**OBJECT ORIENTED PROGRAMMING STRUCTURE OR SYSTEM(OOPS):**

* Class
* Object
* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

**CLASS**

* A **class** is a blueprint or template for creating objects.
* It defines a set of properties (fields/variables) and methods (functions) that the objects created from the class will have.
* It does not occupy memory on its own but provides the structure for creating objects that do.

**Example:**

// Defining a class

public class Car {

// Fields (attributes)

String model;

String color;

int year;

// Constructor (special method to initialize objects)

public Car(String model, String color, int year) {

this.model = model;

this.color = color;

this.year = year;

}

// Method (behavior)

public void displayInfo() {

System.out.println("Model: " + model + ", Color: " + color + ", Year: " + year);

}

}

**Explanation:**

* The Car class defines the attributes (model, color, year) and a behavior (displayInfo()) that all cars will have.

**OBJECT**

* An **object** is an instance of a class.
* When a class is defined, no memory is allocated until an object of that class is created.
* Each object has its own state (values of the variables) and can perform the behaviors defined by its class.

**Example:**

public class Main {

public static void main(String[] args) {

// Creating an object (instance) of the Car class

Car myCar = new Car("Tesla Model S", "Red", 2023);

// Accessing the method of the object

myCar.displayInfo(); // Output: Model: Tesla Model S, Color: Red, Year: 2023

}

}

**Explanation:**

* myCar is an object of the class Car.
* It has its own values for model, color, and year.
* The method displayInfo() is used to access the behavior of the object.

**ENCAPSULATION**

**Definition:**

* Encapsulation is the practice of wrapping data (variables) and methods (functions) into a single unit or class, and restricting access to some of the object's components.

**Purpose:**

* Encapsulation allows for data hiding, protecting the internal state of an object, and providing a controlled interface for external interaction.

**Methods used:**

* Getter method🡪get()🡪also called as Accessor.
* Setter method🡪set()🡪also called as Mutator.

**Usage as in project:**

* **JavaBean🡪**Model or Bean🡪used in servlet framework.
* **POJO class** format🡪called as entity🡪Plain Old Java Object🡪used in ***springboot*** framework.

**Example1:**

package com.demo.JavaCode;  
  
public class Encapsulation {  
 private String name;  
 private String college;  
 private int number;  
 private String dept;  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getCollege() {  
 return college;  
 }  
  
 public void setCollege(String college) {  
 this.college = college;  
 }  
  
 public int getNumber() {  
 return number;  
 }  
  
 public void setNumber(int number) {  
 this.number = number;  
 }  
  
 public String getDept() {  
 return dept;  
 }  
  
 public void setDept(String dept) {  
 this.dept = dept;  
 }  
}

**Example2:**

package com.demo.JavaCode;  
  
public class AccessEncapsulation {  
 public static void main(String[] args){  
 Encapsulation ee = new Encapsulation();  
 ee.setName("Mugilan");  
 ee.setCollege("VEC");  
 ee.setNumber(987654321);  
 ee.setDept("BE-CSE");  
  
 System.out.println(ee.getName());  
 System.out.println(ee.getCollege());  
 System.out.println(ee.getNumber());  
 System.out.println(ee.getDept());  
  
 }  
}

**INHERITANCE:**

**Definition:**

* Inheritance allows one class to inherit properties and behaviors (fields and methods) from another class, promoting code reuse.

**Purpose:**

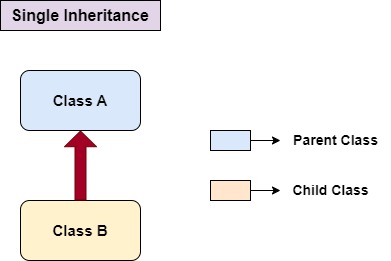
* It helps in creating a hierarchical classification and reduces redundancy by enabling subclasses to reuse methods and fields from the parent class.

**Why use of Inheritance:**

* For method overriding (so run time polymorphism can be achieved)
* For code reusability

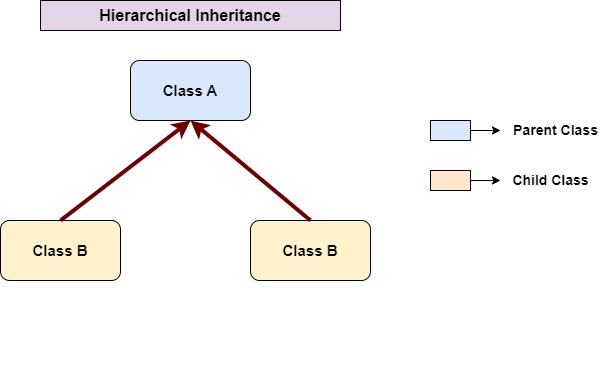
**TYPES OF INHERITANCE:**

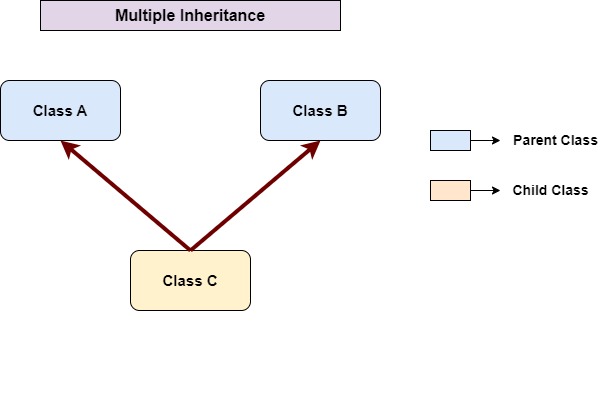
* Single Inheritance
* Multi-level Inheritance
* Hierarchical Inheritance
* Multiple Inheritance (achieved through interface)
* Hybrid Inheritance (achieved through interface)

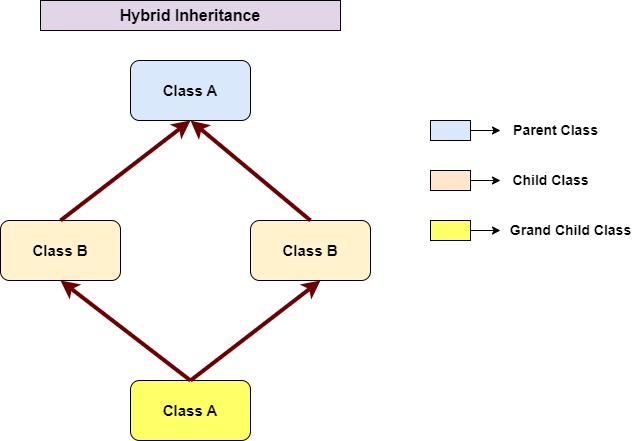


A diagram of a multi-level inheritance

Description automatically generated







**Syntax:**

public class sub-class-name **extends** super-class-name{

//Methods and fields

//Statement

}

**KEYWORDS TO BE USED TO CALL CLASS AND INTERFACE**

A diagram of a computer

Description automatically generated

**SINGLE INHERITANCE:**

**Definition:**

* A subclass inherits from a single parent class.
* The child class acquires the properties and methods of the parent class.

**Example: Parent class**

package com.demo.JavaCode;  
  
public class ParentInheritanceExample {  
  
 int a = 5;  
 int b = 6;  
 int c;  
  
 public void add(){  
 c = a + b;  
 System.out.println("I just built my dream house after many years:"+c);  
 }  
  
 public void sub(){  
 c = a - b;  
 System.out.println("I have my own bike RE:"+c);  
 }  
  
 public void mul(){  
 c = a \* b;  
 System.out.println("Startup level company:"+c);  
 }  
  
}

**Example: Child class**

package com.demo.JavaCode;  
  
public class ChildInheritanceExample **extends** ParentInheritanceExample {  
 public void car(){  
 System.out.println("I have bought my dream car ferrari fe");  
 }  
 public static void main(String[] args){  
 ChildInheritanceExample che = new ChildInheritanceExample();  
 che.car();  
 che.add(); //inherited from parent class  
 che.sub(); //inherited from parent class  
 che.mul(); //inherited from parent class  
 }  
}

**Explanation:**

* The ChildInheritanceExample class inherits the add(), sub(), mul() method from the ParentInheritanceExample class, allowing the ChildInheritanceExample to use the behavior of ParentInheritanceExample.

**MULTI-LEVEL INNHERITANCE:**

**Definition:**

* A subclass inherits from another subclass, forming a multi-level hierarchy.
* This means a class can inherit from a parent class, and then another class can inherit from that subclass.

**Example: Parent class**

package com.demo.JavaCode;  
  
public class ParentInheritanceExample {  
  
 int a = 5;  
 int b = 6;  
 int c;  
  
 public void add(){  
 c = a + b;  
 System.out.println("I just built my dream house after many years:"+c);  
 }  
  
 public void sub(){  
 c = a - b;  
 System.out.println("I have my own bike RE:"+c);  
 }  
  
 public void mul(){  
 c = a \* b;  
 System.out.println("Startup level company:"+c);  
 }  
  
}

**Example: Child class**

package com.demo.JavaCode;  
  
public class ChildInheritanceExample **extends** ParentInheritanceExample {  
 public void car(){  
 System.out.println("I have bought my dream car ferrari fe");  
 }  
}

**Example: Grandchild class**

package com.demo.JavaCode;  
  
public class GrandChildInheritanceExample **extends** ChildInheritanceExample {  
 public static void main(String[] args){  
 GrandChildInheritanceExample gci = new GrandChildInheritanceExample();  
 gci.add();  
 gci.sub();  
 gci.mul();  
 gci.car();  
 }  
}

**Explanation:**

* The GrandChildInheritanceExample class inherits from ChildInheritanceExample, which in turn inherits from ParentInheritanceExample. The GrandChildInheritanceExample object has access to methods from both ChildInheritanceExample and ParentInheritanceExample.

**HIERARCHICAL INHERITANCE:**

**Definition:**

* Multiple classes inherit from the same parent class.
* The child classes are distinct, but they share common behavior or properties from the parent.

**Example: Parent class**

package com.demo.JavaCode;  
  
public class Animal {  
 public void eat(){  
 System.out.println("Eating food");  
 }  
}

**Example: Child class1**

package com.demo.JavaCode;  
  
public class Cat **extends** Animal {  
 public void food(){  
 System.out.println("Cat loves fish");  
 }  
}

**Example: Child class2**

package com.demo.JavaCode;  
  
public class Dog **extends** Animal{  
  
 public void dogFood(){  
 System.out.println("Dog loves meat");  
 }  
 public static void main(String[] args){  
 Cat ct = new Cat();

Dog dt = new Dog();

ct.food();  
 ct.eat();  
// ct.dogFood();//Compile Time Error

dt.dogFood();

dt.eat();  
 }  
}

**Explanation:**

* Both Dog and Cat inherit the eat() method from the Animal class but have their unique behaviors, dogFood() and food().

**MULTIPLE INHERITANCE:**

* Java doesn't support multiple inheritance through classes to avoid ambiguity, but it allows it through **interfaces**.
* A class can implement multiple interfaces, thus inheriting behaviors from multiple sources.

**Example: Parent Interface1**

package com.demo.JavaCode;  
  
public interface Chicken {  
 public void chickenTandoori();  
 public void grillChicken();  
}

**Example: Parent Interface2**

package com.demo.JavaCode;  
  
public interface Lion {  
 public abstract void run();  
 public abstract void eat();  
}

**Example: Child class**

package com.demo.JavaCode;  
  
public class LionActivitiesMultipleInheritanceEx implements Lion, Chicken{  
  
 @Override  
 public void chickenTandoori() {  
 System.out.println("Chicken tandoori made with chicken masala, corn floor powder, with chilli powder and lemon with water");  
 }  
  
 @Override  
 public void grillChicken() {  
 System.out.println("Grill chicken with masala made with low flame");  
 }  
  
 @Override  
 public void run() {  
 System.out.println("Lion runs fast and attack the prey in group");  
 }

@Override  
 public void eat()  
 {  
 System.out.println("Lion eats pure raw meat");  
 }  
 public static void main(String[] args){  
 LionActivitiesMultipleInheritanceEx la = new LionActivitiesMultipleInheritanceEx();  
 la.chickenTandoori();  
 la.grillChicken();  
 la.run();  
 la.eat();  
 }  
}

**Explanation:**

* The LionActivitiesMultipleInheritanceEx class implements two interfaces, Lion and Chicken, demonstrating multiple inheritance through interfaces.

**HYBRID INHERITANCE:**

**Definition:**

* Hybrid inheritance is a combination of more than one type of inheritance (single, multilevel, hierarchical, etc.).
* In Java, this can be achieved using interfaces.

**Example: Parent Interface1**

// Interface 1

public interface Flyable {

void fly();

}

**Example: Parent class**

// Parent class

class Animal {

void eat() {

System.out.println("Eating...");

}

}

**Example: Child class**

// Class inheriting from a class and implementing an interface

class Bird extends Animal implements Flyable {

public void fly() {

System.out.println("Flying...");

}

}

public class Main {

public static void main(String[] args) {

Bird bird = new Bird();

bird.eat(); // Output: Eating... (from Animal class)

bird.fly(); // Output: Flying... (from Flyable interface)

}

}

**Explanation:**

* The Bird class inherits from the Animal class (single inheritance) and implements the Flyable interface, showcasing hybrid inheritance.

**POLYMORPHISM:**

* **Polymorphism** in Java is one of the four fundamental concepts of Object-Oriented Programming (OOP), allowing one interface to be used for different data types or actions.
* In simpler terms, polymorphism means "many forms."
* It enables a single method, object, or operator to behave in multiple ways based on the context.

**TYPES OF POLYMORPHISM IN JAVA**

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

**COMPILE-TIME POLYMORPHISM (METHOD OVERLOADING)**

**Definition:**

* This type of polymorphism occurs when multiple methods in the same class have the same name but different parameter lists (method signatures).
* The method to be invoked is determined at compile time.

**Key Features:**

* Methods have the same name but differ in the number or type of parameters.
* Return type may or may not be the same.
* Achieved through **method overloading**.

**2 ways to overload method in java:**

* By changing number of arguments
* By changing the datatype

**Example: By changing number of arguments**

public class Calculator {

// Overloaded method with two parameters

public int add(int a, int b) {

return a + b;

}

// Overloaded method with three parameters

public int add(int a, int b, int c) {

return a + b + c;

}

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

// Different forms of the add() method called

System.out.println(calc.add(5, 10)); // Output: 15

System.out.println(calc.add(5, 10, 15)); // Output: 30

}

}

**Explanation:**

* The Calculator class has three overloaded methods called add() that vary in the number of parameters.
* Based on the arguments provided, the appropriate method is called at compile-time.

**Example: By changing the datatype**

package com.demo.JavaCode;  
import java.util.Scanner;  
public class OverLoadingExample {  
 int c; //Global variable

public void add(int a, int b){  
 c = a + b;  
 System.out.println("The value of integer c is: "+c);  
 }  
  
 public void add(float a, int b){  
 float c = a + b;  
 System.out.println("The value of float c is: "+c);  
 }  
  
 public void add(float a, double b){  
 double c = a + b;  
 System.out.println("The value of double c is: "+c);  
 }  
 public static void main(String[] args){  
 OverLoadingExample ole = new OverLoadingExample();  
 Scanner sc = new Scanner(System.in);  
  
 int a, b;  
 System.out.println("Enter the value for a and b:");  
 a = sc.nextInt();  
 b = sc.nextInt();  
 ole.add(a,b);  
  
 float c; int d;  
 System.out.println("Enter the value for c and d:");  
 c = sc.nextFloat();  
 d = sc.nextInt();  
 ole.add(c,d);  
  
 float e; double f;  
 System.out.println("Enter the value for e and f:");  
 e = sc.nextFloat();  
 f = sc.nextDouble();  
 ole.add(e,f);  
  
 }  
}

**Explanation:**

* The OverLoadingExample class has three overloaded methods called add() that vary in the datatype but has same number of parameters.
* Based on the arguments provided, the appropriate method is called at compile-time.

**RUN-TIME POLYMORPHISM (METHOD OVERRIDING)**

**Definition:**

* This type of polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its parent class.
* The method that gets invoked is determined at runtime based on the object type.

**Key Features:**

* Requires inheritance (one class extends another).
* The method in the child class must have the same name, return type, and parameter list as the method in the parent class.
* Achieved through method overriding.
* It is a form of dynamic dispatch.

**Example:**

package com.demo.JavaCode;  
  
public class PersonOverrideExample {  
 public void personDetails(){  
 System.out.println("In the person class method role=administrator");  
 }  
 public void employeeDetails(){  
 System.out.println("In the employee class method role=junior java developer");  
 }  
 public void studentDetails(){  
 System.out.println("In student class method role=intern");  
 }  
}

**Example:**

package com.demo.JavaCode;  
  
public class Overriding extends PersonOverrideExample {  
 public void personDetails(){  
 System.out.println("Change role administrator to senior administrator");  
 }  
 public void employeeDetails(){  
 System.out.println("Change role junior java developer to software engineer");  
 }  
 public void studentDetails(){  
 System.out.println("Change role intern to employee as junior developer");  
 }  
 public static void main(String[] args){  
 Overriding obj1 = new Overriding();  
 obj1.personDetails();  
 obj1.employeeDetails();  
 obj1.studentDetails();  
 }  
}

**Explanation:**

* In the above example, the personDetails(), employeeDetails(), studentDetails() method is overridden in Overriding class.
* Based on the type of object (Overriding), the respective personDetails(), employeeDetails(), studentDetails() method is invoked at runtime.

**Key Differences Between Compile-time and Run-time Polymorphism**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Compile-time Polymorphism** | **Run-time Polymorphism** |
| **Achieved By** | Method Overloading | Method Overriding |
| **Time of Resolution** | At Compile Time | At Run Time |
| **Binding** | Early Binding(static binding) | Late Binding(Dynamic binding) |
| **Inheritance Required** | No | Yes |
| **Method Signature** | Methods differ in parameters (number/type) | Methods have the same signature |
| **Example** | add(int a, int b) and add(int a, int b, int c) | PersonOverrideExample class overridden by Overriding class |

**POLYMORPHISM VIA INTERFACES**

**Definition:**

* Interfaces in Java allow polymorphism by enabling multiple classes to implement the same interface methods in different ways.

**Example: Parent interface**

// Interface

public interface Animal {

void sound();

}

**Example: Child Interface 1**

// Class 1 implementing interface

class Dog implements Animal {

public void sound() {

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

}

}

**Example: Child class**

// Class 2 implementing interface

class Cat implements Animal {

public void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myDog = new Dog();

Animal myCat = new Cat();

myDog.sound(); // Output: Dog barks

myCat.sound(); // Output: Cat meows

}

}

**Explanation:**

* Both Dog and Cat implement the Animal interface.
* The sound() method is called dynamically based on the object type (either Dog or Cat).

**OPERATOR OVERLOADING(NOT SUPPORTED IN JAVA)**

**NOTE:**

* Java does not support operator overloading (unlike languages like C++).
* The meaning of operators like +, -, \*, etc., cannot be changed for user-defined types.

**Key Point:**

* Polymorphism in Java provides flexibility and enhances code reusability, making object-oriented systems more dynamic and efficient.

**ABSTRACTION**

* **Abstraction** in Java is one of the key concepts of Object-Oriented Programming (OOP), where we hide the implementation details and expose only the functionality to the user.
* In simpler terms, abstraction allows you to focus on what an object does rather than how it does it.

**Java supports 2 types of abstraction:**

* Abstract classes
* Interfaces

**ABSTRACT CLASS**

**Definition:**

* An abstract class in Java is a class that is declared with the abstract keyword.
* It can have both abstract methods (methods without a body) and non-abstract methods (methods with a body).
* Abstract classes cannot be instantiated directly, meaning you cannot create objects of an abstract class.

**Key features:**

* Can have both abstract methods (without implementation) and concrete methods (with implementation).
* Can have instance variables and constructors.
* Must be extended by a subclass, and the subclass must provide implementations for the abstract methods.

**Example: Abstract class**

package com.demo.JavaCode;  
  
public abstract class AbstractEmployeeExample {  
 public void name(){  
 System.out.println("This is Mugilan");  
 }  
  
 public void designation(){  
 System.out.println("Java Developer and Full Stack trainer");  
 }  
  
 public abstract void futureDesignation();  
}

**Example: Non-abstract class**

package com.demo.JavaCode;  
  
public class AbstractNewEmployeeExample extends AbstractEmployeeExample {  
 public void futureDesignation(){  
 System.out.println("Senior java Developer and full stack and mern stack trainer");  
 System.out.println("Else...can start a new company");  
 }  
 public static void main(String[] args){  
 AbstractNewEmployeeExample ane = new AbstractNewEmployeeExample();  
 ane.name();  
 ane.designation();  
 ane.futureDesignation();  
 }  
}

**Explanation:**

* In this example, the AbstractEmployeeExample class is abstract, and it has one abstract method futureDesignation() and two concrete method name(), designation().
* The AbstractNewEmployeeExample class extends AbstractEmployeeExample and provides an implementation for the futureDesignation() method.

**INTERFACE**

**Definition:**

* An interface in Java is a blueprint of a class that contains static constants and abstract methods.
* It is used to achieve complete abstraction, meaning interfaces can only have method signatures (without bodies).
* Java 8 and beyond allow default and static methods in interfaces with method bodies.

**Key features:**

* Can only contain abstract methods (prior to Java 8) and default or static methods (from Java 8 onward).
* All methods are implicitly public and abstract.
* Can be implemented by any class, and a class can implement multiple interfaces (allowing multiple inheritance in Java).

**Example: Parent Interface**

// Interface

public interface Animal {

// Abstract method

public void sound();

}

**Example: Child class1**

// Implementing the interface in a class

public class Dog implements Animal {

public void sound() {

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

}

}

**Example: Child class2**

public class Cat implements Animal {

public void sound() {

System.out.println("Cat meows.");

}

}

**Example: Main class to access**

public class AccessInfo {

public static void main(String[] args) {

Animal dog = new Dog();

dog.sound(); // Output: Dog barks.

Animal cat = new Cat();

cat.sound(); // Output: Cat meows.

}

}

**Explanation:**

* Here, the Animal interface defines a sound() method that is implemented by the Dog and Cat classes.
* Both classes provide different implementations of the sound() method.
* Interfaces allow flexibility because any class can implement multiple interfaces.

**Differences Between Abstract class and Interface**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Abstract Class** | **Interface** |
| **Keyword Used** | abstract | interface |
| **Methods** | Can have both abstract and non-abstract methods | Only abstract methods (Java 7 and below), can have default and static methods (Java 8 above) |
| **Variables** | Can have instance variables | Can only have static and final variables |
| **Multiple Inheritance** | Does not support multiple inheritance | Support multiple inheritance(a class can implement multiple interfaces) |
| **Constructor** | Can have constructors | Cannot have constructors |
| **Implementation** | A subclass extends the abstract class | A class implements the interface |