1. □ Factory Pattern

[00:48] Video Explanation: The Factory Pattern allows the creation of objects without exposing the instantiation logic to the client. Instead of calling a constructor directly, you call a method that returns an instance of the class.

□ **My Notes:** Useful when you have a superclass with multiple subclasses, and based on some input or configuration, you need to return one of the subclasses.

□ Java Example:

```java
interface Animal {
    void speak();
}

class Dog implements Animal {
    public void speak() { System.out.println("Woof!"); }
}

class Cat implements Animal {
    public void speak() { System.out.println("Meow!"); }
}

class AnimalFactory {
    public static Animal getAnimal(String type) {
        if (type.equals("dog")) return new Dog();
        else if (type.equals("cat")) return new Cat();
        throw new IllegalArgumentException("Unknown animal type");
    }
}

public class FactoryDemo {
    public static void main(String[] args) {
        Animal a1 = AnimalFactory.getAnimal("dog");
        Animal a2 = AnimalFactory.getAnimal("cat");
        a1.speak(); // Woof!
        a2.speak(); // Meow!
    }
}
```

---

2. □ Builder Pattern

[01:33] Video Explanation: Builder Pattern constructs complex objects step by step. It separates the construction of an object from its representation.

□ **My Notes:** Ideal when you have a class with many optional parameters.

□ Java Example:

```java
class Pizza {
    private String dough;
    private String sauce;
    private String topping;

    public static class Builder {
        private String dough;
        private String sauce;
        private String topping;

        public Builder setDough(String d) { dough = d; return this; }
        public Builder setSauce(String s) { sauce = s; return this; }
        public Builder setTopping(String t) { topping = t; return this; }

        public Pizza build() {
            Pizza p = new Pizza();
            p.dough = this.dough;
            p.sauce = this.sauce;
            p.topping = this.topping;
            return p;
        }
    }

    public void showPizza() {
        System.out.println("Pizza with " + dough + ", " + sauce + ", " + topping);
    }
}

public class BuilderDemo {
    public static void main(String[] args) {
        Pizza p = new Pizza.Builder()
                        .setDough("Thin Crust")
                        .setSauce("Tomato")
                        .setTopping("Cheese")
                        .build();
        p.showPizza();
    }
}
```

---

3. □ Singleton Pattern

[02:24] Video Explanation: Ensures that only one instance of a class is created and provides a global point of access to it.

□ **My Notes:** Commonly used for logging, configuration, or managing connections.

□ **Java Example:**

```java
class Singleton {
    private static Singleton instance = null;

    private Singleton() {} // private constructor

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

    public void show() {
        System.out.println("Single instance object");
    }
}
```

```java
public class SingletonDemo {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        s1.show();
        System.out.println(s1 == s2); // true
    }
}
```

---

4. □ Observer Pattern (Pub-Sub)

[03:34] Video Explanation: Allows objects (observers) to subscribe to a subject. When the subject changes, all observers are notified.

□ **My Notes:** Useful in event handling systems like UI frameworks or message brokers.

□ **Java Example:**

```java
import java.util.*;

interface Observer {
    void update(String message);
}

class User implements Observer {
    private String name;
    public User(String name) { this.name = name; }
    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}

class Channel {
    private List<Observer> observers = new ArrayList<>();

    public void subscribe(Observer o) { observers.add(o); }
    public void unsubscribe(Observer o) { observers.remove(o); }

    public void notifyObservers(String message) {
        for (Observer o : observers) {
            o.update(message);
        }
    }
}

public class ObserverDemo {
    public static void main(String[] args) {
        Channel news = new Channel();
        User a = new User("Alice");
        User b = new User("Bob");

        news.subscribe(a);
        news.subscribe(b);
        news.notifyObservers("New video uploaded!");
    }
}
```

---

5. □ Iterator Pattern

[05:14] Video Explanation: Provides a way to access elements in a collection sequentially without exposing its internals.

□ **My Notes:** You use it every day in `for-each` loops!

□ **Java Example:**

```
import java.util.*;

public class IteratorDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("one", "two", "three");
        Iterator<String> it = list.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

---

6. □ Strategy Pattern

[06:26] Video Explanation: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

□ **My Notes:** Good for swapping out algorithms dynamically (e.g., sorting strategies).

□ **Java Example:**

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

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

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

class ShoppingCart {
    private PaymentStrategy strategy;
    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }
    public void checkout(int amount) {
        strategy.pay(amount);
    }
}

public class StrategyDemo {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        cart.setStrategy(new CreditCardPayment());
        cart.checkout(100);
        cart.setStrategy(new PayPalPayment());
        cart.checkout(250);
    }
}
```

---

7. □ Adapter Pattern

[07:21] Video Explanation: Allows incompatible interfaces to work together.

□ **My Notes:** Use this to connect legacy code with new systems.

□ **Java Example:**

```
interface MediaPlayer {
    void play(String audioType, String fileName);
}

class MP3Player implements MediaPlayer {
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp3"))
            System.out.println("Playing mp3 file: " + fileName);
    }
}

class VLCPlayer {
    public void playVLC(String fileName) {
        System.out.println("Playing VLC file: " + fileName);
    }
}

class MediaAdapter implements MediaPlayer {
    VLCPlayer vlc = new VLCPlayer();

    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("vlc"))
            vlc.playVLC(fileName);
    }
}

public class AdapterDemo {
    public static void main(String[] args) {
        MediaPlayer player = new MP3Player();
        player.play("mp3", "song.mp3");
        MediaPlayer adapter = new MediaAdapter();
        adapter.play("vlc", "movie.vlc");
    }
}
```

---

8. □ Facade Pattern

[08:24] Video Explanation: Provides a unified interface to a set of interfaces in a subsystem.

□ My Notes: Helps reduce complexity and dependencies from external code.

□ Java Example:

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

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

class HardDrive {
    void read() { System.out.println("Reading from hard drive"); }
}

class ComputerFacade {
    private CPU cpu = new CPU();
    private Memory memory = new Memory();
    private HardDrive hd = new HardDrive();

    public void startComputer() {
        cpu.start();
        memory.load();
        hd.read();
        System.out.println("Computer started");
    }
}

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