What is OOPS ? Explain.

OOPS stands for Object-Oriented Programming System or Object-Oriented Programming Paradigm. It is a programming paradigm that organizes data and behavior into reusable structures called objects.

The main concept behind OOPS is to model real-world entities as objects and define their characteristics (attributes or properties) and behaviors (methods or functions) within a programming language.

OOPS stands for Object-Oriented Programming System, a programming paradigm that focuses on organizing code around objects, which are instances of classes. OOPS provides a set of principles and concepts to structure and design code for better modularity, reusability, and maintainability.

Let's explore some key OOPS concepts with real-world examples and provide sample Java programs for each concept:

1. Class:

A class is a blueprint or template that defines the properties and behaviors of objects. It encapsulates data (attributes) and methods (functions) that operate on that data. For example, consider a "Car" class:

```java
public class Car {
    // Attributes
    private String brand;
    private String color;

    // Methods
    public void startEngine() {
        System.out.println("Engine started!");
    }

    public void drive() {
        System.out.println("Car is driving.");
    }
}
```

2. Object:

An object is an instance of a class that represents a specific entity in the real world. It has its own state (attribute values) and can perform actions (methods). For example, creating objects of the "Car" class:

```
public class Main {
    public static void main(String[] args) {
        // Creating objects
        Car car1 = new Car();
        Car car2 = new Car();

        // Using object methods
        car1.startEngine();
        car2.drive();
    }
}
```

3. Inheritance:

Inheritance allows a class (child/subclass) to inherit properties and behaviors from another class (parent/superclass). It promotes code reuse and hierarchical relationships. For example, consider a "SportsCar" class inheriting from the "Car" class:

```
public class SportsCar extends Car {
    public void turboBoost() {
        System.out.println("Turbo boost activated!");
    }
}
```

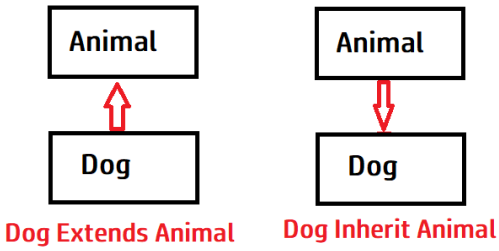
Types of Inheritance

1. Single Inheritance:

Single inheritance is a type of inheritance where a class inherits properties and behaviors from a single parent class.

Example:

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

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



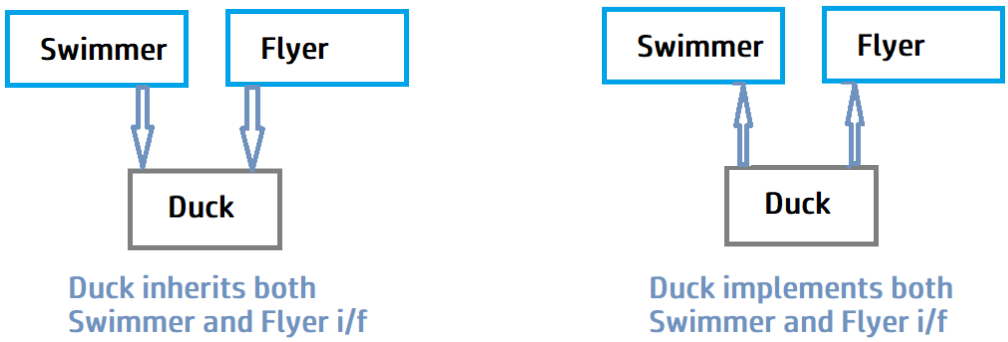
In this example, the class `Dog` inherits from the class `Animal`. The `Dog` class can access the `eat()` method defined in the `Animal` class and also define its own method `bark()`.

2. Multiple Inheritance (through interfaces):

Multiple inheritance is a type of inheritance where a class can inherit from multiple interfaces, but not from multiple classes.

Example:

```
interface Swimmer {
    void swim();
}

interface Flyer {
    void fly();
}

class Duck implements Swimmer, Flyer {
    public void swim() {
        System.out.println("Duck is swimming.");
    }

    public void fly() {
        System.out.println("Duck is flying.");
    }
}
```



In this example, the class `Duck` implements both the `Swimmer` and `Flyer` interfaces. The `Duck` class can provide its own implementations for the `swim()` and `fly()` methods defined in the interfaces.

Why Java class doesn't support multiple inheritance?

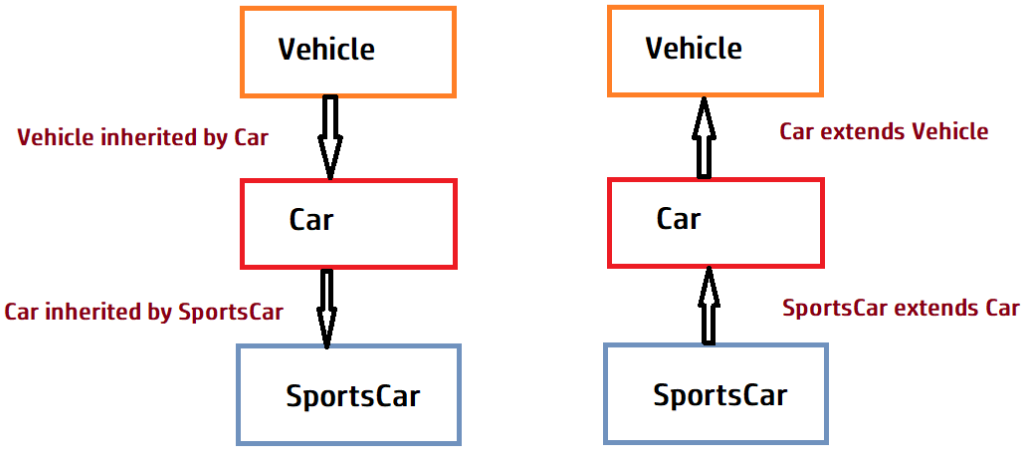
There are chances of ambiguity, for avoiding that **ambiguity** Java doesn't support multiple inheritance.

3. Multilevel Inheritance:

Multilevel inheritance is a type of inheritance where a class inherits from another class, and that class further inherits from another class.

Example:

```
class Vehicle {
    void start() {
        System.out.println("Vehicle started.");
    }
}

class Car extends Vehicle {
    void drive() {
        System.out.println("Car is being driven.");
    }
}

class SportsCar extends Car {
    void accelerate() {
        System.out.println("Sports car is accelerating.");
    }
}
```



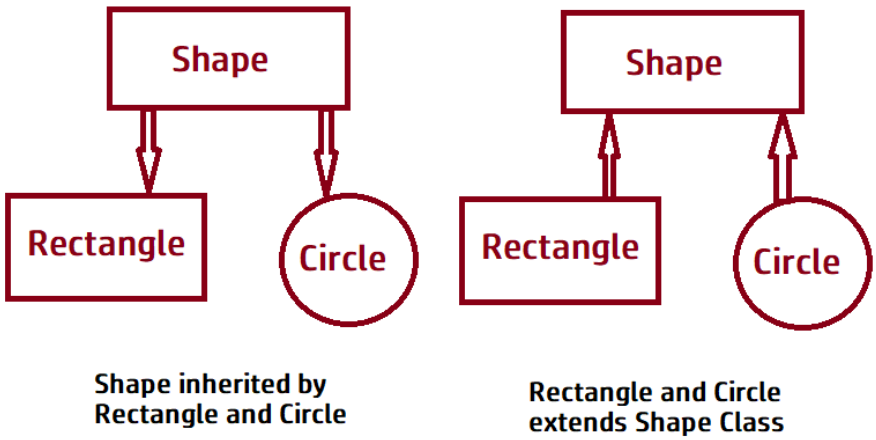
In this example, the class `Car` inherits from the class `Vehicle`, and the class `SportsCar` inherits from the class `Car`. The `SportsCar` class can access methods from both the `Car` and `Vehicle` classes.

4. Hierarchical Inheritance:

Hierarchical inheritance is a type of inheritance where multiple classes inherit from a single parent class.

Example:

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



In this example, both the `Circle` and `Rectangle` classes inherit from the `Shape` class. Each subclass provides its own implementation of the `draw()` method, customizing the behavior for their specific shape.

4. Abstraction

Abstraction is the process of simplifying complex systems by representing essential features while hiding unnecessary details.

It allows us to create abstract classes and interfaces that define the common characteristics and behaviors of related objects. Abstraction helps in managing complexity and allows us to focus on essential aspects.

Real-world example:

Consider a Car as an example. We can abstract the car to focus on its essential features, such as its ability to start, stop, accelerate, and turn. We don't need to know the internal workings of the engine, transmission, or other intricate details to understand and use the car.

Sample program in Java:

```
// Abstract class representing a vehicle
abstract class Vehicle {
    // Abstract method for starting the vehicle
    public abstract void start();

    // Abstract method for stopping the vehicle
    public abstract void stop();
}

// Concrete class representing a Car
class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Car started.");
        // Code for starting the car
    }

    @Override
    public void stop() {
        System.out.println("Car stopped.");
        // Code for stopping the car
    }
}
```

```java
// Main class
public class Main {
    public static void main(String[] args) {
        // Creating an instance of Car
        Car car = new Car();

        // Using abstraction to start and stop the car
        car.start();
        car.stop();
    }
}
```

In this example,

the Vehicle class is an abstract class that defines the common behavior of vehicles. It has abstract methods start() and stop() which represent the essential actions of any vehicle.

The Car class extends the Vehicle class and provides concrete implementations of the start() and stop() methods.

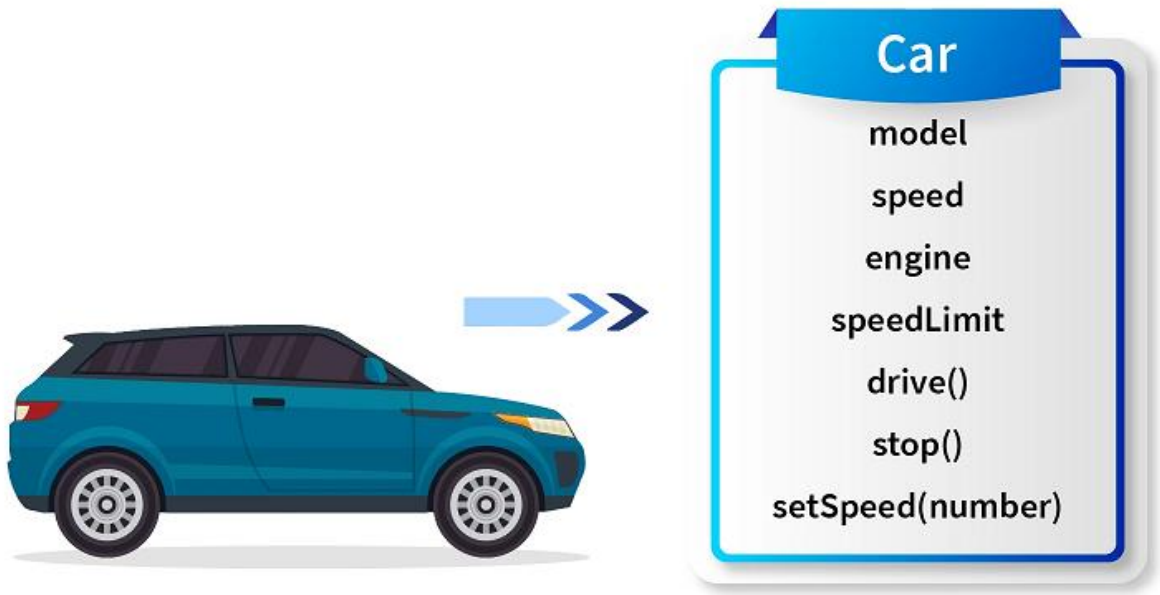
5. Encapsulation:

Encapsulation = Data Hiding + Abstraction

Encapsulation bundles data and methods within a class, hiding the internal details and providing access through public methods. It ensures data integrity and protects the internal state. For example, encapsulating attributes of the "Car" class:

```
public class Car {
    private String brand;
    private String color;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    // ... (similarly for other attributes)
}
```

These are just a few key concepts of OOPS. Java provides a rich set of features and syntax to implement and leverage object-oriented programming concepts effectively.



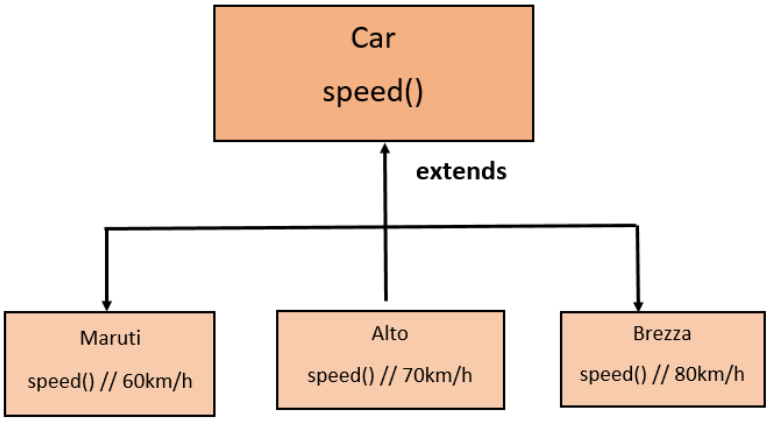
4. Polymorphism:

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and dynamic behavior based on the actual object type.

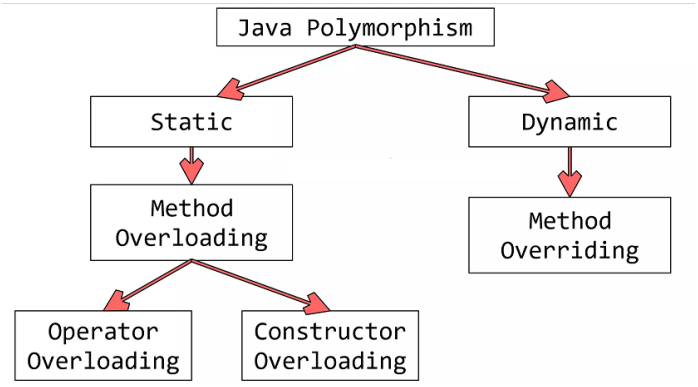
For example, using polymorphism with a "Car" and "SportsCar" class:

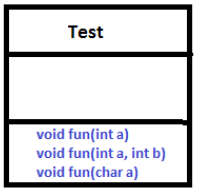
```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        SportsCar sportsCar1 = new SportsCar();

        // Polymorphic behavior
        Car car2 = sportsCar1;
        car2.drive();      // Calls drive() method of SportsCar

        Car car3 = new SportsCar();
        car3.startEngine();  // Calls startEngine() method of Car
    }
}
```



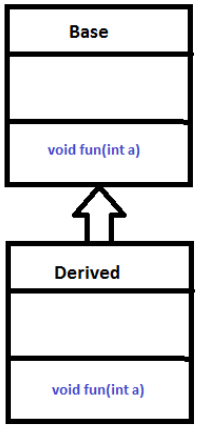
Concept that allows objects of different classes to be treated as objects of a common superclass or interface. It provides the ability to write code that can work with objects of multiple types, leading to flexibility, reusability, and extensibility. Polymorphism is achieved through method overriding and method overloading.

There are two main types of polymorphism:





Overloading



Overriding

Method Overloading	MethodOverriding
1. It occurs with in the same class.	1. It occurs between two classes i.e., Super class and a subclass.
2. Inheritance is not involved.	2. Inheritance is involved.
3.One method does not hide another.	3. child method hides that of the parent class method.
4.Parameters must be different.	4. Parameters must be same.
5.return type may or may not be same.	5. return type must be same.
6. Access modifier & Non access modifier can also be changed.	6. Access modifier should be same or increases the scope of the access modifier.

1. Compile-time Polymorphism (Static Polymorphism):

- Method Overloading: It allows a class to have multiple methods with the same name but different parameter lists. The appropriate method is selected based on the number, types, and order of the arguments during compile-time.

Example:

```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    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();
        System.out.println(calc.add(2, 3));          // Output: 5
        System.out.println(calc.add(2, 3, 4));       // Output: 9
    }
}
```

2. Runtime Polymorphism (Dynamic Polymorphism):

- Method Overriding: It allows a subclass to provide its own implementation of a method that is already defined in its superclass. The appropriate method is selected based on the actual type of the object at runtime.

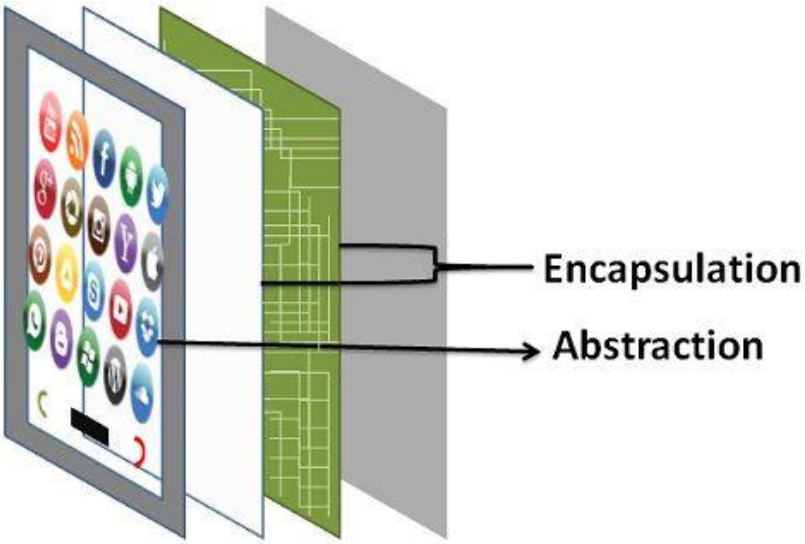
Example:

```
 class Animal {

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Cat extends Animal {
    @Override
```

```
    public void makeSound() {
        System.out.println("Meow");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.makeSound();       // Output: "Animal makes a sound"

        Animal cat = new Cat();
        cat.makeSound();          // Output: "Meow"

        Animal dog = new Dog();
        dog.makeSound();          // Output: "Woof"
    }
}
```

In the example, the `makeSound()` method is overridden in the `Cat` and `Dog` classes. When invoking the `makeSound()` method on an object, the appropriate implementation is determined based on the actual type of the object at runtime. This allows objects of different classes to exhibit different behaviors while being treated as objects of a common superclass (`Animal` in this case).

Difference Between Abstraction And Encapsulation



Encapsulation:-- **Information hiding**.
Abstraction:-- **Implementation hiding**.

Abstraction	Encapsulation
Process of hiding the implementation details from the user and showing only the functionality to the user.	Process of wrapping code and data together into a single unit and keeps both **safe from outside interference** and **misuse**.
You can use abstraction using **Interface** and **Abstract** Class	You can implement encapsulation using **Access Modifiers** (Public, Protected & Private)
Abstraction solves the problem in **Design** Level	Encapsulation solves the problem in **Implementation** Level
For simplicity, abstraction means hiding implementation using Abstract class and Interface	For simplicity, encapsulation means hiding data using getters and setters

What is Abstract Class?

An abstract class in object-oriented programming (OOP) is a class that cannot be instantiated on its own and is meant to serve as a base or blueprint for other classes.

It provides common functionality and defines the structure and behavior that derived classes should implement. Abstract classes can have both abstract and non-abstract methods.

1. **Cannot be instantiated:** You cannot create objects of an abstract class directly. It is designed to be inherited by other classes.

2. **May contain abstract methods:** Abstract methods are declared in an abstract class but do not have an implementation. Subclasses that inherit from the abstract class must provide implementations for these abstract methods.

3. **Can have non-abstract methods:** Abstract classes can also have regular methods with implementations. Subclasses inherit these methods along with the abstract methods.

4. **Can have instance variables:** Abstract classes can define instance variables that are inherited by subclasses.
In the belove class example, the **name** variable declared within the **Animal** class is an instance variable.

5. **Unimplemented abstract method:** If child class is unable to implement then that create another child to implement abstract method.

6. **Serve as a contract or template:** Abstract classes provide a template for derived classes, ensuring that they implement specific methods or adhere to a certain structure.

7. **Constructor:** Constructor is present inside the abstract class, but constructors even when they are only called from their concrete subclasses. (If we do not define any constructor inside the abstract class then JVM will give a default constructor to the abstract class.)

Example:

```
abstract class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    abstract void sound(); // Abstract method

    void sleep() { // Concrete method
        System.out.println(name + " is sleeping.");
    }
}
```

```java
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    void sound() {
        System.out.println(name + " barks.");
    }
}

class Cat extends Animal {
    Cat(String name) {
        super(name);
    }

    void sound() {
        System.out.println(name + " meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sound();
        dog.sleep();

        Cat cat = new Cat("Whiskers");
        cat.sound();
        cat.sleep();
    }
}
```

What is Interface ?

An interface in Java is a blueprint of a class that defines a set of methods (with or without default implementations) that a class implementing the interface must adhere to.

It defines a contract specifying the methods that must be implemented by the implementing classes.

In Java, interfaces provide a way to achieve abstraction, multiple inheritance of type, and support for polymorphism.

Key characteristics of an interface:

1. Abstract methods: Interfaces can have abstract methods that are declared without any implementation. These methods serve as a contract that implementing classes must fulfill by providing their own implementation.

2. Constant fields: Interfaces can have constant fields, which are implicitly `public`, `static`, and `final`. These fields define constants that can be accessed by implementing classes.

3. Default methods: Starting from Java 8, interfaces can also have default methods. Default methods have an implementation within the interface itself and can be optionally overridden by implementing classes.

4. Multiple inheritance of type: Unlike classes, a class can implement multiple interfaces, allowing it to inherit the behavior and contracts of multiple interfaces.

Example:

```
interface Animal {
    void eat();
    void sleep();
}

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

    public void sleep() {
        System.out.println("Dog is sleeping.");
    }
}

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

    public void sleep() {
        System.out.println("Cat is sleeping.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();
        dog.sleep();

        Cat cat = new Cat();
        cat.eat();
        cat.sleep();
    }
}
```

In this example, the `Animal` interface defines the `eat()` and `sleep()` methods. The `Dog` and `Cat` classes implement the `Animal` interface, providing their own implementations for the methods.

Difference between Abstract class and Interface

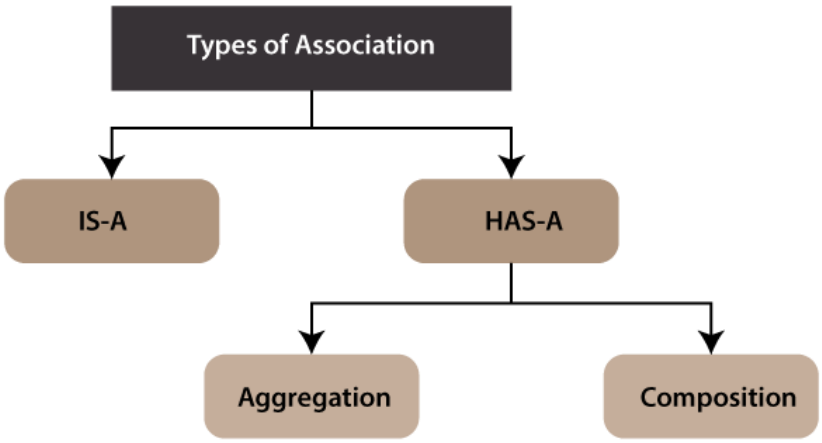
Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

What is Association?

Association is a relationship between two or more classes in object-oriented programming. It represents a connection or interaction between objects of different classes. In association, objects of one class are aware of the existence of objects of another class and can interact with them.

In Java, two types of **Association** are possible:



Aggregation	Composition
Aggregation is a weak Association.	Composition is a strong Association.
Class can exist independently without owner.	Class can not meaningfully exist without owner.
Have their own Life Time.	Life Time depends on the Owner.
A uses B.	A owns B.
Child is not owned by 1 owner.	Child can have only 1 owner.
Has-A relationship. A has B.	Part-Of relationship. B is part of A.
Denoted by a empty diamond in UML.	Denoted by a filled diamond in UML.
We do not use "final" keyword for Aggregation.	"final" keyword is used to represent Composition.
Examples: - Car has a Driver. - A Human uses Clothes. - A Company is an aggregation of People. - A Text Editor uses a File. - Mobile has a SIM Card.	Examples: - Engine is a part of Car. - A Human owns the Heart. - A Company is a composition of Accounts. - A Text Editor owns a Buffer. - IMEI Number is a part of a Mobile.

Associations can have different types based on the nature of the relationship between classes. Here are a few types of associations with examples:

1. One-to-One Association:

In a one-to-one association, each instance of one class is associated with exactly one instance of another class.

Example:

```
class Person {
    private String name;
    private Passport passport;

    public Person(String name, Passport passport) {
        this.name = name;
        this.passport = passport;
    }
```

```
    // Other methods and attributes
}

class Passport {
    private String passportNumber;

    public Passport(String passportNumber) {
        this.passportNumber = passportNumber;
    }

    // Other methods and attributes
}

public class Main {
    public static void main(String[] args) {
        Passport passport = new Passport("ABC123");
        Person person = new Person("John Doe", passport);

        // Code using person and passport
    }
}
```

2. One-to-Many Association:

In a one-to-many association, an instance of one class is associated with multiple instances of another class.

Example:

```
class Department {
    private String name;
    private List<Employee> employees;

    public Department(String name) {
        this.name = name;
        this.employees = new ArrayList<>();
    }

    public void addEmployee(Employee employee) {
        employees.add(employee);
    }

    // Other methods and attributes
}

class Employee {
    private String name;

    public Employee(String name) {
        this.name = name;
    }

    // Other methods and attributes
}
```

```
public class Main {
    public static void main(String[] args) {
        Department department = new Department("Sales");
        Employee employee1 = new Employee("John Doe");
        Employee employee2 = new Employee("Jane Smith");

        department.addEmployee(employee1);
        department.addEmployee(employee2);

        // Code using department and employees
    }
}
```

3. Many-to-Many Association:

In a many-to-many association, instances of one class are associated with multiple instances of another class, and vice versa.

Example:

```
class Student {
    private String name;
    private List<Course> courses;

    public Student(String name) {
        this.name = name;
        this.courses = new ArrayList<>();
    }

    public void enrollCourse(Course course) {
        courses.add(course);
        course.addStudent(this);
    }

    // Other methods and attributes
}

class Course {
    private String name;
    private List<Student> students;

    public Course(String name) {
        this.name = name;
        this.students = new ArrayList<>();
    }

    public void addStudent(Student student) {
        students.add(student);
    }

    // Other methods and attributes
}
```

```
public class Main {
    public static void main(String[] args) {
        Student student1 = new Student("John Doe");
        Student student2 = new Student("Jane Smith");
        Course course1 = new Course("Math");
        Course course2 = new Course("Science");

        student1.enrollCourse(course1);
        student1.enrollCourse(course2);
        student2.enrollCourse(course1);

        // Code using students and courses
    }
}
```

4. Aggregation:

Aggregation is a specialized form of association where one class represents a part of or is composed of another class. It is a "has-a" relationship, where one class contains instances of another class as part of its structure. The aggregated class can exist independently of the container class.

Example:

```
class University {
    private String name;
    private List<Department> departments;

    public University(String name) {
        this.name = name;
        this.departments = new ArrayList<>();
    }

    public void addDepartment(Department department) {
        departments.add(department);
    }

    // Other methods and attributes
}

class Department {
    private String name;

    public Department(String name) {
        this.name = name;
    }

    // Other methods and attributes
}

public class Main {
    public static void main(String[] args) {
        University university = new University("ABC University");
```

```
        Department department1 = new Department("Computer Science");
        Department department2 = new Department("Physics");

        university.addDepartment(department1);
        university.addDepartment(department2);

        // Code using university and departments
    }
}
```

5. Composition:

Composition is a stronger form of aggregation where the lifetime of the contained class is controlled by the container class. It is a "owns-a" relationship, where one class is composed of another class and takes responsibility for the creation and destruction of the composed object.

Example:

```
class Car {
    private String model;
    private Engine engine;

    public Car(String model) {
        this.model = model;
        this.engine = new Engine();
    }

    // Other methods and attributes
}

class Engine {
    // Engine class implementation
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Sedan");

        // Code using car
    }
}
```

6. Dependency:

Dependency represents a weaker form of association, where one class depends on another class, but there is no ownership or structural relationship between them. It signifies that one class uses the services or features provided by another class.

Example:

The `Order` class depends on the `EmailService` class to send confirmation emails.

```
class Order {
    private EmailService emailService;

    public Order() {
        this.emailService = new EmailService();
    }

    public void placeOrder() {
        // Code to place the order

        emailService.sendEmail("Order placed successfully.");
    }

    // Other methods and attributes
}

class EmailService {
    public void sendEmail(String message) {
        // Code to send an email
    }

    // Other methods and attributes
}

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