Object Oriented Programming (OOPS)

1. **Classes and Objects**

Definition: A class is a blueprint for creating objects, which are instances of the class. A class defines properties (attributes) and behaviors (methods) that the objects created from the class will have.

Example Code:

```java
class Car {
    String brand;
    String model;
    int year;

    void displayInfo() {
        System.out.println("Brand: " + brand + ", Model: " + model + ", Year: " + year);
    }
}

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

```
        car1.brand = "Toyota";
        car1.model = "Corolla";
        car1.year = 2021;
        car1.displayInfo();
    }
}
```

Explanation: Here, `Car` is a class with attributes `brand`, `model`, and `year`, and a method `displayInfo()`. `car1` is an object (instance) of the `Car` class.

2. **Constructors**

Definition: A constructor is a special method used to initialize objects. It has the same name as the class and does not have a return type. Constructors can be parameterized or non-parameterized.

Example Code:

```java
class Car {
    String brand;
    String model;
```

```
    int year;

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

    void displayInfo() {
        System.out.println("Brand: " + brand + ", Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota", "Corolla", 2021);
        car1.displayInfo();
    }
}
```

Explanation: This example uses a parameterized constructor to initialize the `Car` object with specific values.

3. **Methods (static and non-static)**

Definition: Methods define behaviors for objects. Non-static methods operate on objects, while static methods belong to the class and can be called without creating an instance.

Example Code:

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

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

public class Main {
```

```
    public static void main(String[] args) {
        // Calling static method
        int sum = MathOperations.add(5, 10);
        System.out.println("Sum: " + sum);

        // Calling non-static method
        MathOperations operations = new MathOperations();
        int product = operations.multiply(5, 10);
        System.out.println("Product: " + product);
    }
}
```

Explanation: The `add` method is static and can be called without an instance, while `multiply` is non-static and requires an object.

4. **`this` Keyword**

Definition: The `this` keyword refers to the current object. It is used to access instance variables, methods, and constructors of the current object.

Example Code:

```java
class Car {
    String brand;
    String model;
    int year;

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

    void displayInfo() {
        System.out.println("Brand: " + this.brand + ", Model: " + this.model + ", Year: " + this.year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Toyota", "Corolla", 2021);
        car1.displayInfo();
```

```
    }
}
```

Explanation: The `this` keyword is used to distinguish between instance variables and parameters with the same name.

5. **Encapsulation**

Definition: Encapsulation is the practice of bundling data (variables) and methods that operate on the data into a single unit (class) and restricting access to some of the object's components (using access modifiers).

Example Code:

```java
class Employee {
    private String name;
    private int age;

    public String getName() {
        return name;
    }
```

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

    public int getAge() {
        return age;
    }

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

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.setName("John");
        emp.setAge(30);
        System.out.println("Name: " + emp.getName());
        System.out.println("Age: " + emp.getAge());
```

```
    }
}
```

Explanation: The `Employee` class encapsulates its data using private variables and provides public getter and setter methods for access and modification.

6. **Inheritance**

Definition: Inheritance allows one class (subclass) to inherit the fields and methods of another class (superclass), enabling code reusability and the creation of hierarchical relationships.

Example Code:

```java
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
```

```
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Subclass method
    }
}
```

Explanation: The `Dog` class inherits the `eat` method from the `Animal` class, demonstrating inheritance.

7. **Polymorphism (Method Overloading and Overriding)**

Definition: Polymorphism allows objects to be treated as instances of their parent class. Method overloading allows multiple methods with the same name but different parameters, while method

overriding allows a subclass to provide a specific implementation of a method already defined in its superclass.

Example Code (Method Overloading):

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

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

public class Main {
    public static void main(String[] args) {
        MathOperations ops = new MathOperations();
        System.out.println("Sum (2 args): " + ops.add(5, 10));
        System.out.println("Sum (3 args): " + ops.add(5, 10, 15));
    }
}
```

```

## **Example Code (Method Overriding):**

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

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Polymorphism
        myAnimal.sound(); // Calls Dog's overridden method
    }
}
```
```

```

**Explanation:** The first example demonstrates method overloading, and the second demonstrates method overriding with polymorphism.

### 8. **Abstraction**

**Definition:** Abstraction is the process of hiding the implementation details and showing only the functionality to the user. In Java, abstraction is achieved using abstract classes and interfaces.

**Example Code (Abstract Class):**

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

    void sleep() {
        System.out.println("This animal sleeps.");
    }
}

class Dog extends Animal {
    void sound() {
```
```

```
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Abstract method implementation
        dog.sleep(); // Concrete method from abstract class
    }
}
```

Explanation: The `Animal` class is abstract, and the `Dog` class provides the implementation for the abstract method `sound`.

9. **Interfaces**

Definition: An interface in Java is a reference type, similar to a class, that can contain only abstract methods and final static variables. Interfaces are used to achieve abstraction and multiple inheritance in Java.

Example Code:

```java
interface Animal {
    void sound(); // Abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("The dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Interface method implementation
    }
}
```

Explanation: The `Dog` class implements the `Animal` interface and provides the implementation for the `sound` method.

10. **Packages and Access Modifiers**

Definition: Packages are namespaces that organize classes and interfaces, while access modifiers control the visibility of classes, methods, and variables. The four access modifiers in Java are `public`, `private`, `protected`,