1. What are object-oriented concepts? What is the difference between object-based, object-oriented, and fully object-oriented language?

Object-Oriented Concepts:

- **Class**: A blueprint for creating objects.
- **Object**: A specific instance of a class, like a car being an instance of a Car class.
- **Encapsulation**: Keeping data (variables) and code (methods) together in a single unit (class), and restricting direct access to some components.
- **Abstraction**: Hiding the complexity of code and showing only the important details.
- **Inheritance**: One class can inherit features from another class.
- **Polymorphism**: The ability to take many forms, like a function that behaves differently based on input.

Differences:

- **Object-based language**: Supports objects but doesn't have all features like inheritance or polymorphism (e.g., JavaScript, VBScript).
- **Object-oriented language**: Fully supports all OOP features like classes, inheritance, etc. (e.g., Java, C++).
- **Fully object-oriented language**: Everything is treated as an object, including primitive types (e.g., Smalltalk).

```
1. Object-Oriented Concepts (Class, Object, Encapsulation,
Inheritance, Polymorphism)

// Class Example
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Inheritance Example
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");  // Polymorphism (Method
Overriding)
    }
}

// Encapsulation Example
class Person {
    private String name;  // Private variable (data hiding)

    public String getName() {  // Getter
        return name;
```

```
    }

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

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound();  // Outputs "Dog barks"

        Person person = new Person();
        person.setName("John");
        System.out.println(person.getName());  // Outputs "John"
    }
}

1. Object-Oriented Concepts: Class, Object, Encapsulation,
Inheritance, Polymorphism

// Base class (Parent class)
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Derived class (Child class) - Inheritance
class Dog extends Animal {
    // Polymorphism (Method Overriding): The child class provides a
specific implementation
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Class demonstrating Encapsulation
class Person {
    // Private attribute - cannot be accessed directly outside the
class
    private String name;
```

```
    // Getter method to access the private attribute
    public String getName() {
        return name;
    }

    // Setter method to modify the private attribute
    public void setName(String name) {
        this.name = name;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object (instance) of Dog class
        Dog myDog = new Dog();
        myDog.sound();  // Outputs "Dog barks" (Polymorphism through
method overriding)

        // Demonstrating Encapsulation with Person class
        Person person = new Person();
        person.setName("John");  // Setting name using the setter
method
        System.out.println(person.getName());  // Getting name using
the getter method; Outputs "John"
    }
}
```

2. What are the advantages of Object-Oriented Programming? What is data security?

Advantages:

- **Modularity**: Code is organized into objects, making it easier to manage and reuse.
- **Reusability**: Objects and classes can be reused across programs.
- **Maintainability**: Easier to update or modify without affecting other parts of the program.
- **Flexibility**: Through polymorphism, code can adapt to new changes without rewriting.

Data Security:

- OOP uses encapsulation to hide data, protecting it from unauthorized access. Only specific methods can change or retrieve the values, providing security.

```
2. Advantages of OOP and Data Security (Encapsulation)

class BankAccount {
```

```
    private double balance;  // Encapsulation for data security

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }
}
```

3. Advantages of OOP and Data Security (Encapsulation)

```
// Class to represent a Bank Account, demonstrating Encapsulation
class BankAccount {
    // Private variable to hold account balance (data hiding)
    private double balance;

    // Public method to retrieve balance (Getter method)
    public double getBalance() {
        return balance;
    }

    // Public method to deposit money (provides controlled access to
modify balance)
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;  // Balance increases by the deposit
amount
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
```

```
            balance -= amount;  // Balance decreases by the
withdrawal amount
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a new BankAccount object
        BankAccount myAccount = new BankAccount();

        // Deposit money into the account
        myAccount.deposit(500);  // Deposits 500
        System.out.println("Balance: " + myAccount.getBalance());  //
Outputs "Balance: 500.0"

        // Withdraw money from the account
        myAccount.withdraw(200);  // Withdraws 200
        System.out.println("Balance: " + myAccount.getBalance());  //
Outputs "Balance: 300.0"
    }
}
```

3. What is a class and object? Give a real-life example.

- **Class**: A template or blueprint for creating objects. It defines properties (attributes) and methods (functions).
- **Object**: A specific instance created from a class.

Example:

- A **class** "Car" might have properties like "color," "model," and methods like "start" or "stop."
- An **object** could be "MyCar," which is a red, 2020 model that can be started or stopped.

3.Class and Object with Real-Life Example

```
class Car {
    String model;
    String color;

    void start() {
        System.out.println("Car is starting");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Object creation
        myCar.model = "Honda Civic";
        myCar.color = "Red";
        myCar.start();  // Outputs "Car is starting"
    }
}
```

3.Class and Object with Real-Life Example

```
// Class representing a Car
class Car {
    String model;  // Attribute to store car model
    String color;  // Attribute to store car color

    // Method to simulate starting the car
    void start() {
        System.out.println("Car is starting");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object (instance) of the Car class
        Car myCar = new Car();
        myCar.model = "Honda Civic";  // Assigning values to object
attributes
        myCar.color = "Red";

        // Calling the start method
        myCar.start();  // Outputs "Car is starting"
    }
}
```

---

4. What are the characteristics of an object? Explain them.

- **Identity**: Each object is unique, like each person having a different name.
- **State**: An object has attributes that represent its current situation, like a car's color or speed.
- **Behavior**: Objects can perform actions, like a car accelerating or braking, which are defined by its methods.

4. Characteristics of an Object

```
class Dog {
    String breed;  // State (attribute)

    void bark() {  // Behavior (method)
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog myDog = new Dog();  // Identity (unique instance)
        myDog.breed = "Labrador";  // State
        myDog.bark();  // Behavior
    }
}
```

4. Characteristics of an Object (State, Behavior, and Identity)

```
// Class representing a Dog
class Dog {
    String breed;  // State (Attribute representing the breed of the
dog)

    // Behavior (Method to simulate the dog barking)
    void bark() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        // Creating an object (instance) of Dog class (Identity)
        Dog myDog = new Dog();

        // Assigning state to the object
        myDog.breed = "Labrador";  // Setting breed of the dog

        // Calling the behavior (method)
        myDog.bark();  // Outputs "Dog barks"
    }
}
```

5. What is the need for getter and setter functions in a class?

- **Getter**: A method that allows you to access a variable from outside the class safely.
- **Setter**: A method that allows you to change the value of a variable safely.
- **Need**: Directly accessing variables can lead to errors. Getters and setters provide control over how a variable is read or modified, ensuring that any changes are valid.

```
5. Getter and Setter Functions (Encapsulation)


class Student {
    private int age;  // Private attribute

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

    public void setAge(int age) {  // Setter
        if (age > 0) {
            this.age = age;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setAge(20);  // Setting value using setter
        System.out.println(student.getAge());  // Getting value using
getter
    }
}
```

```
5. Getter and Setter Functions (Encapsulation)

// Class representing a Student
class Student {
    // Private attribute (Encapsulation: cannot be accessed directly)
    private int age;
```

```
    // Getter method to retrieve the value of the private attribute
    public int getAge() {
        return age;
    }

    // Setter method to set the value of the private attribute
    public void setAge(int age) {
        if (age > 0) {
            this.age = age;  // Only assign if age is positive
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object (instance) of Student class
        Student student = new Student();

        // Setting age using setter
        student.setAge(20);  // Sets age to 20

        // Getting age using getter
        System.out.println("Student Age: " + student.getAge());  //
Outputs "Student Age: 20"
    }
}
```

6. What is abstraction and encapsulation? Give a real-life example.

- **Abstraction**: Hiding complex details and showing only the necessary parts.
    - **Example**: When you drive a car, you only need to know how to use the steering wheel and pedals, not how the engine works.
- **Encapsulation**: Wrapping data and methods into a single unit and restricting access to them.
    - **Example**: A car's internal mechanisms (like the engine) are hidden from the user; you interact with simple controls, which is encapsulation.

```
6. Abstraction and Encapsulation

abstract class Appliance {
    abstract void turnOn();  // Abstract method (Abstraction)
}
```

```
class WashingMachine extends Appliance {
    @Override
    void turnOn() {
        System.out.println("Washing machine is turning on");
    }
}

public class Main {
    public static void main(String[] args) {
        Appliance myMachine = new WashingMachine();
        myMachine.turnOn();  // Outputs "Washing machine is turning
on"
    }
}
```

6.Abstraction and Encapsulation

```
// Abstract class representing an Appliance (provides abstraction)
abstract class Appliance {
    // Abstract method (no implementation)
    abstract void turnOn();
}

// Concrete class representing a Washing Machine, extending Appliance
class WashingMachine extends Appliance {
    @Override
    void turnOn() {
        System.out.println("Washing machine is turning on");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of WashingMachine (Abstract Appliance
type)
        Appliance myMachine = new WashingMachine();

        // Calling the abstract method (now implemented by
WashingMachine)
        myMachine.turnOn();  // Outputs "Washing machine is turning
on"
    }
}
```

7. What is polymorphism? What are its types? Explain with examples.

Polymorphism: The ability of a function or object to take different forms or behave differently based on the context.

Types:

- **Compile-time polymorphism (Method overloading)**: When two or more methods have the same name but different parameters.
    - **Example**: A "print" function might print a number or a string based on the input type.
- **Run-time polymorphism (Method overriding)**: When a subclass modifies a method from its parent class to give it new behavior.

    - **Example**: A "draw" method in a "Shape" class can be overridden by subclasses like "Circle" or "Square" to draw the respective shape.

7. Polymorphism (Method Overloading and Method Overriding)

```
class MathOperation {
    // Method Overloading
    int add(int a, int b) {
        return a + b;
    }

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

class Parent {
    void display() {
        System.out.println("Display from Parent");
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Display from Child");  // Method
Overriding
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MathOperation math = new MathOperation();
        System.out.println(math.add(2, 3));  // Outputs 5 (Method
Overloading)
        System.out.println(math.add(2, 3, 4));  // Outputs 9 (Method
Overloading)

        Parent obj = new Child();
        obj.display();  // Outputs "Display from Child" (Method
Overriding)
    }
}
```

7.Polymorphism (Method Overloading and Method Overriding)

```
// Class demonstrating method overloading (multiple methods with the
same name but different parameters)
class MathOperation {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers (overloading)
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

// Base class for demonstrating method overriding
class Parent {
    void display() {
        System.out.println("Display from Parent");
    }
}

// Derived class demonstrating method overriding
class Child extends Parent {
    @Override
```

```
    void display() {
        System.out.println("Display from Child");  // Overrides
Parent's display method
    }
}

public class Main {
    public static void main(String[] args) {
        // Demonstrating Method Overloading
        MathOperation math = new MathOperation();
        System.out.println(math.add(2, 3));  // Outputs 5 (add with 2
parameters)
        System.out.println(math.add(2, 3, 4));  // Outputs 9 (add
with 3 parameters)

        // Demonstrating Method Overriding
        Parent obj = new Child();  // Parent reference, Child object
        obj.display();  // Outputs "Display from Child" (overridden
method)
    }
}
```

8. What is method overloading? What are the rules of method overloading? Why is the return type not considered in method overloading?

Method Overloading: Having multiple methods with the same name but different parameters within the same class.

Rules:

- Methods must differ in the number of parameters or type of parameters.
- It does not depend on the return type of the method.

Why return type is not considered:

- The compiler uses the method signature (method name + parameters) to identify which method to call. Since the return type is not part of the method signature, it doesn't help in distinguishing overloaded methods.

```
8. Method Overloading

class Calculator {
    int add(int a, int b) {
```

```
        return a + b;
    }

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

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(3, 4));  // Outputs 7
        System.out.println(calc.add(3.5, 4.5));  // Outputs 8.0
    }
}
```

8.Method Overloading Example

```
// Class demonstrating method overloading
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add two double numbers (method overloading)
    double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        // Creating an object of Calculator class
        Calculator calc = new Calculator();

        // Calling overloaded methods
        System.out.println(calc.add(3, 4));  // Outputs 7 (add with
integers)
        System.out.println(calc.add(3.5, 4.5));  // Outputs 8.0 (add
with doubles)
    }
}
```

9. What are the different types of hierarchy? When to use which one?

Types of hierarchy:

- **Single inheritance**: One class inherits from one superclass.
  - Use when there's only one parent class.
- **Multiple inheritance**: A class inherits from more than one class.
  - Use with caution in some languages as it can cause complexity (e.g., C++ allows it, but Java doesn't).
- **Multilevel inheritance**: A class is derived from another derived class.
  - Use when there's a chain of inheritance.
- **Hierarchical inheritance**: Multiple classes inherit from the same base class.
  - Use when you have multiple child classes that share common functionality.

```
9. Types of Hierarchy (Single, Multilevel, Hierarchical Inheritance)

// Single Inheritance
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

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

// Multilevel Inheritance
class Puppy extends Dog {
    void weep() {
        System.out.println("Weeping...");
    }
}

// Hierarchical Inheritance
```

```
class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.eat();  // Outputs "Eating..." (inherited from Animal)
        puppy.bark();  // Outputs "Barking..." (inherited from Dog)
        puppy.weep();  // Outputs "Weeping..."

        Cat cat = new Cat();
        cat.eat();  // Outputs "Eating..." (from Animal)
        cat.meow();  // Outputs "Meowing..."
    }
}
```

9. Types of Hierarchy: Single, Multilevel, Hierarchical Inheritance

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

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

// Multilevel Inheritance Example
class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy is weeping");
    }
}
```

```
// Hierarchical Inheritance Example
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing");
    }
}

public class Main {
    public static void main(String[] args) {
        // Multilevel Inheritance Example
        Puppy puppy = new Puppy();
        puppy.eat();  // Inherited from Animal; Outputs "Animal is
eating"
        puppy.bark();  // Inherited from Dog; Outputs "Dog is
barking"
        puppy.weep();  // Outputs "Puppy is weeping"

        // Hierarchical Inheritance Example
        Cat cat = new Cat();
        Cat
```

10. What is the difference between method overloading and method overriding?

- **Method Overloading**: Same method name but different parameters (within the same class). It occurs at compile time.
    - Example: `add(int a, int b)` and `add(int a, int b, int c)`
- **Method Overriding**: A method in a subclass has the same name and parameters as a method in its parent class but provides a different implementation. It occurs at runtime.
    - Example: A subclass `Dog` might override a method `sound()` from class `Animal` to make it bark.

10. Method Overloading vs Method Overriding

```
// Method Overloading Example

class MathOperation {

    int multiply(int a, int b) {

        return a * b;
```

```
    }

    int multiply(int a, int b, int c) {
        return a * b * c;
    }
}

// Method Overriding Example
class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Bike extends Vehicle {
    @Override
    void run() {
        System.out.println("Bike is running");  // Overriding run
method
    }
}

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

```
        MathOperation math = new MathOperation();
        System.out.println(math.multiply(2, 3));  // Outputs 6
        System.out.println(math.multiply(2, 3, 4));  // Outputs 24

        Vehicle myBike = new Bike();
        myBike.run();  // Outputs "Bike is running"
    }
}
```

11. What is object slicing? Explain object slicing in the context of up-casting.

Object Slicing: This happens when an object of a subclass is assigned to a variable of a superclass, and the subclass-specific data gets "sliced off."

Up-casting: When a reference to a subclass is treated as a reference to its superclass. When this happens, only the part of the object defined in the superclass is accessible.

Example:

- If `Car` is a subclass of `Vehicle`, and you store a `Car` object in a `Vehicle` variable, only the properties and methods of `Vehicle` will be accessible, and any additional data in `Car` (like a special feature) will be "sliced" away.

11. Object Slicing in the Context of Up-casting

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

class Car extends Vehicle {
    int doors = 4;  // Car-specific attribute
}

public class Main {
    public static void main(String[] args) {
        Vehicle myVehicle = new Car();  // Up-casting Car object to Vehicle type
        myVehicle.start();  // Accessing method from Vehicle class (Works fine)

        // Object slicing: We cannot access Car-specific features like doors
        // myVehicle.doors; // This will cause a compilation error
    }
}
```

12. What is down-casting, and when is it required? Explain with code.

Down-casting: Converting a reference of a superclass back into a reference of a subclass. It allows you to access subclass-specific properties and methods.

When required: It is needed when you up-cast an object (from subclass to superclass) and later want to access subclass-specific features again.

Example in Java:

```
java
Copy code
class Vehicle {
    void start() {
        System.out.println("Vehicle starts");
    }
}
class Car extends Vehicle {
    void openTrunk() {
        System.out.println("Trunk is open");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle myVehicle = new Car(); // Up-casting
        myVehicle.start(); // Works fine

        Car myCar = (Car) myVehicle; // Down-casting
        myCar.openTrunk(); // Now we can access Car-specific methods
    }
}
```

12. Down-casting with Code Example

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");  // Dog-specific method
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Animal myAnimal = new Dog();  // Up-casting Dog object to
Animal
        myAnimal.sound();  // Works (from Animal class)

        // Down-casting Animal object back to Dog
        Dog myDog = (Dog) myAnimal;
        myDog.bark();  // Now we can access Dog-specific methods
    }
}
```

13. What do you know about association, composition, and aggregation? Explain with an example.

Association: A general relationship between two classes, where they can be linked to each other. It doesn't define who "owns" the other.

- **Example**: A teacher and a student can have an association in a school, where the teacher teaches the student.

Aggregation: A "has-a" relationship where one class contains another, but the contained object can exist independently.

- **Example**: A class `Library` can have many `Books`, but even if the `Library` is destroyed, the `Books` can still exist separately.

Composition: A strong form of aggregation where one class owns the other, and the contained object cannot exist without the owner class.

- **Example**: A `Car` and its `Engine`. If the `Car` is destroyed, the `Engine` is also destroyed since the engine is an integral part of the car.

13. Association, Aggregation, and Composition Example

```
// Association Example

class Teacher {

    String name;

    Teacher(String name) {
```

```
        this.name = name;
    }
}

class Student {
    String name;
    Teacher teacher;  // Association between Student and Teacher
    Student(String name, Teacher teacher) {
        this.name = name;
        this.teacher = teacher;
    }
}

// Aggregation Example
class Library {
    String name;
    Library(String name) {
        this.name = name;
    }
}

class Book {
    String title;
    Library library;  // Aggregation (Library can exist without books)
```

```
    Book(String title, Library library) {
        this.title = title;
        this.library = library;
    }
}

// Composition Example
class Engine {
    Engine() {
        System.out.println("Engine created");
    }
}

class Car {
    private Engine engine;  // Composition: Engine cannot exist without Car
    Car() {
        engine = new Engine();  // Car creates Engine
        System.out.println("Car created with engine");
    }
}

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

```
        // Association Example
        Teacher teacher = new Teacher("Mr. Smith");
        Student student = new Student("John", teacher);

        // Aggregation Example
        Library library = new Library("Central Library");
        Book book = new Book("Java Programming", library);

        // Composition Example
        Car myCar = new Car();  // Car creates an engine
automatically
    }
}
```

14. What are the different types of inheritance? Explain with an example. What are the problems with multiple inheritance?

Types of Inheritance:

- **Single Inheritance**: A class inherits from one parent class.
  - **Example**: `class Dog extends Animal.`
- **Multilevel Inheritance**: A class is derived from another derived class.
  - **Example**: `class Puppy extends Dog extends Animal.`
- **Hierarchical Inheritance**: Multiple classes inherit from the same base class.
  - **Example**: `class Cat extends Animal` and `class Dog extends Animal.`
- **Multiple Inheritance**: A class inherits from more than one parent class. Some languages (like Java) do not support this directly.
  - **Example in C++**: `class Amphibian extends Animal, Vehicle.`

Problems with Multiple Inheritance:

- **Ambiguity**: If two parent classes have the same method, the compiler may get confused about which method to use.
    - **Example**: If both parent classes have a method `start()`, the child class might face ambiguity about which one to call.
- **Diamond Problem**: This happens when two classes inherit from the same parent class and a third class inherits from both, creating confusion about inheritance paths.

14. Types of Inheritance and Issues with Multiple Inheritance

```
// Single Inheritance
class Animal {
    void eat() {
        System.out.println("Animal eats");
    }
}

class Dog extends Animal {  // Single Inheritance
    void bark() {
        System.out.println("Dog barks");
    }
}

// Multilevel Inheritance
class Puppy extends Dog {  // Inheriting from Dog, which inherits from Animal
    void weep() {
        System.out.println("Puppy weeps");
```

```
    }
}

// Hierarchical Inheritance
class Cat extends Animal {  // Multiple classes inherit from the same superclass
    void meow() {
        System.out.println("Cat meows");
    }
}

// Multiple Inheritance in C++ (Java does not support multiple inheritance)
class Vehicle {
    void start() {
        System.out.println("Vehicle starts");
    }
}

class Amphibian /*extends Vehicle, Animal*/ {  // This would cause ambiguity in Java
    void swim() {
        System.out.println("Amphibian swims");
    }
}
```

```
public class Main {

    public static void main(String[] args) {

        Puppy puppy = new Puppy();

        puppy.eat();  // Outputs "Animal eats"

        puppy.bark();  // Outputs "Dog barks"

        puppy.weep();  // Outputs "Puppy weeps"



        Cat cat = new Cat();

        cat.eat();  // Outputs "Animal eats"

        cat.meow();  // Outputs "Cat meows"

    }

}
```

15. What is the difference between interface, abstract class, and non-abstract class? Which one to use where?

-

Interface: A contract where all methods are abstract (no implementation). Classes that implement the interface must provide their own implementation of the methods.

-
  - **When to use**: Use an interface when you want different classes to agree on method names but let them define how those methods work.
  - **Example**: A `Flyable` interface could be implemented by both `Bird` and `Airplane` classes.
-

Abstract Class: A class that can have both abstract methods (without implementation) and concrete methods (with implementation). It cannot be instantiated directly.

-

- **When to use**: Use an abstract class when you want to share some code between related classes but also force subclasses to provide certain method implementations.
- **Example**: An abstract class `Animal` might have an abstract method `sound()` that subclasses must implement (e.g., `Dog` class implements `sound()` as `bark()`).
-

Non-Abstract Class: A regular class where all methods have implementations. You can create objects of this class.

-

- **When to use**: Use when you want a fully functional class that doesn't need to be extended or overridden.
- **Example**: A `Car` class where all methods are fully defined and can be used to create `Car` objects.

15. Interface, Abstract Class, and Non-Abstract Class

```
// Interface Example

interface Flyable {

   void fly();  // Abstract method (No implementation)

}



class Bird implements Flyable {

   @Override

   public void fly() {

      System.out.println("Bird flies");

   }

}



// Abstract Class Example

abstract class Animal {

   abstract void sound();  // Abstract method (no implementation)
```

```
    void sleep() {
        System.out.println("Animal sleeps");  // Concrete method
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Non-Abstract Class Example
class Car {
    void drive() {
        System.out.println("Car is driving");
    }
}

public class Main {
    public static void main(String[] args) {
        // Using Interface
        Flyable bird = new Bird();
        bird.fly();  // Outputs "Bird flies"

        // Using Abstract Class
        Dog dog = new Dog();
```

```
        dog.sound();  // Outputs "Dog barks"

        dog.sleep();  // Outputs "Animal sleeps"



        // Using Non-Abstract Class

        Car car = new Car();

        car.drive();  // Outputs "Car is driving"

    }

}
```

16. Which are the different types of design patterns? Explain the singleton design pattern.

Design Patterns: Reusable solutions to common problems in software design. Some common types are:

- **Creational Patterns**: Deal with object creation.
    - **Example**: Singleton, Factory, Builder.
- **Structural Patterns**: Deal with the composition of classes and objects.
    - **Example**: Adapter, Composite, Proxy.
- **Behavioral Patterns**: Deal with object interaction and responsibility.
    - **Example**: Observer, Strategy, Command.

Singleton Design Pattern:

- **Definition**: Ensures that a class has only one instance and provides a global point of access to that instance.
- **Usage**: Used when you need to ensure that there is only one instance of a class throughout the application (e.g., a configuration manager or a database connection).

Example:

```
java
Copy code
public class Singleton {
    private static Singleton instance;

    // Private constructor prevents instantiation
    private Singleton() {}

    // Global point of access to the instance
```

```
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

In this example, `getInstance()` ensures only one instance of `Singleton` is created.

15. Singleton Design Pattern Example

```
class Singleton {

    // Static variable to hold the single instance of the class

    private static Singleton instance = null;



    // Private constructor to prevent direct instantiation

    private Singleton() {

        System.out.println("Singleton instance created");

    }



    // Public method to provide access to the single instance

    public static Singleton getInstance() {

        if (instance == null) {

            instance = new Singleton();  // Create the instance if
not already created

        }

        return instance;

    }
```

```
}



public class Main {

    public static void main(String[] args) {

        // Try to get the singleton instance

        Singleton obj1 = Singleton.getInstance();  // Outputs
"Singleton instance created"

        Singleton obj2 = Singleton.getInstance();  // No new instance
is created



        // Verify both references point to the same instance

        System.out.println(obj1 == obj2);  // Outputs true

    }

}
```