1. Data Abstraction

Concept:

Data abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. In Java, abstraction is achieved using abstract classes and interfaces.

Example:

Let's say we have an abstract class `Shape` with an abstract method `draw()`.

abstract class Shape {

abstract void draw();

}

class Circle extends Shape {

void draw() {

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

}

}

class Rectangle extends Shape {

void draw() {

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

}

}

public class Main {

public static void main(String[] args) {

Shape circle = new Circle();

Shape rectangle = new Rectangle();

circle.draw();

rectangle.draw();

}

}

Explanation:

In this example, `Shape` is an abstract class with an abstract method `draw()`. The `Circle` and `Rectangle` classes extend the `Shape` class and provide implementations for the `draw()` method. This way, we achieve abstraction by hiding the implementation details of different shapes.

2. Encapsulation

Concept:

Encapsulation is the concept of wrapping data (variables) and code (methods) together as a single unit. In Java, encapsulation is achieved using access modifiers like private, protected, and public.

Example:

Let's create a `Person` class with private fields and public getter and setter methods.

class Person {

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) {

this.age = age;

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person();

person.setName("Alice");

person.setAge(30);

System.out.println("Name: " + person.getName());

System.out.println("Age: " + person.getAge());

}

}

Explanation:

In this example, the `Person` class has private fields `name` and `age`, and public getter and setter methods to access and modify these fields. This way, the fields are encapsulated within the class, and we can control access to them.

3. Inheritance

Concept:

Inheritance is the concept where one class inherits the properties and behaviors of another class. In Java, inheritance is achieved using the `extends` keyword.

Example:

Let's create a `Vehicle` class and a `Car` class that inherits from `Vehicle`.

class Vehicle {

void start() {

System.out.println("Vehicle started");

}

}

class Car extends Vehicle {

void honk() {

System.out.println("Car honking");

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car();

car.start();

car.honk();

}

}

Explanation:

In this example, the `Vehicle` class has a method `start()`, and the `Car` class extends `Vehicle` and adds a new method `honk()`. When we create an instance of `Car`, we can call both `start()` and `honk()` methods, demonstrating inheritance.

4. Polymorphism

Concept:

Polymorphism is the concept where a single action can be performed in different ways. In Java, polymorphism is achieved through method overloading and method overriding.

Example:

Let's demonstrate method overriding with a `Animal` class and its subclasses `Dog` and `Cat`.

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void sound() {

System.out.println("Dog barks");

}

}

class Cat extends Animal {

void sound() {

System.out.println("Cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal myDog = new Dog();

Animal myCat = new Cat();

myDog.sound();

myCat.sound();

}

}

Explanation:

In this example, the `Animal` class has a method `sound()`, which is overridden in the `Dog` and `Cat` classes. When we create instances of `Dog` and `Cat` and call the `sound()` method, it produces different outputs, demonstrating polymorphism.