1. What is OOPs?

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of objects. Objects are instances of classes that encapsulate data and behavior. OOP is designed to promote code reusability, scalability, and efficiency. Key characteristics of OOP include encapsulation, abstraction, inheritance, and polymorphism.

Example: In a banking application, you can create objects like `Account`, `Customer`, `Transaction`, each having specific attributes (data) and methods (functions or behaviors).

2. What are the key principles of OOPs?

The four key principles of OOPs are:

- **Encapsulation**: Bundling the data (attributes) and methods (functions) that operate on the data within a single unit, i.e., a class, and restricting access to some of the object's components.
- **Abstraction**: Hiding the complex implementation details and exposing only the essential features of an object.
- **Inheritance**: Creating new classes from existing ones, allowing code reuse and establishing a relationship between parent and child classes.
- **Polymorphism**: Allowing objects to be treated as instances of their parent class and enabling methods to do different things based on the object it is acting on.

3. What is a class in OOPs?

A **class** is a blueprint for creating objects. It defines a datatype by bundling data and methods that work on the data. A class encapsulates attributes and behaviors that are shared among all objects of that class.

Example:

```
public class Car {
    String make;
    String model;
    int year;

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

    void stop() {
        System.out.println("Car is stopping");
    }
}
```

4. What is an object in OOPs?

An **object** is an instance of a class. It is a real-world entity that has a state (attributes) and behavior (methods). Each object has its own copy of the attributes defined in the class.

Example:

```
Car myCar = new Car();
myCar.make = "Toyota";
myCar.model = "Camry";
myCar.year = 2020;
myCar.start();
```

5. What is inheritance in OOPs?

Inheritance is a mechanism where a new class (child class) inherits attributes and methods from an existing class (parent class). This promotes code reuse and establishes a hierarchy between classes.

Example:

```
public class Vehicle {
    String fuelType;

    void refuel() {
        System.out.println("Refueling the vehicle");
    }
}

public class Car extends Vehicle {
    int numberOfDoors;

    void openTrunk() {
        System.out.println("Opening the trunk");
    }
}
```

6. What is encapsulation in OOPs?

Encapsulation is the practice of hiding the internal state of an object and requiring all interaction to be performed through an object's methods. This protects the integrity of the data and makes the code more secure and modular.

Example:

```
public class BankAccount {
    private double balance;

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

    public double getBalance() {
        return balance;
    }
}
```

7. What is abstraction in OOPs?

Abstraction involves hiding the complex implementation details and exposing only the necessary parts of the object. This simplifies interaction with the object and reduces complexity.

Example:

```
public abstract class Shape {
    abstract void draw();
}

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

8. What is polymorphism in OOPs?

Polymorphism allows methods to do different things based on the object it is acting upon. It can be achieved through method overloading (compile-time polymorphism) or method overriding (runtime polymorphism).

Example:

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

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

Animal myAnimal = new Dog();
myAnimal.makeSound();  // Output: Dog barks
```

9. What is method overloading? Explain with an example.

Method overloading allows multiple methods to have the same name but different parameters (different type or number of parameters). This allows methods to perform similar but slightly different functions.

Example:

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

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

10. What is method overriding? Explain with an example.

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is used to implement runtime polymorphism.

Example:

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

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

11. What is a constructor?

A **constructor** is a special method that is called when an object is instantiated. It initializes the object and sets up its initial state. Constructors do not have a return type and have the same name as the class.

Example:

```
public class Car {
    String make;
    String model;
    int year;

    Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}
```

12. What are different types of constructors?

- **Default Constructor**: A constructor with no parameters. If no constructor is defined, the compiler provides a default one.
- **Parameterized Constructor**: A constructor that takes parameters to initialize the object with specific values.

- **Copy Constructor**: A constructor that creates a new object as a copy of an existing object.

13. What is a destructor?

A **destructor** is a method called automatically when an object is destroyed. It is used to clean up resources (like closing files or releasing memory). In languages like Java, garbage collection handles destruction, so explicit destructors are not needed.

14. What is a static method?

A **static method** belongs to the class rather than an instance of the class. It can be called without creating an object of the class and can only access static data members of the class.

Example:

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

// Calling the static method
int sum = MathOperations.add(5, 3);
```

15. What is a static variable?

A **static variable** is a class-level variable shared among all instances of the class. It retains its value between multiple method calls and is initialized only once.

Example:

```
public class Counter {
    static int count = 0;

    Counter() {
        count++;
    }

    public static int getCount() {
        return count;
    }
}
```

16. What is the difference between an instance variable and a class variable?

- **Instance Variable**: Belongs to an instance of a class (object), each object has its own copy.
- **Class Variable**: Also known as a static variable, it belongs to the class and is shared among all instances.

17. What is a package in Java?

A **package** is a namespace that organizes classes and interfaces, helping to avoid name conflicts and making code modular. Packages allow for better code organization and access protection.

Example:

```
package com.example;

public class MyClass {
    // Class content
}
```

18. What is the difference between abstraction and encapsulation?

- **Abstraction**: Focuses on hiding the complexity by exposing only the essential features of an object. It deals with the interface to the user.
- **Encapsulation**: Focuses on bundling the data and methods that manipulate the data and restricting access to some of the object's components. It deals with the implementation details.

19. What is the difference between inheritance and polymorphism?

- **Inheritance**: A mechanism for a new class to inherit properties and methods from an existing class, promoting code reuse and establishing a parent-child relationship.
- **Polymorphism**: A concept that allows methods to do different things based on the object it is acting upon. It enables one interface to be used for a general class of actions.

20. What is the role of the final keyword in Java?

The `final` keyword in Java can be used in three contexts:

- **Final Variable**: The value cannot be changed (constant).
- **Final Method**: The method cannot be overridden.
- **Final Class**: The class cannot be extended (no subclasses can be created).

21. Share an example that shows the difference between instance and class variables in OOPs.

```
public class Employee {
    static int numberOfEmployees; // class variable
    String name; // instance variable

    Employee(String name) {
        this.name = name;
        numberOfEmployees++;
    }

    public static void main(String[] args) {
        Employee emp1 = new Employee("John");
        Employee emp2 = new Employee("Jane");

        System.out.println(Employee.numberOfEmployees); // Output: 2
        System.out.println(emp1.name); // Output: John
        System.out.println(emp2.name); // Output: Jane
    }
}
```

22. What is the difference between private, protected, and public access modifiers in OOPs?

- **Private**: The member is accessible only within the class in which it is declared. It is not visible outside the class.
- **Protected**: The member is accessible within the same package and subclasses (even if they are in different packages).
- **Public**: The member is accessible from any other class, making it visible to all classes.