student3();
Student3 s2=new Student3();
s1.display();
s2.display();
} }
```

3. Java Parameterized Constructor

```
class Student4
{
    int id;
    String name;
    Student4(int i, String n)
    {
    id = i;
    name = n;
    }
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[])
  {
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

Practice program

```
class main {
  private String name;

main() {
    System.out.println("Constructor
                Called:");
    name = "PESU";
  }
public static void main(String[] args)
 {
    main obj = new main();
    System.out.println("The name is " +
          obj.name);
  }
}
```

Java program to initialize the values from one object to another object.

Copy constructor

```
class Student6
{
   int id;
   String name;
   Student6(int i,String n)
   {
   id = i;
   name = n;
   }
```

```
Student6(Student6 s)
   {
   id = s.id;
   name =s.name;
   }
```

Cont…

```
void display()
{
System.out.println(id+" "+name);
}
public static void main(String args[]){
Student6 s1 = new Student6(111,"Karan");
Student6 s2 = new Student6(s1);
s1.display();
s2.display();  } }
```

Practice program:

Design a class Ticket with attributes: passengerName, trainNumber, and seatNumber.

- Write a parameterized constructor to assign values when creating an object.
- Create multiple tickets and print passenger details.

Practice program:

Create a class Product with attributes: productName, price, and quantity.

- Use a constructor to initialize product details.
- Write a method to calculate and display the total cost (price * quantity).

Write JAVA program to demonstrate constructor overloading

Note:

1. You can create any number of objects of a class.
2. The process of creating an object of a particular class is called **instantiating of an object**. (instantiation)
3. The object is called an instance of class.

Four Pillars of OOPs in Java

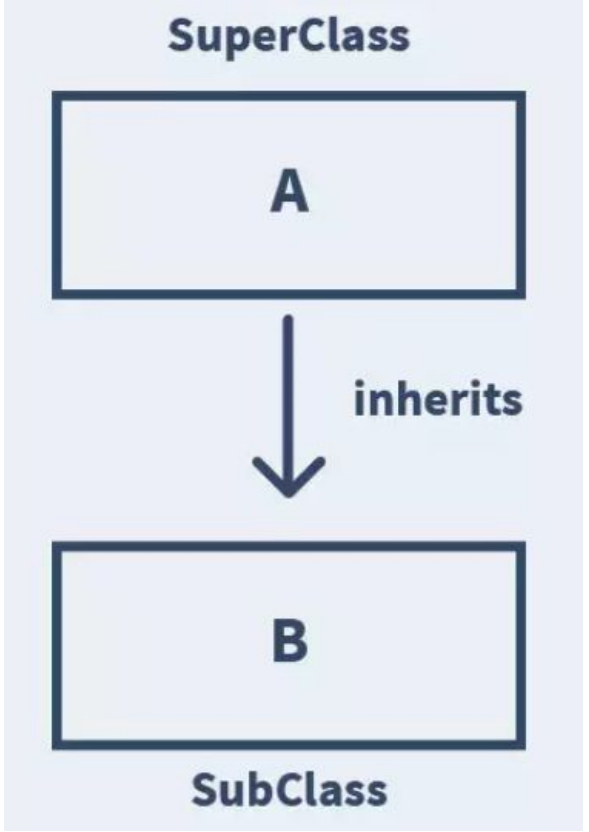
The following are the basic principles of OOP in Java. They are known as the principles as the entire OOP lays its foundation on these principles.

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Inheritance

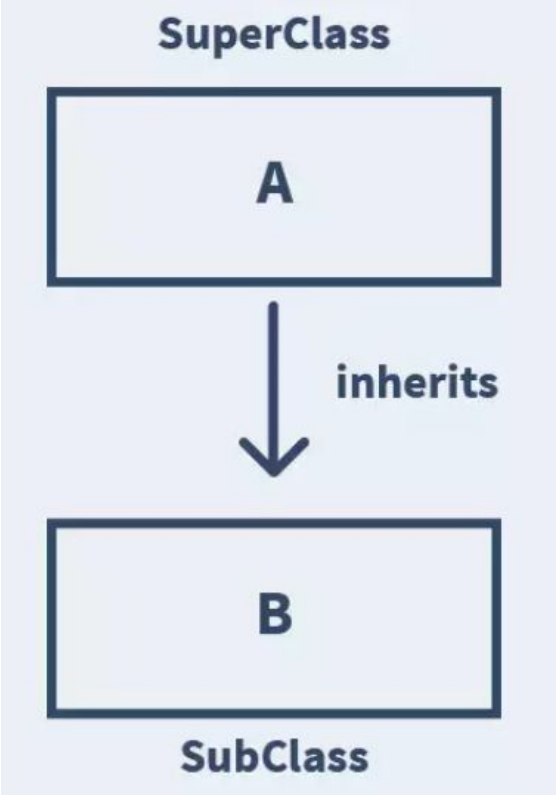
What is Inheritance in Java

- Inheritance is a mechanism of deriving one class from another.
- It allows the derived class to inherit features from a parent class, the features include (fields and methods).
- The parent class is also known as
  the superclass or base whereas the derived class is known as a subclass or child class.
- Inheritance is a property of OOPs in which member functions and data members of one class can be used by another class.



Inheritance

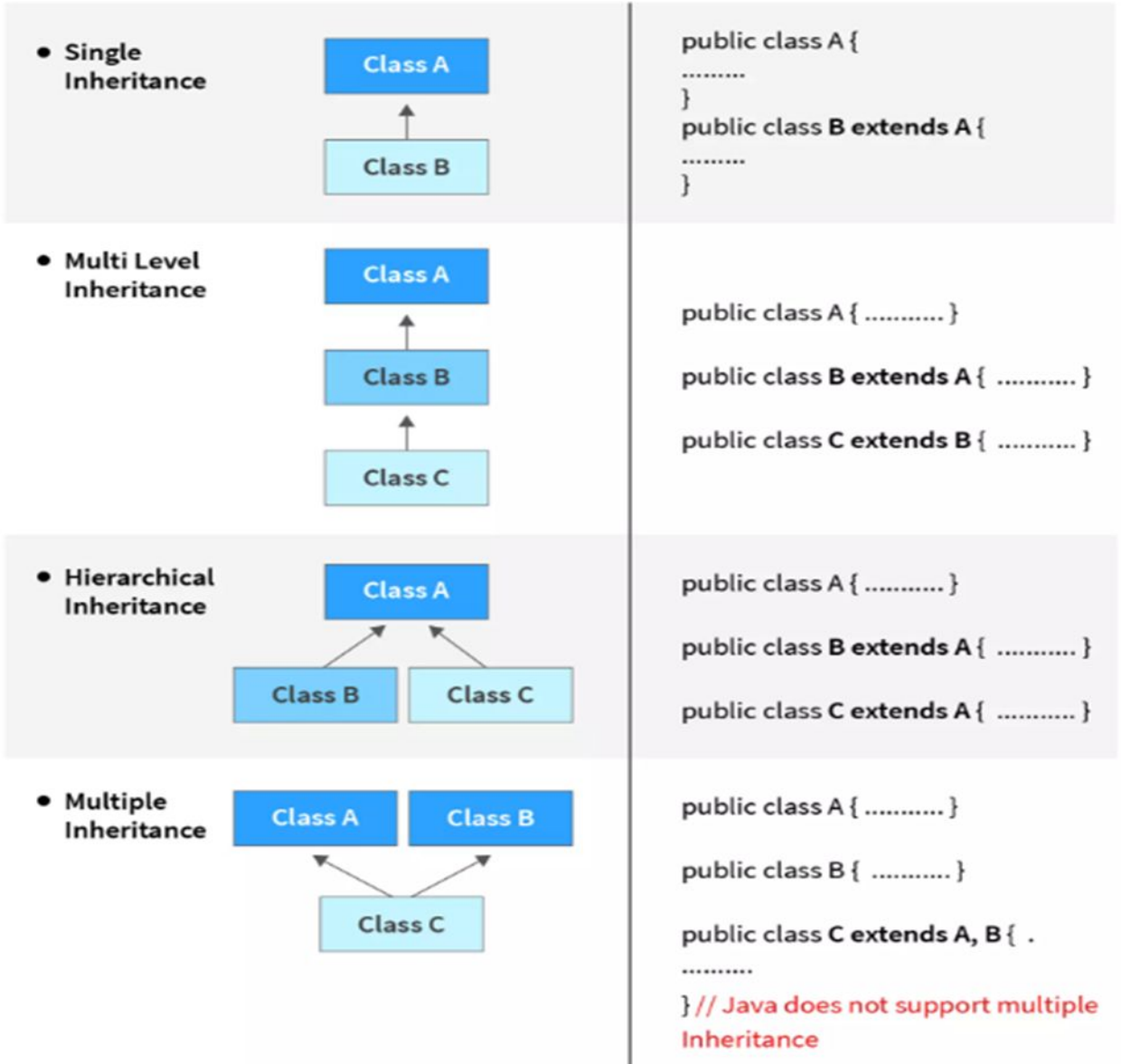
- Reusability of code can be achieved because of inheritance. If we want to create a class that already has some common methods which we want to define within another class, we can use the already existing class as a base class or a parent class and then inherit from it.
- Hence the code can be reused as we don't need to write the same piece of code again and can use it in the derived class or child class.
- Base class- a class from which member functions and data members are used by another class.
- Derived class- a class that uses the properties of the base class and can also add its properties.



Inheritance

Types of Inheritance :

- Single
- Multilevel
- Multiple
- Hybrid
- Hierarchical



Single Inheritance

Syntax

```
class ParentClass {
   // parent class members (variables + methods)
}
class ChildClass extends ParentClass {
   // child class members (variables + methods)
}
```

- **extends** keyword is used for inheritance.
- The child class can use all non-private members of the parent class.

Single Inheritance

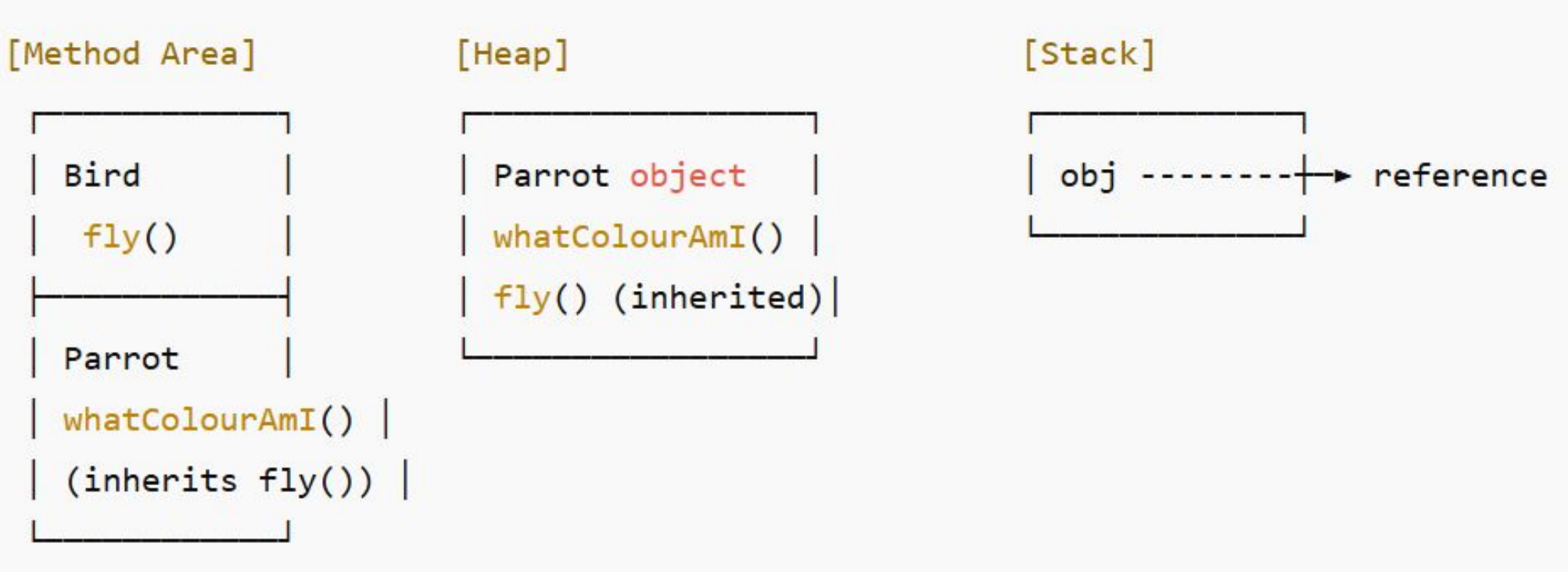
Example

```
class Bird {
  void fly() {
    System.out.println("I am a Bird");
  } }
// Inheriting SuperClass to SubClass
class Parrot extends Bird {
  void whatColourAmI() {
    System.out.println("I am green!");
  } }
```

```
class Main {
  public static void main(String args[]) {
    Parrot obj = new Parrot();
    obj.whatColourAmI();
    obj.fly();
  }
}
```

Single Inheritance

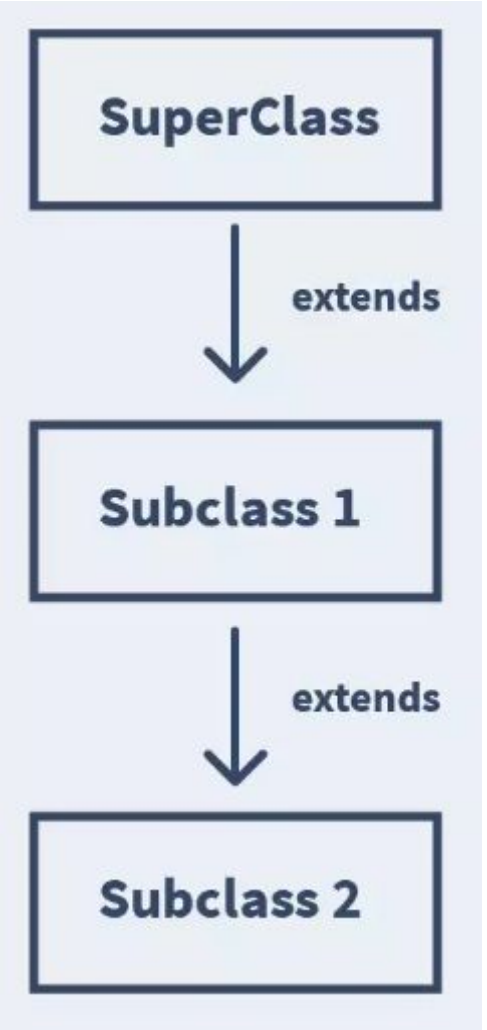
How JVM Works



Multilevel Inheritance

- This is an extension to single inheritance in Java, where another class again inherits the subclass, which inherits the superclass.

```
class ClassA {
}
class ClassB extends ClassA {
}
class ClassC extends ClassB {
}
class Main {
    public static void main(String[] args) {
        ClassC obj = new ClassC();
 }  }
```



Multilevel Inheritance

```
class Bird {
  void fly() {
    System.out.println("I am a Bird");
  } }

class Parrot extends Bird {
  void whatColourAmI() {
    System.out.println("I am green!");
  }
}
```

```
class SingingParrot extends Parrot {
  void whatCanISing() {
    System.out.println("I can sing Opera!");
  } }

class Main {
  public static void main(String args[]) {
    SingingParrot obj = new SingingParrot();
    obj.whatCanISing();
    obj.whatColourAmI();
    obj.fly();
  } }
```

Access modifiers using Inheritance

```
class Person {
    public String name;
    protected int age;
    String address;
    private String aadharNo;
    public void setAadhar(String aadhar) {
        aadharNo = aadhar;
    }
```

Multilevel Inheritance

```
  public void showPerson() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Address: " + address);
        System.out.println("Aadhar: " + aadharNo);
    }
}
```

Multilevel Inheritance

```
class Student extends Person {
    public String studentId; // Public field in child

    public void showStudent() {
        System.out.println("\n--- Student Details ---");
        System.out.println("Name: " + name);       // public
        System.out.println("Age: " + age);         // protected
        System.out.println("Address: " + address); // default
        System.out.println("Aadhar: " + aadharNo);
        System.out.println("Student ID: " + studentId);
    }
}
```

Multilevel Inheritance

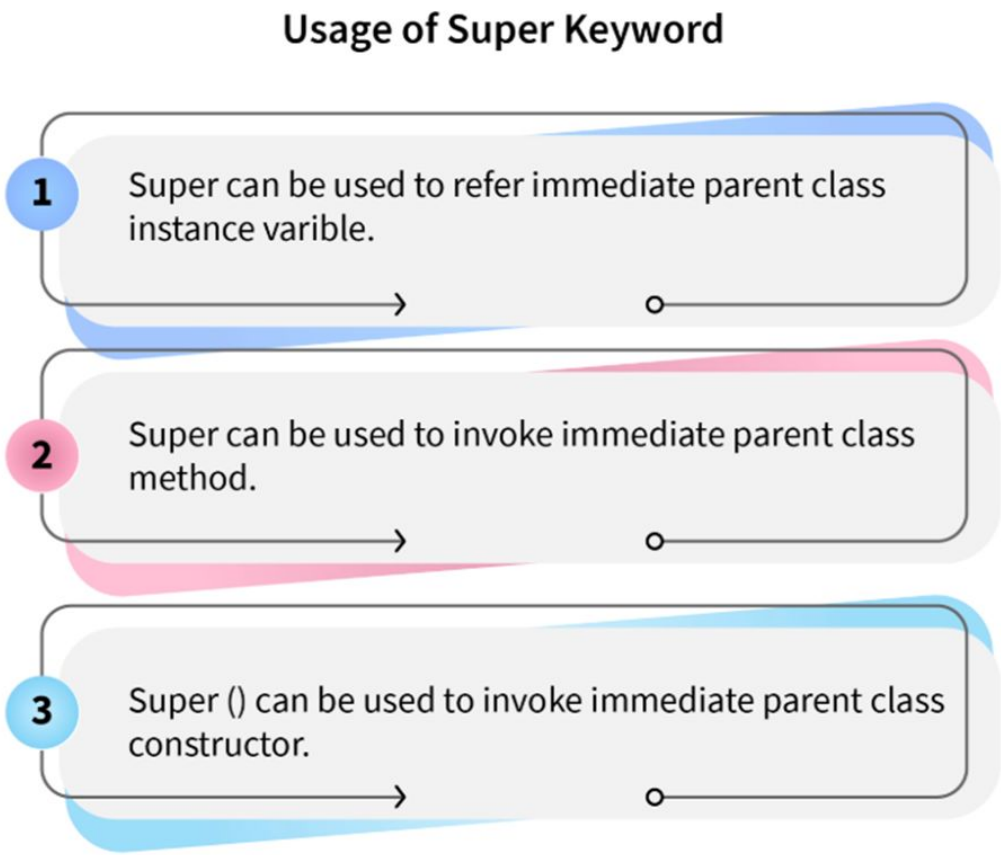
```
class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "Ananya";
        s.age = 19;
        s.address = "Mysore";
        s.studentId = "ST101";

        s.setAadhar("XXXX-YYYY-ZZZZ");
        s.showStudent();
        s.showPerson();
    }
}
```

Multilevel Inheritance

```
class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "Ananya";
        s.age = 19;
        s.address = "Mysore";
        s.studentId = "ST101";

        s.setAadhar("XXXX-YYYY-ZZZZ");
        s.showStudent();
        s.showPerson();
    }
}
```

Super Keyword in Inheritance

- The super keyword in Java is used as a reference variable to access objects from parent classes.
- It allows access to parent class constructors, members, and methods in the derived class.
- The keyword plays a crucial role in inheritance and polymorphism concepts.



Super Keyword in Inheritance

```
class School {
   int id;
  String name = "SSM Public School";
}

class Teacher extends School {
   int id;
  String name=“Dehli Public School”
  void printSchoolName() {
  System.out.println(“School Name:”+name);
  System.out.println("School Name: " + super.name);
  }

 public static void main(String[] args) {
   Teacher ob = new Teacher();
   ob.printSchoolName();
  }
}
```

Super Keyword in Inheritance : super()

```
class School {
    int id;
    String name;
        School() {
        name = "SSM Public School";
        System.out.println("School
constructor called");
    }
}
```

```
class Teacher extends School {
    int id;
    String name;
      Teacher() {
       super(); // calls School() constructor first
       name = "Delhi Public School";
       System.out.println("Teacher constructor
called");
    }
```

Super Keyword in Inheritance

```
    void printSchoolName() {
        System.out.println("Child class name: " + name);
        System.out.println("Parent class name: " + super.name);
    }

    public static void main(String[] args) {
        Teacher ob = new Teacher();
        ob.printSchoolName();
    }
}
```

Super Keyword in Inheritance : calling parent class method

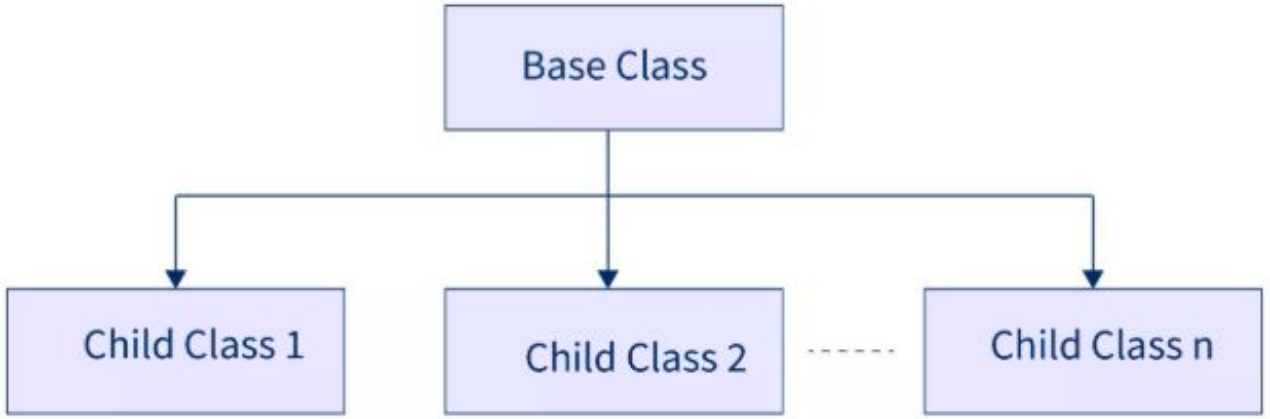
```
class School {
String name = "SSM Public School";
 void showSchool() {
    System.out.println("School Name from Parent: " + name);
}  }

class Teacher extends School {
   String name = "Delhi Public School";
  void showSchool() {
       System.out.println("School Name from Child: " + name);
        super.showSchool();
   }
```

Super Keyword in Inheritance : calling parent class method

```
    public static void main(String[] args) {
        Teacher ob = new Teacher();
        ob.showSchool();
    }
}
```

Hierarchical Inheritance in Java

- Hierarchical inheritance is **one** of the types of inheritance where multiple child classes inherit the methods and properties of the same parent class.
- Hierarchical inheritance reduces the code length and increases the code modularity.
- For Hierarchical inheritance to occur, there must be at least **two or more** sub-classes that extend (or inherit) the same superclass.



Hierarchical Inheritance in Java

```
class BaseClass
{
  int parentNum = 10;
}

class SubClass1 extends BaseClass
{
  int childNum1 = 1;
}
```

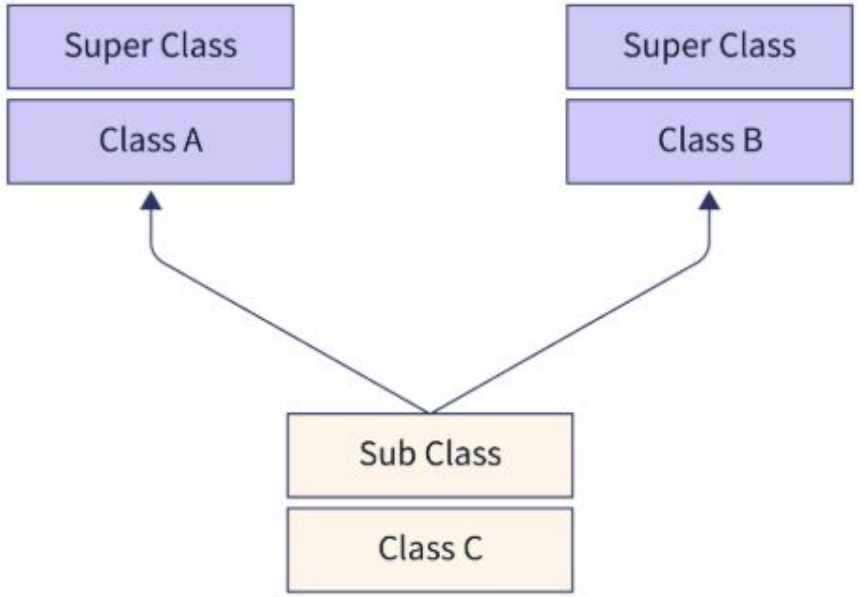
```
class SubClass2 extends BaseClass
{
  int childNum2 = 2;
}

class SubClass3 extends BaseClass
{
  int childNum3 = 3;
}
```

Hierarchical Inheritance in Java

```
public class Main
{
public static void main(String args[])
  {
    SubClass1 childObj1 = new SubClass1 ();
    SubClass2 childObj2 = new SubClass2 ();
    SubClass3 childObj3 = new SubClass3 ();
    System.out.println("parentNum * childNum1 = " + childObj1.parentNum *
childObj1.childNum1);
    System.out.println("parentNum * childNum2 = " + childObj2.parentNum *
childObj2.childNum2);
    System.out.println("parentNum * childNum3 = " + childObj3.parentNum *
childObj3.childNum3);      } }
```

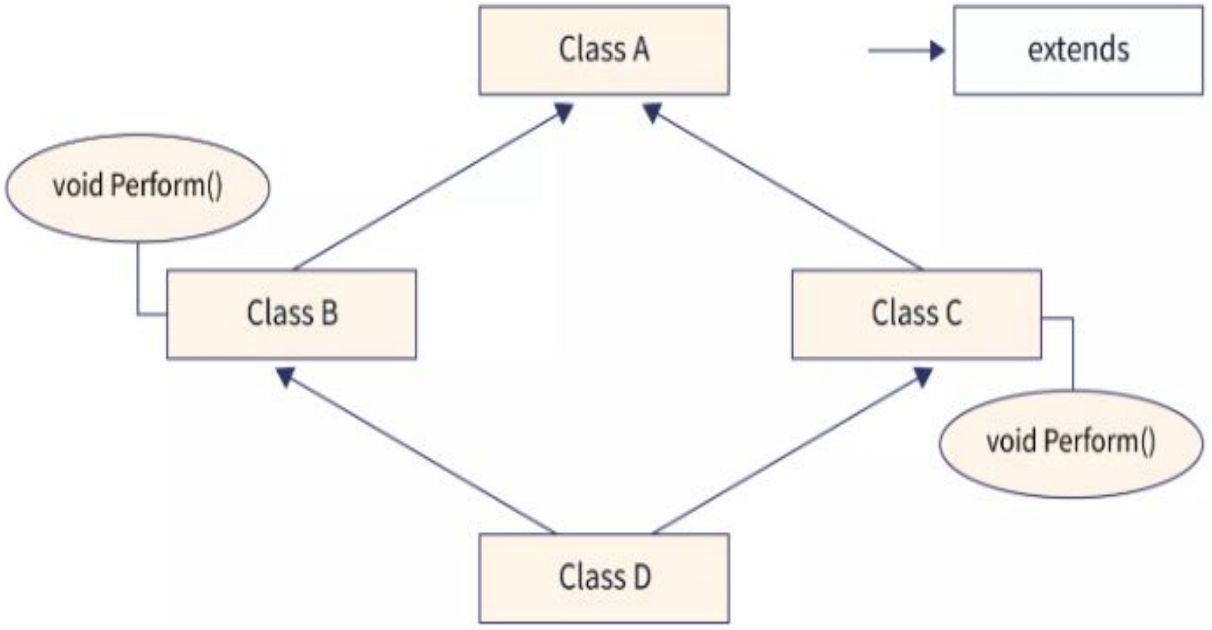
Multiple Inheritance in Java

- Multiple Inheritance is the process in which a subclass inherits more than one superclass.
- we can observe that Class C(sub-class) inherits from more than one superclass i.e., Class A, Class B.
- Java does not support Multiple Inheritance, but we can use Interfaces to achieve the same purpose.



Multiple Inheritance in Java

- Implementation of Multiple Inheritance in Java is not supported by default to avoid several ambiguity issues.
- One kind of ambiguity is the **Diamond Problem.**



Multiple Inheritance in Java

```
class A
{
   public void execute()  {
      System.out.println("Hi.. Executing
From Class A");
   } }
class B
{
   public void execute()    {
      System.out.println("Hi.. Executing
From Class B");
   } }
```

```
class C extends A, B
{

}

public class Main
{
public static void main(String[] args)
{
   C obj = new C();
   obj.execute();     //Ambiguity
} }
```

Interface in Java

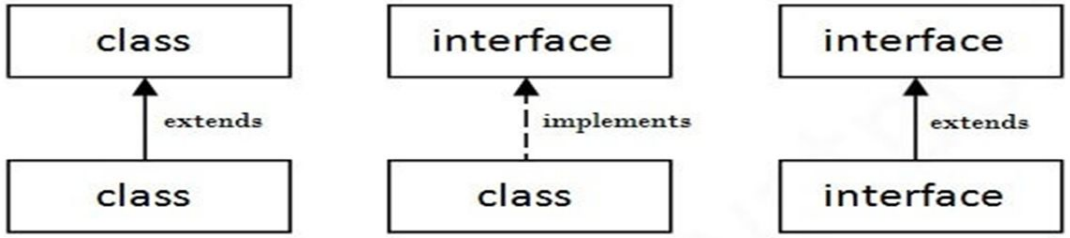
- Interfaces in Java are a set of abstract and public methods we want our classes to implement.
- It is the blueprint of a class and contains static constants and abstract methods.
- Interfaces are used to achieve abstraction and implement multiple inheritance.
- These methods are public and abstract by default(you don't have to explicitly use the "abstract" keyword), and any class implementing your interface will need to provide implementations of those methods.
- Cannot create objects of an interface directly (only references allowed).
- Interfaces are used to represent capabilities/behavior (e.g., Runnable, Comparable).

Interface in Java

- A class implementing an interface must override all abstract methods.
- Interfaces help achieve loose coupling between code modules.
- An interface can be used when we want to achieve 100% abstraction. On the other hand, abstract classes can be used to achieve anything between 0–100% abstraction.
- An interface cannot have constructors because we cannot create objects of an interface.
- If you want a class to achieve multiple inheritances, there is only one way: interfaces.
- If the methods in it are made private or protected, then a compilation error will be thrown.

Interface in Java

- Declared using the interface keyword.
- A class uses the implements keyword to use an interface.
- From Java 8, interfaces can also have default and static methods.
- From Java 9, they can also have private methods.
- Variables in an interface are always:public, static, and final (constants).
- Interfaces support multiple inheritance (a class can implement multiple interfaces).
- Interfaces provide a way to achieve abstraction.



Interface in Java

Syntax:

```
interface InterfaceName {
datatype CONSTANT_NAME = value;
    returnType methodName1();
    returnType methodName2();
        default void defaultMethod() {
            }
         static void staticMethod() {
            }
} }
```

Interface in Java

Syntax:

```
interface Car {
      void start();
  }
```

```
interface Vehicle {
    void start(); // implicitly public & abstract
    void fuelType();
}
```

- Methods declared inside an interface are implicitly marked as public and abstract, and
- variables declared inside an interface are implicitly marked as public, static, final by the compiler.

Interface in Java

```
class HybridCar implements Car, Vehicle {

    public void start() {
        System.out.println("Hybrid Car starting...");
    }

    public void fuelType() {
        System.out.println("Fuel type: Petrol + Electric");
    }
}
```

Interface in Java

```
class Main {
    public static void main(String[] args) {
        HybridCar h = new HybridCar();
        h.start();
        h.fuelType();

        Car c = h;
        c.start();

        Vehicle v = h;
        v.start();
        v.fuelType();
    }  }
```

Note:

- Compile-time (reference type) ,decides what methods you can call.
- Runtime (object type), decides which method's body executes.

Example on Diamond problem in Java

```
interface Vehicle {
    void start();
}

interface Car extends Vehicle {
    void fuelType();
}

interface Bike extends Vehicle {
    void wheels();
}
```

Diamond problem in Java

```
class HybridVehicle implements Car, Bike {
    public void start() {
    System.out.println("Hybrid Vehicle starting...");
  }

  public void fuelType() {
    System.out.println("Fuel type: Petrol + Electric");
  }

  public void wheels() {
    System.out.println("This vehicle has 4 wheels");
  }
}
```

Diamond problem in Java

```
public class Main {
    public static void main(String[] args) {
        HybridVehicle hv = new HybridVehicle();
        hv.start();
        hv.fuelType();
        hv.wheels();
    }
}
```

Diamond problem in Java

Example 2: Overriding interface methods

```
interface Vehicle {
    default void start() {
        System.out.println("Vehicle starting...");
    }
}
interface Car extends Vehicle {
    default void start() {
        System.out.println("Car starting...");
    }
}
```

//Note: An interface method is always abstract (unless it's default/static).

Diamond problem in Java

```
interface Bike extends Vehicle {
    default void start() {
        System.out.println("Bike starting...");
    } }

class HybridVehicle implements Car, Bike {
        public void start() {
        System.out.println("Hybrid Vehicle starting...");
        Car.super.start();
        Bike.super.start();
    }
}
```

Diamond problem in Java

```
public class Main {
    public static void main(String[] args) {
        HybridVehicle hv = new HybridVehicle();
        hv.start();
    }
}
```

Abstract Classes

- An abstract class cannot be instantiated and is primarily meant to be subclassed by other classes.
- In Java, an abstract class is a class declared with the abstract keyword. It can have abstract and non-abstract methods.
- An abstract method is a method declared without any implementation.
- Abstract classes in Java implement the concept of Abstraction in Object-Oriented Programming.
- Abstract is a small description that gives you an imaginative idea.
- Abstract Classes in Java simply declares the methods that need to be implemented by the class, which extends the abstract class.

Abstract Classes

- The abstract class hides the internal details of method implementation showing only essential information to the user - the method name.
- Can contain instance variables, static variables, and constants like a normal class.
- Supports single inheritance directly.
- Abstract methods cannot be private (because then they cannot be overridden).They can be public or protected.
- A class cannot be both final and abstract (final classes cannot be extended).
- An abstract method cannot be final or static (because they need to be overridden).

Abstract Classes

- You cannot create objects of an abstract class directly.
  Student s=new Student(); // **ERROR**
- Abstract class references can point to subclass objects.
  Student a=new MCA_Student();

Upcasting : Create reference variable of abstract class student, assign object of subclass to reference variable, i.e object is stored in the reference variable

- Supports polymorphism.

Abstract Classes

Syntax :

```
abstract class ClassName {
    abstract returnType methodName(parameters);
     returnType normalMethod(parameters) {
      }
}
class SubClassName extends ClassName {
@Override
returnType methodName(parameters) {
          }
}
```

@Override tells the compiler that this method overrides a superclass method.

Example : Abstract Classes

```
abstract class College {

    String collegeName = "PES University";
    abstract void department();
    void showCollege() {
        System.out.println("Welcome to " + collegeName);
    }   }

class ComputerScience extends College {
    @Override
    void department() {
        System.out.println("Department: Computer Science & Engineering");
    }   }
```

```
class Electronics extends College {
    @Override
    void department() {
    System.out.println("Department: Electronics & Communication Engineering");
    }    }

public class Main {
    public static void main(String[] args) {
        College c = new ComputerScience();
        c.showCollege();       // Non-abstract method
        c.department();        // Implemented abstract method
        College c = new Electronics();
        c.showCollege();
        c.department();
}     }
```

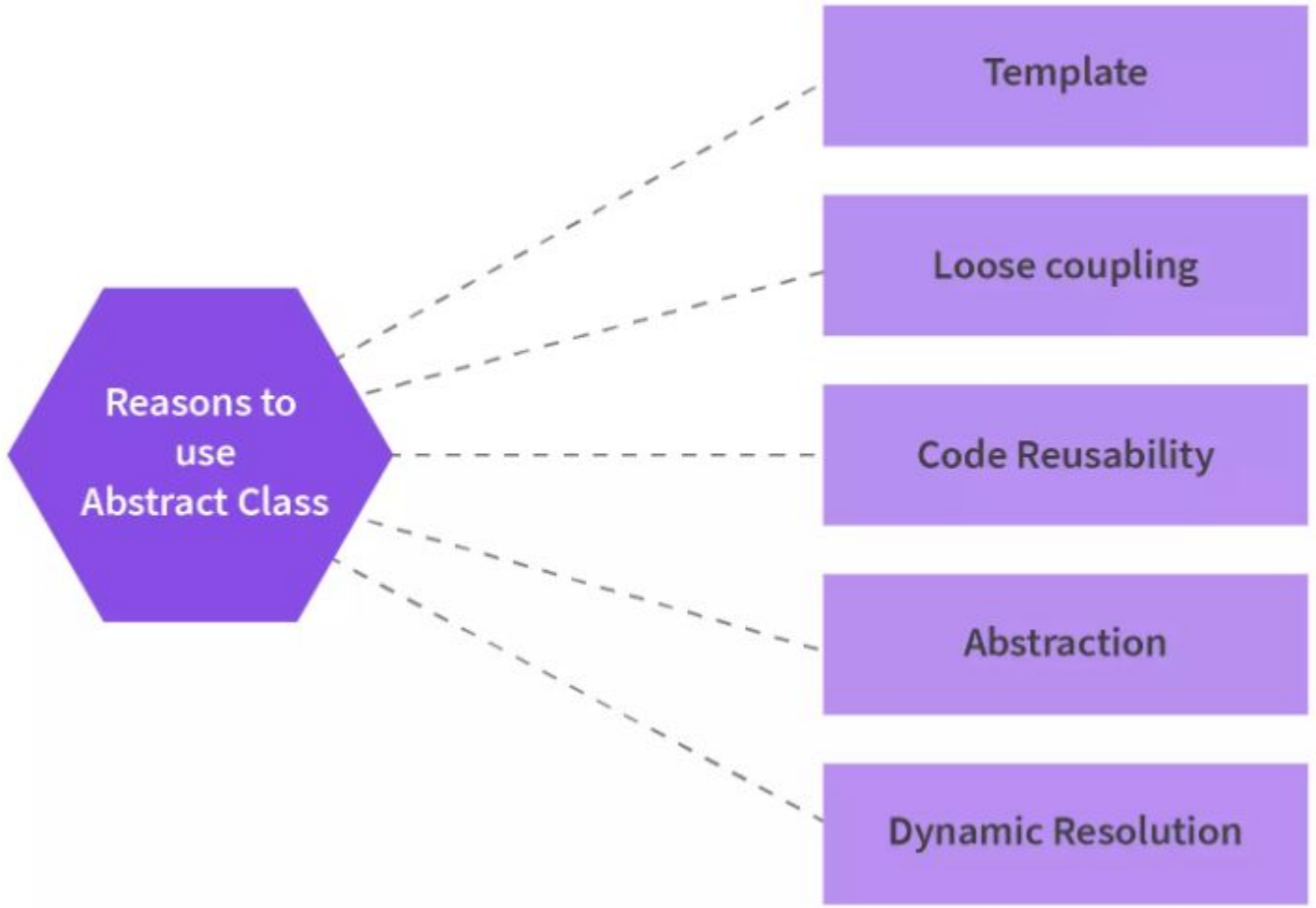
Constructors in Abstract Class

- An abstract class cannot be instantiated directly, but it can have a constructor.
- The constructor is called when a subclass object is created.
- It is mainly used to initialize common fields of the abstract class.

Constructors in Abstract Class

```
abstract class DisplayTest {
   protected final String P;
   abstract void display(String str);

   DisplayTest(String P){
      this.P = P;
   }  }

public class Main extends DisplayTest
{
   public void display(String str) {
      System.out.println(str + " " + P );
}

Main(String P){
      super(P);
   }

public static void main(String[] args) {
DisplayTest displayTest = new Main(“University");
displayTest.display(“PES");

}  }
```

Advantages of abstract Class



Advantages of abstract Class

Template Programming

Abstract classes provide a blueprint to be followed by the classes that extend the Abstract class. Abstract Class because it gives a predefined template for any future specific class that you might need.

Loose Coupling

A method or class is almost independent in loose coupling, and they are less dependent on each other. In other words, the more one class or method knows about another class or method, the more tightly coupled the structure becomes. The more loosely connected the structure, the less the classes or methods know about each other.

Advantages of abstract Class

Dynamic Method Resolution

- The Abstract Classes enable us with dynamic method resolution or the **dynamic method dispatch process.**
- The dynamic Method Resolution is a procedure where, at runtime, the calling of an overridden method is resolved. This is how **RunTime polymorphism** is implemented.
- Whenever an overridden method is called by reference, the JVM is responsible for checking the version of the method to execute depending upon the type of object.

JAVA TECHNOLOGIES

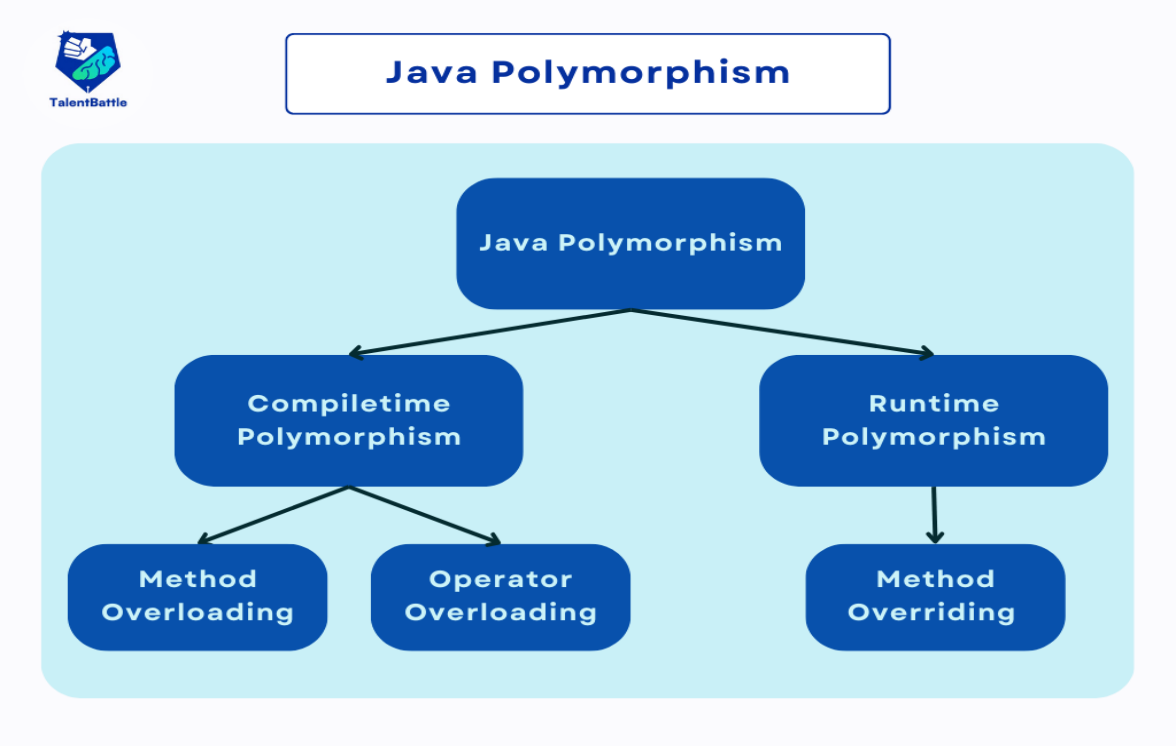
Polymorphism

- Word comes from "poly" (many) + "morph" (forms).
- Polymorphism means the ability of an object to take many forms.
- In Java, it allows one interface to be used for different implementations.

Types of Polymorphism in Java

Java supports two types of polymorphism:

1. Compile-time polymorphism
2. Runtime polymorphism



Polymorphism

Compile-time Polymorphism (Static binding / Method Overloading)

- Achieved by method overloading (same method name, different parameters).
- Resolved at compile time.
- Compile-time polymorphism in Java, also known as static polymorphism or static method dispatch, is a critical feature in Java that enables multiple methods within the same class to share the same name but differ in their parameter lists.
- This polymorphism is facilitated through method overloading, where the decision on which method to invoke is made during compile time rather than at runtime.

Example : Compile-time Polymorphism (Static binding / Method Overloading)

```
class CompileTime {

static int perimeter(int a) {
    return 4 * a;
  }

static int perimeter(int l, int b) {
    return 2 * (l + b);
  }
}
```

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

  System.out.println("Side of square : 4\n Perimeter of square will be : " +
  Compiletime.perimeter(4) + "\n");

  System.out.println("Sides of rectangle are : 10, 13\n Perimeter of rectangle will be : " +
  Compiletime.perimeter(10, 13));

}
}
```

Polymorphism

Runtime Polymorphism (Dynamic binding / Method Overriding)

- Achieved by method overriding (subclass provides its own implementation of a method defined in superclass).
- Resolved at runtime using **dynamic method dispatch.**
- Dynamic polymorphism arises in situations where multiple classes are linked through inheritance.
- Methods overridden in a subclass are accessed via a reference from the parent class, and the specific method to be executed is determined based on the actual type of the object at runtime. The Java Virtual Machine (JVM) identifies the object's type and the corresponding method to invoke.

Example : Dynamic Method dispatch

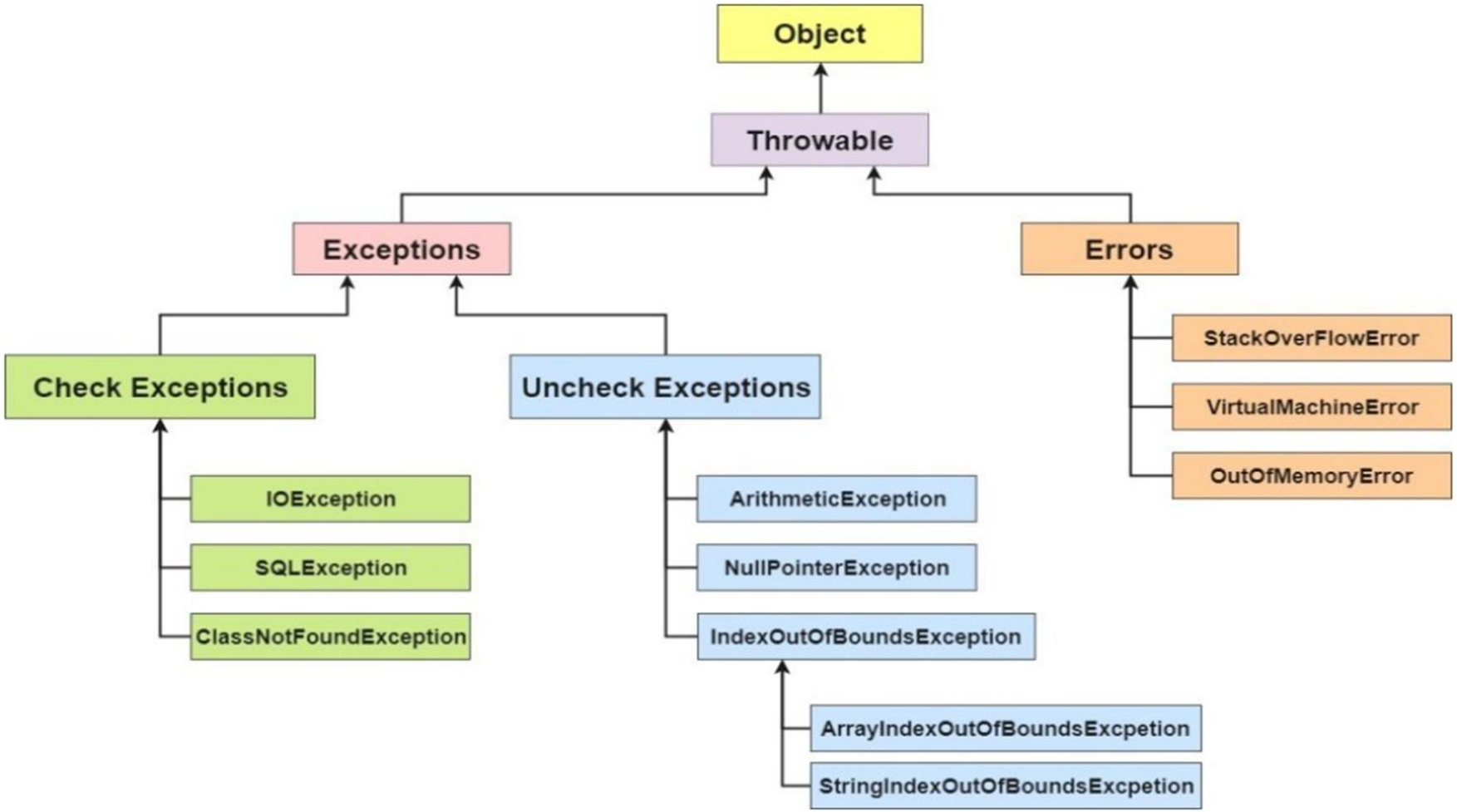
```
class College {
    void department() {
        System.out.println("General Department of College");
    }
}

class ComputerScience extends College {
    @Override
    void department() {
        System.out.println("Department: Computer Science & Engineering");
    }
}
```

JAVA TECHNOLOGIES

```
class Mechanical extends College {
    @Override
    void department() {
        System.out.println("Department: Mechanical Engineering");
    }
}
public class Main {
    public static void main(String[] args) {
        College c;   // Reference of superclass
        c = new ComputerScience();
        c.department();
        c = new Mechanical();
        c.department();
    }
}
```

JAVA TECHNOLOGIES

Exception Handling :



Exception Handling :

Types of Exceptions Handling in Java

1. Checked Exceptions

- Checked exceptions are those exceptions that are checked at compile time by the compiler.
- The program will not compile if they are not handled.
- These exceptions are child classes of the Exception class.
- IOException, ClassNotFoundException, and SQL Exception are a few of the checked exceptions in Java.

Exception Handling :

2. Unchecked Exceptions

- Unchecked exceptions are those exceptions that are checked at run time by JVM, as the compiler cannot check unchecked exceptions,
- The programs with unchecked exceptions get compiled successfully but they give runtime errors if not handled.
- These are child classes of Runtime Exception Class.
- ArithmeticException, NullPointerException, NumberFormatException, IndexOutOfBoundException are a few of the unchecked exceptions in Java.

Exception Handling :

Error = a type of Throwable, usually **not handled** with try–catch.

- An error is also an unwanted condition but it is caused due to lack of resources and indicates a serious problem.
- Errors are irrecoverable, they cannot be handled by the programmers.
- Errors are of unchecked type only.
- They can occur only at run time.
- In java, errors belong to java.lang.error class.
- Eg: OutOfMemmoryError.

Exception Handling :

Exception

- An exception is an unwanted or unexpected event that occurs during the execution of the program, that disrupts the flow of the program
- Exceptional handling in Java is a very powerful mechanism as it helps to identify exceptional conditions and maintain the flow of the program as expected, by handling/avoiding the errors if occurred. In some cases, it is used to make the program user-friendly.
- Exceptions can be of both checked(exceptions that are checked by the compiler) and unchecked(exceptions that cannot be checked by the compiler) type.
- They can occur at both run time and compile time.
- In Java, exceptions belong to java.lang.Exception class.

Exception Handling

Exception handling in java is achieved using five keywords: try, catch, throw, throws, and finally. Here is how these keywords work in short.

- **Try block** contains the program statements that may raise an exception.
- **Catch block** catches the raised exception and handles it.
- **Throw keyword** is used to explicitly throw an exception.
- **Throws keyword** is used to declare an exception.
- **Finally block** contains statements that must be executed after the try block.

Exception Handling

```
try {
   code
}
catch and finally blocks . . .
```

```
try
{  }
catch (ExceptionType name)
{  }
catch (ExceptionType name)
{  }
```

```
catch (IOException | SQLException ex)
{  }
```

Exception Handling

Try block

- try block is used to execute doubtful statements which can throw exceptions.
- try block can have multiple statements.
- Try block cannot be executed on itself, there has to be at least one catch block or finally block with a try block.
- When any exception occurs in a try block, the appropriate exception object will be redirected to the catch block, this catch block will handle the exception according to statements in it and continue the further execution.
- The control of execution goes from the try block to the catch block once an exception occurs.

Exception Handling

```
class ExceptionExample {
  public static void main(String args[]) {
    try {
      // Code that can raise exception
      int div = 509 / 0;
    }
catch (ArithmeticException e) {
      System.out.println(e);
    }
    System.out.println("End of code");
  }
}
```

Exception Handling

```
class ExceptionExample {
  public static void main(String args[]) {
    try {
      // Code that can raise exception
      int div = 509 / 0;
    }
catch (ArithmeticException e) {
      System.out.println(e);
    }
    System.out.println("End of code");
  }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
End of code
```

Exception Handling

Catch block

- catch block is used to give a solution or alternative for an exception.
- catch block is used to handle the exception by declaring the type of exception within the parameter.
- The declared exception must be the parent class exception or the generated exception type in the exception class hierarchy or a user-defined exception.
- You can use multiple catch blocks with a single try block.

Exception Handling

```
public class EH2 {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {10, 1, 2, 3, 5, 11};
      System.out.println(myNumbers[10]);
   }
catch (Exception e) {
      System.out.println("Something went wrong.");
    }
 System.out.println(“End of the Code");
  } }
```

Exception Handling

```
public class EH3 {
  public static void main(String[] args) {
    try {
      int a[] = new int[5];
      a[5] = 30 / 0;    }
   catch (ArithmeticException e) {
     System.out.println("Arithmetic Exception occurs");     }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("ArrayIndexOutOfBounds Exception occurs");    }
   catch (Exception e) {
     System.out.println("Parent Exception occurs");   }
    System.out.println("End of the code");  }
}
```

Exception Handling

After execution of the try block, the Arithmetic Exception is raised and JVM starts to search for the catch block to handle the same.

After the exception is handled the flow of the program comes out from try-catch block and it will execute the rest of the code.

Exception Handling

```
try {   // main try

    //try-block2
    try {

      //try-block3
      try {
          }

    catch ( ) {
          }

        }
```

```
catch () {
        }
    }

catch () {
        }

catch () {
        }

catch () {
        }
```

Exception Handling

```
class EH4 {
  public static void main(String args[]) {
    //main try-block
    try {
      //try-block2
      try {
        //try-block3
        try {
          int arr[] = {1,2,3,4};
               System.out.println(arr[10]); }
catch (ArithmeticException e) {
System.out.print("Arithmetic Exception handled in try-block3");
} }
```

```
catch (ArithmeticException e) {
        System.out.print("Arithmetic Exception handled in try-block2"); } }

catch (ArithmeticException e3) {
System.out.print("Arithmetic Exception handled in main try-block"); }

catch (ArrayIndexOutOfBoundsException e4) {
System.out.print("ArrayIndexOutOfBoundsException handled in main try-
block"); }

catch (Exception e5) {
System.out.print("Exception handled in main try-block");  }
} }
```

Exception Handling

finally block

- finally block is associated with a try, catch block.
- It is executed every time irrespective of exception is thrown or not.
- finally block is used to execute important statements such as closing statement, release the resources, and release memory also.
- finally block can be used with try block with or without catch block.

```
try
{
   //Doubtful Statements
}
catch(Exception e)
{   }

finally
{
   //Close resources
}
```

Exception Handling

```
public class EH5 {
  public static void main(String[] args) {
    try {
      int data = 100/0;
      System.out.println(data);
    }
    catch (Exception e) {
      System.out.println("Can't divide integer by 0!");
    }
   finally {
      System.out.println("The 'try catch' is finished.");
    } }
}
```

JAVA TECHNOLOGIES

Java Final Vs Finally Vs Finalize

final	finally	finalize
final is the keyword and access modifier	finally block is used in java Exception Handling to execute the important code after try-catch blocks.	finalize is the method in Java.
final access modifier is used to apply restrictions on the variables, methods, classes.	finally block executes whether an exception occurs or not. It is used to close resources.	finalize() method is used to perform clean-up processing just before an object is a garbage collected.

Throw Keyword

- throw keyword in java is used to throw an exception explicitly.
- We can throw checked as well as unchecked exceptions (compile-time and runtime) using it.
- We specify the class of exception object which is to be thrown. The exception has some error message with it that provides the error description.
- We can also define our own set of conditions for which we can throw an exception explicitly using the throw keyword.

Throw Keyword

- The flow of execution of the program stops immediately after the throw statement is executed and the nearest try block is checked to see if it has a catch statement that matches the type of exception.
- It tries to find all the catch blocks until it finds the respective handler, else it transfers the control to the default handler which will halt the program.

Syntax:

```
throw new exception_class("error message");
```

Throw Keyword

```
public class EH6
{
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException(" You must be at least 18 years old for voting.");
    }
   else {
      System.out.println(" You are eligible for voting!");
    } }

  public static void main(String[] args) {
    checkAge(15); // Set age to 15 (which is below 18...)
  }    }
```

Throw Keyword

throw keyword is used to inform the user that he/she does not fit in the required criteria. If age is less than 18 then the throw keyword explicitly throws an arithmetic exception

Throws Keyword

- throws keyword in java is used in the signature of the method to indicate that this method might throw one of the exceptions from java exception class hierarchy.
- throws keyword is used only for **checked exceptions** like IOException as using it with unchecked exceptions is meaningless(Unchecked exceptions can be avoided by correcting the programming mistakes.)
- throws keyword can be used to declare an exception.

Throws Keyword

Syntax:

```
returnType methodName(parameters) throws ExceptionType {
    // method body
}
```

Throws Keyword

```
public class EH7 {
  static void checkAge(int age) throws ArithmeticException {
    if (age < 18) {
    //  throw new ArithmeticException(" You must be at least 18 years old for voting.");
        System.out.println("Arithmetic Error");
} else {
     System.out.println(" You are eligible for voting!");
    }
  }

  public static void main(String[] args) {
    checkAge(15);
  }
}
```

Throws Keyword

```
class Example2 {
    static void test() throws ArithmeticException, NullPointerException {
        int a = 10 / 0;
        String s = null;
        System.out.println(s.length());
    }
    public static void main(String[] args) {
        try {
            test();
        } catch (Exception e) {
            System.out.println("Exception handled");
        }
    }  }
```

JAVA TECHNOLOGIES

Throw and Throws Keyword

Throw	Throws
Java throw keyword is used to throw an exception explicitly in the program, inside any block of code	Java throws keyword is used in method or function signature to declare an exception that the method may throw while execution of code.
throw keyword can be used to throw both checked and unchecked exceptions.	throws keyword can be used only with checked exceptions.
throw is used within the method.	throws is used within the method signature.

Throw and Throws Keyword

Syntax: throw new exception_class("error message");	**Syntax:** void method() throws ArithmeticException
We can throw only one exception at a time.	We can declare multiple exceptions using the throws keyword that the method can throw.

Common Scenarios of Java Exceptions

1. ArithmeticException

Arithmetic exceptions is raised by JVM when we try to perform any arithmetic operation which is not possible in mathematics. One of the most common arithmetic exception that occurs is when we divide any number with zero.

```
int div = 100/0;
**Exception Raised:**
java.lang.ArithmeticException: / by zero
```

Common Scenarios of Java Exceptions

2. NullPointerException

NullPointerException occurs when a user tries to access variable that stores null values. For example, if a variable stores null value and the user tries to perform any operation on that variable throws NullPointerException.

```
String s = null;
System.out.println(s.length());
**Exception Raised:**
java.lang.NullPointerException: Cannot invoke "String.length()" because
"local1" is null
```

Common Scenarios of Java Exceptions

3. NumberFormatException

In java, variables have data types and certain operations are compatible with specific data types. Some functions are to be performed on numeric values, but if a variable with an incompatible data type like string is given as an input, it results in NumberFormatException.

```
String s = "Java";
int i = Integer.parseInt(s);
**Exception Raised:**
java.lang.NumberFormatException: For input string: "Java"
```

Common Scenarios of Java Exceptions

4. ArrayIndexOutOfBoundsException

While accessing an array if we access an element that is present in an array it will execute properly without throwing any exceptions, but accessing an index that is not present throws ArrayIndexOutOfBoundsException.

```
int arr[] = new int[3];
System.out.println(arr[3]);
*//Maximum index we can access is 2 as indexing in array starts from 0.*
**Exception Raised:**
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for
length 3
```

Common Scenarios of Java Exceptions

5. StringIndexOutOfBoundsException

It is the same as ArrayIndexOutOfBoundsException but it is for strings instead of arrays. Here if the length of a string is less than what we are trying to access then we get StringIndexOutOfBoundsException.

.

```
String s1 = "I am learning Java.";
System.out.println("String length is:" + s1.length());
System.out.println("Length of substring is:" + s1.substring(32));
**Exception Raised:**
String length is:19
java.lang.StringIndexOutOfBoundsException: begin 32, end 19, length 19
```

Advantages

- Program execution continues if an exception is raised and handled, it does not terminate the code being executed abruptly.
- With exception handling it is easy to identify errors that occurred while execution.
- The exceptions thrown in a Java program are objects of a class. Since the Throwable class overrides the toString() method, you can obtain a description of an exception in the form of a string.
- Java provides several super classes and subclasses that group exceptions based on their type it is easy to differentiate and group exceptions.

Practice Program

ATM Withdrawal Simulation

Simulate an ATM system with the following tricky cases:

1. Custom checked exception if balance is too low.
2. Unchecked exception (NullPointerException) if ATM card is not inserted.
3. Nested try–catch where inner block handles one exception but outer handles another.
4. A finally block that always runs (important in real systems).

JAVA TECHNOLOGIES

Threads

- A thread is an extremely lightweight process, or the smallest component of the process, that enables software to work more effectively by doing numerous tasks concurrently.

- Multithreading in Java is a process of executing multiple threads simultaneously. The main reason for incorporating threads into an application is to improve its performance.

- Java supports multithreading through its built-in features for creating and managing threads.

- It provides a Thread class that can be extended to create custom threads or Runnable interface to define tasks for threads.

Threads

In Java, there are two main types of threads:

1. user threads
2. daemon threads.

User threads are created by the application and continue to run until their task is completed or the application terminates.

On the other hand, **daemon threads are background threads** that are automatically terminated when there are no more user threads running. These daemon threads are useful for tasks like garbage collection, etc.

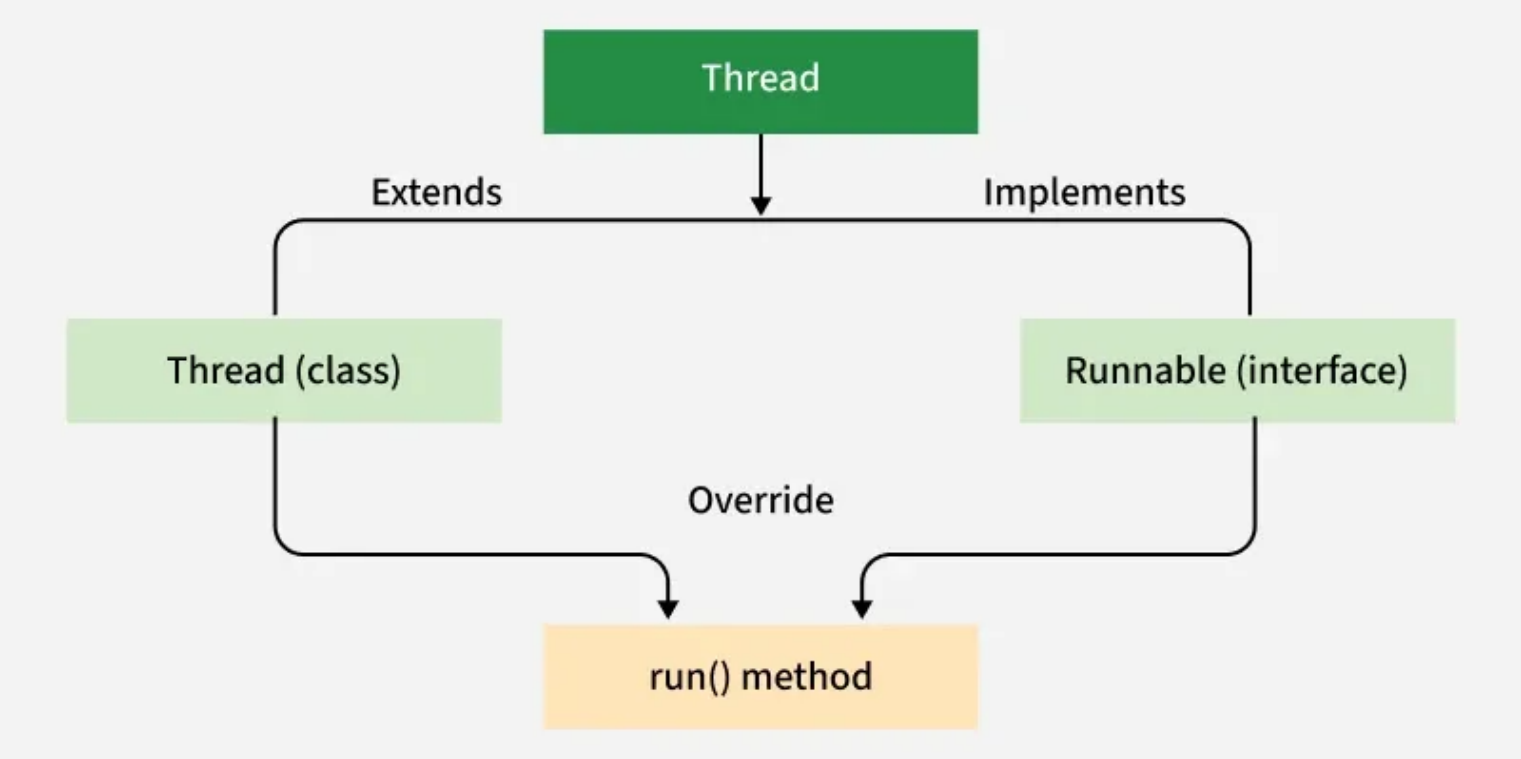
Threads

- **Using Extends**

Every thread class has run method called by start()

- **Using Runnable()**

Create thread object because runnable method does not have thread methods



Threads

Thread Methods:

- **start()** – Starts the thread.
- **getState()** – It returns the state of the thread.
- **getName()** – It returns the name of the thread.
- **getPriority()** – It returns the priority of the thread.
- **sleep()** – Stop the thread for the specified time.
- **Join()** – Stop the current thread until the called thread gets terminated.
- **isAlive()** – Check if the thread is alive.
- **run() –** runs the actual code

Threads

```
public class Tsimple {
    public static void main(String args[]) {
        // creating an object of the Thread class.
        Thread t1 = new Thread("Thread number 1");
        Thread t2 = new Thread("Thread number 2");
        // executing the threads using start method.
        t1.start();
        t2.start();
        // getting the name of the threads.
        String name1 = t1.getName();
        String name2 = t2.getName();
        System.out.println(name1);
        System.out.println(name2);
    } }
```

JAVA TECHNOLOGIES

Threads

- run() : A method in the Runnable interface (and in Thread class) that contains the actual code to be executed by the thread.
- start(): A method in the Thread class that creates a new thread and then calls the run() method internally.
- Calling run() : directly Executes like a normal method call, in the current thread, no new thread is created.
- Calling start() : Creates a new thread of execution, then the JVM calls run() on that new thread.
- run() : Can be called multiple times (it's just a method).
- start() : Can only be called once per thread object; otherwise, it throws IllegalThreadStateException.
- The run() method has to be public when you override it.

Threads

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running: " + Thread.currentThread().getName());
    }
 }
public class ThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.run(); // Runs in main thread
        t1.start(); // Runs in new thread
        //MyThread t2 = new MyThread();
        // t2.start();
    }
}
```

Threads

```
class A extends Thread
{
public void run()// running state
{
for(int i=0;i<20;i++)
System.out.println("A");
try{
        Thread.sleep(2);
}
catch(InterruptedException e)
{

}}}
```

```
class B extends Thread
{
public void run()
{
        for(int i=0;i<20;i++)
        System.out.println("B");
        try{
        Thread.sleep(2);
}
catch(InterruptedException e)
{

}}}
```

JAVA TECHNOLOGIES

Threads

```
public class Tmain
{
        public static void main(String args[])
{
A obja=new A();
B objb=new B();
//System.out.println(obja.getPriority());
obja.start();
objb.start();// runnable state
}
}
```

Threads

```
class A implements Runnable
{
public void run()
{
for(int i=0;i<15;i++)
System.out.println("A");
} }
class B implements Runnable
{
public void run()
{
        for(int i=0;i<15;i++)
        System.out.println("B");
} }
```

```
public class Tmain
{
public static void main(String args[])
{
Runnable obj1=new A();
Runnable obj2=new B();
//System.out.println(obj1.getPriority());
//obj1.start();
//obj2.start();
Thread t1=new Thread(obj1);
Thread t2=new Thread(obj2);
t1.start();
t2.start();
} }
```

JAVA TECHNOLOGIES

Threads

- Runnable is just an interface.
- It defines only the run() method.
- It does not have start() or getPriority() methods.
- The compiler does not allow calling getPriority() or start() on a Runnable reference.
- Runnable = **what to do**
- Thread = **who does it**

Threads

Threads can go through five different status in its life cycle as shown below.

- **New:** When the thread instance is created, it will be in “New” state.
- **Runnable:** When the thread is started, it is called “Runnable” state.
- **Running:** When the thread is running, it is called “Running” state.
- **Waiting:** When the thread is put on hold or it is waiting for the other thread to complete, then that state will be known as “waiting” state.
- **Terminated:** When the thread is dead, it will be known as “terminated” state.

Threads

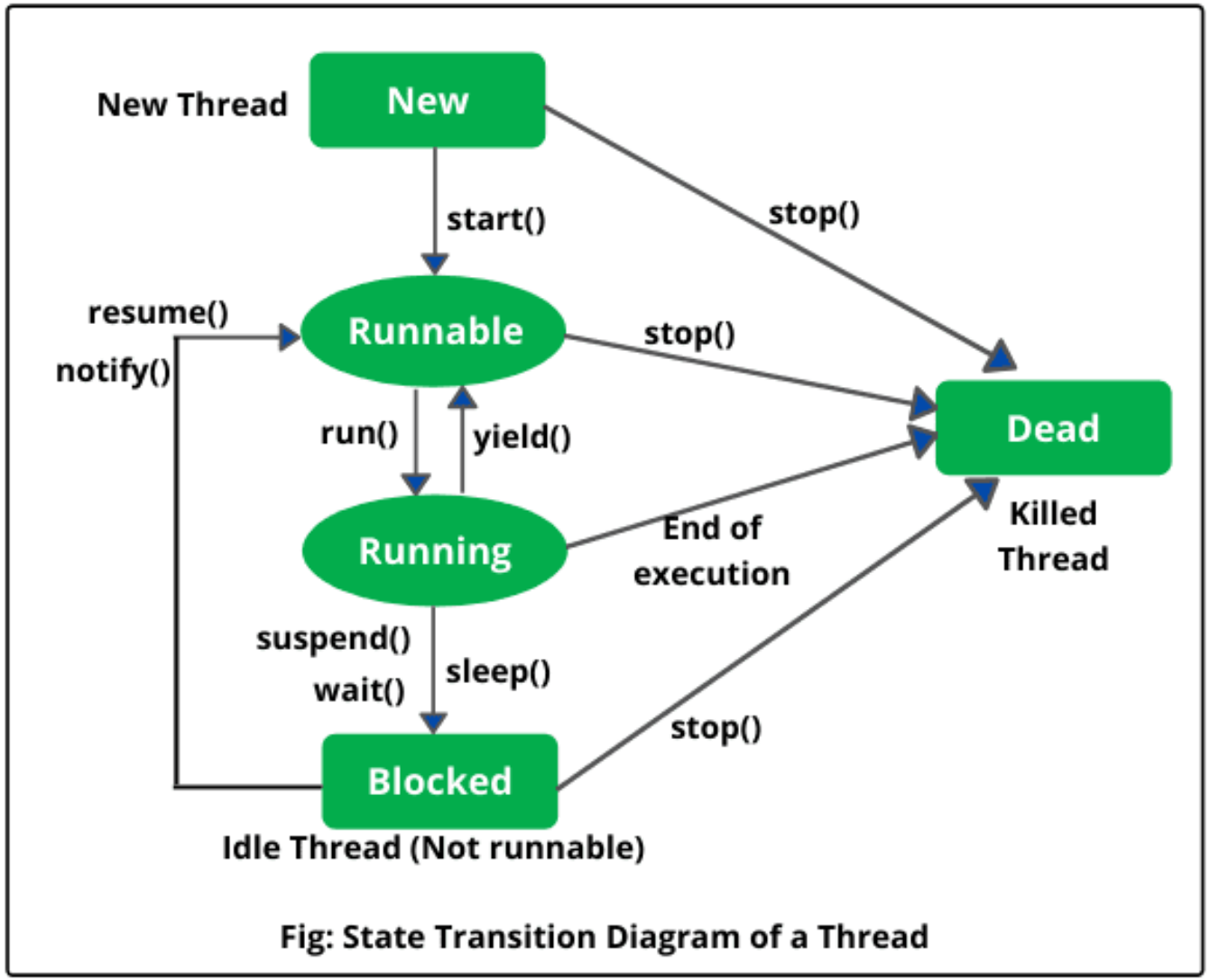


Fig: State Transition Diagram of a Thread

THANK YOU

Samyukta D Kumta
Department of Computer Applications
samyuktad@pes.edu