Creational Design Patterns

1. Factory Design Pattern

Step 1: Create a common interface

|  |
| --- |
| public interface BankAccount {  void accountType();  } |

Step 2: Implement concrete classes

|  |
| --- |
| public class SavingsAccount implements BankAccount {  @Override  public void accountType() {  System.out.println("Savings Account");  }  }  public class CurrentAccount implements BankAccount {  @Override  public void accountType() {  System.out.println("Current Account");  }  }  public class FixedDepositAccount implements BankAccount {  @Override  public void accountType() {  System.out.println("Fixed Deposit Account");  }  } |

**Step 3: Create the factory class**

|  |
| --- |
| public class BankAccountFactory {  public static BankAccount getBankAccount(String type) {  if (type == null) {  return null;  }  switch (type.toLowerCase()) {  case "savings":  return new SavingsAccount();  case "current":  return new CurrentAccount();  case "fixed":  return new FixedDepositAccount();  default:  throw new IllegalArgumentException("Unknown account type: " + type);  }  }  } |

Step 4 : **Use the factory in a client class**

|  |
| --- |
| **public class Main {**  **public static void main(String[] args) {**  **BankAccount account1 = BankAccountFactory.getBankAccount("savings");**  **account1.accountType();**  **BankAccount account2 = BankAccountFactory.getBankAccount("current");**  **account2.accountType();**  **BankAccount account3 = BankAccountFactory.getBankAccount("fixed");**  **account3.accountType();**  **}**  **}** |

1. **Abstract Factory Pattern**

The **Abstract Factory Pattern** is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It's an extension of the Factory Pattern — useful when you have multiple factories producing related products.

**💡 Scenario: Bank Accounts with Region-Specific Rules**

Let's say a bank offers different account types (Savings, Current) and the rules differ by **region** (e.g., Domestic vs. International). So, we want a factory for each region that can create both types of accounts.

**Step-by-step Abstract Factory Pattern in Java for Bank Accounts**

**Step 1. Product Interfaces**

|  |
| --- |
| **public interface SavingsAccount {**  **void create();**  **}**  **public interface CurrentAccount {**  **void create();**  **}** |

**Step 2 : Concrete Products (Domestic)**

|  |
| --- |
| **public class DomesticSavingsAccount implements SavingsAccount {**  **@Override**  **public void create() {**  **System.out.println("Domestic Savings Account created.");**  **}**  **}**  **public class DomesticCurrentAccount implements CurrentAccount {**  **@Override**  **public void create() {**  **System.out.println("Domestic Current Account created.");**  **}**  **}** |

**Step 3 : Concrete Products (International)**

|  |
| --- |
| **public class InternationalSavingsAccount implements SavingsAccount {**  **@Override**  **public void create() {**  **System.out.println("International Savings Account created.");**  **}**  **}**  **public class InternationalCurrentAccount implements CurrentAccount {**  **@Override**  **public void create() {**  **System.out.println("International Current Account created.");**  **}**  **}** |

**Step 4 : Abstract Factory Interface**

|  |
| --- |
| **public interface AccountFactory {**  **SavingsAccount createSavingsAccount();**  **CurrentAccount createCurrentAccount(); }** |

**Step 5 : Concrete Factories**

|  |
| --- |
| **public class DomesticAccountFactory implements AccountFactory {**  **public SavingsAccount createSavingsAccount() {**  **return new DomesticSavingsAccount();**  **}**  **public CurrentAccount createCurrentAccount() {**  **return new DomesticCurrentAccount();**  **}**  **}**  **public class InternationalAccountFactory implements AccountFactory {**  **public SavingsAccount createSavingsAccount() {**  **return new InternationalSavingsAccount();**  **}**  **public CurrentAccount createCurrentAccount() {**  **return new InternationalCurrentAccount();**  **}**  **}** |

**Step 6 : Client Code**

|  |
| --- |
| **public class Main {**  **public static void main(String[] args) {**  **AccountFactory domesticFactory = new DomesticAccountFactory();**  **SavingsAccount ds = domesticFactory.createSavingsAccount();**  **ds.create();**  **CurrentAccount dc = domesticFactory.createCurrentAccount();**  **dc.create();**  **AccountFactory internationalFactory = new InternationalAccountFactory();**  **SavingsAccount is = internationalFactory.createSavingsAccount();**  **is.create();**  **CurrentAccount ic = internationalFactory.createCurrentAccount();**  **ic.create();**  **}**  **}** |

1. **Builder**

**The Builder Design Pattern is a creational pattern used to construct complex objects step by step. It's particularly useful when an object needs to be created with many optional parameters or configurations.**

**Example: Bank Account with Builder Pattern**

**Let’s say a BankAccount object has both required and optional fields:**

**Step-by-Step Java Example:**

**Step 1: BankAccount Class with Builder**

|  |
| --- |
| **public class BankAccount {**  **// Required parameters**  **private final String accountNumber;**  **private final String accountHolderName;**  **// Optional parameters**  **private final String accountType;**  **private final double balance;**  **private final String branch;**  **private final boolean isSalaryAccount;**  **// Private constructor**  **private BankAccount(Builder builder) {**  **this.accountNumber = builder.accountNumber;**  **this.accountHolderName = builder.accountHolderName;**  **this.accountType = builder.accountType;**  **this.balance = builder.balance;**  **this.branch = builder.branch;**  **this.isSalaryAccount = builder.isSalaryAccount;**  **}**  **// Static nested Builder class**  **public static class Builder {**  **private final String accountNumber;**  **private final String accountHolderName;**  **private String accountType = "Savings";**  **private double balance = 0.0;**  **private String branch = "Main";**  **private boolean isSalaryAccount = false;**  **public Builder(String accountNumber, String accountHolderName) {**  **this.accountNumber = accountNumber;**  **this.accountHolderName = accountHolderName;**  **}**  **public Builder accountType(String accountType) {**  **this.accountType = accountType;**  **return this;**  **}**  **public Builder balance(double balance) {**  **this.balance = balance;**  **return this;**  **}**  **public Builder branch(String branch) {**  **this.branch = branch;**  **return this;**  **}**  **public Builder isSalaryAccount(boolean isSalaryAccount) {**  **this.isSalaryAccount = isSalaryAccount;**  **return this;**  **}**  **public BankAccount build() {**  **return new BankAccount(this);**  **}**  **}**  **public void printDetails() {**  **System.out.println("Account Number: " + accountNumber);**  **System.out.println("Account Holder: " + accountHolderName);**  **System.out.println("Account Type: " + accountType);**  **System.out.println("Balance: " + balance);**  **System.out.println("Branch: " + branch);**  **System.out.println("Salary Account: " + isSalaryAccount);**  **}**  **}** |

**Step 2 : Client Code**

|  |
| --- |
| **public class Main {**  **public static void main(String[] args) {**  **BankAccount account1 = new BankAccount.Builder("1234567890", "Alice")**  **.accountType("Current")**  **.balance(10000.0)**  **.branch("Downtown")**  **.isSalaryAccount(true)**  **.build();**  **BankAccount account2 = new BankAccount.Builder("9876543210", "Bob")**  **.balance(5000.0)**  **.build();**  **account1.printDetails();**  **System.out.println("-----");**  **account2.printDetails();**  **}**  **}** |

**Sample Output**

|  |
| --- |
| **Account Number: 1234567890**  **Account Holder: Alice**  **Account Type: Current**  **Balance: 10000.0**  **Branch: Downtown**  **Salary Account: true**  **-----**  **Account Number: 9876543210**  **Account Holder: Bob**  **Account Type: Savings**  **Balance: 5000.0**  **Branch: Main**  **Salary Account: false** |

1. **Prototype**

**Banking Example: Cloning a Bank Account Template**

Suppose a bank wants to quickly create new bank accounts based on predefined templates (e.g., Standard Savings Account, Premium Account). Instead of reinitializing everything each time, the system clones a "prototype" account.

**Step-by-Step Java Example**

**Step 1 : Step 1: BankAccount Class implements Cloneable**

|  |
| --- |
| **public class BankAccount implements Cloneable {**  **private String accountHolder;**  **private String accountType;**  **private String branch;**  **private double balance;**  **public BankAccount(String accountHolder, String accountType, String branch, double balance) {**  **this.accountHolder = accountHolder;**  **this.accountType = accountType;**  **this.branch = branch;**  **this.balance = balance;**  **}**  **// Getters and Setters**  **public void setAccountHolder(String accountHolder) {**  **this.accountHolder = accountHolder;**  **}**  **public void setBalance(double balance) {**  **this.balance = balance;**  **}**  **@Override**  **public BankAccount clone() {**  **try {**  **return (BankAccount) super.clone();**  **} catch (CloneNotSupportedException e) {**  **throw new AssertionError("Cloning not supported", e);**  **}**  **}**  **public void printDetails() {**  **System.out.println("Account Holder: " + accountHolder);**  **System.out.println("Account Type: " + accountType);**  **System.out.println("Branch: " + branch);**  **System.out.println("Balance: " + balance);**  **System.out.println("--------");**  **}**  **}** |

**Step 2: Use Prototype in Client Code**

|  |
| --- |
| **public class Main {**  **public static void main(String[] args) {**  **// Create a prototype for a Standard Savings Account**  **BankAccount savingsPrototype = new BankAccount("Template", "Savings", "Main", 0.0);**  **// Clone for a real customer**  **BankAccount aliceAccount = savingsPrototype.clone();**  **aliceAccount.setAccountHolder("Alice");**  **aliceAccount.setBalance(5000.0);**  **BankAccount bobAccount = savingsPrototype.clone();**  **bobAccount.setAccountHolder("Bob");**  **bobAccount.setBalance(10000.0);**  **// Print details**  **aliceAccount.printDetails();**  **bobAccount.printDetails();**  **}**  **}** |

**Sample Output**

|  |
| --- |
| **Account Holder: Alice**  **Account Type: Savings**  **Branch: Main**  **Balance: 5000.0**  **--------**  **Account Holder: Bob**  **Account Type: Savings**  **Branch: Main**  **Balance: 10000.0**  **--------** |

**Structural Design patterns**

1. **Adapter Design Pattern**

**The Adapter design pattern is a structural design pattern that allows two incompatible interfaces to work together. It acts as a bridge between two interfaces, making them compatible and enabling them to work seamlessly. This pattern is useful when you need to integrate existing code or libraries that have different interfaces with your codebase without making significant modifications.**

**Step 1 : Define a Common Interface**

**The first step in implementing the Adapter pattern is to define a common interface that**

**Your application understands, in our case, we will create a Payment Gateway Interface.**

|  |
| --- |
| **public interface PaymentGateway {  void processPayment(double amount); }** |

**This interface defines a single method, processPayment, which is responsible for processing payments.**

**Step 2: Create Payment Gateway Adapters**

**Next, we’ll create adapter classes for each specific payment gateway provider (HyperPay and Checkout) to make them compatible with our PaymentGateway interface. Let's start with the HyperPay adapter:**

**HyperPay Adapter:**

|  |
| --- |
| **Public class HyperPayAdapter implements PaymentGateway {  private HyperPayPaymentGateway hyperPaypaymentGateway;   public HyperPayAdapter (HyperPayPaymentGateway hyperPaypaymentGateway) {  this. hyperPaypaymentGateway = hyperPaypaymentGateway;  }   @Override  public void processPayment(double amount) {  // Convert our application's method to HyperPaypaymentGateway method  hyperPaypaymentGateway.makePayment(amount);  } }** |

**Checkout Adapter:**

**Similarly, we create an adapter for the Checkout payment gateway:**

|  |
| --- |
| **public class CheckoutAdapter implements PaymentGateway {  private CheckoutPaymentGateway checkoutPaymentGateway;   public CheckoutAdapter (CheckoutPaymentGateway checkoutPaymentGateway) {  this. checkoutPaymentGateway = checkoutPaymentGateway;  }   @Override  public void processPayment(double amount) {  // Convert our application's method to checkoutPaymentGateway method  checkoutPaymentGateway.charge(amount);  } }** |

**These adapter classes take the specific payment gateway instances as constructor parameters and implement the processPayment method by converting the method calls to the corresponding methods of the payment gateway providers.**

**Step 3: Implement Concrete Payment Gateway Providers**

**In our example, we’ll create hypothetical implementations of the PayPal and Stripe payment gateways.**

**HyperPay Implementation**

|  |
| --- |
| **public class HyperPay {  public void makePayment(double amount) {  // HyperPay-specific payment processing logic  System.out.println("Paid $" + amount + " via Hyperpay.");  } }** |

**Checkout Implemtation**

|  |
| --- |
| **public class CheckoutPaymentGateway {  public void charge(double amount) {  // Stripe-specific payment processing logic  System.out.println("Charged $" + amount + " using Checkout.");  } }** |

**These classes represent the specific payment gateway providers and define their unique payment processing logic.**

**Step 4: Client Code**

**Now that we have defined the common interface (PaymentGateway), created adapter classes for payment gateways, and implemented concrete payment gateway providers, we can use these components in our client code:**

|  |
| --- |
| **public class PaymentApp {  public static void main(String[] args) {  PaymentGateway hyperPayGateway = new HyperPayAdapter(new HyperPayPaymentGateway ());  PaymentGateway checkoutGateway = new checkoutAdapter(new CheckoutPaymentGateway());   double amount = 100.0;   // Process payments using different payment gateways  hyperPayGateway.processPayment(amount);  checkoutGateway.processPayment(amount);  } }** |

1. **Bridge Pattern**

**Decouples an abstraction from its implementation so that the two can vary independently.**

**Key Components**

| **Component** | **Java Representation** | **Purpose** |
| --- | --- | --- |
| **Abstraction** | **Abstract class/interface** | **Defines the high-level control logic** |
| **RefinedAbstraction** | **Extends Abstraction** | **Adds additional behavior or customization** |
| **Implementor** | **Interface** | **Defines low-level implementation operations** |
| **ConcreteImplementor** | **Implements Implementor** | **Provides specific implementation** |

**Step-by-Step Implementation**

1. **Implementor Interface – MessageSender**

|  |
| --- |
| **public interface MessageSender {**  **void sendMessage(String message);**  **}** |

1. **Concrete Implementors – Different sending platforms**

**EmailSender**

|  |
| --- |
| **public class EmailSender implements MessageSender {**  **@Override**  **public void sendMessage(String message) {**  **System.out.println("Email sent: " + message);**  **}**  **}** |

**SMSSender**

|  |
| --- |
| **public class SMSSender implements MessageSender {**  **@Override**  **public void sendMessage(String message) {**  **System.out.println("SMS sent: " + message);**  **}**  **}** |

1. **Abstraction – Message**

|  |
| --- |
| **public abstract class Message {**  **protected MessageSender sender;**  **public Message(MessageSender sender) {**  **this.sender = sender;**  **}**  **public abstract void send(String message);**  **}** |

1. **Refined Abstractions – Types of messages**

**UrgentMessage**

|  |
| --- |
| **public class UrgentMessage extends Message {**  **public UrgentMessage(MessageSender sender) {**  **super(sender);**  **}**  **@Override**  **public void send(String message) {**  **sender.sendMessage("[URGENT] " + message);**  **}**  **}** |

**NormalMessage**

|  |
| --- |
| **public class NormalMessage extends Message {**  **public NormalMessage(MessageSender sender) {**  **super(sender);**  **}**  **@Override**  **public void send(String message) {**  **sender.sendMessage("[Normal] " + message);**  **}**  **}** |

1. **Client Code**

|  |
| --- |
| **public class Main {**  **public static void main(String[] args) {**  **MessageSender email = new EmailSender();**  **MessageSender sms = new SMSSender();**  **Message urgentEmail = new UrgentMessage(email);**  **Message normalSMS = new NormalMessage(sms);**  **urgentEmail.send("System is down!");**  **normalSMS.send("Your order has shipped.");**  **}**  **}** |

Decorator Design Pattern – eCommerce Order Example

Decorator Design Pattern is a structural pattern that allows you to add new behavior to an object dynamically without altering its structure. In the context of an eCommerce system, it can be used to add additional features to an order, such as gift wrapping, express shipping, or discounts, without modifying the base order class.

# 1. Component Interface – Order

public interface Order {  
 double getPrice(); // Returns the total price of the order  
}

# 2. Concrete Component – BasicOrder

public class BasicOrder implements Order {  
 @Override  
 public double getPrice() {  
 return 100.0; // Base price for the order  
 }  
}

# 3. Decorator Class – OrderDecorator

public abstract class OrderDecorator implements Order {  
 protected Order decoratedOrder; // Holds a reference to the order component  
  
 public OrderDecorator(Order order) {  
 this.decoratedOrder = order;  
 }  
  
 @Override  
 public double getPrice() {  
 return decoratedOrder.getPrice(); // Delegate the price calculation to the wrapped order  
 }  
}

# 4. Concrete Decorators – Additional Features for Order

## GiftWrapDecorator

public class GiftWrapDecorator extends OrderDecorator {  
 public GiftWrapDecorator(Order order) {  
 super(order);  
 }  
  
 @Override  
 public double getPrice() {  
 return decoratedOrder.getPrice() + 5.0; // Add cost for gift wrapping  
 }  
}

## ExpressShippingDecorator

public class ExpressShippingDecorator extends OrderDecorator {  
 public ExpressShippingDecorator(Order order) {  
 super(order);  
 }  
  
 @Override  
 public double getPrice() {  
 return decoratedOrder.getPrice() + 20.0; // Add cost for express shipping  
 }  
}

## DiscountDecorator

public class DiscountDecorator extends OrderDecorator {  
 private double discountPercentage;  
  
 public DiscountDecorator(Order order, double discountPercentage) {  
 super(order);  
 this.discountPercentage = discountPercentage;  
 }  
  
 @Override  
 public double getPrice() {  
 double price = decoratedOrder.getPrice();  
 return price - (price \* discountPercentage / 100); // Apply discount  
 }  
}

# 5. Client Code

public class Main {  
 public static void main(String[] args) {  
 Order order = new BasicOrder();  
 System.out.println("Base Order Price: $" + order.getPrice());  
  
 // Add gift wrapping  
 order = new GiftWrapDecorator(order);  
 System.out.println("Order Price with Gift Wrapping: $" + order.getPrice());  
  
 // Add express shipping  
 order = new ExpressShippingDecorator(order);  
 System.out.println("Order Price with Express Shipping: $" + order.getPrice());  
  
 // Add discount  
 order = new DiscountDecorator(order, 10);  
 System.out.println("Order Price with 10% Discount: $" + order.getPrice());  
 }  
}

# Output:

Base Order Price: $100.0  
Order Price with Gift Wrapping: $105.0  
Order Price with Express Shipping: $125.0  
Order Price with 10% Discount: $112.5

# Summary

In this example, the Decorator Design Pattern allows us to dynamically add features like gift wrapping, express shipping, and discounts to a basic order object. By using decorators, we can avoid subclassing the `BasicOrder` class for each combination of features, making the system more flexible and maintainable.

Proxy Design Pattern – Product Inventory Check (eCommerce)

# Use Case:

In an eCommerce system, checking real-time stock (e.g., from a warehouse or third-party system) can be slow or expensive. The Proxy acts as an intermediary to:  
- Cache product stock  
- Reduce API calls  
- Block unavailable products from checkout

# 1. Subject Interface – InventoryService

public interface InventoryService {  
 int getStock(String productId);  
}

# 2. RealSubject – RealInventoryService

public class RealInventoryService implements InventoryService {  
 @Override  
 public int getStock(String productId) {  
 System.out.println("Fetching real-time stock for: " + productId);  
 // Simulate external system call  
 return 5; // Always return 5 units in stock for demo  
 }  
}

# 3. Proxy – InventoryServiceProxy

import java.util.HashMap;  
import java.util.Map;  
  
public class InventoryServiceProxy implements InventoryService {  
 private RealInventoryService realInventoryService;  
 private Map<String, Integer> cache = new HashMap<>();  
  
 public InventoryServiceProxy() {  
 this.realInventoryService = new RealInventoryService();  
 }  
  
 @Override  
 public int getStock(String productId) {  
 if (cache.containsKey(productId)) {  
 System.out.println("Returning cached stock for: " + productId);  
 return cache.get(productId);  
 }  
  
 int stock = realInventoryService.getStock(productId);  
 cache.put(productId, stock);  
 return stock;  
 }  
}

# 4. Client Code

public class Main {  
 public static void main(String[] args) {  
 InventoryService inventoryService = new InventoryServiceProxy();  
  
 System.out.println("Stock for PROD1: " + inventoryService.getStock("PROD1"));  
 System.out.println("Stock for PROD1: " + inventoryService.getStock("PROD1"));  
 System.out.println("Stock for PROD2: " + inventoryService.getStock("PROD2"));  
 }  
}

# Output:

Fetching real-time stock for: PROD1  
Stock for PROD1: 5  
  
Returning cached stock for: PROD1  
Stock for PROD1: 5  
  
Fetching real-time stock for: PROD2  
Stock for PROD2: 5

# Why Use Proxy Here?

| Component | Role |  
|-----------------------|----------------------------------------------------------------------|  
| InventoryService | Defines the method to get stock |  
| RealInventoryService | Calls a real-time backend/warehouse API |  
| InventoryServiceProxy | Adds caching to reduce external API calls |

# Benefits:

- ✅ Performance Boost – Prevents frequent hits to the slow backend API  
- ✅ Saves Cost – Reduces paid API usage  
- ✅ Clean Design – Keeps logic separated from external system management

Chain of Responsibility Design Pattern – Payment Processing Example (Java)

# Use Case:

In an eCommerce system, when a customer tries to pay, we might want to try different payment methods (like Wallet, Credit Card, or Bank Account) in sequence until one succeeds. The Chain of Responsibility pattern allows passing the request along a chain of handlers.

# 1. Handler Interface – PaymentHandler

public abstract class PaymentHandler {  
 protected PaymentHandler next;  
  
 public void setNext(PaymentHandler next) {  
 this.next = next;  
 }  
  
 public abstract void handlePayment(double amount);  
}

# 2. Concrete Handlers

* WalletPaymentHandler:

public class WalletPaymentHandler extends PaymentHandler {  
 private double walletBalance = 100.0;  
  
 @Override  
 public void handlePayment(double amount) {  
 if (walletBalance >= amount) {  
 System.out.println("Paid $" + amount + " using Wallet.");  
 } else {  
 System.out.println("Insufficient Wallet balance. Passing to next handler.");  
 if (next != null) next.handlePayment(amount);  
 }  
 }  
}

* CreditCardPaymentHandler:

public class CreditCardPaymentHandler extends PaymentHandler {  
 private double creditLimit = 500.0;  
  
 @Override  
 public void handlePayment(double amount) {  
 if (creditLimit >= amount) {  
 System.out.println("Paid $" + amount + " using Credit Card.");  
 } else {  
 System.out.println("Insufficient Credit Card limit. Passing to next handler.");  
 if (next != null) next.handlePayment(amount);  
 }  
 }  
}

* BankPaymentHandler:

public class BankPaymentHandler extends PaymentHandler {  
 private double bankBalance = 1000.0;  
  
 @Override  
 public void handlePayment(double amount) {  
 if (bankBalance >= amount) {  
 System.out.println("Paid $" + amount + " using Bank Account.");  
 } else {  
 System.out.println("Insufficient Bank balance. Payment failed.");  
 }  
 }  
}

# 3. Client Code

public class Main {  
 public static void main(String[] args) {  
 PaymentHandler wallet = new WalletPaymentHandler();  
 PaymentHandler creditCard = new CreditCardPaymentHandler();  
 PaymentHandler bank = new BankPaymentHandler();  
  
 // Setup the chain: Wallet → Credit Card → Bank  
 wallet.setNext(creditCard);  
 creditCard.setNext(bank);  
  
 // Try to pay $120 (Wallet has only $100)  
 System.out.println("Attempting to pay $120...");  
 wallet.handlePayment(120.0);  
 }  
}

# Output:

Insufficient Wallet balance. Passing to next handler.  
Paid $120.0 using Credit Card.

# Why Use Chain of Responsibility?

| Component | Responsibility |  
|--------------------------|---------------------------------------------------------------|  
| PaymentHandler | Defines a method and reference to the next handler |  
| WalletPaymentHandler | Tries to handle the request using wallet funds |  
| CreditCardPaymentHandler | Tries credit card if wallet fails |  
| BankPaymentHandler | Fallback to bank account if others fail |

# Benefits:

- ✅ Loose coupling between sender and receiver  
- ✅ Flexible order of processing  
- ✅ Simplifies control flow logic

Command Design Pattern – E-Commerce Order Example (Java)

# Use Case:

An eCommerce platform might want to encapsulate order actions (like PlaceOrder, CancelOrder, TrackOrder) as command objects so they can be executed, logged, queued, or undone independently.

# 1. Command Interface – OrderCommand

public interface OrderCommand {  
 void execute();  
}

# 2. Receiver – Order

public class Order {  
 public void place() {  
 System.out.println("Order has been placed.");  
 }  
  
 public void cancel() {  
 System.out.println("Order has been canceled.");  
 }  
  
 public void track() {  
 System.out.println("Tracking order...");  
 }  
}

# 3. Concrete Commands

* PlaceOrderCommand:

public class PlaceOrderCommand implements OrderCommand {  
 private Order order;  
  
 public PlaceOrderCommand(Order order) {  
 this.order = order;  
 }  
  
 @Override  
 public void execute() {  
 order.place();  
 }  
}

* CancelOrderCommand:

public class CancelOrderCommand implements OrderCommand {  
 private Order order;  
  
 public CancelOrderCommand(Order order) {  
 this.order = order;  
 }  
  
 @Override  
 public void execute() {  
 order.cancel();  
 }  
}

* TrackOrderCommand:

public class TrackOrderCommand implements OrderCommand {  
 private Order order;  
  
 public TrackOrderCommand(Order order) {  
 this.order = order;  
 }  
  
 @Override  
 public void execute() {  
 order.track();  
 }  
}

# 4. Invoker – OrderInvoker

public class OrderInvoker {  
 private OrderCommand command;  
  
 public void setCommand(OrderCommand command) {  
 this.command = command;  
 }  
  
 public void executeCommand() {  
 command.execute();  
 }  
}

# 5. Client Code

public class Main {  
 public static void main(String[] args) {  
 Order order = new Order();  
  
 OrderCommand place = new PlaceOrderCommand(order);  
 OrderCommand cancel = new CancelOrderCommand(order);  
 OrderCommand track = new TrackOrderCommand(order);  
  
 OrderInvoker invoker = new OrderInvoker();  
  
 invoker.setCommand(place);  
 invoker.executeCommand();  
  
 invoker.setCommand(track);  
 invoker.executeCommand();  
  
 invoker.setCommand(cancel);  
 invoker.executeCommand();  
 }  
}

# Output:

Order has been placed.  
Tracking order...  
Order has been canceled.

# Why Use the Command Pattern?

| Component | Role |  
|--------------------------|-----------------------------------------------------|  
| OrderCommand | Common interface for commands |  
| PlaceOrderCommand, etc. | Encapsulate actions as objects |  
| Order | The receiver that knows how to perform the action |  
| OrderInvoker | Calls the command without knowing what it does |

# Benefits:

- ✅ Decouples objects that invoke operations from those that perform them  
- ✅ Allows for queuing, logging, or undoing operations  
- ✅ Improves extensibility and separation of concerns

**🔁 Iterator Design Pattern – Java Example (Shopping Cart)**

**✅ Use Case:**

You want to **traverse through a collection** (like items in a shopping cart) without exposing the internal structure of the collection.

**🔹 1. Iterator Interface**

java

CopyEdit

public interface CartIterator {

boolean hasNext();

Item next();

}

**🔹 2. Item Class (Cart Item)**

java

CopyEdit

public class Item {

private String name;

private double price;

public Item(String name, double price) {

this.name = name;

this.price = price;

}

public String getName() { return name; }

public double getPrice() { return price; }

}

**🔹 3. Aggregate Interface**

java

CopyEdit

public interface Cart {

CartIterator createIterator();

}

**🔹 4. Concrete Aggregate – ShoppingCart**

java

CopyEdit

import java.util.ArrayList;

import java.util.List;

public class ShoppingCart implements Cart {

private List<Item> items = new ArrayList<>();

public void addItem(Item item) {

items.add(item);

}

@Override

public CartIterator createIterator() {

return new ShoppingCartIterator(items);

}

}

**🔹 5. Concrete Iterator – ShoppingCartIterator**

java

CopyEdit

import java.util.List;

public class ShoppingCartIterator implements CartIterator {

private List<Item> items;

private int position = 0;

public ShoppingCartIterator(List<Item> items) {

this.items = items;

}

@Override

public boolean hasNext() {

return position < items.size();

}

@Override

public Item next() {

return items.get(position++);

}

}

**🔹 6. Client Code**

java

CopyEdit

public class Main {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

cart.addItem(new Item("Laptop", 1500));

cart.addItem(new Item("Phone", 800));

cart.addItem(new Item("Mouse", 25));

CartIterator iterator = cart.createIterator();

while (iterator.hasNext()) {

Item item = iterator.next();

System.out.println("Item: " + item.getName() + ", Price: $" + item.getPrice());

}

}

}

**🟢 Output:**

yaml

CopyEdit

Item: Laptop, Price: $1500.0

Item: Phone, Price: $800.0

Item: Mouse, Price: $25.0

**🧠 Why Use the Iterator Pattern?**

| **Component** | **Role** |
| --- | --- |
| CartIterator | Interface for traversing a collection |
| ShoppingCart | Aggregate holding collection of items |
| ShoppingCartIterator | Iterator that implements traversal |
| Main | Uses the iterator to access items cleanly |

**✅ Benefits:**

* Hides internal structure of collections
* Supports multiple traversals
* Cleaner and more modular traversal logic

**Observer Design Pattern – E-Commerce Order Example (Java)**

**Use Case:**

You need to notify multiple parts of your system (observers) whenever there is a change in the order status (subject).

**1. Observer Interface**

public interface Observer {

void update (String orderStatus);

}

**2. Concrete Observer – Customer**

java

CopyEdit

public class Customer implements Observer {

private String name;

public Customer(String name) {

this.name = name;

}

@Override

public void update(String orderStatus) {

System.out.println(name + " has been notified. Order status: " + orderStatus);

}

}

**3. Concrete Observer – ShippingSystem**

java

CopyEdit

public class ShippingSystem implements Observer {

@Override

public void update(String orderStatus) {

System.out.println("Shipping System updated with Order status: " + orderStatus);

}

}

**4. Subject Interface**

java

CopyEdit

public interface Subject {

void addObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

**5. Concrete Subject – Order**

java

CopyEdit

import java.util.ArrayList;

import java.util.List;

public class Order implements Subject {

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

private String orderStatus;

public void setOrderStatus(String status) {

this.orderStatus = status;

notifyObservers();

}

@Override

public void addObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(orderStatus);

}

}

}

**6. Client Code**

java

CopyEdit

public class Main {

public static void main(String[] args) {

// Create observers

Customer customer = new Customer("John Doe");

ShippingSystem shippingSystem = new ShippingSystem();

// Create subject (Order)

Order order = new Order();

// Add observers to the order

order.addObserver(customer);

order.addObserver(shippingSystem);

// Change order status, which will notify observers

order.setOrderStatus("Order Placed");

order.setOrderStatus("Shipped");

order.setOrderStatus("Delivered");

}

}

**Output:**

sql

CopyEdit

John Doe has been notified. Order status: Order Placed

Shipping System updated with Order status: Order Placed

John Doe has been notified. Order status: Shipped

Shipping System updated with Order status: Shipped

John Doe has been notified. Order status: Delivered

Shipping System updated with Order status: Delivered

**Why Use the Observer Pattern?**

| **Component** | **Role** |
| --- | --- |
| **Observer** | Interface for classes that will receive updates |
| **ConcreteObserver** | Classes that implement the Observer interface (like Customer, ShippingSystem) |
| **Subject** | Interface for adding/removing observers and notifying them |
| **ConcreteSubject** | Class that maintains state and notifies observers (like Order) |

**Benefits:**

* ✅ Loose coupling between subject and observers.
* ✅ Easy to add new observers without modifying the subject.
* ✅ Supports dynamic changes in the system.

**Strategy Design Pattern**

**Here’s an example of the Strategy Design Pattern in Java, which allows a class to choose an algorithm at runtime.**

**Scenario: Payment System**

**In this example, we have a payment processing system where a user can select different payment methods (Credit Card, PayPal, etc.) at runtime. The strategy pattern allows us to switch between these payment methods without changing the context.**

**1. PaymentStrategy Interface**

**public interface PaymentStrategy {**

**void pay(int amount);**

**}**

**This interface declares a method pay() that will be implemented by all concrete payment methods.**

**2. Concrete Payment Methods (Strategies)**

**CreditCardPayment**

**public class CreditCardPayment implements PaymentStrategy {**

**private String cardNumber;**

**public CreditCardPayment(String cardNumber) {**

**this.cardNumber = cardNumber;**

**}**

**@Override**

**public void pay(int amount) {**

**System.out.println("Paid " + amount + " using Credit Card: " + cardNumber);**

**}**

**}**

**PayPalPayment**

**public class PayPalPayment implements PaymentStrategy {**

**private String email;**

**public PayPalPayment(String email) {**

**this.email = email;**

**}**

**@Override**

**public void pay(int amount) {**

**System.out.println("Paid " + amount + " using PayPal: " + email);**

**}**

**}**

**BitcoinPayment**

**public class BitcoinPayment implements PaymentStrategy {**

**private String bitcoinAddress;**

**public BitcoinPayment(String bitcoinAddress) {**

**this.bitcoinAddress = bitcoinAddress;**

**}**

**@Override**

**public void pay(int amount) {**

**System.out.println("Paid " + amount + " using Bitcoin: " + bitcoinAddress);**

**}**

**}**

**3. Context Class (PaymentProcessor)**

**The PaymentProcessor class uses the PaymentStrategy interface. It can switch between different payment methods dynamically.**

**java**

**CopyEdit**

**public class PaymentProcessor {**

**private PaymentStrategy paymentStrategy;**

**// Set payment method dynamically**

**public void setPaymentStrategy(PaymentStrategy paymentStrategy) {**

**this.paymentStrategy = paymentStrategy;**

**}**

**// Process payment using the current payment strategy**

**public void processPayment(int amount) {**

**paymentStrategy.pay(amount);**

**}**

**}**

**4. Client Code**

**Here’s how the client can use the PaymentProcessor and switch between different payment methods:**

**java**

**CopyEdit**

**public class Main {**

**public static void main(String[] args) {**

**// Create a PaymentProcessor**

**PaymentProcessor paymentProcessor = new PaymentProcessor();**

**// Set payment strategy to CreditCardPayment**

**paymentProcessor.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));**

**paymentProcessor.processPayment(100);**

**// Switch to PayPalPayment**

**paymentProcessor.setPaymentStrategy(new PayPalPayment("user@example.com"));**

**paymentProcessor.processPayment(200);**

**// Switch to BitcoinPayment**

**paymentProcessor.setPaymentStrategy(new BitcoinPayment("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa"));**

**paymentProcessor.processPayment(50);**

**}**

**}**

**Output:**

**sql**

**CopyEdit**

**Paid 100 using Credit Card: 1234-5678-9876-5432**

**Paid 200 using PayPal: user@example.com**

**Paid 50 using Bitcoin: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa**

**Explanation:**

* **PaymentStrategy: This is the interface that defines the pay() method for all concrete strategies.**
* **Concrete Strategies (CreditCardPayment, PayPalPayment, BitcoinPayment): These classes implement the PaymentStrategy interface and provide specific implementations for how payments are processed.**
* **PaymentProcessor: This class acts as the context. It holds a reference to a PaymentStrategy and can switch strategies dynamically using the setPaymentStrategy() method.**
* **Main: The client code demonstrates how the payment strategy can be changed at runtime.**

**Benefits of the Strategy Pattern:**

* **Flexibility: You can add new strategies (payment methods) without changing the context (PaymentProcessor).**
* **Loose Coupling: The context (PaymentProcessor) does not need to know about the specific implementation of payment methods.**
* **Open/Closed Principle: You can extend the functionality by adding new payment methods (strategies) without modifying the existing code.**