Design pattern	Category	Explanation
Factory	Creational	יוצר מופע של אובייקט לפי הצורך
Singleton	Creational	יוצר מופע יחיד למחלקה
Delegation	Structural\ Behavioral	הכלה של אובייקט במחלקה אחרת במקום ירושה של אותה מחלקה
Façade	Structural	חושף למשתמש מחלקה אחת שמכילה מופעים של שאר המחלקות כך שלא ייחשף קוד (API)
Decorator	Structural	"לעטוף" אובייקט במחלקות של אובייקטים פשוטים יותר על מנת להוסיף פונקציונליות
Iterator	Behavioral	כשרוצים לעבור על כל האלמנטים של אוסף של אובייקטים מסוג מסוים
Strategy	Behavioral	מאגר של אלגוריתמים (כל אלגוריתם הוא מחלקה) שלוקח בעיה ומצמיד לה אלגוריתם מתאים
Observer	Behavioral	לתת עדכונים (למשתמשי קצה) לאובייקטים לפי צורכי המערכת
Adapter	Structural	מתאם פונקציונליות בין מחלקות שלא יודעות לעבוד טוב ביחד בלעדיו
Flyweight	Structural	בדרך כלל בשימוש ביחד עם פקטורי, מונע כפילויות של מופעים בעלי אותם פרמטרים
Builer	Creational	עוזר ליצירת אובייקטים מסובכים בעלי הרבה פרמטרים.
model view controller	Architectural patterns	מחבר מודל גרפי לממשק שלו עם התוכנה וקלט המשתמש
Composite	Structural	יוצר היררכיה בין מחלקות. מחזיק אוסף מרוכז של אוספים אחרים של מופעים של מחלקות. (בדרך כלל בנוי כמו עץ)

Interface, implements, static getIns

Factory:

```java
// Step 1: Create an interface
interface Vehicle {
    void drive();
}

// Step 2: Concrete classes
class Car implements Vehicle {
    public void drive() {
        System.out.println("Driving a car!");
    }
}

class Bike implements Vehicle {
    public void drive() {
        System.out.println("Riding a bike!");
    }
}

// Step 3: Factory class
class VehicleFactory {
    static Vehicle getVehicle(String type) {
        if (type.equalsIgnoreCase("Car")) return new Car();
        else if (type.equalsIgnoreCase("Bike")) return new Bike();
        return null;
    }
}

// Step 4: Usage
public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = VehicleFactory.getVehicle("Car");
        vehicle.drive(); // Output: Driving a car!
    }
}
```

Singleton Private (Inst + Const), Static getInst

Public Methods

```java
class Database {
    private static Database instance;

    private Database() {} // Private constructor

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

    public void connect() {
        System.out.println("Connected to database.");
    }
}

public class Main {
    public static void main(String[] args) {
        Database db1 = Database.getInstance();
        Database db2 = Database.getInstance();
        db1.connect(); // Output: Connected to database.
        System.out.println(db1 == db2); // true (same instance)
    }
}
```

Class + Method

class + private Instance + overriding Method

Delegation

```
class Printer {
    void print(String message) {
        System.out.println(message);
    }
}

class Document {
    private Printer printer = new Printer(); // Delegation

    void print(String message) {
        printer.print(message);
    }
}

public class Main {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print("Hello!"); // Output: Hello!
    }
}
```

Façade Bunch of Classes, One Big Class + Private Inst. Void Meth..

```
class CPU {
    void start() {
        System.out.println("CPU started.");
    }
}

class Memory {
    void load() {
        System.out.println("Memory loaded.");
    }
}

class HardDrive {
    void boot() {
        System.out.println("OS booted.");
    }
}

// Façade class
class Computer {
    private CPU cpu = new CPU();
    private Memory memory = new Memory();
    private HardDrive hardDrive = new HardDrive();

    void startComputer() {
        cpu.start();
        memory.load();
        hardDrive.boot();
    }
}

public class Main {
    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.startComputer();
    }
}
```

Decorator

```java
interface Coffee {
    String getDescription();
}

class SimpleCoffee implements Coffee {
    public String getDescription() {
        return "Simple coffee";
    }
}

class MilkDecorator implements Coffee {
    private Coffee coffee;

    MilkDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public String getDescription() {
        return coffee.getDescription() + ", with Milk";
    }
}

public class Main {
    public static void main(String[] args) {
        Coffee coffee = new MilkDecorator(new SimpleCoffee());
        System.out.println(coffee.getDescription()); // Output: Simple coffee, with Milk
    }
}
```

// Interface

// implementation

//added func.

Iterator

```java
class CustomIterator implements Iterator<Integer> {
    private int[] numbers;
    private int index = 0;

    CustomIterator(int[] numbers) {
        this.numbers = numbers;
    }

    public boolean hasNext() {
        return index < numbers.length;
    }

    public Integer next() {
        return numbers[index++];
    }
}

public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        Iterator<Integer> iterator = new CustomIterator(numbers);

        while (iterator.hasNext()) {
            System.out.println(iterator.next()); // Output: 1, 2, 3
        }
    }
}
```

Strategy

```java
interface PaymentStrategy {
    void pay(int amount);
}

class CreditCardPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " via Credit Card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid $" + amount + " via PayPal.");
    }
}

class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    ShoppingCart(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}

public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart(new PayPalPayment());
        cart.checkout(100); // Output: Paid $100 via PayPal.
    }
}
```

// Interface

// Private Interface obj

Observer

```java
interface Observer {
    void update(String message);
}

class Subscriber implements Observer {
    private String name;

    Subscriber(String name) {
        this.name = name;
    }

    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}

class NewsChannel {
    private List<Observer> subscribers = new ArrayList<>();

    void subscribe(Observer o) {
        subscribers.add(o);
    }

    void notifySubscribers(String news) {
        for (Observer o : subscribers) {
            o.update(news);
        }
    }
}
```

// Interface Obs
Void u

```java
public class Main {
    public static void main(String[] args) {
        NewsChannel channel = new NewsChannel();
        Subscriber s1 = new Subscriber("Alice");
        channel.subscribe(s1);
        channel.notifySubscribers("Breaking News!"); // Output: Alice received: Breaking News
    }
}
```

Adapter

```
interface USB {
    void connectWithUsbCable();
}

class USBDevice implements USB {
    public void connectWithUsbCable() {
        System.out.println("Connected via USB.");
    }
”

// Adapter to connect USB device to Type-C port
class USBToTypeCAdapter {
    private USB usbDevice;

    USBToTypeCAdapter(USB usbDevice) {
        this.usbDevice = usbDevice;
    }

    void connectWithTypeC() {
        usbDevice.connectWithUsbCable();
        System.out.println("Adapted to Type-C.");
    }
}

public class Main {
    public static void main(String[] args) {
        USB usb = new USBDevice();
        USBToTypeCAdapter adapter = new USBToTypeCAdapter(usb);
        adapter.connectWithTypeC();
    }
}
```

Flyweight

```java
class ShapeFactory {
    private static final HashMap<String, Circle> circleMap = new HashMap<>();

    static Circle getCircle(String color) {
        Circle circle = circleMap.get(color);
        if (circle == null) {
            circle = new Circle(color);
            circleMap.put(color, circle);
            System.out.println("Creating new " + color + " circle.");
        }
        return circle;
    }
}

class Circle {
    private String color;

    Circle(String color) {
        this.color = color;
    }

    void draw() {
        System.out.println("Drawing a " + color + " circle.");
    }
}

public class Main {
    public static void main(String[] args) {
        Circle redCircle1 = ShapeFactory.getCircle("Red");
        redCircle1.draw();

        Circle redCircle2 = ShapeFactory.getCircle("Red");
        redCircle2.draw(); // Same instance as redCircle1
    }
}
```

Builder

```java
class Car {
    private String engine;
    private int wheels;
    private boolean sunroof;

    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.wheels = builder.wheels;
        this.sunroof = builder.sunroof;
    }

    static class CarBuilder {
        private String engine;
        private int wheels;
        private boolean sunroof;

        CarBuilder setEngine(String engine) {
            this.engine = engine;
            return this;
        }

        CarBuilder setWheels(int wheels) {
            this.wheels = wheels;
            return this;
        }

        CarBuilder setSunroof(boolean sunroof) {
            this.sunroof = sunroof;
            return this;
        }

        Car build() {
            return new Car(this);
        }
    }
```

```java
public class Main {
    public static void main(String[] args) {
        // Create a car with only an engine specified
        Car car1 = new Car.CarBuilder()
                .setEngine("V8") // Only setting engine
                .build();

        // Create a car with engine and sunroof specified
        Car car2 = new Car.CarBuilder()
                .setEngine("Electric")
                .setSunroof(true) // Not setting wheels, so default is used
                .build();
```