Contents

Introduction

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects and data rather than actions and logic. It emphasizes the concept of "objects," which are instances of classes, and allows for the encapsulation, abstraction, inheritance, and polymorphism of data and behavior. OOP facilitates modular, reusable, and maintainable code by modeling real-world entities as objects with properties (attributes) and behaviors (methods), enabling easier understanding, design, and development of complex systems.

Design principles

SOLID principles & The Theory

Single Responsibility Principle (SRP)

- **Description:** A class should have only one reason to change.
- **Example:** Separating data access logic from business logic.

Open/Closed Principle (OCP)

- **Description:** Objects should be open for extension but closed for modification.
- **Example:** Using interfaces and abstract classes to allow for extension without modifying existing code.

Liskov Substitution Principle (LSP)

- **Description:** Subtypes must be substitutable for their base types.
- **Example:** Ensuring that derived classes can be used interchangeably with their base class without affecting the program's behavior.

Interface Segregation Principle (ISP)

- **Description:** Clients should not be forced to depend on interfaces they do not use.
- **Example:** Breaking down large interfaces into smaller, more specific interfaces to avoid imposing unnecessary dependencies.

Dependency Inversion Principle (DIP)

- **Description:** the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes.
- **Example:** Using dependency injection to decouple components and promote loose coupling.

SOLID Principles & The Real-World

Single Responsibility Principle (SRP)

- **Problem:** An application has a class responsible for both logging user activities and processing user data.
- **Solution:** Separate the logging functionality into its own class, creating two classes: one for processing user data and another for logging user activities.

Open/Closed Principle (OCP)

- **Problem:** An application has a class responsible for calculating various types of discounts, but new types of discounts keep getting added, requiring modification of the existing class.
- **Solution:** Create an abstract class or interface for discounts and implement specific discount types as subclasses. This allows adding new discount types without modifying the existing class.

Liskov Substitution Principle (LSP)

- **Problem:** An application expects all shapes to have an area method, but the behavior of the area method for some shapes (e.g., circles) cannot be accurately represented by a generic area method.
- **Solution:** Ensure that derived classes (e.g., Circle, Square) can be substituted for their base class (e.g., Shape) without altering the correctness of the program's behavior.

Interface Segregation Principle (ISP)

- **Problem:** An interface for a reporting service includes methods for generating multiple types of reports, but not all clients need all types of reports, leading to unnecessary dependencies.
- **Solution:** Split the large interface into smaller, more specific interfaces (e.g., IExcelReportGenerator, IPDFReportGenerator) to avoid forcing clients to depend on methods they don't use.

Dependency Inversion Principle (DIP)

- **Problem:** An application has tightly coupled components, making it difficult to replace or extend one component without affecting others.
- **Solution:** Use dependency injection to invert the dependencies, allowing high-level modules to depend on abstractions (e.g., interfaces) rather than concrete implementations, promoting loose coupling and easier maintenance.

Other principles

Encapsulation

Encapsulation refers to the bundling of data with the methods that operate on that data, or the restriction of access to some of an object's components.

Abstraction

Abstraction is the concept of representing essential features without including the background details.

Inheritance

Inheritance is a mechanism in which a new class is derived from an existing class. It promotes code reusability and establishes a relationship between different classes.

Polymorphism

Polymorphism allows methods to perform different tasks based on the object that calls them. It is the ability of an object to take on many forms.

Composition

Favor composition over inheritance to achieve code reuse and flexibility in design. This principle suggests that you should prefer creating classes that are composed of other classes or modules rather than inheriting from them.

More principles

Clean Code Principles (CCp)

These principles include concepts such as meaningful names, small functions, single responsibility, minimal comments, and others that contribute to writing high-quality code.Cohesion and Coupling

Cohesion and Coupling (CaC)

Cohesion refers to the degree to which elements within a module or class belong together. High cohesion indicates that the elements of a module are closely related and work together to achieve a single purpose.

Coupling refers to the degree of interdependence between modules or classes. Loose coupling means that modules are relatively independent of each other, which promotes modularity and easier maintenance.

Inversion of Control (IoC)

Inversion of Control (IoC) is a design principle where the control of object creation and flow is inverted from the application code to a framework or container. This promotes loose coupling and allows for better testability and scalability.

Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development process where tests are written before the actual implementation code. The development cycle involves writing a failing test, writing the code to make the test pass, and then refactoring the code while ensuring that all tests still pass. TDD promotes a more iterative and incremental approach to development, leading to code that is more robust, maintainable, and thoroughly tested.

Domain-Driven Design (DDD)

Domain-Driven Design (DDD) is an approach to software development that focuses on understanding and modeling the domain of the problem at hand. It emphasizes close collaboration between domain experts and developers to create a shared understanding of the problem domain. DDD promotes the use of a common language (ubiquitous language) to bridge the communication gap between technical and non-technical stakeholders, leading to better software solutions.

Design patterns

1. **Singleton Pattern** Ensures that a class has only one instance and provides a global point of access to that instance.

2. **Factory Method Pattern** Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
3. **Observer Pattern** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
4. **Builder Pattern** Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.
5. **Strategy Pattern** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
6. **Decorator Pattern** Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.
7. **Bridge Pattern** Allows and is used to separate the abstraction from its implementation, allowing them to vary independently.
8. **Adapter Pattern** Allows incompatible interfaces to work together by converting the interface of a class into another interface.
9. **Command Pattern** Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.
10. **Template Method Pattern** Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.
11. **State Pattern** Allows an object to alter its behavior when its internal state changes, appearing to change its class.
12. **Proxy Pattern** Provides a surrogate or placeholder for another object to control access to it.
13. **Composite Pattern** Composes objects into tree structures to represent part-whole hierarchies, treating individual objects and compositions of objects uniformly.
14. **Iterator Pattern** Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
15. **Mediator Pattern** Defines an object that encapsulates how a set of objects interact, promoting loose coupling.
16. **Memento Pattern** Captures and externalizes an object's internal state so that it can be restored to this state later.
17. **Flyweight Pattern** Uses sharing to support a large number of fine-grained objects efficiently.
18. **Chain of Responsibility Pattern** Allows an object to send a command without knowing which object will handle it, chaining the receiving objects.
19. **Visitor Pattern** Represents an operation to be performed on the elements of an object structure, letting you define new operations without changing the classes of the elements.

Singleton Pattern

Description: Ensures that a class has only one instance and provides a global point of access to that instance.

Example: Singleton pattern is commonly used in scenarios where there should be only one instance of a class. For instance, a configuration manager in a web application:

```
public class ConfigurationManager {
    private static ConfigurationManager instance;

    private ConfigurationManager() {}

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

```
        }

        // Other methods and properties
    }
```

Builder Pattern

Description: Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.

Example: Consider a builder for creating a pizza with various toppings:

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

        public Pizza(Builder builder) {
            this.dough = builder.dough;
            this.sauce = builder.sauce;
            this.topping = builder.topping;
        }

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

            public Builder() {}

            public Builder dough(String dough) {
                this.dough = dough;
                return this;
            }

            public Builder sauce(String sauce) {
                this.sauce = sauce;
                return this;
            }

            public Builder topping(String topping) {
                this.topping = topping;
                return this;
            }

            public Pizza build() {
                return new Pizza(this);
            }
        }
    }

    // Usage
```

```
Pizza pizza = new Pizza.Builder()
                .dough("Thin crust")
                .sauce("Tomato")
                .topping("Cheese")
                .build();
```

Factory Pattern

Description: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Example: Consider a factory for creating different types of cars:

```
interface Car {
    void drive();
}

class Sedan implements Car {
    @Override
    public void drive() {
        System.out.println("Driving Sedan");
    }
}

class SUV implements Car {
    @Override
    public void drive() {
        System.out.println("Driving SUV");
    }
}

class CarFactory {
    public static Car createCar(String type) {
        switch (type) {
            case "Sedan":
                return new Sedan();
            case "SUV":
                return new SUV();
            default:
                throw new IllegalArgumentException("Invalid car type:
                   " + type);
        }
    }
}

Car sedan = CarFactory.createCar("Sedan");
sedan.drive();

Car suv = CarFactory.createCar("SUV");
suv.drive();
```

Strategy Pattern

Description: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Example: Consider a sorting strategy interface with different implementations:

```
import java.util.Arrays;

interface SortingStrategy {
    void sort(int[] array);
}

class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting array using Bubble Sort");
        // Bubble sort implementation
    }
}

class QuickSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting array using Quick Sort");
        // Quick sort implementation
    }
}

class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(int[] array) {
        strategy.sort(array);
    }
}

// Usage
int[] array = {5, 2, 7, 1, 9};
Sorter sorter = new Sorter(new BubbleSort());
sorter.sort(array);
sorter.setStrategy(new QuickSort());
sorter.sort(array);
```

Decorator Pattern

Description: Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.

Example: Consider a simple coffee order system with different decorators for adding condiments:

```
interface Coffee {
    double cost();
    String getDescription();
}

class SimpleCoffee implements Coffee {
    @Override
    public double cost() {
        return 2.0;
    }

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
}

abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public abstract double cost();

    public abstract String getDescription();
}

class Milk extends CoffeeDecorator {
    public Milk(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return coffee.cost() + 0.5;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
}
```

```
class Sugar extends CoffeeDecorator {
    public Sugar(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double cost() {
        return coffee.cost() + 0.2;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }
}

Coffee coffee = new SimpleCoffee();
coffee = new Milk(coffee);
coffee = new Sugar(coffee);
System.out.println("Description: " + coffee.getDescription());
System.out.println("Cost: $" + coffee.cost());
```

Bridge Pattern

Description: The Bridge pattern decouples an abstraction from its implementation, allowing them to vary independently.

Example: Consider a drawing application where shapes can be drawn using different drawing tools. We'll use the Bridge pattern to separate the abstraction (Shape) from its implementation (DrawingAPI).

```
// Abstraction
interface Shape {
    void draw();
}

// Concrete Abstraction
class Circle implements Shape {
    private final DrawingAPI drawingAPI;

    public Circle(DrawingAPI drawingAPI) {
        this.drawingAPI = drawingAPI;
    }

    @Override
    public void draw() {
        drawingAPI.drawCircle();
    }
}

// Implementation
interface DrawingAPI {
    void drawCircle();
```

```
}

// Concrete Implementation
class DrawingAPI1 implements DrawingAPI {
    @Override
    public void drawCircle() {
        System.out.println("Drawing circle using API1");
    }
}

class DrawingAPI2 implements DrawingAPI {
    @Override
    public void drawCircle() {
        System.out.println("Drawing circle using API2");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Shape circle1 = new Circle(new DrawingAPI1());
        Shape circle2 = new Circle(new DrawingAPI2());

        circle1.draw(); // Output: Drawing circle using API1
        circle2.draw(); // Output: Drawing circle using API2
    }
}
```

Adapter Pattern

Description: Allows incompatible interfaces to work together. It converts the interface of a class into another interface that a client expects.

Example: Consider an adapter to convert a Fahrenheit temperature reading to Celsius:

```
interface CelsiusTemperature {
    double getCelsius();
}

class FahrenheitTemperature {
    private double fahrenheit;

    public FahrenheitTemperature(double fahrenheit) {
        this.fahrenheit = fahrenheit;
    }

    public double getFahrenheit() {
        return fahrenheit;
    }
}

class TemperatureAdapter implements CelsiusTemperature {
```

```
    private FahrenheitTemperature temperature;

    public TemperatureAdapter(FahrenheitTemperature temperature) {
        this.temperature = temperature;
    }

    @Override
    public double getCelsius() {
        return (temperature.getFahrenheit() - 32) * 5 / 9;
    }
}

// Usage
FahrenheitTemperature fahrenheitTemp = new FahrenheitTemperature(98.6);
CelsiusTemperature celsiusTemp = new
   TemperatureAdapter(fahrenheitTemp);
System.out.println("Celsius temperature: " + celsiusTemp.getCelsius());
```

Command Pattern

Description: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

Example: Consider a simple remote control with buttons that execute commands:

```
// Command interface
interface Command {
    void execute();
}

// Concrete Command
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is on");
    }
}

// Invoker
```

```
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

// Usage
Light livingRoomLight = new Light();
Command lightOnCommand = new LightOnCommand(livingRoomLight);
RemoteControl remoteControl = new RemoteControl();
remoteControl.setCommand(lightOnCommand);
remoteControl.pressButton(); // Output: "Light is on"
```

Template Pattern

Description: Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Example: Consider a template method for making tea:

```
abstract class Beverage {
    // Template method
    public final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    // Hook method (optional)
    boolean customerWantsCondiments() {
        return true;
```

```
    }
}

class Tea extends Beverage {
    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }

    // Overriding hook method
    @Override
    boolean customerWantsCondiments() {
        return false;
    }
}

class Coffee extends Beverage {
    @Override
    void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

// Usage
Beverage tea = new Tea();
tea.prepareRecipe();

Beverage coffee = new Coffee();
coffee.prepareRecipe();
```

State Pattern

Description: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Example: Consider a simple vending machine that dispenses different items based on its current state:

```
interface State {
    void handleRequest();
}

class TrafficLight {
```

```
        private State state;

        public TrafficLight() {
            state = new RedState();
        }

        public void changeState(State newState) {
            state = newState;
        }

        public void request() {
            state.handleRequest();
        }
    }

    class RedState implements State {
        @Override
        public void handleRequest() {
            System.out.println("Stop");
        }
    }

    // Similarly, implement YellowState and GreenState classes

    // Usage
    TrafficLight trafficLight = new TrafficLight();
    trafficLight.request(); // Outputs: Stop
```

Proxy Pattern

Description: Provides a surrogate or placeholder for another object to control access to it.

Example: Consider a proxy controlling access to a sensitive bank account:

```
    interface BankAccount {
        void deposit(double amount);

        void withdraw(double amount);
    }

    class RealBankAccount implements BankAccount {
        private double balance;

        @Override
        public void deposit(double amount) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        }

        @Override
        public void withdraw(double amount) {
            if (balance >= amount) {
```

```
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Insufficient balance");
        }
    }
}

class BankAccountProxy implements BankAccount {
    private RealBankAccount realAccount;
    private String password;

    public BankAccountProxy(String password) {
        this.password = password;
        realAccount = new RealBankAccount();
    }

    @Override
    public void deposit(double amount) {
        authenticate();
        realAccount.deposit(amount);
    }

    @Override
    public void withdraw(double amount) {
        authenticate();
        realAccount.withdraw(amount);
    }

    private void authenticate() {
        if (!password.equals("secret")) {
            throw new SecurityException("Access denied!");
        }
    }
}

// Usage
BankAccount account = new BankAccountProxy("secret");
account.deposit(1000); // Output: Deposited: $1000
account.withdraw(500); // Output: Withdrawn: $500
```

Composite Pattern

Description: Composes objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

Example: Consider a simple file system representation using the composite pattern:

```
import java.util.ArrayList;
import java.util.List;

interface FileSystemComponent {
```

```
    void print();
}

class File implements FileSystemComponent {
    private String name;

    public File(String name) {
        this.name = name;
    }

    @Override
    public void print() {
        System.out.println("File: " + name);
    }
}

class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> components;

    public Directory(String name) {
        this.name = name;
        components = new ArrayList<>();
    }

    public void addComponent(FileSystemComponent component) {
        components.add(component);
    }

    @Override
    public void print() {
        System.out.println("Directory: " + name);
        for (FileSystemComponent component : components) {
            component.print();
        }
    }
}

// Usage
Directory root = new Directory("Root");
Directory documents = new Directory("Documents");
Directory music = new Directory("Music");
File resume = new File("Resume.docx");
File song = new File("Song.mp3");

documents.addComponent(resume);
music.addComponent(song);

root.addComponent(documents);
root.addComponent(music);
```

```
    root.print();
```

Iterator Pattern

Description: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Example: Consider a simple implementation of an iterator for a list:

```
    interface Iterator<T> {
        boolean hasNext();
        T next();
    }

    interface Aggregate<T> {
        Iterator<T> iterator();
    }

    class ListAggregate<T> implements Aggregate<T> {
        private List<T> items;

        public ListAggregate(List<T> items) {
            this.items = items;
        }

        @Override
        public Iterator<T> iterator() {
            return new ListIterator<>(this);
        }

        // Other methods to manipulate the list
    }

    class ListIterator<T> implements Iterator<T> {
        private ListAggregate<T> aggregate;
        private int index;

        public ListIterator(ListAggregate<T> aggregate) {
            this.aggregate = aggregate;
            this.index = 0;
        }

        @Override
        public boolean hasNext() {
            return index < aggregate.size();
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
```

```
            return aggregate.get(index++);
        }
    }

    // Usage
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    Aggregate<Integer> aggregate = new ListAggregate<>(numbers);
    Iterator<Integer> iterator = aggregate.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
```

Mediator Pattern

Description: Defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly.

Example: Consider a simple chat room where users communicate through a mediator:

```
    import java.util.ArrayList;
    import java.util.List;

    interface ChatMediator {
        void sendMessage(User user, String message);
    }

    class ConcreteChatMediator implements ChatMediator {
        private List<User> users;

        public ConcreteChatMediator() {
            users = new ArrayList<>();
        }

        public void addUser(User user) {
            users.add(user);
        }

        @Override
        public void sendMessage(User user, String message) {
            for (User u : users) {
                if (u != user) {
                    u.receiveMessage(message);
                }
            }
        }
    }

    class User {
        private String name;
        private ChatMediator mediator;

        public User(String name, ChatMediator mediator) {
```

```
        this.name = name;
        this.mediator = mediator;
    }

    public void sendMessage(String message) {
        mediator.sendMessage(this, message);
    }

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

// Usage
ConcreteChatMediator mediator = new ConcreteChatMediator();
User user1 = new User("John", mediator);
User user2 = new User("Alice", mediator);
User user3 = new User("Bob", mediator);

mediator.addUser(user1);
mediator.addUser(user2);
mediator.addUser(user3);

user1.sendMessage("Hello everyone!");
```

Memento Pattern

Description: Captures and externalizes an object's internal state so that the object can be restored to this state later, without violating encapsulation.

Example: Consider a simple text editor with undo functionality using the memento pattern:

```
class Editor {
    private String content;

    public Editor(String content) {
        this.content = content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public EditorMemento save() {
        return new EditorMemento(content);
    }

    public void restore(EditorMemento memento) {
```

```
        this.content = memento.getContent();
    }
}

class EditorMemento {
    private String content;

    public EditorMemento(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

// Usage
Editor editor = new Editor("Initial content");
EditorMemento memento = editor.save(); // Save initial state

editor.setContent("Updated content");
System.out.println("Current content: " + editor.getContent());

editor.restore(memento); // Restore initial state
System.out.println("Restored content: " + editor.getContent());
```

Flyweight Pattern

Description: The Flyweight pattern is used to minimize memory usage and improve performance by sharing common state among multiple objects.

Example: Consider a GUI application that displays a large number of icons on the screen. Each icon may have different properties such as size, color, and position. Instead of creating a separate object for each icon, we can use the Flyweight pattern to share common properties among icons of the same type.

```
import java.awt.Color;
import java.awt.Graphics;
import java.util.HashMap;
import java.util.Map;

// Flyweight interface
interface Icon {
    void draw(int x, int y);
}

// Concrete flyweight class
class ImageIcon implements Icon {
    private final String filename;

    public ImageIcon(String filename) {
        this.filename = filename;
    }
```

```
    @Override
    public void draw(int x, int y) {
        // Load and draw image at (x, y)
        System.out.println("Drawing image '" + filename + "' at (" + x
           + ", " + y + ")");
    }
}

// Flyweight factory
class IconFactory {
    private final Map<String, Icon> icons = new HashMap<>();

    public Icon getIcon(String filename) {
        if (!icons.containsKey(filename)) {
            icons.put(filename, new ImageIcon(filename));
        }
        return icons.get(filename);
    }
}

// Usage
IconFactory factory = new IconFactory();
// Make icon, first time
Icon icon1 = factory.getIcon("icon.png");
icon1.draw(100, 100);
// Reuse existing icon
Icon icon2 = factory.getIcon("icon.png");
icon2.draw(200, 200);
```

Chain of Responsibility Pattern

Description: Allows an object to send a command without knowing which object will handle it. It chains the receiving objects and passes the request along the chain until an object handles it.

Example: Consider a simple ATM withdrawal process with multiple handlers:

```
abstract class WithdrawalHandler {
    protected WithdrawalHandler nextHandler;

    public void setNextHandler(WithdrawalHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public abstract void handleRequest(int amount);
}

class FiftyDollarHandler extends WithdrawalHandler {
    @Override
    public void handleRequest(int amount) {
        if (amount >= 50) {
            int numNotes = amount / 50;
```

```
            int remainingAmount = amount % 50;
            System.out.println("Dispensing " + numNotes + " $50
               notes");
            if (remainingAmount > 0 && nextHandler != null) {
                nextHandler.handleRequest(remainingAmount);
            }
        } else if (nextHandler != null) {
            nextHandler.handleRequest(amount);
        }
    }
}

class TwentyDollarHandler extends WithdrawalHandler {
    @Override
    public void handleRequest(int amount) {
        if (amount >= 20) {
            int numNotes = amount / 20;
            int remainingAmount = amount % 20;
            System.out.println("Dispensing " + numNotes + " $20
               notes");
            if (remainingAmount > 0 && nextHandler != null) {
                nextHandler.handleRequest(remainingAmount);
            }
        } else if (nextHandler != null) {
            nextHandler.handleRequest(amount);
        }
    }
}

// Usage
WithdrawalHandler fiftyHandler = new FiftyDollarHandler();
WithdrawalHandler twentyHandler = new TwentyDollarHandler();

fiftyHandler.setNextHandler(twentyHandler);

// Request handling
fiftyHandler.handleRequest(80);
```

Visitor Pattern

Description: Represents an operation to be performed on the elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates.

Example: Consider a simple implementation of a visitor pattern for a list of shapes:

```
interface Shape {
    void accept(Visitor visitor);
}

class Circle implements Shape {
    @Override
    public void accept(Visitor visitor) {
```

```
        visitor.visit(this);
    }
}

class Rectangle implements Shape {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

interface Visitor {
    void visit(Circle circle);
    void visit(Rectangle rectangle);
}

class AreaCalculator implements Visitor {
    double totalArea;

    @Override
    public void visit(Circle circle) {
        // Calculate area of circle
        totalArea += Math.PI * Math.pow(5, 2); // Assuming radius = 5
    }

    @Override
    public void visit(Rectangle rectangle) {
        // Calculate area of rectangle
        totalArea += 10 * 20; // Assuming width = 10, height = 20
    }
}

// Usage
List<Shape> shapes = new ArrayList<>();
shapes.add(new Circle());
shapes.add(new Rectangle());

AreaCalculator areaCalculator = new AreaCalculator();
for (Shape shape : shapes) {
    shape.accept(areaCalculator);
}
System.out.println("Total area: " + areaCalculator.totalArea);
```

Exception handling

Exception Types

Checked Exceptions

- **Checked exceptions** are exceptions that the compiler forces you to handle. These exceptions are subclasses of `Exception` but not subclasses of `RuntimeException`.
- Examples include `IOException`, `SQLException`, and `ClassNotFoundException`.

Unchecked Exceptions

- **Unchecked exceptions** are exceptions that don't need to be declared in a method's throws clause. These exceptions are subclasses of `RuntimeException`.
- Examples include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `ArithmeticException`.

Errors

- **Errors** are exceptional situations that are typically beyond the control of the programmer. They are not meant to be caught or handled by applications.
- Examples include `OutOfMemoryError`, `StackOverflowError`, and `AssertionError`.

Checked exceptions must be either caught using a try-catch block or declared in the throws clause of the method signature, while unchecked exceptions are not required to be handled explicitly. Checked exceptions are checked at compile time, meaning that the compiler ensures that they are either handled or declared, whereas unchecked exceptions are not checked at compile time.

- Checked Exceptions: Typically used for recoverable conditions where the caller is expected to handle the exception.
- Unchecked Exceptions: Often used for programming errors or conditions that cannot be reasonably handled, such as null pointer or array index out of bounds.

Exception Handling

Try-Catch & Finally

- Java provides the `try-catch` block for handling exceptions. Code that may throw an exception is enclosed within a `try` block, and any exceptions thrown are caught and handled in the `catch` block.
- Multiple `catch` blocks can be used to handle different types of exceptions. Remember that the catch blocks are executed sequentially, the way they appear in code, meaning that it is advised to catch exceptions from more fine grained to more generic ones.
- `Finally` Block The finally block is used to execute code that should always run, regardless of whether an exception occurred or not. It's often used for cleanup tasks like closing resources.

```
try {
    // Code that may throw an exception
} catch (IOException e) {
    // Handle IOException
} catch (SQLException e) {
    // Handle SQLException
} finally {
    // Optional: Code that always executes, regardless of whether an
       exception occurred
```

```
}
```

Try-With-Resources

- Introduced in Java 7, the try-with-resources statement ensures that a resource is closed at the end of the statement, regardless of whether an exception is thrown or not.
- Resources that implement the AutoCloseable interface can be used within the try-with-resources statement.

```
try (BufferedReader reader = new BufferedReader(new
   FileReader("file.txt"))) {
    // Code that uses the BufferedReader
} catch (IOException e) {
    // Handle IOException
}
```

Best practices

- Use specific exceptions: Catch specific exceptions rather than catching generic Exception whenever possible.
- Handle exceptions appropriately: Handle exceptions at an appropriate level of abstraction and provide meaningful error messages.
- Avoid swallowing exceptions: Avoid catching exceptions without taking any action or logging them.
- Clean up resources: Ensure that resources are properly closed in a finally block or using try-with-resources.
- Don't catch Throwable: Avoid catching Throwable, which includes both Exception and Error subclasses, unless absolutely necessary.

Streams & Transforms

1. **Filtering**
   - Filtering is a common operation that allows you to select elements from a stream based on a specified predicate.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
```

2. **Mapping**
   - Mapping transforms each element of a stream using a specified function.

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
List<Integer> nameLengths = names.stream()
                                 .map(String::length)
                                 .collect(Collectors.toList());
```

3. **Sorting**
   - Sorting arranges the elements of a stream in a specified order.

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
List<String> sortedNames = names.stream()
                                .sorted()
                                .collect(Collectors.toList());
```

4. **Reducing**
   - Reducing performs a reduction operation on the elements of a stream and produces a single result.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, Integer::sum);
```

5. **Grouping**
   - Grouping collects elements of a stream into groups based on a specified classifier function.

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
Map<Integer, List<String>> namesByLength = names.stream()
                                                .collect(Collectors.groupin
```

6. **Flattening**
   - Flattening is used to flatten nested collections produced by mapping operations.

```
List<List<Integer>> nestedLists = Arrays.asList(Arrays.asList(1, 2),
   Arrays.asList(3, 4));
List<Integer> flattenedList = nestedLists.stream()
                                         .flatMap(List::stream)
                                         .collect(Collectors.toList());
```