1. What is polymorphism and how can it be achieved?

Polymorphism in object-oriented programming (OOP) refers to the ability of a variable, function, or object to take on multiple forms. It allows objects of different classes to be treated as objects of a common superclass. There are two types of polymorphism: compile-time (method overloading) and runtime (method overriding).

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

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

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

2. What is method overriding? Explain access specifiers with an example.

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass should have the same signature as the method in the superclass.

Access Specifiers:

- `public`: Accessible from anywhere.
- `protected`: Accessible within the same package and subclasses.
- `default` (no specifier): Accessible only within the same package.
- `private`: Accessible only within the same class.

Example:

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

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

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();
        obj.display();  // Outputs "Display from Child"
    }
}
```

3. How to use method overriding with exception handling? Provide an example.

When overriding methods, the overridden method in the subclass can throw the same exceptions or subclasses of those exceptions declared in the superclass method.

```
class Parent {
    void show() throws IOException {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    @Override
    void show() throws FileNotFoundException {  // FileNotFoundException is a subclass of IOException
        System.out.println("Child method");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        Parent obj = new Child();
        try {
            obj.show();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. What is an abstract class with a constructor, and what is the use of a constructor if we don’t instantiate the abstract class?

An abstract class can have a constructor, and it is called when a subclass object is created. The constructor is used to initialize common properties of all subclasses.

```
abstract class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
        System.out.println("Vehicle constructor called");
    }

    abstract void run();
}

class Car extends Vehicle {
    Car(String brand) {
        super(brand);
    }

    @Override
    void run() {
        System.out.println(brand + " car is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car("Toyota");
        myCar.run();  // Outputs "Vehicle constructor called" followed by "Toyota car is running"
    }
}
```

5. What are the four principles of OOP?

The four principles of OOP are:

1. **Encapsulation**: Hiding internal states and requiring all interaction to be performed through an object's methods.
2. **Abstraction**: Hiding complex implementation details and showing only the necessary features.
3. **Inheritance**: Creating new classes from existing ones to promote code reuse.
4. **Polymorphism**: Allowing entities to be represented in multiple forms.

6. What exactly is abstraction? Difference between Abstract class over a concrete class?

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. It allows focusing on what an object does rather than how it does it.

Advantages:

- Reduces complexity by hiding unnecessary details.
- Improves code maintainability.
- Promotes code reuse and cleaner code architecture.

Concrete Class vs. Abstract Class:

- Concrete classes provide full implementation and can be instantiated.
- Abstract classes provide partial implementation (or none) and cannot be instantiated, focusing on defining common characteristics.

7. Can an interface have no methods in it?

Yes, an interface can have no methods. Such an interface is called a marker interface. Examples include `Serializable` and `Cloneable`.

```
interface MarkerInterface {
}

class MyClass implements MarkerInterface {
    // Class implementation
}
```

8. How will you use encapsulation to design your code?

Encapsulation involves bundling the data (variables) and methods (functions) that operate on the data into a single unit, i.e., a class, and restricting access to some of the object's components.

```
class Account {

    private double balance;



    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;

        }

    }

}
```

9. Explain some tricks to encapsulate code?

1. **Use private access specifiers** for member variables.
2. **Provide public getter and setter methods** to access and update the private variables.
3. **Validate data in setter methods** to ensure object integrity.
4. **Hide internal implementation details** by exposing only necessary methods.
5. **Use final classes and methods** to prevent subclassing if necessary.

10. Can we have overloaded methods with different return types?

No, method overloading cannot be achieved by changing only the return type of the methods. Method overloading requires changing the method signature, i.e., the number or type of parameters.

11. What is abstraction and ways to achieve it?

Abstraction is the process of hiding the complex implementation details and showing only the essential features. It can be achieved through:

- **Abstract classes**: Using abstract methods that must be implemented by subclasses.
- **Interfaces**: Defining a contract that implementing classes must follow.

Example using abstract class:

```
abstract class Shape {

    abstract void draw();

}



class Circle extends Shape {

    @Override

    void draw() {

        System.out.println("Drawing Circle");

    }

}



class Rectangle extends Shape {
```

```
    @Override
    void draw() {
        System.out.println("Drawing Rectangle");
    }
}
```

12. Abstract class vs. interface?

- **Abstract Class**:
  - Can have both abstract and concrete methods.
  - Can have constructors.
  - Can have instance variables.
  - Supports inheritance (single inheritance).
- **Interface**:
  - Can only have abstract methods (Java 8 introduced default and static methods).
  - Cannot have constructors.
  - Cannot have instance variables, only constants.
  - Supports multiple inheritance.

13. Inheritance vs. Composition?

- **Inheritance**: A mechanism where a new class inherits properties and behavior from an existing class, promoting code reuse and establishing an "is-a" relationship.
- **Composition**: A design principle where a class is composed of one or more objects of other classes, promoting code reuse and establishing a "has-a" relationship.

Example of Inheritance:

```
class Vehicle {
    void move() {
        System.out.println("Vehicle is moving");
    }
}

class Car extends Vehicle {
```

```
    void honk() {
        System.out.println("Car is honking");
    }
}
```

Example of Composition:

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


class Car {
    private Engine engine = new Engine();

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

14. What is abstraction and how is it different from encapsulation?

Abstraction focuses on hiding complex implementation details and exposing only essential features, making it easier to manage complexity. Encapsulation involves bundling data and methods that operate on the data into a single unit and restricting access to some of the object's components to protect the object's state.

15. What are SOLID principles in Java?

The SOLID principles are five design principles intended to make software designs more understandable, flexible, and maintainable:

1. **Single Responsibility Principle**: A class should have only one reason to change.
2. **Open/Closed Principle**: Software entities should be open for extension but closed for modification.
3. **Liskov Substitution Principle**: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness.
4. **Interface Segregation Principle**: Clients should not be forced to depend on interfaces they do not use.
5. **Dependency Inversion Principle**: High-level modules should not depend on low-level modules. Both should depend on abstractions.

16. Explain about access modifiers in Java.

Access modifiers in Java determine the scope of access to classes, methods, and variables.

- **public**: Accessible from anywhere.
- **protected**: Accessible within the same package and subclasses.
- **default** (no modifier): Accessible only within the same package.
- **private**: Accessible only within the same class.

17. Create a parent P and child C class, in the child class create a list, and discuss if we can add the object of parent in that list, or if the parent class can add an object of the child class in that list?

Example:

```
import java.util.ArrayList;

import java.util.List;



class Parent {

    // Parent class implementation

}
```

```
class Child extends Parent {

    List<Parent> list = new ArrayList<>();


    void addParent(Parent p) {

        list.add(p);

    }

}


public class Main {

    public static void main(String[] args) {

        Child child = new Child();

        Parent parent = new Parent();

        child.addParent(parent);  // Yes, a parent object can be added to the list in the child class.


        // Parent class cannot add child objects directly without casting or specifying child list.

    }

}
```

18. If Parent class has methods m1, m2, m3 and Child has methods m1, m2, m3, m4, m5:

- **Can parent call methods m4, m5 of the child?** No, the parent class cannot call methods defined only in the child class unless the reference is of the child type.
- **Can the child class call methods m1, m2, m3 of the parent class?** Yes, the child class can call methods m1, m2, m3 of the parent class.

Example:

```
class Parent {
```

```
    void m1() { System.out.println("Parent m1"); }
    void m2() { System.out.println("Parent m2"); }
    void m3() { System.out.println("Parent m3"); }
}

class Child extends Parent {
    void m1() { System.out.println("Child m1"); }
    void m4() { System.out.println("Child m4"); }
    void m5() { System.out.println("Child m5"); }

    void callParentMethods() {
        super.m1();
        m2();
        m3();
    }
}

public class Main {
    public static void main(String[] args) {
        Parent parent = new Parent();
        Child child = new Child();

        child.callParentMethods();  // Calls parent methods
```

```
        // Parent reference cannot directly call child-specific methods
        // parent.m4();  // Compile-time error
    }
}
```

19. Can an interface have no methods in it?

Yes, an interface can have no methods in it. Such an interface is called a marker interface.

Example:

```
interface Serializable {
    // No methods
}


class MyClass implements Serializable {
    // Implementation of the class
}
```

20. Consider a scenario with a parent class Parent and a child class Child. In the Child class, a list is created. Can an object of the Parent class be added to that list, or can an object of the Child class be added to a list declared in the Parent class?

You can add an object of the `Parent` class to a list declared in the `Child` class (`List<Parent>` in `Child`).

You cannot directly add an object of the `Child` class to a list declared in the `Parent` class (`List<Parent>` in `Parent`), unless the list is declared to accept objects of the subclass (`List<? extends Parent>`).