Object oriented programming in java

In Java, a class is a blueprint or template that defines the properties (attributes) and behavior (methods) of objects. It's like a cookie cutter that you use to create similar cookies. The class itself doesn't hold any data or perform any actions, but it serves as a specification for creating objects that do.

Key aspects of class in Java

- **Blueprint:** A class defines the characteristics of its objects. These characteristics include variables (to store data) and methods (to define actions).
- **Objects:** An object is an instance of a class. It's like a single cookie created from the cookie cutter. Objects hold their own data and can execute the methods defined in the class.
- **Properties:** These are also known as attributes or member variables. They represent the data or state of an object. For example, a Car class might have properties like color, weight, and speed.
- **Behavior:** These are defined by methods, which are functions associated with the class. Methods define the actions that objects can perform. Following the car example, a Car class might have methods like accelerate(), brake(), and turn().

> *Imagine a class named Dog. This class would define the general characteristics of dogs, like breed, color, and age (properties). It might also have methods like bark(), wagTail(), and playFetch() (to define the behavior of dogs). Individual dogs (like your pet Fido) would be objects created from the*

> *Dog class. Each dog would have its own specific fur color, breed, and age (data stored in the object's properties).*

By using classes, you can create multiple objects with similar characteristics and behaviors, promoting code reusability and organization in your Java programs.

Getting Started with OOP in Java:

1. **Define a Class:** Use the `class` keyword followed by the class name to define a class. Inside the class, you declare its properties (using variables) and methods (using functions).
2. **Create Objects:** Use the `new` keyword followed by the class name to create an object of that class. You can then access the object's properties and methods.
3. **Inheritance:** Use the `extends` keyword to create a subclass that inherits properties and methods from a superclass.

Example: Let's Build a Car!

```
public class Car {
  // Properties (data)
  private String color;
  private int speed;

  // Methods (actions)
  public void accelerate() {
    speed++;
  }

  public void brake() {
    speed--;
  }

  // Getter and setter methods (controlled access)
  public String getColor() {
        return color;
    }

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

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Create an object of the Car class
        myCar.setColor("Red");  // Set the color using a setter meth
        myCar.accelerate();
        System.out.println("Car color: " + myCar.getColor() + ", Spe
    }
}
```

In this example, the `Car` class defines the blueprint for car objects with properties like color and speed, and methods like accelerate and brake. We create a `myCar` object and interact with its properties and methods.

Types of member in java class

In Java, a class can have several types of members that define its structure and behavior. These members determine how the class interacts with other parts of your program. Here's a breakdown of the main types of class members:

1. Fields (Variables):

- Fields store the data (attributes) associated with an object of the class.
- They can be declared with different access modifiers (`public`, `private`, `protected`, or package-private) to control their visibility.
- Example:

```
public class Person {
  private String name;
  int age; // Package-private field (accessible within the same
}
```

2. Methods (Functions):

- Methods define the actions (behaviors) that objects of a class can perform.
- They can take arguments (parameters) and may return values.
- Access modifiers control their visibility as well.
- Example:

```
public class Car {
  public void accelerate() {
    // Code to increase speed
  }

  private void brake() {
    // Code to decrease speed (only accessible within the Car cl
  }
}
```

3. Constructors:

- Special methods with the same name as the class that are invoked when you create an object using `new`.
- They are used to initialize the object's state (set initial values for fields).
- Can have arguments to allow for customized object creation.
- Example:

```
public class Animal {
  public Animal(String name) {
    this.name = name;
  }

  private String name;
}
```

4. Nested Classes:

- Inner classes are defined within another class, creating a hierarchical relationship.
- They can access members of the enclosing class, including private members.
- There are four types of nested classes:
    - Static nested classes - Don't require an instance of the outer class to be created.
    - Member (non-static) nested classes - Must be created within an instance of the outer class.
    - Local inner classes - Defined within a method and can only access final local variables of the enclosing method.
    - Anonymous inner classes - Defined and instantiated at the same time, without a separate class name.
- Example (Static nested class):

```
public class OuterClass {
  public static class InnerClass {
    public void someMethod() {
      System.out.println("Inner class method");
    }
  }
}
```

5. Methods and Fields declared with `static` keyword:

- Static members belong to the class itself, not to individual objects.
- They can be accessed directly using the class name without creating an object.
- Often used for utility methods or constants shared by all objects of the class.
- Example:

```
public class MathUtil {
  public static double PI = 3.14159;

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

Choosing the Right Member Type:

- Use fields to store data associated with objects.
- Use methods to define actions (behaviors) of objects.
- Use constructors to initialize objects during creation.
- Use nested classes for specific functionalities related to the outer class.
- Use static members for class-level functionality or constants.

By understanding these different types of class members, you can effectively design and build well-structured Java classes that encapsulate data and behavior for your programs.

Static members of a class.

In Java, a static member refers to a class element that belongs to the class itself rather than to individual objects of that class. There are three main types of static members:

Static Fields (Variables): These are variables declared with the static keyword within a class. There exists only one copy of a static field shared by all instances of the class. Any changes made to the value of a static field through one object will be reflected for all other objects as well. Static fields are useful for constant values (like Math.PI) or values that need to be shared across all objects (like a counter).

Static Methods: Similar to static fields, static methods are declared with the static keyword. They can access other static members of the class directly, but cannot access non-static members (instance variables) without an object reference. Static methods are typically utility functions that don't rely on the specific state of an object and can be called without creating an object first. For instance, a Circle class might have a static method calculateArea(double radius) to compute the area of a circle.

Static Nested Classes: These are classes defined within another class using the static keyword. A static nested class can access only static members of the enclosing class, including private ones. It cannot access instance members of the enclosing class directly. Static nested classes are useful for helper classes that are tightly coupled to the outer class but don't need access to its object-specific data.

Here are some key points to remember about static members:

Shared State: Static members provide a way to share state (data) across all instances of a class.
Accessed Without Objects: You can access static fields and methods directly using the class name, without needing to create an object first.

Use Cases: Static members are useful for constants, utility functions, and helper classes that don't depend on object state.

Types of classes in java

Concrete Class:

A fundamental class type that serves as a blueprint for creating objects.
Defines both properties (variables) and behavior (methods) of the objects.

Can be directly instantiated to create objects.
Example:

```
public class Car {
    String color;
    int speed;

    public void accelerate() {
        speed++;
    }

    public void brake() {
        speed--;
    }

    public static void main(String[] args) {
        Car myCar = new Car(); // Create an object of Car
        myCar.color = "Red";
        myCar.accelerate();
        System.out.println("Car color: " + myCar.color + ", Spee
    }
}
```

Abstract Class:

Acts as a template for subclasses and cannot be directly instantiated.
Defines a common structure and behavior for related classes.
Can have abstract methods (without a body) that subclasses must implement.
Enforces consistency and ensures subclasses provide required functionality.
Example:

```java
public abstract class Animal {
    public abstract void makeSound(); // Abstract method

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

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

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

Interface:

Similar to an abstract class, but defines only abstract methods (no implementation).
Acts as a contract that specifies what functionality a class must provide.
A class can implement multiple interfaces.
Example:

```java
public interface Shape {
    double calculateArea();
}

public class Circle implements Shape {
    double radius;

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

public class Square implements Shape {
    double side;

    @Override
    public double calculateArea() {
        return side * side;
    }
}
```

Final Class:

Cannot be subclassed. Prevents further inheritance from that class.
Useful for representing fixed concepts or preventing unintended modifications.
Methods within a final class can also be declared as final to prevent overriding in subclasses.
Example:

```
public final class MathUtil {
    public static final double PI = 3.14159; // Final static va

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

Inner Class:

A class defined within another class. Provides access to the enclosing class's members.

There are four types of inner classes:

Static Nested Class: Defined with static keyword. Can access only static members of the enclosing class.

```
public class OuterClass {
    public static class StaticNestedClass {
        public void printMessage() {
            System.out.println("Message from Static Nested Clas
        }
    }
}
```

Member Inner Class: Defined without static keyword. Can access both static and non-static members of the enclosing class. Needs an object reference of the outer class to be accessed.

```
public class OuterClass {
    private int data = 10;

    public class MemberInnerClass {
        public void printData() {
            System.out.println("Data from Outer Class: " + data
        }
    }
}
```

Local Inner Class: Defined within a method. Short-lived and can access local variables of the enclosing method.

```
public class OuterClass {
    public void someMethod() {
        int value = 5;
        class LocalInnerClass {
            public void printValue() {
                System.out.println("Local variable from enclosin
            }
        }

        LocalInnerClass inner = new LocalInnerClass();
        inner.printValue();




    }
}
```

Anonymous Inner Class: Defined and instantiated at the same time, without a separate class name. Often used for implementing interfaces or simple functionality on the fly.

```
Syntax
new InterfaceName() {
// Implement interface methods here
}


Button myButton = new Button("Click Me");

// Anonymous inner class implementing ActionListener
myButton.addActionListener(new ActionListener() {
@Override
public void actionPerformed(ActionEvent e) {
System.out.println("Button clicked!");
// You can add more logic here, like opening a new window, etc.
}
});
```

Sealed Class (Java 15):

Restricts inheritance hierarchy by specifying allowed subclasses.
Enhances control over class relationships and prevents unintended extensions.

```
public sealed class Shape permits Circle, Square {
    public abstract double calculateArea();
}

public final class Circle extends Shape {
    double radius;

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

// Square can inherit from Shape because it's listed in the `pe
public class Square extends Shape {
    double side;

    @Override
    public double calculateArea() {
        return side * side;
    }
}

// This would cause a compile-time error because Triangle is no
// public class Triangle extends Shape { ... }
```

POJO (Plain Old Java Object):

This is not a technical class type but rather a design pattern.
A POJO class is a simple class that focuses on data storage and retrieval.
It typically has private member variables and public getter and setter methods to access and modify the data.

POJOs are often used for data transfer between applications or persistence with databases.

Here's a POJO example of a Person class with name and age in Java:

```
Java
public class Person {

    private String name;
    private int age;

    // Default constructor (optional, but good practice)
    public Person() {}

    // Constructor with arguments (allows initializing name and
    public Person(String name, int age) {
                this.name = name;
            this.age = age;
        }

        // Getter for name
        public String getName() {
            return name;
        }

        // Setter for name
        public void setName(String name) {
            this.name = name;
            }

            // Getter for age
            public int getAge() {
                return age;
            }

            // Setter for age (can add validation here to en
            public void setAge(int age) {
                if (age < 0) {
                    throw new IllegalArgumentException("Age
                }
                this.age = age;
            }

            // You can add other methods specific to Person

            @Override
            public String toString() {
                return "Person [name=" + name + ", age=" + a
            }
        }
```

Record Class (Java 14):

Concise way to define immutable data holder classes.
Reduces boilerplate code for getters, equals(), hashCode(), and toString().

```
public record Person(String name, int age) {
// Implicit constructor, getters (name(), age()), equals(), hash
}
```

Access modifiers

Access modifiers in Java are keywords that define the accessibility (visibility) of classes, fields (variables), methods, and constructors within a Java program. They control which parts of your code can access these elements and how they can be used. Here's a breakdown of the four main access modifiers:

1. Public:

- Members declared as `public` are accessible from anywhere in your program, regardless of the package or class they are in.
- This is the most permissive access modifier and should be used cautiously for elements that need to be widely accessible throughout your application.

Example:

```
public class PublicClass {
  public String publicField;
  public void publicMethod() {
    System.out.println("This is a public method!");
  }
}
```

2. Private:

- Members declared as `private` are only accessible within the class they are defined in.
- They are invisible to other classes, even within the same package.
- This is the most restrictive access modifier and promotes encapsulation by protecting internal implementation details.

Example:

```
class PrivateClass {
  private int privateField;
  private void privateMethod() {
    System.out.println("This is a private method!");
  }
}
```

3. Protected:

- Members declared as `protected` are accessible within the class they are defined in, as well as in subclasses of that class (even if the subclasses are in different packages).
- This allows controlled inheritance and sharing of implementation details between a class and its subclasses.

Example:

```
class ParentClass {
  protected String protectedField;
  protected void protectedMethod() {
    System.out.println("This is a protected method!");
  }
}

class ChildClass extends ParentClass {
  public void someMethod() {
    System.out.println(protectedField); // Accessing inherited
    protectedMethod(); // Calling inherited protected method
  }
}
```

4. Default (Package-Private):

- If you don't explicitly use any access modifier, the member inherits the default access modifier, which is package-private.
- Members with default access are only accessible from classes within the same package.
- This provides a balance between visibility and encapsulation, allowing access within the same package but restricting access from outside.

Example:

```
package com.example;

class DefaultClass {
  String defaultField;
  void defaultMethod() {
    System.out.println("This is a default method!");
  }
}

// Another class in the same package can access these members
class SamePackageClass {
  public void someMethod() {
    DefaultClass obj = new DefaultClass();
    System.out.println(obj.defaultField);
    obj.defaultMethod();
  }
}
```

Choosing the Right Access Modifier:

- Use `public` sparingly, only for members that truly need to be accessed from anywhere.
- Favor `private` for internal implementation details to promote data hiding and encapsulation.
- Use `protected` for members that should be shared and potentially overridden in subclasses.
- Use `default` for members that should be accessible within the same package but hidden from outside packages.

By effectively using access modifiers, you can improve code organization, maintainability, and control the reusability of your classes and their components in Java.

Class vs Object

The main difference between an object and a class in Java lies in their concept and how they are used in your program:

Class:

- A class acts as a blueprint or template that defines the properties (variables) and behavior (methods) of objects.
- It specifies the characteristics that all objects of that class will share.
- You can think of a class as a recipe that outlines the ingredients (variables) and steps (methods) needed to create a dish (object).
- A class itself doesn't exist in memory until you create an instance of it.

Object:

- An object is an instance of a class. It's a concrete realization of the blueprint defined by the class.
- Each object has its own set of values for the properties (variables) defined in the class.
- Objects can interact with each other by calling methods on each other.
- When you create an object using the `new` keyword, memory is allocated for the object, and the initial values for its properties are set.

Here's a table summarizing the key differences:

Feature	**Class**	**Object**
Concept	Blueprint, template	Instance of a class
Properties	Defines properties (variables)	Has its own set of values for properties
Behavior	Defines methods (functions)	Can call methods on itself and other objects
Memory allocation	No memory allocated when class is defined	Memory allocated when object is created
Creation	Defined using the `class` keyword	Created using the `new` keyword
Reusability	Reusable - can create multiple objects	Not directly reusable - represents a single entity

Example:

```
public class Car { // Class definition (blueprint)
  String color;
  int speed;

  public void accelerate() {
    speed++;
  }
}

Car myCar = new Car(); // Object creation (instance)
myCar.color = "Red";
myCar.accelerate();
System.out.println("Car color: " + myCar.color + ", Speed: " + r
```

In this example:

- The `Car` class defines the properties (`color` and `speed`) and a method (`accelerate`) that all `Car` objects will have.
- The `myCar` object is an instance of the `Car` class. It has its own specific color ("Red") and speed (initially 0).
- We can call the `accelerate` method on `myCar` to modify its speed.

By understanding the distinction between classes and objects, you can effectively structure your Java programs and create objects that represent real-world entities or concepts.

Class vs Abstract class vs interface

Here's a breakdown of the key differences between classes, abstract classes, and interfaces in Java:

Class:

- A fundamental blueprint for creating objects.
- Defines both properties (variables) and behavior (methods) of the objects.

- Can be directly instantiated (using `new`) to create objects.
- Provides a complete implementation for its functionalities.

Abstract Class:

- Acts as a template for subclasses and cannot be directly instantiated.
- Defines a common structure and behavior for related classes.
- Can have abstract methods (without a body) that subclasses must implement.
- Enforces consistency and ensures subclasses provide required functionality.
- May also have concrete (implemented) methods that can be reused by subclasses.

Interface:

- Similar to an abstract class, but defines only abstract methods (no implementation).
- Acts as a contract that specifies what functionality a class must provide.
- A class can implement multiple interfaces, inheriting the required methods from each.
- Focuses on what a class can do, not how it does it (implementation details are left to the class).

Here's a table summarizing the key points:

Feature	Class	Abstract Class	Interface
Instantiation	Can be directly instantiated (using `new`)	Cannot be directly instantiated	Cannot be directly instantiated
Abstract methods	No (all methods must be implemented)	Can have abstract methods (without a body)	Only abstract methods (no implementation)
Concrete methods	Can have concrete methods (with body)	Can have concrete methods (with body)	No concrete methods (only abstract methods)

Inheritance	Can be inherited from	Can be inherited from (but cannot be instantiated)	Can be implemented by multiple classes
Implementation details	Can have full implementation details	Can have partial implementation details	No implementation details (only method declarations)
Example	`Car`, `Employee`	`Animal` (abstract methods like `makeSound()`)	`Shape` (abstract methods like `calculateArea()`)

Choosing the Right Approach:

- Use a class when you have a complete and concrete definition of an object with its properties and behavior.
- Use an abstract class when you want to define a common structure and behavior for related classes, enforcing certain functionalities through abstract methods.
- Use an interface when you want to specify a contract (what functionalities a class must provide) for achieving a specific behavior, allowing for different implementations in different classes.

By understanding these differences, you can effectively model real-world entities and their relationships in your Java programs using classes, abstract classes, and interfaces.

Interface vs Abstract class

Here's a breakdown of when to use interfaces and abstract classes in Java, along with examples:

When to Use Interfaces:

- **Standardizing behavior:** Use interfaces when you want to define a set of methods that multiple unrelated classes can implement in different ways. This promotes code reusability and ensures consistency in functionality across different implementations.

Example:

```
public interface Shape {
  double calculateArea();
}

public class Circle implements Shape {
  double radius;

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

public class Square implements Shape {
  double side;

  @Override
  public double calculateArea() {
    return side * side;
  }
}
```

In this example, the `Shape` interface defines a single method `calculateArea()`. Both `Circle` and `Square` classes implement this interface, providing their own implementations for calculating area based on their specific shapes.

- **Achieving loose coupling:** Interfaces promote loose coupling between classes. A class that uses an interface doesn't care about the specific implementation details. It just relies on the fact that any class implementing the interface will provide the required methods. This improves maintainability and testability.

Example:

```
public interface DatabaseConnection {
  void connect();
  void disconnect();
  void executeQuery(String query);
}

public class MySQLConnection implements DatabaseConnection {
  // Specific implementation for connecting to a MySQL database
  @Override
  public void connect() { ... }
  @Override
  public void disconnect() { ... }
  @Override
  public void executeQuery(String query) { ... }
}

public class OracleConnection implements DatabaseConnection {
  // Specific implementation for connecting to an Oracle databas
  @Override
  public void connect() { ... }
  @Override
  public void disconnect() { ... }
  @Override
  public void executeQuery(String query) { ... }
}
```

The `DatabaseConnection` interface defines methods for database interactions. Both `MySQLConnection` and `OracleConnection` implement this interface, but their specific implementation details for connecting and querying data might differ. This allows switching between database implementations without modifying the code that uses the interface.

When to Use Abstract Classes:

- **Modeling inheritance and partial implementation:** Use abstract classes when you have a set of related classes with a common structure and behavior. The abstract class can define some common methods and properties, while subclasses can extend and potentially override these methods to provide specific implementations.

Example:

```
public abstract class Animal {
  public abstract void makeSound(); // Abstract method

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

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

public class Cat extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Meow!");
  }
}
```

The `Animal` abstract class defines a common `makeSound` method (abstract) and a `sleep` method (concrete). Both `Dog` and `Cat` classes extend `Animal` and implement the `makeSound` method to provide their specific sounds, while inheriting the `sleep` behavior.

- **Enforcing specific behavior in subclasses:** Abstract classes can enforce certain behavior in subclasses by declaring abstract methods that must be implemented. This ensures consistency within the related classes.

Key Points:

- Interfaces focus on "what" functionalities a class should provide, while abstract classes can provide some "how" through concrete methods.
- Interfaces promote multiple inheritance, while abstract classes follow a single inheritance hierarchy.
- Use interfaces for behavior contracts, and abstract classes for defining a common foundation with partial implementation for related classes.