Design Patterns

Creational:

Singleton Design Pattern

The Singleton Design Pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when you need to control access to shared resources like databases, loggers, thread pools, etc.

1. Eager Initialization

In this approach, the instance of the Singleton class is created at the time of class loading. It is the simplest and thread-safe but has a disadvantage: it may create an instance even if it is not required.

```
public class SingletonEager {
    // Creating the single instance of the class at the time of class loading
    private static final SingletonEager instance = new SingletonEager();

    // Private constructor to prevent instantiation
    private SingletonEager() { }

    public static SingletonEager getInstance() {
        return instance;
    }
}

class TestSingletonEager {
    public static void main(String[] args) {
        SingletonEager singleton = SingletonEager.getInstance();
        System.out.println(singleton);
    }
}
```

2. Static Block Initialization

Here, the instance is created in a static block, which allows exception handling during instance creation. This is a form of eager initialization that gives more control.

```java
public class SingletonStaticBlock {
    private static SingletonStaticBlock instance;

    // Private constructor to prevent instantiation
    private SingletonStaticBlock() { }

    // Static block to handle exceptions
    static {
        try {
            instance = new SingletonStaticBlock();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static SingletonStaticBlock getInstance() {
        return instance;
    }
}

class TestSingletonStaticBlock {
    public static void main(String[] args) {
        SingletonStaticBlock singleton = SingletonStaticBlock.getInstance();
        System.out.println(singleton);
    }
}
```

3. Lazy Initialization

In lazy initialization, the Singleton instance is created only when it is needed (i.e., when `getInstance()` is called). This can result in better performance if the Singleton is not used often, but it requires synchronization for thread safety.

```java
public class SingletonLazy {
    private static SingletonLazy instance;

    // Private constructor to prevent instantiation
    private SingletonLazy() { }

    public static SingletonLazy getInstance() {
        if (instance == null) {
            instance = new SingletonLazy();  // Lazy initialization
        }
        return instance;
    }
}

class TestSingletonLazy {
    public static void main(String[] args) {
        SingletonLazy singleton = SingletonLazy.getInstance();
        System.out.println(singleton);
    }
}
```

4. Thread Safe Singleton

Thread-safe singleton ensures that only one instance of the Singleton class is created even in a multi-threaded environment. The instance is created lazily, but synchronization ensures that only one thread can create the instance at a time.

```java
public class SingletonThreadSafe {
    private static SingletonThreadSafe instance;

    // Private constructor to prevent instantiation
    private SingletonThreadSafe() { }

    public static synchronized SingletonThreadSafe getInstance() {
        if (instance == null) {
            instance = new SingletonThreadSafe();  // Lazy initialization with synchronization
        }
        return instance;
    }
}

class TestSingletonThreadSafe {
    public static void main(String[] args) {
        SingletonThreadSafe singleton = SingletonThreadSafe.getInstance();
        System.out.println(singleton);
    }
}
```

5. Bill Pugh Singleton Implementation (Best Practice)

The Bill Pugh Singleton implementation uses an inner static helper class to implement the Singleton. This approach is thread-safe, lazy-loaded, and ensures that the instance is created only when the class is referenced.

```java
public class SingletonBillPugh {
    // Inner static class responsible for holding the Singleton instance
    private static class SingletonHelper {
        private static final SingletonBillPugh instance = new SingletonBillPugh();
    }

    // Private constructor to prevent instantiation
    private SingletonBillPugh() { }

    public static SingletonBillPugh getInstance() {
        return SingletonHelper.instance;
    }
}

class TestSingletonBillPugh {
    public static void main(String[] args) {
        SingletonBillPugh singleton = SingletonBillPugh.getInstance();
        System.out.println(singleton);
    }
}
```

6. Using Reflection to Destroy Singleton Pattern

Reflection allows us to break the Singleton pattern. If we access the constructor via reflection, we can create a new instance of the Singleton class, which violates the Singleton principle.

```java
import java.lang.reflect.Constructor;

public class SingletonReflective {
    private static final SingletonReflective instance = new SingletonReflective();

    private SingletonReflective() {
        if (instance != null) {
            throw new IllegalStateException("Instance already created");
        }
    }

    public static SingletonReflective getInstance() {
        return instance;
    }
}

class TestSingletonReflection {
    public static void main(String[] args) throws Exception {
        SingletonReflective singleton1 = SingletonReflective.getInstance();
        System.out.println(singleton1);

        // Using reflection to create a second instance
        Constructor<SingletonReflective> constructor = SingletonReflective.class.getDeclaredCo
        constructor.setAccessible(true);
        SingletonReflective singleton2 = constructor.newInstance();
        System.out.println(singleton2);
    }
}
```

7. Enum Singleton

Using an enum is considered the best way to implement a Singleton pattern in Java. Enums provide built-in thread safety and ensure that the instance is created only once.

```java
public enum SingletonEnum {
    INSTANCE;

    // Add any methods you need to implement
    public void someMethod() {
        System.out.println("Some method");
    }
}

class TestSingletonEnum {
    public static void main(String[] args) {
        SingletonEnum singleton = SingletonEnum.INSTANCE;
        singleton.someMethod();
    }
}
```

8. Serialization and Singleton

When a Singleton class is serialized, the deserialization process can break the Singleton pattern by creating a new instance. To prevent this, you can implement `readResolve()` to return the existing instance.

```java
import java.io.Serializable;

public class SingletonSerializationSafe implements Serializable {
   private static final long serialVersionUID = 1L;

   // Singleton instance
   private static final SingletonSerializationSafe instance = new SingletonSerializationSafe()

   // Private constructor to prevent instantiation
   private SingletonSerializationSafe() { }

   // Method to return the instance during deserialization
   public static SingletonSerializationSafe getInstance() {
      return instance;
   }

   // Implement readResolve to ensure Singleton instance is preserved after deserialization
   protected Object readResolve() {
      return instance;
   }
}

class TestSingletonSerializationSafe {
   public static void main(String[] args) {
      SingletonSerializationSafe singleton = SingletonSerializationSafe.getInstance();
      System.out.println(singleton);

      // Serialization and deserialization logic can be implemented here
   }
}
```

Exercises for Singleton Design Pattern

Exercise 1: Basic Singleton Implementation Objective: Implement a Singleton class using eager initialization and demonstrate its usage by creating multiple instances and checking if they refer to the same object.

Exercise 2: Lazy Initialization with Synchronization Objective: Implement a Singleton class with lazy initialization, but use the synchronized block in the getInstance() method to ensure thread safety. Test it by calling the getInstance() method from multiple threads and verify that only one instance is created.

Exercise 3: Singleton with Double-Checked Locking Objective: Implement a Singleton class with lazy initialization and optimize it using double-checked locking. This should reduce the overhead of synchronization in multithreaded environments.

Exercise 4: Singleton for Database Connection Objective: Create a Singleton class for managing a database connection. Simulate the process of getting a connection from a pool and demonstrate that only one connection instance is used throughout the application.

Exercise 5: Singleton with Enum Type Objective: Implement a Singleton using the Enum type, which provides a simple and effective way to create a thread-safe Singleton. Verify that the getInstance() method returns the same instance each time by printing out object references.

Java Classes that Use Singleton Design Pattern

java.lang.Runtime

The Runtime class provides a runtime environment in which the Java application runs. It allows interaction with the Java runtime environment, such as memory management, executing system processes, and garbage collection. Singleton Implementation: The Runtime class is a Singleton to ensure that there is only one instance controlling the application's runtime environment.

Usage:

```
Runtime runtime = Runtime.getRuntime();
```

Reason: Ensures that all parts of the program access the same instance, providing consistent and efficient management of runtime resources.

java.lang.System

The System class provides utility methods for system-related operations, such as standard input/output, environment variables, and time management. Singleton Implementation: The System class is implicitly a Singleton because its methods are static, meaning there's no need for an instance of the class to access them.

Usage:

```
System.out.println("Hello, World!");
```

Reason: System-level operations need to be centralized and consistent across all parts of the application.

java.util.Calendar

The Calendar class is used to work with dates and times. While it's not a typical Singleton (because it can return different types of calendars), its default getInstance() method behaves similarly to a

Singleton, returning a single instance for a particular locale and time zone.

Usage:

```
Calendar calendar = Calendar.getInstance();
```

Reason: The Calendar class uses getInstance() to provide access to a default calendar instance. This method ensures that the same instance is used for all date/time manipulations within the default locale.

java.util.Properties

The Properties class is used for reading and writing configuration data in key-value pairs. It is often used for application settings or configuration management. Singleton Implementation: While Properties itself is not strictly a Singleton, it's often used in a Singleton fashion, with a single Properties instance managing all configuration data.

Usage:

```
Properties properties = new Properties();
properties.load(new FileInputStream("config.properties"));
```

Reason: A single instance of Properties is often used globally in applications to handle settings and configurations, so it functions similarly to a Singleton.

java.nio.file.FileSystems

The FileSystems class provides access to the file system abstraction, such as creating and managing FileSystem objects. Singleton Implementation: The FileSystems class uses the Singleton pattern for managing the system's default file system and returning the same instance whenever it is needed.

Usage:

```
FileSystem fs = FileSystems.getDefault();
```

Reason: The default file system should be shared globally, avoiding unnecessary creation of new file system instances.

java.util.TimeZone

The TimeZone class represents a time zone, and it can be used to manage the time zone data for different locations. Singleton Implementation: The TimeZone class has a Singleton-like behavior because it caches time zone data and provides a single instance for a given time zone.

Usage:

```
TimeZone tz = TimeZone.getDefault();
```

Reason: Having a single instance of the time zone ensures consistency when working with time-based data.

javax.xml.parsers.DocumentBuilderFactory

The DocumentBuilderFactory is used to create DocumentBuilder objects that parse XML documents. Singleton Implementation: It follows a Singleton-like pattern by ensuring that a single factory instance is used to create DocumentBuilder instances. The factory is typically retrieved through a static method.

Usage:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

Reason: It centralizes XML parsing, ensuring that configuration and behavior are consistent across the application.

java.util.logging.Logger

The Logger class provides a simple API for logging messages in Java applications. While each logger can be configured with its own name and level, the global logging system is usually treated as a Singleton, ensuring that logging behavior is consistent across an entire application. Usage:

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

Reason: To ensure that all logging across an application comes from a single, centralized logger, providing consistent formatting, logging levels, and outputs.

javax.naming.InitialContext

The InitialContext class is used in Java's naming and directory interface (JNDI) for looking up resources such as database connections, environment variables, and services. Singleton Implementation: The InitialContext class is often implemented as a Singleton to provide a consistent context for looking up resources.

Usage:

```
InitialContext ctx = new InitialContext();
```

Reason: The context for resource lookup needs to be consistent and shared throughout the application to avoid inconsistencies and redundant resource lookups.

Builder Design Pattern

The **Builder Design Pattern** is used to construct a complex object step by step. It allows the creation of an object with a flexible construction process, making it easy to create different variations of the object using the same construction process.

Key Benefits of the Builder Pattern:

1. **Separation of construction and representation**: The builder class handles object construction, which allows you to build complex objects step by step.
2. **Immutable objects**: Typically, the constructed objects are immutable.
3. **Clear and readable code**: Avoids constructor overloading and provides a clean, readable API for object creation.

Example 1: Builder for a Complex Object (e.g., `Person`)

This is a simple example where a `Person` object is constructed using a builder to avoid constructor overloading.

```
public class Person {
    private String name;
    private int age;
    private String address;
    private String phoneNumber;

    private Person(PersonBuilder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.address = builder.address;
        this.phoneNumber = builder.phoneNumber;
    }

    @Override
```

```java
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", address=" + address + ", phoneNumb
    }

    public static class PersonBuilder {
        private String name;
        private int age;
        private String address;
        private String phoneNumber;

        public PersonBuilder setName(String name) {
            this.name = name;
            return this;
        }

        public PersonBuilder setAge(int age) {
            this.age = age;
            return this;
        }

        public PersonBuilder setAddress(String address) {
            this.address = address;
            return this;
        }

        public PersonBuilder setPhoneNumber(String phoneNumber) {
            this.phoneNumber = phoneNumber;
            return this;
        }

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

public class Main {
    public static void main(String[] args) {
        Person person = new Person.PersonBuilder()
                .setName("John Doe")
                .setAge(30)
                .setAddress("123 Main St")
                .setPhoneNumber("555-1234")
                .build();
        System.out.println(person);
    }
}
```

Explanation:

- The `PersonBuilder` class allows you to set individual attributes of the `Person` object.
- The `build()` method constructs the final `Person` object.
- The object is immutable after construction.

Example 2: Builder for `Computer` Class

This example demonstrates a more complex object (`Computer`) being built using a builder.

```
public class Computer {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;

    private Computer(ComputerBuilder builder) {
        this.CPU = builder.CPU;
        this.GPU = builder.GPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", GPU=" + GPU + ", RAM=" + RAM + "GB, storage=" + sto
    }

    public static class ComputerBuilder {
        private String CPU;
        private String GPU;
        private int RAM;
        private int storage;

        public ComputerBuilder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public ComputerBuilder setGPU(String GPU) {
            this.GPU = GPU;
            return this;
        }

        public ComputerBuilder setRAM(int RAM) {
```

```java
            this.RAM = RAM;
            return this;
        }

        public ComputerBuilder setStorage(int storage) {
            this.storage = storage;
            return this;
        }

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

public class Main {
    public static void main(String[] args) {
        Computer computer = new Computer.ComputerBuilder()
                .setCPU("Intel i7")
                .setGPU("NVIDIA GTX 3080")
                .setRAM(16)
                .setStorage(512)
                .build();
        System.out.println(computer);
    }
}
```

Explanation:

- The `ComputerBuilder` class builds the `Computer` object step by step.
- Each method in the builder class returns the builder itself (`return this`), allowing method chaining.
- The `build()` method returns the constructed `Computer` object.

Example 3: Builder for `House` Class with Optional Fields

Here, we build a `House` object with optional fields, making it easy to create different configurations of a house.

```java
public class House {
    private String foundation;
    private String structure;
    private String roof;
    private boolean hasGarage;
    private boolean hasSwimmingPool;

    private House(HouseBuilder builder) {
        this.foundation = builder.foundation;
        this.structure = builder.structure;
        this.roof = builder.roof;
        this.hasGarage = builder.hasGarage;
        this.hasSwimmingPool = builder.hasSwimmingPool;
    }

    @Override
    public String toString() {
        return "House [foundation=" + foundation + ", structure=" + structure + ", roof=" + ro
                ", hasGarage=" + hasGarage + ", hasSwimmingPool=" + hasSwimmingPool + "]";
    }

    public static class HouseBuilder {
        private String foundation;
        private String structure;
        private String roof;
        private boolean hasGarage;
        private boolean hasSwimmingPool;

        public HouseBuilder setFoundation(String foundation) {
            this.foundation = foundation;
            return this;
        }

        public HouseBuilder setStructure(String structure) {
            this.structure = structure;
            return this;
        }

        public HouseBuilder setRoof(String roof) {
            this.roof = roof;
            return this;
        }

        public HouseBuilder setHasGarage(boolean hasGarage) {
            this.hasGarage = hasGarage;
            return this;
        }

        public HouseBuilder setHasSwimmingPool(boolean hasSwimmingPool) {
            this.hasSwimmingPool = hasSwimmingPool;
            return this;
        }
        public House build() {
            return new House(this);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        House house = new House.HouseBuilder()
                .setFoundation("Concrete")
                .setStructure("Wood")
                .setRoof("Tile")
                .setHasGarage(true)
                .build();
        System.out.println(house);
    }
}
```

Explanation:

- The builder allows you to customize which features (e.g., garage, swimming pool) are present in the house.
- Fields like `hasGarage` and `hasSwimmingPool` are optional, so they can be omitted if not needed.

Exercise 1: Basic Builder Pattern

- **Objective**: Create a `Book` class using the builder pattern. The `Book` should have attributes like title, author, publisher, and price. Demonstrate its use in building a book object.

Exercise 2: Builder for `Car` Class

- **Objective**: Implement a builder pattern for a `Car` class, which includes optional features like air conditioning, GPS, and leather seats. Demonstrate creating a car with some optional features.

Exercise 3: Nested Builder Pattern

- **Objective**: Create a `Computer` class with different components (`Processor` , `GraphicsCard` , `RAM`) as nested objects. Use the builder pattern to construct a computer with various combinations of these components.

Exercise 4: Complex Object Creation with Validation

- **Objective**: Design a `Meal` class with optional items like soup, salad, and dessert. Implement the builder pattern to construct a `Meal` object, ensuring that no invalid combinations are allowed (e.g., a meal without any food).

Exercise 5: Builder for `Document` Class

- **Objective**: Implement a `Document` class using the builder pattern where the document can have different types of content such as text, images, and tables. Show how to create a document with different combinations of content.

Java Classes that Use Builder Design Pattern

Several standard Java classes and libraries use the **Builder Design Pattern** to simplify the creation of complex objects. Here are some examples:

1. **`StringBuilder` and `StringBuffer`**
   - `StringBuilder` and `StringBuffer` are used to create mutable sequences of characters. These classes allow the construction of strings by appending or modifying them step by step, though they aren't strictly following the builder pattern.
2. **`java.time.LocalDateTime` (in Java 8 and later)**
   - The `LocalDateTime` class uses a builder-like approach for creating instances of date and time. You can construct a `LocalDateTime` object by setting various fields like year, month, day, hour, minute, etc.
3. **`java.util.List` (e.g., `ArrayList`)**
   - The `List` interface, particularly `ArrayList`, is often used in a builder pattern context when constructing lists with specific elements. However, this is more of a "collection builder" concept rather than a strict implementation of the Builder Design Pattern.
4. **`java.lang.StringBuilder` (Used in String Concatenation)**
   - `StringBuilder` is used to build strings step by step, particularly when there are multiple concatenations or modifications to a string. While it’s not an exact match to the traditional Builder pattern, it shares a similar philosophy in constructing an object incrementally.
5. **`javax.ws.rs.client.Client` (JAX-RS Client API)**
   - In JAX-RS (Java API for RESTful Web Services), the `Client` class uses the Builder pattern to configure and create client instances for interacting with REST services.

Prototype Design Pattern

The **Prototype Design Pattern** is a creational design pattern that allows you to create a new object by copying an existing object, known as the prototype. This is useful when creating an object from scratch is expensive, and it is more efficient to copy an existing instance and modify it.

Key Benefits of the Prototype Pattern:

1. **Efficiency**: It is more efficient to clone an object rather than create a new instance from scratch.
2. **Avoiding Repetition**: Useful in scenarios where many similar objects are required, as it avoids repeating the creation process.
3. **Flexibility**: Allows easy creation of objects with variations without needing to create new classes or constructors.

Example 1: Cloning a `Person` Object

Here, we define a `Person` class that implements the `Cloneable` interface, allowing us to clone an object instead of creating a new one.

```
public class Person implements Cloneable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public Person clone() {
        try {
            return (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }

    public static void main(String[] args) {
        Person original = new Person("John", 25);
        Person clone = original.clone();

        System.out.println("Original: " + original);
        System.out.println("Clone: " + clone);
    }
}
```

Explanation:

- The `Person` class implements `Cloneable`, making it possible to create a clone of an existing `Person` object.
- The `clone()` method creates a shallow copy of the object, and you can modify it further if needed.

Example 2: Cloning a `Car` Object (Deep Clone)

In this example, a deep cloning approach is used to create a clone of a `Car` object, including all nested objects like `Engine` and `Wheel`.

```
public class Engine implements Cloneable {
    private String model;

    public Engine(String model) {
        this.model = model;
    }

    @Override
    protected Engine clone() {
        try {
            return (Engine) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "Engine{" + "model='" + model + '\'' + '}';
    }
}

public class Car implements Cloneable {
    private String make;
    private Engine engine;

    public Car(String make, Engine engine) {
        this.make = make;
        this.engine = engine;
    }

    @Override
    protected Car clone() {
        try {
            Car cloned = (Car) super.clone();
            cloned.engine = this.engine.clone();  // Deep clone the Engine
            return cloned;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "Car{" + "make='" + make + '\'' + ", engine=" + engine + '}';
    }

    public static void main(String[] args) {
        Engine engine = new Engine("V8");
        Car originalCar = new Car("Ford", engine);
        Car clonedCar = originalCar.clone();

        System.out.println("Original Car: " + originalCar);
        System.out.println("Cloned Car: " + clonedCar);
    }
}
```

Explanation:

- The `Car` class contains an `Engine` object. In the `clone()` method, we perform a deep copy by cloning the `Engine` object as well.
- The result is that both the `Car` and its `Engine` are fully cloned.

Example 3: Cloning a `Book` Object with Additional Properties

In this example, we demonstrate a scenario where an object has more complex attributes, and we need to clone an object with mutable properties.

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

public class Book implements Cloneable {
    private String title;
    private String author;
    private List<String> chapters;

    public Book(String title, String author) {
        this.title = title;
        this.author = author;
        this.chapters = new ArrayList<>();
    }

    public void addChapter(String chapter) {
        chapters.add(chapter);
    }

    @Override
    public Book clone() {
        try {
            Book cloned = (Book) super.clone();
            cloned.chapters = new ArrayList<>(this.chapters);  // Clone the chapters list
            return cloned;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    @Override
    public String toString() {
        return "Book{title='" + title + "', author='" + author + "', chapters=" + chapters + '
    }

    public static void main(String[] args) {
        Book originalBook = new Book("Design Patterns", "Erich Gamma");
        originalBook.addChapter("Introduction");
        originalBook.addChapter("Creational Patterns");

        Book clonedBook = originalBook.clone();

        // Modify the cloned book's chapters
        clonedBook.addChapter("Structural Patterns");

        System.out.println("Original Book: " + originalBook);
        System.out.println("Cloned Book: " + clonedBook);
    }
}
```

Explanation:

- The `Book` class contains a list of chapters. In the `clone()` method, we perform a deep clone of the chapters list to avoid sharing references between the original and cloned books.
- This results in a cloned object where the lists of chapters are independently mutable.

Exercise 1: Clone a `Student` Object

- **Objective**: Create a `Student` class with attributes like name, age, and a list of subjects. Implement the Prototype pattern to clone a `Student` object, and modify the list of subjects in the cloned object without affecting the original.

Exercise 2: Cloning a `Laptop` Object with Nested Objects

- **Objective**: Design a `Laptop` class that contains objects like `Processor` , `RAM` , and `Storage` . Implement the Prototype pattern to perform both shallow and deep cloning of the `Laptop` object and observe the differences.

Exercise 3: Prototype for `Person` Class with Address

- **Objective**: Create a `Person` class that has nested objects like `Address` . Use the Prototype pattern to clone the `Person` object and ensure that the cloned object has a distinct copy of the `Address` object (deep clone).

Exercise 4: Cloning a `Library` Object

- **Objective**: Create a `Library` class with a list of `Books` . Implement the Prototype pattern to clone a `Library` and observe how the list of `Books` is handled during cloning (both shallow and deep).

Exercise 5: Prototype with Immutable Objects

- **Objective**: Create a class `ImmutableCar` with attributes like model, make, and price. Implement the Prototype pattern and ensure that the cloned `ImmutableCar` object is a deep clone and does not mutate the original object's state.

Java Classes that Use Prototype Design Pattern

The Prototype pattern is not as commonly seen in the Java standard library, but some classes in Java and popular libraries use or follow the prototype-like behavior for cloning or copying objects.

1. **`java.lang.Object.clone()`** :
   - The `Object` class in Java provides a `clone()` method, which is part of the Prototype pattern. This method allows creating a copy of an object. Any class that wants to be cloned must implement the `Cloneable` interface.
2. **`java.util.ArrayList`** :
   - While not explicitly a Prototype, the `ArrayList` class in Java uses a clone method to create a shallow copy of the list.
3. **`java.util.Date`** :
   - The `Date` class implements the `Cloneable` interface, and the `clone()` method is used to create a duplicate `Date` object.
4. **`javax.swing.ImageIcon`** :
   - The `ImageIcon` class (used for handling image icons in Swing applications) uses the Prototype pattern for copying or cloning an `ImageIcon` object.
5. **Apache Commons `BeanUtils`** :
   - The `BeanUtils` class from Apache Commons has methods that allow copying or cloning beans (objects). These methods implement a version of the Prototype pattern.

Factory Design Pattern

The **Factory Design Pattern** is a creational design pattern that provides a way to create objects without specifying the exact class of the object that will be created. Instead, a factory method is used to create an instance of the required class, often based on input or other conditions.

Key Benefits of the Factory Pattern:

1. **Encapsulation**: It hides the instantiation logic of an object, making the system more modular.
2. **Decoupling**: The client code is decoupled from the specific class types and their instantiation.
3. **Maintainability**: It centralizes object creation, making it easier to manage or update the process.
4. **Flexibility**: You can extend the system by adding new types of products without changing the client code.

Example 1: Simple Factory for Animals

In this example, a `Factory` class is used to create different types of `Animal` objects (e.g., `Dog`, `Cat`).

```java
// Product Interface
public interface Animal {
    void speak();
}

// Concrete Product 1
public class Dog implements Animal {
    @Override
    public void speak() {
        System.out.println("Woof!");
    }
}

// Concrete Product 2
public class Cat implements Animal {
    @Override
    public void speak() {
        System.out.println("Meow!");
    }
}

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

// Main Class
public class FactoryExample {
    public static void main(String[] args) {
        Animal dog = AnimalFactory.createAnimal("dog");
        dog.speak();  // Output: Woof!

        Animal cat = AnimalFactory.createAnimal("cat");
        cat.speak();  // Output: Meow!
    }
}
```

Explanation:

- The `AnimalFactory` class is responsible for creating instances of different `Animal` types based on the input.
- The `createAnimal()` method decides which concrete class to instantiate based on the type of the animal requested.

Example 2: Factory for Different Vehicle Types

In this example, the `Vehicle` interface defines common operations, and the `VehicleFactory` is used to create different types of vehicles (`Car` , `Truck`).

```java
// Product Interface
public interface Vehicle {
    void drive();
}

// Concrete Product 1
public class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}

// Concrete Product 2
public class Truck implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a truck");
    }
}

// Factory
public class VehicleFactory {
    public static Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        } else if (type.equalsIgnoreCase("truck")) {
            return new Truck();
        } else {
            throw new IllegalArgumentException("Unknown vehicle type");
        }
    }
}

// Main Class
public class FactoryExample {
    public static void main(String[] args) {
        Vehicle car = VehicleFactory.createVehicle("car");
        car.drive();  // Output: Driving a car

        Vehicle truck = VehicleFactory.createVehicle("truck");
        truck.drive();  // Output: Driving a truck
    }
}
```

Explanation:

- The `VehicleFactory` creates different `Vehicle` types (`Car` , `Truck`) based on the provided string input.
- The client code does not need to know the concrete class being instantiated.

Example 3: Factory with Abstract Factory for Furniture

This example uses an abstract factory pattern to create families of related objects (`Chair` , `Sofa`) based on the type of furniture.

```
// Abstract Product 1
public interface Chair {
    void sitOn();
}

// Abstract Product 2
public interface Sofa {
```

```java
    void layOn();
}

// Concrete Product 1A
public class VictorianChair implements Chair {
    @Override
    public void sitOn() {
        System.out.println("Sitting on a Victorian chair");
    }
}

// Concrete Product 1B
public class ModernChair implements Chair {
    @Override
    public void sitOn() {
        System.out.println("Sitting on a modern chair");
    }
}

// Concrete Product 2A
public class VictorianSofa implements Sofa {
    @Override
    public void layOn() {
        System.out.println("Laying on a Victorian sofa");
    }
}

// Concrete Product 2B
public class ModernSofa implements Sofa {
    @Override
    public void layOn() {
        System.out.println("Laying on a modern sofa");
    }
}

// Abstract Factory
public interface FurnitureFactory {
    Chair createChair();
    Sofa createSofa();
}

// Concrete Factory 1
public class VictorianFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new VictorianChair();
    }
    @Override
    public Sofa createSofa() {
        return new VictorianSofa();
    }
}

// Concrete Factory 2
public class ModernFurnitureFactory implements FurnitureFactory {
    @Override
    public Chair createChair() {
        return new ModernChair();
    }

    @Override
    public Sofa createSofa() {
        return new ModernSofa();
    }
}

// Main Class
public class FactoryExample {
    public static void main(String[] args) {
        FurnitureFactory factory = new VictorianFurnitureFactory();

        Chair chair = factory.createChair();
        Sofa sofa = factory.createSofa();

        chair.sitOn();  // Output: Sitting on a Victorian chair
        sofa.layOn();   // Output: Laying on a Victorian sofa

        factory = new ModernFurnitureFactory();

        chair = factory.createChair();
        sofa = factory.createSofa();

        chair.sitOn();  // Output: Sitting on a modern chair
        sofa.layOn();   // Output: Laying on a modern sofa
    }
}
```

Explanation:

- The `FurnitureFactory` interface defines methods for creating different products (e.g., `Chair` and `Sofa`).
- Different concrete factories (`VictorianFurnitureFactory` , `ModernFurnitureFactory`) are responsible for creating specific types of furniture.

- The client code can switch between different families of related furniture objects without knowing the concrete classes.

Exercise 1: Simple Animal Factory

- **Objective**: Create a factory that produces animals (`Dog` , `Cat` , `Bird`). Define common methods for animals (e.g., `makeSound()`) and implement a factory class to instantiate these objects.

Exercise 2: Vehicle Factory with Price

- **Objective**: Implement a `Vehicle` factory that creates different types of vehicles (`Car` , `Truck` , `Motorcycle`) with associated prices. Add a `getPrice()` method for each vehicle type.

Exercise 3: Shape Factory

- **Objective**: Create a factory that produces shapes (`Circle` , `Rectangle` , `Square`). Each shape should have an area calculation method. Use the factory to create the shapes and calculate the area.

Exercise 4: Electronics Factory

- **Objective**: Design an electronics factory that creates different types of products like `Phone` , `Laptop` , and `Tablet` . Each product should have a `turnOn()` method, and the factory should handle the creation based on input type.

Exercise 5: Furniture Factory with Discount

- **Objective**: Create a factory for furniture objects (`Table` , `Chair` , `Sofa`). Add a `getDiscountedPrice()` method for each product, and implement a discount strategy based on the type of product.

Java Classes that Use Factory Design Pattern

Many classes in Java utilize the Factory Design Pattern to simplify object creation and to provide flexibility in the codebase. Some of these include:

1. **`java.util.Calendar`** :
    - `Calendar.getInstance()` is a factory method that returns a specific instance of `Calendar` (e.g., `GregorianCalendar`), depending on the locale or system settings.

2. **`java.text.NumberFormat`**:
    - `NumberFormat.getInstance()` is a factory method used to obtain an instance of a `NumberFormat` object, which formats numbers based on the default locale.
3. **`javax.xml.parsers.DocumentBuilderFactory`**:
    - `DocumentBuilderFactory.newInstance()` is a factory method that provides a way to instantiate a `DocumentBuilderFactory` object, which is used to create XML parsers.
4. **`java.nio.file.Files`**:
    - `Files.newBufferedReader()` and other `Files` factory methods are used to create various types of readers and writers for files.
5. **`java.sql.DriverManager`**:
    - `DriverManager.getConnection()` is a factory method used to obtain a connection to a database, abstracting the implementation details of the database connection.

Structural:

Adapter Design Pattern

The **Adapter Design Pattern** is a structural design pattern that allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting one interface into another that a client expects. This pattern is particularly useful when you have existing classes that need to work together, but their interfaces don't match.

Key Benefits of the Adapter Pattern:

1. **Reusability**: Allows classes with incompatible interfaces to work together.
2. **Flexibility**: You can use the adapter to integrate with third-party classes or legacy code.
3. **Separation of Concerns**: Keeps the code modular and encapsulates the integration logic.

Example 1: Adapter for Media Players

In this example, we have two classes `AudioPlayer` and `VideoPlayer` that represent different types of media players. The `MediaAdapter` allows `AudioPlayer` to play `Video` files, which it wouldn't normally support.

```java
// Target Interface
public interface MediaPlayer {
    void play(String mediaType, String fileName);
}

// Adaptee - This class can only play audio
public class AudioPlayer implements MediaPlayer {
    @Override
    public void play(String mediaType, String fileName) {
        if(mediaType.equalsIgnoreCase("mp3")){
            System.out.println("Playing audio file: " + fileName);
        } else {
            System.out.println("Invalid media type for AudioPlayer");
        }
    }
}

// Adaptee - This class can only play video
public class VideoPlayer {
    public void playVideo(String fileName) {
        System.out.println("Playing video file: " + fileName);
    }
}

// Adapter Class - Allows AudioPlayer to play videos by using the VideoPlayer
public class MediaAdapter implements MediaPlayer {
    private VideoPlayer videoPlayer;

    public MediaAdapter() {
        videoPlayer = new VideoPlayer();
    }

    @Override
    public void play(String mediaType, String fileName) {
        if(mediaType.equalsIgnoreCase("mp4")){
            videoPlayer.playVideo(fileName);
        } else {
            System.out.println("Invalid media type for MediaAdapter");
        }
    }
}

// Client Class
public class AdapterPatternExample {
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();
        audioPlayer.play("mp3", "song.mp3");
        MediaPlayer mediaAdapter = new MediaAdapter();
        mediaAdapter.play("mp4", "movie.mp4");
    }
}
```

Explanation:

- `AudioPlayer` can only play `mp3` files.
- The `VideoPlayer` can only play `mp4` files.
- The `MediaAdapter` allows the `AudioPlayer` to play `mp4` files by forwarding the request to `VideoPlayer`.

Example 2: Adapter for Shape Drawing

In this example, we use the Adapter pattern to adapt a `Rectangle` class (which has a `draw()` method) to a `Circle` class, which requires a different method to draw.

```
// Target Interface
public interface Shape {
    void draw();
}

// Adaptee - This class draws a Rectangle
public class Rectangle {
    public void drawRectangle() {
        System.out.println("Drawing a rectangle");
    }
}

// Adapter Class - Adapts Rectangle to Shape
public class RectangleAdapter implements Shape {
    private Rectangle rectangle;

    public RectangleAdapter(Rectangle rectangle) {
        this.rectangle = rectangle;
    }

    @Override
    public void draw() {
        rectangle.drawRectangle();  // Delegating to the Rectangle class
    }
}
```

```
// Client Class
public class AdapterPatternExample {
    public static void main(String[] args) {
        Shape shape = new RectangleAdapter(new Rectangle());
        shape.draw();  // Output: Drawing a rectangle
    }
}
```

Explanation:

- The `Rectangle` class has a method `drawRectangle()`.
- The `Shape` interface requires a `draw()` method.
- The `RectangleAdapter` class adapts `Rectangle` to implement `Shape`.

Example 3: Adapter for Charging Different Devices

In this example, we use the Adapter pattern to allow a `USBAdapter` to charge devices using different charging ports (USB-C, Lightning).

```
// Target Interface
public interface Charger {
    void charge(String deviceType);
}

// Adaptee 1 - USB Charger
public class USBCharger {
    public void chargeUsingUSB(String deviceType) {
        System.out.println("Charging " + deviceType + " using USB charger");
    }
}

// Adaptee 2 - Lightning Charger
public class LightningCharger {
    public void chargeUsingLightning(String deviceType) {
        System.out.println("Charging " + deviceType + " using Lightning charger");
    }
}

// Adapter Class
public class USBAdapter implements Charger {
    private USBCharger usbCharger;

    public USBAdapter(USBCharger usbCharger) {
        this.usbCharger = usbCharger;
    }

    @Override
    public void charge(String deviceType) {
        usbCharger.chargeUsingUSB(deviceType);
    }
}

public class LightningAdapter implements Charger {
    private LightningCharger lightningCharger;

    public LightningAdapter(LightningCharger lightningCharger) {
        this.lightningCharger = lightningCharger;
    }

    @Override
    public void charge(String deviceType) {
        lightningCharger.chargeUsingLightning(deviceType);
    }
}

// Client Class
public class AdapterPatternExample {
    public static void main(String[] args) {
        Charger usbCharger = new USBAdapter(new USBCharger());
        usbCharger.charge("Smartphone");  // Charging Smartphone using USB charger

        Charger lightningCharger = new LightningAdapter(new LightningCharger());
        lightningCharger.charge("iPhone");  // Charging iPhone using Lightning charger
    }
}
```

Explanation:

- The `USBCharger` and `LightningCharger` are two different chargers.
- The `USBAdapter` and `LightningAdapter` are used to make these chargers compatible with the common `Charger` interface.

Exercise 1: Adapter for Different Payment Systems

- **Objective**: Implement a payment system where different payment methods (e.g., Credit Card, PayPal) need to be adapted to a common `PaymentProcessor` interface. Implement adapters for each payment system.

Exercise 2: Adapter for Different Database Connections

- **Objective**: You have two types of database connections (`MySQLConnection` , `OracleConnection`) that should implement a common `DatabaseConnection` interface. Create adapters for each database type to work with the same interface.

Exercise 3: Adapter for Different File Formats

- **Objective**: Create a file reader that can read both `CSV` and `XML` files. Use the Adapter pattern to adapt the file reading process to a common `FileReader` interface.

Exercise 4: Adapter for Different Audio Formats

- **Objective**: Implement an audio player that can play different audio formats (e.g., `MP3` , `WAV` , `FLAC`). Create adapters for each format to implement a common `AudioPlayer` interface.

Exercise 5: Adapter for Legacy System Integration

- **Objective**: You have a legacy system with a method `oldMethod()` that performs a specific task, but you need to integrate it into a new system that uses a `newMethod()` . Implement an adapter to make the legacy method compatible with the new method.

Java Classes That Use Adapter Design Pattern

The **Adapter Design Pattern** is used extensively in Java libraries, especially when working with different libraries or systems that need to be integrated. Below are some examples of Java classes that use the Adapter pattern:

1. `java.util.Collections` (Adapter Pattern via Collections Utilities)

- **Example**: `Collections.synchronizedList()`
- The `Collections.synchronizedList()` method adapts any `List` to be thread-safe by wrapping it with an adapter class that synchronizes the operations.

```
List<String> list = new ArrayList<>();
List<String> syncList = Collections.synchronizedList(list);
```

Explanation: The `Collections.synchronizedList()` method acts as an adapter, adapting a `List` to a synchronized version.

2. `java.util.Arrays` (Adapter Pattern via Arrays Utilities)

- **Example**: `Arrays.asList()`
- The `asList()` method adapts an array into a `List` interface.

```
String[] array = {"A", "B", "C"};
List<String> list = Arrays.asList(array);
```

Explanation: `asList()` adapts the array type to a `List` interface, which provides more flexibility for operations.

3. `java.sql.DriverManager` (Adapter Pattern for Database Drivers)

- **Example**: `DriverManager.getConnection()`
- The `DriverManager` class uses the Adapter pattern to adapt different database drivers into a standard `Connection` interface.

```
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/testdb", "user", "pa
```

Explanation: `DriverManager` adapts various database-specific connection mechanisms (like MySQL, Oracle) to the `Connection` interface.

4. `javax.swing.JFrame` (Adapter Pattern for Event Handling)

- **Example**: Using `ActionListener` adapter for event handling.
- In Swing, event listeners are often implemented via adapters, such as `MouseAdapter` and `KeyAdapter`.

```
JButton button = new JButton("Click me!");
button.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        System.out.println("Button clicked");
    }
});
```

Explanation: The `MouseAdapter` class is an adapter for the `MouseListener` interface, providing default implementations for some methods, allowing the user to only override the methods they care about.

Bridge Design Pattern

The **Bridge Design Pattern** is a structural design pattern that separates an abstraction from its implementation. The pattern allows the abstraction and implementation to vary independently by creating a bridge between them. This helps to avoid a large number of subclasses that are typically created when you try to mix different variations of an abstraction and its implementation.

Key Benefits of the Bridge Pattern:

1. **Separation of Concerns**: Separates the abstraction and implementation into different class hierarchies, making the code more modular.
2. **Flexibility**: Allows changing the abstraction or implementation independently without affecting the other.
3. **Avoids Class Explosion**: Prevents the creation of a large number of subclasses by decoupling the abstraction and implementation.

Example 1: Shape and Color Example

In this example, the `Shape` class represents the abstraction, and the `Color` class represents the implementation. The `Shape` class can be extended to create different shapes like `Circle` and `Square`, and `Color` can represent different colors like `Red` and `Blue`.

```
// Implementor Interface
public interface Color {
    void applyColor();
}

// Concrete Implementors
public class Red implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying Red color");
    }
}

public class Blue implements Color {
    @Override
    public void applyColor() {
        System.out.println("Applying Blue color");
    }
}

// Abstraction
```

```java
public abstract class Shape {
    protected Color color;  // This is the bridge to the implementor

    public Shape(Color color) {
        this.color = color;
    }

    abstract void draw();
}

// Refined Abstraction
public class Circle extends Shape {
    public Circle(Color color) {
        super(color);
    }

    @Override
    public void draw() {
        System.out.print("Drawing Circle - ");
        color.applyColor();
    }
}

public class Square extends Shape {
    public Square(Color color) {
        super(color);
    }

    @Override
    public void draw() {
        System.out.print("Drawing Square - ");
        color.applyColor();
    }
}

// Client Class
public class BridgePatternExample {
    public static void main(String[] args) {
        Shape redCircle = new Circle(new Red());
        redCircle.draw();  // Output: Drawing Circle - Applying Red color

        Shape blueSquare = new Square(new Blue());
        blueSquare.draw();  // Output: Drawing Square - Applying Blue color
    }
}
```

Explanation:

- The `Shape` class is the abstraction, and `Color` is the implementation.
- `Circle` and `Square` are different shapes (refined abstractions) that can be painted in different colors (implementations).

Example 2: Remote Control and Devices Example

In this example, the `RemoteControl` class is the abstraction, and different devices like `TV` and `Radio` are the implementations. The remote control can control various devices through the bridge, enabling flexibility.

```
// Implementor Interface
public interface Device {
    void turnOn();
    void turnOff();
    void setVolume(int volume);
}

// Concrete Implementors
public class TV implements Device {
    @Override
    public void turnOn() {
        System.out.println("Turning on the TV");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the TV");
    }

    @Override
    public void setVolume(int volume) {
        System.out.println("Setting TV volume to " + volume);
    }
}

public class Radio implements Device {
    @Override
    public void turnOn() {
        System.out.println("Turning on the Radio");
    }

    @Override
    public void turnOff() {
        System.out.println("Turning off the Radio");
    }
```

```java
    @Override
    public void setVolume(int volume) {
        System.out.println("Setting Radio volume to " + volume);
    }
}

// Abstraction
public abstract class RemoteControl {
    protected Device device;

    public RemoteControl(Device device) {
        this.device = device;
    }

    abstract void powerOn();
    abstract void powerOff();
    abstract void adjustVolume(int volume);
}

// Refined Abstraction
public class ConcreteRemoteControl extends RemoteControl {
    public ConcreteRemoteControl(Device device) {
        super(device);
    }

    @Override
    void powerOn() {
        device.turnOn();
    }

    @Override
    void powerOff() {
        device.turnOff();
    }

    @Override
    void adjustVolume(int volume) {
        device.setVolume(volume);
    }
}

// Client Class
public class BridgePatternExample {
    public static void main(String[] args) {
        Device tv = new TV();
        RemoteControl remoteForTV = new ConcreteRemoteControl(tv);
        remoteForTV.powerOn();         // Output: Turning on the TV
        remoteForTV.adjustVolume(15);  // Output: Setting TV volume to 15

        Device radio = new Radio();
        RemoteControl remoteForRadio = new ConcreteRemoteControl(radio);

        remoteForRadio.powerOn();      // Output: Turning on the Radio
        remoteForRadio.adjustVolume(10); // Output: Setting Radio volume to 10
    }
}
```

Explanation:

- The `RemoteControl` class is the abstraction and controls the devices through the `Device` interface.
- Different devices (`TV` and `Radio`) implement the `Device` interface, which is the implementation.
- `ConcreteRemoteControl` is the refined abstraction that can control multiple devices without being tightly coupled to any one device.

Example 3: File System Example

In this example, we have an abstraction for `File` and two different types of file systems (`WindowsFileSystem` and `LinuxFileSystem`). The `File` class uses a bridge to work with different file systems.

```
// Implementor Interface
public interface FileSystem {
    void saveFile(String filename);
    void loadFile(String filename);
}

// Concrete Implementors
public class WindowsFileSystem implements FileSystem {
    @Override
    public void saveFile(String filename) {
        System.out.println("Saving " + filename + " on Windows file system.");
    }

    @Override
    public void loadFile(String filename) {
        System.out.println("Loading " + filename + " from Windows file system.");
    }
}
```

```java
public class LinuxFileSystem implements FileSystem {
    @Override
    public void saveFile(String filename) {
        System.out.println("Saving " + filename + " on Linux file system.");
    }

    @Override
    public void loadFile(String filename) {
        System.out.println("Loading " + filename + " from Linux file system.");
    }
}

// Abstraction
public abstract class File {
    protected FileSystem fileSystem;

    public File(FileSystem fileSystem) {
        this.fileSystem = fileSystem;
    }

    abstract void open(String filename);
    abstract void save(String filename);
}

// Refined Abstraction
public class TextFile extends File {
    public TextFile(FileSystem fileSystem) {
        super(fileSystem);
    }

    @Override
    void open(String filename) {
        fileSystem.loadFile(filename);
    }

    @Override
    void save(String filename) {
        fileSystem.saveFile(filename);
    }
}

public class ImageFile extends File {
    public ImageFile(FileSystem fileSystem) {
        super(fileSystem);
    }

    @Override
    void open(String filename) {
        fileSystem.loadFile(filename);
    }

    @Override
    void save(String filename) {
        fileSystem.saveFile(filename);
    }
}

// Client Class
public class BridgePatternExample {
    public static void main(String[] args) {
        File textFileOnWindows = new TextFile(new WindowsFileSystem());
        textFileOnWindows.open("document.txt");
        textFileOnWindows.save("document.txt");

        File imageFileOnLinux = new ImageFile(new LinuxFileSystem());
        imageFileOnLinux.open("picture.jpg");
        imageFileOnLinux.save("picture.jpg");
    }
}
```

Explanation:

- The `File` class represents the abstraction, while `FileSystem` is the implementation.
- Different file types (`TextFile` , `ImageFile`) and different file systems (`WindowsFileSystem` , `LinuxFileSystem`) are bridged together.
- This allows changing the file system independently of the file type.

Exercise 1: Shape and Color with More Shapes and Colors

- **Objective**: Implement the `Bridge` pattern with more shapes (e.g., `Triangle` , `Rectangle`) and more colors (e.g., `Green` , `Yellow`). Create variations of these shapes and apply different colors.

Exercise 2: Remote Control for Different Devices

- **Objective**: Implement a `Bridge` pattern with a remote control that can operate different devices such as `AirConditioner` , `Fan` , and `SmartLight` . Create appropriate adapters and abstractions.

Exercise 3: File System with Multiple File Types

- **Objective**: Implement a `Bridge` pattern for a file system that works with multiple file types such as `TextFile`, `PDF`, and `Word`. Implement different file systems like `LocalFileSystem` and `CloudFileSystem`.

Exercise 4: Payment System with Multiple Payment Methods

- **Objective**: Implement a payment system where different payment methods (`CreditCard`, `PayPal`, `Bitcoin`) work with different currencies (`USD`, `EUR`, `GBP`). Use the `Bridge` pattern to decouple the payment method from the currency.

Exercise 5: Vehicle and Engine Types

- **Objective**: Implement the `Bridge` pattern for vehicles that can have different types of engines (`ElectricEngine`, `DieselEngine`, `GasolineEngine`) and vehicle types (`Car`, `Motorcycle`, `Truck`). Use the pattern to decouple the engine type from the vehicle type.

Java Classes That Use Bridge Design Pattern

1. **`java.awt.Component`** and **`java.awt.peer.ComponentPeer`**

   - The AWT (`Abstract Window Toolkit`) uses the Bridge design pattern to separate the abstraction (`Component`) from the platform-specific implementation (`ComponentPeer`). The `Component` class defines a platform-independent interface, while the `ComponentPeer` class handles platform-specific details for rendering.

   ```java
   Component component = new Button("Click Me");
   component.setSize(100, 50); // Abstraction
   ```

2. **`javax.swing.AbstractButton`** and **`javax.swing.plaf.ButtonUI`**

   - In the Swing library, buttons (like `JButton`) use the Bridge pattern. `AbstractButton` defines the button's abstraction, while `ButtonUI` handles the look-and-feel rendering, allowing different UI designs to be plugged into the button without changing the button logic.

Decorator Design Pattern

The **Decorator Design Pattern** is a structural pattern that allows you to dynamically add behavior to an object at runtime without altering its structure. The pattern involves creating a decorator class that wraps the original class, adding new behavior while keeping the interface of the original class intact. This is particularly useful when you need to extend the functionality of objects in a flexible and reusable way.

Key Benefits of the Decorator Pattern**:

1. **Extends functionality**: Allows you to add new behavior to an object without modifying its existing code.
2. **Flexible**: You can chain decorators to add multiple behaviors, making the pattern highly flexible.
3. **Open/Closed Principle**: The decorator pattern follows the Open/Closed Principle, meaning that classes can be extended without changing their existing code.

Example 1: Coffee Shop Example

In this example, we have a `Beverage` class and different types of beverages. The decorator class `CondimentDecorator` allows you to add condiments like milk, sugar, or soy to the beverage dynamically.

```java
// Component Interface
public abstract class Beverage {
    public abstract double cost();
}

// Concrete Component
public class Coffee extends Beverage {
    @Override
    public double cost() {
        return 5.0; // basic cost of coffee
    }
}

public class Tea extends Beverage {
    @Override
    public double cost() {
        return 3.0; // basic cost of tea
    }
}

// Decorator Class
public abstract class CondimentDecorator extends Beverage {
    protected Beverage beverage;

    public CondimentDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    public abstract double cost();
}
```

```java
// Concrete Decorators
public class MilkDecorator extends CondimentDecorator {
    public MilkDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public double cost() {
        return beverage.cost() + 1.0; // Adding milk cost
    }
}

public class SugarDecorator extends CondimentDecorator {
    public SugarDecorator(Beverage beverage) {
        super(beverage);
    }

    @Override
    public double cost() {
        return beverage.cost() + 0.5; // Adding sugar cost
    }
}

// Client Class
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Beverage coffee = new Coffee();
        System.out.println("Cost of coffee: " + coffee.cost()); // Output: 5.0

        Beverage coffeeWithMilk = new MilkDecorator(new Coffee());
        System.out.println("Cost of coffee with milk: " + coffeeWithMilk.cost()); // Output: 6

        Beverage coffeeWithMilkAndSugar = new SugarDecorator(new MilkDecorator(new Coffee()));
        System.out.println("Cost of coffee with milk and sugar: " + coffeeWithMilkAndSugar.cos
    }
}
```

Explanation:

- `Beverage` is the base component, and `Coffee` and `Tea` are concrete implementations.
- The `CondimentDecorator` class is the base decorator that wraps a `Beverage` object.
- `MilkDecorator` and `SugarDecorator` are concrete decorators that add additional costs (milk and sugar) to the beverage.

Example 2: Window Decorator

In this example, the `Window` class is the core component, and we add functionality (like scrollbars or borders) using decorators.

```java
// Component Interface
public interface Window {
    void draw();
}

// Concrete Component
public class SimpleWindow implements Window {
    @Override
    public void draw() {
        System.out.println("Drawing simple window.");
    }
}

// Decorator Class
public abstract class WindowDecorator implements Window {
    protected Window window;

    public WindowDecorator(Window window) {
        this.window = window;
    }

    public void draw() {
        window.draw();
    }
}

// Concrete Decorators
public class ScrollbarDecorator extends WindowDecorator {
    public ScrollbarDecorator(Window window) {
        super(window);
    }

    @Override
    public void draw() {
        super.draw();
        System.out.println("Adding scrollbar.");
    }
}

public class BorderDecorator extends WindowDecorator {
    public BorderDecorator(Window window) {
        super(window);
    }

    @Override
    public void draw() {
        super.draw();
        System.out.println("Adding border.");
    }
}

// Client Class
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Window simpleWindow = new SimpleWindow();
        simpleWindow.draw(); // Output: Drawing simple window.

        Window scrollWindow = new ScrollbarDecorator(new SimpleWindow());
        scrollWindow.draw(); // Output: Drawing simple window. Adding scrollbar.

        Window borderedScrollWindow = new BorderDecorator(new ScrollbarDecorator(new SimpleWin
        borderedScrollWindow.draw(); // Output: Drawing simple window. Adding scrollbar. Addin
    }
}
```

Explanation:

- `Window` is the component, and `SimpleWindow` is the basic implementation.
- `WindowDecorator` is the base decorator, which adds functionality to the `Window` .
- `ScrollbarDecorator` and `BorderDecorator` add scrollbars and borders, respectively, to the window.

Example 3: Text Editor Example

In this example, the decorator is used to add functionality to a `TextEditor` such as making the text bold, italic, or underlined.

```
// Component Interface
public interface Text {
    String format();
}

// Concrete Component
public class PlainText implements Text {
    private String text;
```

```java
    public PlainText(String text) {
        this.text = text;
    }

    @Override
    public String format() {
        return text; // Basic text, no formatting
    }
}

// Decorator Class
public abstract class TextDecorator implements Text {
    protected Text text;

    public TextDecorator(Text text) {
        this.text = text;
    }

    public abstract String format();
}

// Concrete Decorators
public class BoldDecorator extends TextDecorator {
    public BoldDecorator(Text text) {
        super(text);
    }

    @Override
    public String format() {
        return "<b>" + text.format() + "</b>"; // Add bold formatting
    }
}

public class ItalicDecorator extends TextDecorator {
    public ItalicDecorator(Text text) {
        super(text);
    }

    @Override
    public String format() {
        return "<i>" + text.format() + "</i>"; // Add italic formatting
    }
}

public class UnderlineDecorator extends TextDecorator {
    public UnderlineDecorator(Text text) {
        super(text);
    }
    @Override
    public String format() {
        return "<u>" + text.format() + "</u>"; // Add underline formatting
    }
}

// Client Class
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Text text = new PlainText("Hello, world!");
        System.out.println(text.format()); // Output: Hello, world!

        Text boldText = new BoldDecorator(new PlainText("Hello, world!"));
        System.out.println(boldText.format()); // Output: <b>Hello, world!</b>

        Text italicBoldText = new ItalicDecorator(new BoldDecorator(new PlainText("Hello, worl
        System.out.println(italicBoldText.format()); // Output: <i><b>Hello, world!</b></i>
    }
}
```

Explanation:

- `Text` is the base component, and `PlainText` is the basic implementation of the `Text` interface.
- `TextDecorator` is the base decorator class.
- `BoldDecorator` , `ItalicDecorator` , and `UnderlineDecorator` add specific formatting to the text.

Exercise 1: Vehicle Rental System

- **Objective**: Implement a `Vehicle` rental system where `Car` , `Motorcycle` , and `Bicycle` are the base components. Use decorators to add additional features like `GPS` , `AirConditioning` , or `Insurance` .

Exercise 2: Notifications System

- **Objective**: Create a notification system where the base notification (`EmailNotification`) can be decorated with additional functionalities such as `SMSNotification` , `PushNotification` , and `LoggingNotification` .

Exercise 3: File Compression

- **Objective**: Implement a `File` interface and create decorators for compressing the file, encrypting it, and adding a password protection layer. Allow combining these decorators to add functionality dynamically.

Exercise 4: Pizza Ordering System

- **Objective**: Create a `Pizza` interface with basic pizza types (`CheesePizza` , `VegPizza`). Use decorators to add extra toppings like `Olives` , `Mushrooms` , or `ExtraCheese` .

Exercise 5: Document Printing System

- **Objective**: Implement a document printing system where a basic document can be printed in `BlackAndWhite` . Use decorators to add `ColorPrinting` , `DoubleSidedPrinting` , and `PaperSizeSelection` .

Java Classes That Use Decorator Design Pattern

1. `java.io.InputStream` and `java.io.OutputStream`

The `Decorator Design Pattern` is widely used in the `java.io` package. For instance, `BufferedInputStream` and `BufferedOutputStream` are decorators for the `InputStream` and `OutputStream` classes, respectively, adding extra functionality like buffering to improve I/O performance.

```
InputStream inputStream = new BufferedInputStream(new FileInputStream("file.txt"));
```

Explanation:

- `BufferedInputStream` is a decorator that adds buffering to the `FileInputStream` without modifying its core functionality.

2. `java.util.Collections`

The `Collections` class in Java uses the decorator pattern to create synchronized collections. For example, `Collections.synchronizedList()` wraps a list to ensure thread-safety.

```
List<String> list = Collections.synchronizedList(new ArrayList<>());
```

Explanation:

- `Collections.synchronizedList()` wraps the `ArrayList` and adds synchronization, ensuring thread-safe operations.

3. **`javax.swing.JComponent`**

In Swing, components like `JButton` and `JLabel` often use decorators to add extra features like borders, tooltips, or background colors. These decorators enhance the functionality of the component dynamically.

```
JButton button = new JButton("Click Me");
button.set

Border(BorderFactory.createLineBorder(Color.BLACK));
```

Explanation:

- `BorderFactory.createLineBorder()` is a decorator that adds a border to the `JButton` component.

Facade Design Pattern

The **Facade Design Pattern** is a structural design pattern that provides a simplified interface to a complex subsystem or set of interfaces. The goal is to hide the complexity of the subsystem and provide a higher-level interface that is easier to use.

Key Benefits of the Facade Pattern:

1. **Simplifies client interaction**: The client only needs to interact with a simplified interface, avoiding the need to understand the complexities of the underlying system.
2. **Reduces dependencies**: The facade decouples the client from the subsystems, reducing dependencies and making the system easier to modify or extend.
3. **Improves code readability**: The facade provides a clear and unified interface, which improves the maintainability of the code.

Example 1: Home Theater System

In a home theater system, there are many components like the `Amplifier`, `DVDPlayer`, `Projector`, and `Lights`. The Facade pattern simplifies the interaction by providing a unified interface to control all the components.

```
// Subsystems
public class Amplifier {
```

```java
    public void on() {
        System.out.println("Amplifier is on");
    }

    public void off() {
        System.out.println("Amplifier is off");
    }
}

public class DVDPlayer {
    public void on() {
        System.out.println("DVD Player is on");
    }

    public void play() {
        System.out.println("DVD Player is playing movie");
    }

    public void off() {
        System.out.println("DVD Player is off");
    }
}

public class Projector {
    public void on() {
        System.out.println("Projector is on");
    }

    public void off() {
        System.out.println("Projector is off");
    }
}

public class Lights {
    public void dim() {
        System.out.println("Lights are dimmed");
    }

    public void on() {
        System.out.println("Lights are on");
    }
}

// Facade Class
public class HomeTheaterFacade {
    private Amplifier amplifier;
    private DVDPlayer dvdPlayer;
    private Projector projector;
    private Lights lights;

    public HomeTheaterFacade(Amplifier amplifier, DVDPlayer dvdPlayer, Projector projector, Li
        this.amplifier = amplifier;
        this.dvdPlayer = dvdPlayer;
        this.projector = projector;
        this.lights = lights;
    }

    public void watchMovie() {
        lights.dim();
        projector.on();
        amplifier.on();
        dvdPlayer.on();
        dvdPlayer.play();
        System.out.println("Ready to watch the movie!");
    }

    public void endMovie() {
        lights.on();
        projector.off();
        amplifier.off();
        dvdPlayer.off();
        System.out.println("Movie has ended.");
    }
}

// Client Code
public class FacadePatternExample {
    public static void main(String[] args) {
        Amplifier amplifier = new Amplifier();
        DVDPlayer dvdPlayer = new DVDPlayer();
        Projector projector = new Projector();
        Lights lights = new Lights();

        HomeTheaterFacade homeTheater = new HomeTheaterFacade(amplifier, dvdPlayer, projector,
        homeTheater.watchMovie();  // Simplified movie-watching process
        homeTheater.endMovie();    // Simplified movie-ending process
    }
}
```

Explanation:

- The `HomeTheaterFacade` class provides a simple interface to interact with the home theater system.
- The client does not need to interact directly with each component (like `Amplifier` , `DVDPlayer` ,

`Projector`); instead, they use the `HomeTheaterFacade` to start and stop the movie experience.

Example 2: Computer Startup Process

In a computer system, starting up involves interacting with various components like the `CPU` , `Memory` , `Disk` , and `OS` . The Facade pattern simplifies the startup process by providing a unified interface.

```
// Subsystems
public class CPU {
    public void freeze() {
        System.out.println("Freezing CPU");
    }

    public void jump(String location) {
        System.out.println("Jumping to " + location);
    }

    public void execute() {
        System.out.println("Executing instructions");
    }
}

public class Memory {
    public void load(String data) {
        System.out.println("Loading data: " + data);
    }
}

public class Disk {
    public String read(String location) {
        return "Data from " + location;
    }
}

public class OS {
    public void boot() {
        System.out.println("Booting the operating system");
    }
}

// Facade Class
public class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private Disk disk;
    private OS os;

    public ComputerFacade(CPU cpu, Memory memory, Disk disk, OS os) {
        this.cpu = cpu;
        this.memory = memory;
        this.disk = disk;
        this.os = os;
    }

    public void startComputer() {
        cpu.freeze();
        memory.load(disk.read("boot location"));
        cpu.jump("start");
        cpu.execute();
        os.boot();
        System.out.println("Computer is now started.");
    }
}

// Client Code
public class FacadePatternExample {
    public static void main(String[] args) {
        CPU cpu = new CPU();
        Memory memory = new Memory();
        Disk disk = new Disk();
        OS os = new OS();

        ComputerFacade computer = new ComputerFacade(cpu, memory, disk, os);
        computer.startComputer();  // Simplified computer startup process
    }
}
```

Explanation:

- The `ComputerFacade` class abstracts the complexity of the computer startup process by interacting with subsystems like `CPU`, `Memory`, `Disk`, and `OS`.
- The client can simply call the `startComputer()` method, and the facade takes care of the complex sequence of operations.

Example 3: Restaurant Ordering System

In a restaurant, a customer might need to interact with the `Menu`, `Order`, `Kitchen`, and `Payment` systems. The Facade pattern simplifies this by providing a single point of contact for the ordering process.

```java
// Subsystems
public class Menu {
    public void displayMenu() {
        System.out.println("Displaying menu");
    }
}

public class Order {
    public void takeOrder() {
        System.out.println("Taking customer order");
    }

    public void prepareOrder() {
        System.out.println("Preparing customer order");
    }
}

public class Kitchen {
    public void cook() {
        System.out.println("Cooking the food");
    }
}

public class Payment {
    public void processPayment() {
        System.out.println("Processing payment");
    }
}

// Facade Class
public class RestaurantFacade {
    private Menu menu;
    private Order order;
    private Kitchen kitchen;
    private Payment payment;

    public RestaurantFacade(Menu menu, Order order, Kitchen kitchen, Payment payment) {
        this.menu = menu;
        this.order = order;
        this.kitchen = kitchen;
        this.payment = payment;
    }

    public void placeOrder() {
        menu.displayMenu();
        order.takeOrder();
        order.prepareOrder();
        kitchen.cook();
        payment.processPayment();
        System.out.println("Order placed successfully.");
    }
}

// Client Code
public class FacadePatternExample {
    public static void main(String[] args) {
        Menu menu = new Menu();
        Order order = new Order();
        Kitchen kitchen = new Kitchen();
        Payment payment = new Payment();

        RestaurantFacade restaurantFacade = new RestaurantFacade(menu, order, kitchen, payment
        restaurantFacade.placeOrder();  // Simplified order placement process
    }
}
```

Explanation:

- The `RestaurantFacade` class provides a simple interface for placing an order, interacting with subsystems like `Menu`, `Order`, `Kitchen`, and `Payment`.
- The client only interacts with the facade, simplifying the order process.

Exercise 1: Online Shopping System

- **Objective**: Create a facade for an online shopping system that interacts with subsystems like `Cart`, `Payment`, `Shipping`, and `Inventory`. Simplify the order process through the facade.

Exercise 2: Library Management System

- **Objective**: Create a facade for a library management system where the client can interact with subsystems like `BookCatalog`, `UserAccount`, `ReservationSystem`, and `NotificationService`.

Exercise 3: Smart Home System

- **Objective**: Implement a facade for a smart home system that controls different devices like `Lighting`, `TemperatureControl`, `SecuritySystem`, and `EntertainmentSystem` with a single interface for the user.

Exercise 4: Bank System

- **Objective**: Create a facade for a banking system that interacts with subsystems like `AccountManagement`, `LoanProcessing`, `TransactionService`, and `NotificationService`.

Exercise 5: Hotel Reservation System

- **Objective**: Design a facade for a hotel reservation system that interacts with subsystems like `RoomBooking`, `PaymentGateway`, `NotificationService`, and `CustomerService`.

Java Classes that Use Facade Design Pattern

1. `javax.faces.FacesContext` (JavaServer Faces)

JavaServer Faces (JSF) is a framework for building web applications, and `FacesContext` is an example of a facade class. It abstracts the complexities of managing requests, responses, and navigation in web applications.

```
FacesContext facesContext = FacesContext.getCurrentInstance();
facesContext.getExternalContext().redirect("home.xhtml");
```

Explanation:

- `FacesContext` provides a simplified interface to manage the lifecycle of a JSF request, hiding the complexity of handling requests and responses.

2. `java.awt.Toolkit` (Java AWT Toolkit)

`Toolkit` is a facade for accessing different graphical and system-related services in Java's AWT library, like interacting with the system's native GUI environment.

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Dimension screenSize = toolkit.getScreenSize();
System.out.println("Screen Size: " + screenSize);
```

Explanation:

- `Toolkit` provides a simple interface to access platform-dependent resources, like screen size or system properties.

3. `java.util.logging.Logger`

The `Logger` class in the `java.util.logging` package serves as a facade for the underlying logging subsystems. It simplifies logging by providing a higher-level interface for different logging levels (info, warn, error).

```
Logger logger = Logger.getLogger("MyLogger");
logger.info("This is an info message");
```

Explanation:

- `Logger` acts as a facade that abstracts the underlying logging system, allowing for simpler logging without worrying about low-level configuration.

Proxy Design Pattern

The **Proxy Design Pattern** is a structural design pattern that provides an object representing another object. The proxy controls access to the real object and can add additional functionality like lazy initialization, access control, logging, or monitoring. It acts as an intermediary between the client and the real object.

Types of Proxy Patterns:

1. **Virtual Proxy**: Used to delay the creation of an expensive object until it's needed.
2. **Protection Proxy**: Controls access to the real object based on permissions or security checks.
3. **Remote Proxy**: Used when an object resides in a different address space (e.g., in a different machine or server).
4. **Cache Proxy**: Caches the result of expensive computations or remote calls to avoid recomputation or re-fetching.

Example 1: Virtual Proxy

A **virtual proxy** can be used to delay the creation of an object that is resource-intensive. For example, consider an image loading system where the actual image is loaded only when it's required.

```
// Real Subject (the object that is expensive to create or load)
public class RealImage {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk(); // Simulating the expensive operation of loading an image
```

```java
    }

    public void loadImageFromDisk() {
        System.out.println("Loading image: " + filename);
    }

    public void display() {
        System.out.println("Displaying image: " + filename);
    }
}

// Proxy Class
public class ImageProxy {
    private RealImage realImage;
    private String filename;

    public ImageProxy(String filename) {
        this.filename = filename;
    }

    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename); // Lazy loading
        }
        realImage.display();
    }
}

// Client Code
public class ProxyPatternExample {
    public static void main(String[] args) {
        ImageProxy image1 = new ImageProxy("image1.jpg");
        image1.display();  // Image will be loaded and displayed

        image1.display();  // Image will not be loaded again, it will just be displayed
    }
}
```

Explanation:

- `RealImage` simulates a resource-heavy object (e.g., an image) that takes time to load.
- `ImageProxy` serves as a proxy to delay the loading of the image until it's needed.

Example 2: Protection Proxy

A **protection proxy** controls access to the real object based on security or permission checks. For example, access to certain resources may be restricted based on user roles.

```java
// Real Subject (the actual service)
public class SensitiveData {
    public void accessData() {
        System.out.println("Accessing sensitive data...");
    }
}

// Proxy Class
public class SecurityProxy {
    private SensitiveData sensitiveData;
    private String userRole;

    public SecurityProxy(String userRole) {
        this.userRole = userRole;
        this.sensitiveData = new SensitiveData();
    }

    public void accessData() {
        if ("Admin".equals(userRole)) {
            sensitiveData.accessData();  // Allows access only for Admins
        } else {
            System.out.println("Access denied: Insufficient permissions");
        }
    }
}

// Client Code
public class ProxyPatternExample {
    public static void main(String[] args) {
        SecurityProxy adminProxy = new SecurityProxy("Admin");
        adminProxy.accessData();  // Access allowed

        SecurityProxy userProxy = new SecurityProxy("User");
        userProxy.accessData();  // Access denied
    }
}
```

Explanation:

- `SensitiveData` is the real object that can access sensitive information.
- `SecurityProxy` checks the user's role and controls access to `SensitiveData`. Only users with the "Admin" role can access the data.

Example 3: Remote Proxy

A **remote proxy** is used when the object resides in a different address space, such as in a different machine (e.g., in a distributed system). This pattern allows the client to interact with an object remotely as if it were local.

```java
// Real Subject (Remote Object)
public class RealSubject {
    public void request() {
        System.out.println("Request received and processed by the real subject.");
    }
}

// Proxy Class
public class RemoteProxy {
    private RealSubject realSubject;
    private String serverAddress;

    public RemoteProxy(String serverAddress) {
        this.serverAddress = serverAddress;
    }

    public void request() {
        if (realSubject == null) {
            System.out.println("Connecting to the remote server at: " + serverAddress);
            realSubject = new RealSubject();  // Simulate remote object connection
        }
        realSubject.request();  // Forward the request to the real object
    }
}

// Client Code
public class ProxyPatternExample {
    public static void main(String[] args) {
        RemoteProxy remoteProxy = new RemoteProxy("192.168.0.1");
        remoteProxy.request();  // Request is forwarded to the real object through the proxy
    }
}
```

Explanation:

- `RealSubject` simulates the actual remote object that can process requests.
- `RemoteProxy` acts as a proxy for `RealSubject` and handles the interaction with the real object, such as connecting to a remote server.

Exercise 1: Bank Account Proxy

- **Objective**: Create a bank account system where a `BankAccountProxy` checks whether the user has sufficient funds before allowing withdrawals. Implement different types of proxy (virtual, protection) for various features.

Exercise 2: File System Proxy

- **Objective**: Implement a file system where a proxy is used to delay the loading of large files. When a file is accessed for the first time, it should be loaded from disk; subsequent accesses should use the cached content.

Exercise 3: Logging Proxy

- **Objective**: Implement a proxy that adds logging functionality to method calls of a service. The proxy should log method calls and parameters before passing the request to the actual service.

Exercise 4: Virtual Proxy for Database Queries

- **Objective**: Create a proxy for a database query service that delays database connection until a query is actually executed. Ensure the proxy initializes the connection only when the real data is needed.

Exercise 5: Proxy for Remote File Access

- **Objective**: Implement a proxy for accessing files from a remote server. The proxy should handle file fetching, managing network connections, and delegating tasks to the remote file service.

Java Classes that Use Proxy Design Pattern

1. `java.lang.reflect.Proxy`

The `Proxy` class in the `java.lang.reflect` package is a dynamic proxy that can be used to create a proxy instance for interfaces. It is often used in libraries like Spring and Hibernate.

```java
import java.lang.reflect.*;

public interface RealSubject {
    void request();
}
```

```java
public class RealSubjectImpl implements RealSubject {
    public void request() {
        System.out.println("Request processed by the real subject.");
    }
}

public class DynamicProxy implements InvocationHandler {
    private RealSubject realSubject;

    public DynamicProxy(RealSubject realSubject) {
        this.realSubject = realSubject;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Before invoking: " + method.getName());
        Object result = method.invoke(realSubject, args);
        System.out.println("After invoking: " + method.getName());
        return result;
    }
}

// Client Code
public class ProxyPatternExample {
    public static void main(String[] args) {
        RealSubject realSubject = new RealSubjectImpl();
        RealSubject proxyInstance = (RealSubject) Proxy.newProxyInstance(
                realSubject.getClass().getClassLoader(),
                new Class[] { RealSubject.class },
                new DynamicProxy(realSubject)
        );
        proxyInstance.request();  // Proxy handles the request with additional functionality
    }
}
```

Explanation:

- `java.lang.reflect.Proxy` is used to create dynamic proxy instances that implement specified interfaces.
- The `DynamicProxy` class intercepts method calls to the real object and allows additional functionality (like logging) to be added.

2. `java.rmi.server.UnicastRemoteObject`

In RMI (Remote Method Invocation), `UnicastRemoteObject` is a class that provides the functionality of creating a proxy for remote objects. It allows the proxy to represent a remote object that lives in a different JVM.

```java
import java.rmi.*;

public interface MyRemote extends Remote {
    void sayHello() throws RemoteException;
}

public class MyRemoteImpl extends UnicastRemoteObject implements MyRemote {
    public MyRemoteImpl() throws RemoteException {
        super();
    }

    public void sayHello() throws RemoteException {
        System.out.println("Hello from the remote object!");
    }
}

// Client Code
public class ProxyPatternExample {
    public static void main(String[] args) {
        try {
            MyRemote remote = (MyRemote) Naming.lookup("//localhost/RemoteHello");
            remote.sayHello();  // Proxy forwards the request to the real remote object
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- `UnicastRemoteObject` provides the functionality of creating a proxy for remote objects, which can be used for communication between different JVMs.

Composite Design Pattern

The **Composite Design Pattern** is a structural pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies. This pattern allows clients to treat individual objects and compositions of objects uniformly. It is particularly useful when you want to represent hierarchical structures, such as file systems, organizational charts, or graphical user interfaces (GUIs), where both single objects and collections of objects are treated the same.

Key Benefits of the Composite Pattern:

1. **Uniformity**: Treats individual objects and compositions of objects uniformly.
2. **Flexibility**: Makes it easy to add new components without changing the client code.
3. **Simplicity**: Simplifies the client code by using the same interface for both individual objects and composites.

Example 1: File System (File and Directory)

A **file system** can be structured with files and directories. A directory can contain files or other directories, forming a tree-like structure. Both `File` and `Directory` can be treated uniformly as `FileSystemComponent` objects.

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

// Component
public abstract class FileSystemComponent {
    public abstract void display();
}

// Leaf Class
public class File extends FileSystemComponent {
    private String name;

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

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

// Composite Class
public class Directory extends FileSystemComponent {
    private String name;
    private List<FileSystemComponent> components = new ArrayList<>();

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

    public void add(FileSystemComponent component) {
        components.add(component);
```

```java
    }

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

// Client Code
public class CompositePatternExample {
    public static void main(String[] args) {
        FileSystemComponent file1 = new File("file1.txt");
        FileSystemComponent file2 = new File("file2.txt");

        Directory directory1 = new Directory("Directory1");
        directory1.add(file1);
        directory1.add(file2);

        FileSystemComponent file3 = new File("file3.txt");
        Directory directory2 = new Directory("Directory2");
        directory2.add(file3);

        Directory mainDirectory = new Directory("MainDirectory");
        mainDirectory.add(directory1);
        mainDirectory.add(directory2);

        mainDirectory.display();  // Display entire structure
    }
}
```

Explanation:

- `FileSystemComponent` is an abstract class that defines the common interface for both leaf objects (`File`) and composite objects (`Directory`).
- `File` is a leaf component representing individual files.
- `Directory` is a composite component that can contain both files and other directories.
- The client interacts with the composite structure (`mainDirectory`), and the system treats all components uniformly.

Example 2: Organization Structure (Employee and Department)

An **organization structure** can be modeled using the Composite pattern, where both individual employees and departments are treated as `OrganizationComponent` . A department can contain employees or other departments.

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

// Component
public abstract class OrganizationComponent {
    public abstract void showDetails();
}

// Leaf Class
public class Employee extends OrganizationComponent {
    private String name;
    private String position;

    public Employee(String name, String position) {
        this.name = name;
        this.position = position;
    }

    @Override
    public void showDetails() {
        System.out.println("Employee: " + name + ", Position: " + position);
    }
}

// Composite Class
public class Department extends OrganizationComponent {
    private String name;
    private List<OrganizationComponent> components = new ArrayList<>();

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

    public void add(OrganizationComponent component) {
        components.add(component);
    }

    @Override
    public void showDetails() {
        System.out.println("Department: " + name);
        for (OrganizationComponent component : components) {
            component.showDetails();
        }
    }
}

// Client Code
public class CompositePatternExample {
    public static void main(String[] args) {
        Employee emp1 = new Employee("Alice", "Developer");
        Employee emp2 = new Employee("Bob", "Tester");

        Department devDepartment = new Department("Development");
        devDepartment.add(emp1);
        devDepartment.add(emp2);

        Employee emp3 = new Employee("Charlie", "HR Manager");
        Department hrDepartment = new Department("HR");
        hrDepartment.add(emp3);

        Department company = new Department("Company");
        company.add(devDepartment);
        company.add(hrDepartment);

        company.showDetails();  // Display the entire organization structure
    }
}
```

Explanation:

- `OrganizationComponent` is an abstract class that provides a common interface for both `Employee` (leaf) and `Department` (composite).
- `Employee` is a leaf object, representing individual employees.
- `Department` is a composite object, representing a department that can contain other departments or employees.
- The client can interact with the entire organization structure uniformly.

Example 3: Graphics System (Shape and Group)

In a **graphics system**, you can use the Composite pattern to represent shapes (such as `Circle`, `Rectangle`) and groups of shapes (`Group`), which can contain other shapes or groups.

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

// Component
```

```java
public abstract class Shape {
    public abstract void draw();
}

// Leaf Class
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

// Composite Class
public class Group extends Shape {
    private List<Shape> shapes = new ArrayList<>();

    public void addShape(Shape shape) {
        shapes.add(shape);
    }

    public void removeShape(Shape shape) {
        shapes.remove(shape);
    }

    @Override
    public void draw() {
        for (Shape shape : shapes) {
            shape.draw();
        }
    }
}

// Client Code
public class CompositePatternExample {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        Group group1 = new Group();
        group1.addShape(circle);
        group1.addShape(rectangle);
        Shape group2 = new Group();
        ((Group) group2).addShape(group1);

        group2.draw();  // Draw the entire group with shapes
    }
}
```

Explanation:

- `Shape` is an abstract class that defines the common interface for both leaf (`Circle` , `Rectangle`) and composite (`Group`) objects.
- `Circle` and `Rectangle` are leaf objects that represent individual shapes.
- `Group` is a composite object that can contain both individual shapes and other groups.
- The client can call `draw()` on the `Group` object to render all contained shapes, demonstrating the composite structure.

Exercise 1: E-commerce Catalog

- **Objective**: Create a catalog for an e-commerce website where both individual products and product categories (which can contain other categories or products) are treated uniformly. Implement the Composite pattern for displaying product details.

Exercise 2: Menu System

- **Objective**: Implement a menu system where both individual menu items (e.g., `Dish` , `Drink`) and menu categories (e.g., `Appetizers` , `Main Course`) are treated uniformly. The system should be able to display all items and categories in the menu.

Exercise 3: File Management System

- **Objective**: Design a file management system where files and directories are treated uniformly. Implement operations like listing files, adding new files, and deleting files using the Composite pattern.

Exercise 4: Tree Data Structure

- **Objective**: Implement a tree structure for representing organizational charts or family trees. Each node can represent either an individual person or a group of people (such as a department or family).

Exercise 5: Graphic Editor

- **Objective**: Create a graphic editor where different shapes (e.g., circles, rectangles) and groups of shapes (e.g., a picture frame) are treated uniformly. Implement operations like `draw()`, `resize()`, and `move()` using the Composite pattern.

Java Classes that Use Composite Design Pattern

1. `javax.swing.JMenu` (Java Swing)

The `JMenu` class in Java Swing is an example of a composite design pattern where individual menu items and submenus (which can contain other menu items or submenus) are treated uniformly.

```java
import javax.swing.*;

public class CompositePatternExample {
    public static void main(String[] args) {
        JMenu fileMenu = new JMenu("File");
        JMenuItem newItem = new JMenuItem("New");
        JMenuItem openItem = new JMenuItem("Open");
        fileMenu.add(newItem);
        fileMenu.add(openItem);

        JMenu editMenu = new JMenu("Edit");
        JMenuItem cutItem = new JMenuItem("Cut");
        JMenuItem copyItem = new JMenuItem("Copy");
        editMenu.add(cutItem);
        editMenu.add(copyItem);

        JMenuBar menuBar = new JMenuBar();
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        JFrame frame = new JFrame("Menu Example");
        frame.setJMenuBar(menuBar);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation

(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Explanation:

- `JMenu` can contain both individual `JMenuItem` objects and other `JMenu` objects (submenus), demonstrating the Composite pattern where both menu items and submenus are treated uniformly.

2. **`java.awt.Container` (Java AWT)**

`Container` in AWT (Abstract Window Toolkit) is a composite class that can hold other components, such as buttons or text fields, which are treated uniformly.

Behavioural

Observer Design Pattern

The Observer pattern is a behavioral design pattern where an object, called the *subject*, maintains a list of its dependents, called *observers*, and notifies them of any state changes, usually by calling one of their methods. It is widely used in implementing distributed event-handling systems.

Example 1: Stock Price Tracker

- **Description**: A subject (`Stock`) updates its observers (e.g., `Investor` , `Broker`) whenever the stock price changes.

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

public interface Observer {
    void update(String stockName, double stockPrice);
}

public class Stock {
    private String name;
    private double price;
    private List<Observer> observers = new ArrayList<>();

    public Stock(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setPrice(double price) {
        this.price = price;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(name, price);
        }
    }
}

public class Investor implements Observer {
    private String name;

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

    @Override
    public void update(String stockName, double stockPrice) {
        System.out.println("Investor " + name + " notified. Stock: " + stockName + ", Price: "
    }
}

// Usage:
public class StockMarket {
    public static void main(String[] args) {
        Stock googleStock = new Stock("Google", 1500.0);
        Investor alice = new Investor("Alice");
        Investor bob = new Investor("Bob");

        googleStock.addObserver(alice);
        googleStock.addObserver(bob);

        googleStock.setPrice(1550.0);
        googleStock.setPrice(1600.0);
    }
}
```

Here are examples and exercises for the **Observer Design Pattern** in Java, along with code snippets demonstrating its implementation.

Example 2: Weather Station

A weather station (subject) notifies its observers (e.g., display devices) whenever there's a change in weather conditions.

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

public interface Observer {
    void update(float temperature, float humidity);
}

public class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;
    private float humidity;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    public void setMeasurements(float temperature, float humidity) {
        this.temperature = temperature;
        this.humidity = humidity;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity);
        }
    }
}

public class WeatherDisplay implements Observer {
    @Override
    public void update(float temperature, float humidity) {
        System.out.println("Weather Updated: Temp=" + temperature + "°C, Humidity=" + humidity
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();
        WeatherDisplay display = new WeatherDisplay();
        station.addObserver(display);
        station.setMeasurements(25.5f, 65.0f);
        station.setMeasurements(22.3f, 70.0f);
    }
}
```

Example 3: Stock Market

A stock price tracker (subject) notifies its investors (observers) when stock prices change.

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

public interface StockObserver {
    void update(String stock, float price);
}

public class StockMarket {
    private List<StockObserver> observers = new ArrayList<>();
    private String stockName;
    private float price;

    public void addObserver(StockObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(StockObserver observer) {
        observers.remove(observer);
    }

    public void setStockPrice(String stock, float price) {
        this.stockName = stock;
        this.price = price;
        notifyObservers();
    }

    private void notifyObservers() {
        for (StockObserver observer : observers) {
            observer.update(stockName, price);
        }
    }
}

public class Investor implements StockObserver {
    private String name;

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

    @Override
    public void update(String stock, float price) {
        System.out.println(name + " notified. Stock: " + stock + ", Price: " + price);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        StockMarket market = new StockMarket();
        Investor investor1 = new Investor("Alice");
        Investor investor2 = new Investor("Bob");

        market.addObserver(investor1);
        market.addObserver(investor2);

        market.setStockPrice("AAPL", 150.0f);
        market.setStockPrice("GOOGL", 2800.0f);
    }
}
```

Example 4: Chat Application

A chatroom notifies its users when a new message is posted.

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

public interface ChatObserver {
    void update(String message);
}

public class ChatRoom {
    private List<ChatObserver> observers = new ArrayList<>();
    public void join(ChatObserver observer) {
        observers.add(observer);
    }

    public void leave(ChatObserver observer) {
        observers.remove(observer);
    }

    public void sendMessage(String message) {
        notifyObservers(message);
    }

    private void notifyObservers(String message) {
        for (ChatObserver observer : observers) {
            observer.update(message);
        }
    }
}

public class User implements ChatObserver {
    private String username;

    public User(String username) {
        this.username = username;
    }

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

// Usage
public class Main {
    public static void main(String[] args) {
        ChatRoom chatRoom = new ChatRoom();
        User user1 = new User("Alice");
        User user2 = new User("Bob");

        chatRoom.join(user1);
        chatRoom.join(user2);

        chatRoom.sendMessage("Hello, everyone!");
    }
}
```

Exercises for Observer Design Pattern

1. **Weather Alert System**
   Implement a weather alert system where different devices (e.g., phone, TV, radio) get notified of weather warnings.

2. **News Publisher**
   Design a news publisher that sends updates to its subscribers when a new article is published.

3. **Real-Time Game Score Updates**
   Create a program where fans get notified of live updates for their favorite sports teams.

4. **Social Media Notifications**
   Implement a social media application where users are notified when their followed accounts post new content.

5. **Traffic Monitoring System**
   Build a traffic monitoring system that notifies users of traffic conditions in their area.

State Design Pattern

The **State Design Pattern** is used to encapsulate varying behavior for the same object based on its internal state. By defining states as separate classes and delegating state-specific behavior to these classes, the pattern makes state transitions explicit and manageable.

Key Benefits of the State Design Pattern

1. **Improved Code Organization**: State-specific behavior is encapsulated in separate classes, reducing clutter in the main context class.
2. **Easier to Add/Modify States**: Adding new states or modifying behavior is straightforward and doesn't affect the context class or other states.
3. **Encapsulation of State Logic**: State transitions and behaviors are encapsulated, making the code easier to understand and maintain.
4. **Eliminates Complex Conditionals**: Replaces large `if-else` or `switch-case` statements with polymorphism.
5. **Improved Scalability**: Supports complex state transitions and behaviors without degrading code quality.

Example 1: Media Player

A media player switches between states like Playing, Paused, and Stopped.

```java
public interface MediaPlayerState {
    void pressPlay(MediaPlayerContext context);
}

public class PlayingState implements MediaPlayerState {
    @Override
    public void pressPlay(MediaPlayerContext context) {
        System.out.println("Pausing the playback.");
        context.setState(new PausedState());
    }
}

public class PausedState implements MediaPlayerState {
    @Override
    public void pressPlay(MediaPlayerContext context) {
        System.out.println("Resuming playback.");
        context.setState(new PlayingState());
    }
}

public class StoppedState implements MediaPlayerState {
    @Override
    public void pressPlay(MediaPlayerContext context) {
        System.out.println("Starting playback.");
        context.setState(new PlayingState());
    }
}

public class MediaPlayerContext {
    private MediaPlayerState state;

    public MediaPlayerContext() {
        state = new StoppedState(); // Default state
    }

    public void setState(MediaPlayerState state) {
        this.state = state;
    }

    public void pressPlay() {
        state.pressPlay(this);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        MediaPlayerContext player = new MediaPlayerContext();
        player.pressPlay(); // Starting playback
        player.pressPlay(); // Pausing playback
        player.pressPlay(); // Resuming playback
    }
}
```

Example 2: Traffic Signal System

A traffic signal transitions between Red, Yellow, and Green states.

```java
public interface TrafficSignalState {
    void changeSignal(TrafficSignalContext context);
}

public class RedSignal implements TrafficSignalState {
    @Override
    public void changeSignal(TrafficSignalContext context) {
        System.out.println("Changing to Green.");
        context.setState(new GreenSignal());
    }
}

public class GreenSignal implements TrafficSignalState {
    @Override
    public void changeSignal(TrafficSignalContext context) {
        System.out.println("Changing to Yellow.");
        context.setState(new YellowSignal());
    }
}

public class YellowSignal implements TrafficSignalState {
    @Override
    public void changeSignal(TrafficSignalContext context) {
        System.out.println("Changing to Red.");
        context.setState(new RedSignal());
    }
}

public class TrafficSignalContext {
    private TrafficSignalState state;

    public TrafficSignalContext() {
        state = new RedSignal(); // Initial state
    }
    public void setState(TrafficSignalState state) {
        this.state = state;
    }

    public void changeSignal() {
        state.changeSignal(this);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        TrafficSignalContext signal = new TrafficSignalContext();
        signal.changeSignal(); // Changing to Green
        signal.changeSignal(); // Changing to Yellow
        signal.changeSignal(); // Changing to Red
    }
}
```

Example 3: ATM Machine

An ATM operates in different states like Idle, Card Inserted, and Transaction Completed.

```java
public interface ATMState {
    void handleRequest(ATMContext context);
}

public class IdleState implements ATMState {
    @Override
    public void handleRequest(ATMContext context) {
        System.out.println("Please insert your card.");
        context.setState(new CardInsertedState());
    }
}

public class CardInsertedState implements ATMState {
    @Override
    public void handleRequest(ATMContext context) {
        System.out.println("Processing your request...");
        context.setState(new TransactionCompletedState());
    }
}

public class TransactionCompletedState implements ATMState {
    @Override
    public void handleRequest(ATMContext context) {
        System.out.println("Transaction completed. Thank you!");
        context.setState(new IdleState());
    }
}

public class ATMContext {
    private ATMState state;

    public ATMContext() {
        state = new IdleState(); // Default state
    }

    public void setState(ATMState state) {
        this.state = state;
    }

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

// Usage
public class Main {
    public static void main(String[] args) {
        ATMContext atm = new ATMContext();
        atm.handleRequest(); // Insert card
        atm.handleRequest(); // Processing request
        atm.handleRequest(); // Transaction completed
    }
}
```

Exercises for State Design Pattern

1. **Order Lifecycle**
   Implement an order management system with states like New Order, Processing, and Delivered.
2. **Elevator System**
   Create an elevator with states such as Moving Up, Moving Down, and Idle.
3. **Printer State**
   Design a printer system with states like Ready, Printing, and Out of Paper.
4. **E-Commerce Cart**

Create an e-commerce cart with states like Empty, Active, and Checked Out.

5. **Game Character Behavior**
   Implement a character in a game with states like Idle, Running, and Jumping.

Java Classes that Use State Design Pattern

1. `javax.faces.lifecycle.Lifecycle`

- **Description**: Manages different states of the JSF lifecycle (Restore View, Apply Request Values, etc.).
- **Snippet**:

```java
Lifecycle lifecycle = FacesContext.getCurrentInstance().getApplication().getLifecycle();
lifecycle.execute(context);
```

2. `javax.swing.JComponent`

- **Description**: Swing components use internal state transitions for enabling/disabling or selection events.
- **Snippet**:

```java
JButton button = new JButton("Click Me");
button.setEnabled(false); // Change state
button.setEnabled(true);  // Change state
```

3. `java.nio.channels.Selector`

- **Description**: Selector transitions between states of readiness for different channels.
- **Snippet**:

```java
Selector selector = Selector.open();
Set<SelectionKey> selectedKeys = selector.selectedKeys();
for (SelectionKey key : selectedKeys) {
    if (key.isReadable()) {
        System.out.println("Channel is ready to read.");
    }
}
```

Strategy Design Pattern

The **Strategy Design Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. This pattern allows the algorithm to vary independently from the clients that use it.

Key Benefits of the Strategy Design Pattern

1. **Flexibility**: Easily switch between different algorithms or strategies at runtime.
2. **Code Reusability**: Encapsulation of algorithms into separate classes promotes code reuse.
3. **Eliminates Conditional Statements**: Replaces complex `if-else` or `switch-case` logic with polymorphism.
4. **Open/Closed Principle**: Adding new strategies does not modify existing code, adhering to the Open/Closed principle.
5. **Improved Maintainability**: Separating behaviors into distinct classes makes the code easier to understand and maintain.

Example 1: Payment Processing

```java
// Strategy Interface
public interface PaymentStrategy {
    void pay(int amount);
}

// Concrete Strategies
public class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

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

// Context
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

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

// Usage
public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(100);

        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(200);
    }
}
```

Example 2: Compression Algorithms

```
// Strategy Interface
public interface CompressionStrategy {
    void compress(String file);
}

// Concrete Strategies
public class ZipCompression implements CompressionStrategy {
    @Override
    public void compress(String file) {
        System.out.println("Compressing " + file + " using ZIP compression.");
    }
}

public class RarCompression implements CompressionStrategy {
    @Override
    public void compress(String file) {
        System.out.println("Compressing " + file + " using RAR compression.");
    }
}

// Context
public class FileCompressor {
```

```java
    private CompressionStrategy strategy;

    public void setCompressionStrategy(CompressionStrategy strategy) {
        this.strategy = strategy;
    }

    public void compress(String file) {
        strategy.compress(file);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        FileCompressor compressor = new FileCompressor();

        compressor.setCompressionStrategy(new ZipCompression());
        compressor.compress("document.txt");

        compressor.setCompressionStrategy(new RarCompression());
        compressor.compress("image.jpg");
    }
}
```

Example 3: Sorting Algorithms

```java
// Strategy Interface
public interface SortingStrategy {
    void sort(int[] numbers);
}

// Concrete Strategies
public class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        System.out.println("Sorting using Bubble Sort.");
        // Implementation of Bubble Sort
    }
}

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

// Context
public class Sorter {
    private SortingStrategy strategy;

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

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

// Usage
public class Main {
    public static void main(String[] args) {
        Sorter sorter = new Sorter();

        sorter.setSortingStrategy(new BubbleSort());
        sorter.sort(new int[] {5, 3, 8, 2});

        sorter.setSortingStrategy(new QuickSort());
        sorter.sort(new int[] {5, 3, 8, 2});
    }
}
```

Exercises for Strategy Design Pattern

1. **Text Formatting**
   Implement a text editor with strategies for formatting text (e.g., Uppercase, Lowercase, Capitalize).

2. **Travel Modes**
   Create a travel planner with strategies like Car, Bus, and Bike, each calculating travel time differently.

3. **Tax Calculation**
   Implement a tax calculator with different strategies for regions or tax regimes.

4. **Authentication Mechanism**
   Create an authentication system with strategies like OAuth, SAML, and JWT.

5. **Discount Calculation**
   Design a shopping system that applies different discount strategies (e.g., Percentage Discount, Buy One Get One).

Java Classes That Use Strategy Design Pattern

1. `java.util.Comparator`

- **Description**: The `Comparator` interface allows sorting using different strategies.
- **Snippet**:

```java
import java.util.*;

public class StrategyComparatorExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 2);

        // Strategy 1: Ascending Order
        numbers.sort((a, b) -> a - b);
        System.out.println("Ascending: " + numbers);

        // Strategy 2: Descending Order
        numbers.sort((a, b) -> b - a);
        System.out.println("Descending: " + numbers);
    }
}
```

2. `javax.servlet.http.HttpServlet`

- **Description**: Servlets use different strategies (`doGet` , `doPost` , etc.) to handle HTTP requests.
- **Snippet**:

```java
import javax.servlet.http.*;

public class StrategyServletExample extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
        resp.getWriter().write("GET request handled.");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) {
        resp.getWriter().write("POST request handled.");
    }
}
```

3. **`javax.crypto.Cipher`**

- **Description**: The `Cipher` class uses different encryption strategies (e.g., AES, DES).
- **Snippet**:

```java
import javax.crypto.Cipher;

public class StrategyCipherExample {
    public static void main(String[] args) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        // Use cipher for encryption/decryption based on strategy
        System.out.println("Cipher initialized with AES.");
    }
}
```

Chain of Responsibility Design Pattern

The **Chain of Responsibility Design Pattern** allows a request to be passed along a chain of handlers until it is processed by an appropriate handler. Each handler in the chain decides whether to handle the request or pass it to the next handler.

Key Benefits of the Chain of Responsibility Design Pattern

1. **Decoupling of Sender and Receiver**: The sender has no knowledge of which handler will process the request.
2. **Flexibility**: Handlers can be added, removed, or reordered dynamically without affecting the client.
3. **Responsibility Sharing**: Multiple handlers can contribute to handling a request, enhancing modularity.
4. **Open/Closed Principle**: New handlers can be added without modifying existing code.
5. **Dynamic Behavior**: The chain can process requests at runtime, enabling dynamic configurations.

Example 1: Logging System

```java
public abstract class Logger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;
    protected Logger nextLogger;

    public void setNextLogger(Logger nextLogger) {
        this.nextLogger = nextLogger;
    }

    public void logMessage(int level, String message) {
        if (this.level <= level) {
            write(message);
        }
        if (nextLogger != null) {
            nextLogger.logMessage(level, message);
        }
    }

    protected abstract void write(String message);
}

public class ConsoleLogger extends Logger {
    public ConsoleLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Console Logger: " + message);
    }
}

public class FileLogger extends Logger {
    public FileLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("File Logger: " + message);
    }
}
```

```java
public class ErrorLogger extends Logger {
    public ErrorLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("Error Logger: " + message);
    }
}

// Usage
public class Main {
    private static Logger getChainOfLoggers() {
        Logger errorLogger = new ErrorLogger(Logger.ERROR);
        Logger fileLogger = new FileLogger(Logger.DEBUG);
        Logger consoleLogger = new ConsoleLogger(Logger.INFO);

        errorLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(consoleLogger);

        return errorLogger;
    }

    public static void main(String[] args) {
        Logger loggerChain = getChainOfLoggers();

        loggerChain.logMessage(Logger.INFO, "This is an information.");
        loggerChain.logMessage(Logger.DEBUG, "This is a debug level message.");
        loggerChain.logMessage(Logger.ERROR, "This is an error message.");
    }
}
```

Example 2: Request Processing

```java
public abstract class RequestHandler {
    protected RequestHandler nextHandler;

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

    public void handleRequest(String request) {
        if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

public class AuthenticationHandler extends RequestHandler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("Authenticated")) {
            System.out.println("Authentication successful.");
        } else {
            System.out.println("Authentication failed.");
        }
        super.handleRequest(request);
    }
}

public class AuthorizationHandler extends RequestHandler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("Authorized")) {
            System.out.println("Authorization successful.");
        } else {
            System.out.println("Authorization failed.");
        }
        super.handleRequest(request);
    }
}

public class ValidationHandler extends RequestHandler {
    @Override
    public void handleRequest(String request) {
        if (request.equals("Valid")) {
            System.out.println("Validation successful.");
        } else {
            System.out.println("Validation failed.");
        }
        super.handleRequest(request);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        RequestHandler authHandler = new AuthenticationHandler();
        RequestHandler authzHandler = new AuthorizationHandler();
        RequestHandler validationHandler = new ValidationHandler();

        authHandler.setNextHandler(authzHandler);
        authzHandler.setNextHandler(validationHandler);

        authHandler.handleRequest("Valid");
    }
}
```

Example 3: ATM Dispensing Machine

```
public abstract class ATMHandler {
    protected ATMHandler nextHandler;

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

    public abstract void dispense(int amount);
}

public class HundredDispenser extends ATMHandler {
    @Override
    public void dispense(int amount) {
        if (amount >= 100) {
            int num = amount / 100;
            int remainder = amount % 100;
            System.out.println("Dispensing " + num + " hundred dollar notes.");
            if (remainder != 0 && nextHandler != null) {
                nextHandler.dispense(remainder);
            }
        } else if (nextHandler != null) {
            nextHandler.dispense(amount);
        }
    }
}

public class FiftyDispenser extends ATMHandler {
    @Override
    public void dispense(int amount) {
        if (amount >= 50) {
            int num = amount / 50;
            int remainder = amount % 50;
            System.out.println("Dispensing " + num + " fifty dollar notes.");
            if (remainder != 0 && nextHandler != null) {
                nextHandler.dispense(remainder);
            }
        } else if (nextHandler != null) {
            nextHandler.dispense(amount);
        }
    }
}

public class TwentyDispenser extends ATMHandler {
    @Override
    public void dispense(int amount) {
        if (amount >= 20) {
            int num = amount / 20;
            int remainder = amount % 20;
            System.out.println("Dispensing " + num + " twenty dollar notes.");
            if (remainder != 0 && nextHandler != null) {
                nextHandler.dispense(remainder);
            }
        } else if (nextHandler != null) {
            nextHandler.dispense(amount);
        }
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        ATMHandler hundredDispenser = new HundredDispenser();
        ATMHandler fiftyDispenser = new FiftyDispenser();
        ATMHandler twentyDispenser = new TwentyDispenser();

        hundredDispenser.setNextHandler(fiftyDispenser);
        fiftyDispenser.setNextHandler(twentyDispenser);

        int amount = 380;
        System.out.println("Requesting $" + amount + " withdrawal.");
        hundredDispenser.dispense(amount);
    }
}
```

Exercises for Chain of Responsibility Design Pattern

1. **Form Validation**
   Create a form validation system where different validators handle specific fields (e.g., Email, Password, Username).

2. **Workflow Automation**
   Implement a workflow system where tasks are handled by specific departments (e.g., HR, Finance,

IT) based on their type.

3. **Customer Support System**
   Develop a support system where requests are escalated through levels of support (e.g., Level 1, Level 2, Manager).

4. **Spam Filter**
   Build an email system where emails are filtered through different stages (e.g., Spam Detection, Priority Classification, Tagging).

5. **Event Handling**
   Create an event-handling system where specific handlers respond to certain types of user events (e.g., Mouse Click, Keyboard Press).

Java Classes That Use Chain of Responsibility Design Pattern

1. **`java.util.logging.Logger`**

- **Description**: Java's logging framework uses a chain of `Logger` objects to process log messages at different levels.
- **Snippet**:

```java
import java.util.logging.*;

public class LoggerExample {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger("ChainLogger");

        ConsoleHandler consoleHandler = new ConsoleHandler();
        FileHandler fileHandler;
        try {
            fileHandler = new FileHandler("app.log");
            logger.addHandler(consoleHandler);
            logger.addHandler(fileHandler);
        } catch (Exception e) {
            e.printStackTrace();
        }

        logger.setLevel(Level.ALL);
        logger.info("This is an info message.");
    }
}
```

2. `javax.servlet.Filter`

- **Description**: Filters in Java Servlets are processed in a chain to handle HTTP requests and responses.
- **Snippet**:

```java
public class AuthenticationFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        System.out.println("Authentication Filter executed.");
        chain.doFilter(request, response);
    }
}
```

Command Design Pattern

The **Command Design Pattern** encapsulates a request as an object, allowing for parameterization of clients with different requests, queuing of requests, and logging of requests. It decouples the invoker of a request from the object that executes the request.

Key Benefits of the Command Design Pattern

1. **Decoupling**: Separates the sender (Invoker) and receiver (Handler) of requests.
2. **Undo/Redo Functionality**: Commands can be easily stored for undo or redo operations.
3. **Command Queueing**: Enables requests to be queued or scheduled.
4. **Dynamic Behavior**: Commands can be created dynamically at runtime.
5. **Adherence to Open/Closed Principle**: Adding new commands does not modify existing code.

Example 1: Remote Control System

```java
// Command Interface
public interface Command {
    void execute();
}

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

// Concrete Commands
public class TurnOnLightCommand implements Command {
    private Light light;

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

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

public class TurnOffLightCommand implements Command {
    private Light light;

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

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

// Invoker
public class RemoteControl {
    private Command command;

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

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

// Usage
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command turnOn = new TurnOnLightCommand(light);
        Command turnOff = new TurnOffLightCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(turnOn);
        remote.pressButton();

        remote.setCommand(turnOff);
        remote.pressButton();
    }
}
```

Example 2: Text Editor with Undo

```
// Command Interface
public interface Command {
    void execute();
    void undo();
}

// Receiver
public class TextEditor {
    private StringBuilder text = new StringBuilder();

    public void write(String words) {
        text.append(words);
    }

    public void eraseLast(int length) {
        if (text.length() >= length) {
            text.delete(text.length() - length, text.length());
        }
    }

    public String getText() {
        return text.toString();
    }
}

// Concrete Command
public class WriteCommand implements Command {
    private TextEditor editor;
```

```java
    private String text;

    public WriteCommand(TextEditor editor, String text) {
        this.editor = editor;
        this.text = text;
    }

    @Override
    public void execute() {
        editor.write(text);
    }

    @Override
    public void undo() {
        editor.eraseLast(text.length());
    }
}

// Invoker
public class EditorInvoker {
    private Command command;

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

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

    public void undoCommand() {
        command.undo();
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        WriteCommand writeCommand = new WriteCommand(editor, "Hello World!");

        EditorInvoker invoker = new EditorInvoker();
        invoker.setCommand(writeCommand);

        invoker.executeCommand();
        System.out.println("Text after execution: " + editor.getText());

        invoker.undoCommand();
        System.out.println("Text after undo: " + editor.getText());
    }
}
```

Example 3: Job Scheduling

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

// Receiver
public class PrintJob {
    public void print() {
        System.out.println("Printing document...");
    }
}

public class EmailJob {
    public void sendEmail() {
        System.out.println("Sending email...");
    }
}

// Concrete Commands
public class PrintCommand implements Command {
    private PrintJob printJob;

    public PrintCommand(PrintJob printJob) {
        this.printJob = printJob;
    }

    @Override
    public void execute() {
        printJob.print();
    }
}

public class EmailCommand implements Command {
    private EmailJob emailJob;

    public EmailCommand(EmailJob emailJob) {
        this.emailJob = emailJob;
    }
    @Override
    public void execute() {
        emailJob.sendEmail();
    }
}

// Invoker
public class JobScheduler {
    private List<Command> commands = new ArrayList<>();

    public void addJob(Command command) {
        commands.add(command);
    }

    public void executeJobs() {
        for (Command command : commands) {
            command.execute();
        }
        commands.clear();
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        PrintJob printJob = new PrintJob();
        EmailJob emailJob = new EmailJob();

        Command printCommand = new PrintCommand(printJob);
        Command emailCommand = new EmailCommand(emailJob);

        JobScheduler scheduler = new JobScheduler();
        scheduler.addJob(printCommand);
        scheduler.addJob(emailCommand);

        scheduler.executeJobs();
    }
}
```

Exercises for Command Design Pattern

1. **Home Automation System**
   Implement a system for controlling multiple devices (e.g., Lights, Fans, Air Conditioners) using a command pattern.

2. **Task Queue Manager**
   Create a task manager that allows adding, executing, and clearing tasks from a queue.
3. **Media Player Controller**
   Implement a media player with commands for Play, Pause, Stop, and Rewind.
4. **Banking Transactions**
   Design a system for handling banking operations like Deposit, Withdraw, and Transfer, with undo functionality.
5. **Game Command System**
   Build a command system for a game where players can perform actions like Move, Jump, Attack, and Undo.

Java Classes That Use Command Design Pattern

1. `javax.swing.Action`

- **Description**: The `Action` interface in Swing encapsulates a command that can be attached to UI components like buttons and menu items.
- **Snippet**:

```java
import javax.swing.*;
import java.awt.event.ActionEvent;

public class CommandExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Command Pattern");
        JButton button = new JButton();

        Action action = new AbstractAction("Click Me") {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        };

        button.setAction(action);
        frame.add(button);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

2. `java.lang.Runnable`

- **Description**: The `Runnable` interface represents a command that encapsulates a unit of work to be executed by a thread.
- **Snippet**:

```java
public class RunnableExample {
    public static void main(String[] args) {
        Runnable command = () -> System.out.println("Executing Runnable command.");
        Thread thread = new Thread(command);
        thread.start();
    }
}
```

Iterator Design Pattern

The **Iterator Design Pattern** provides a way to access elements of a collection sequentially without exposing the underlying representation. It is widely used in the implementation of collection classes in Java.

Key Benefits of the Iterator Design Pattern

1. **Encapsulation**: It abstracts the traversal of the collection from its implementation.
2. **Consistency**: Provides a uniform way to iterate over different types of collections.
3. **Flexibility**: The iteration logic can be customized without modifying the collection.
4. **Separation of Concerns**: Keeps the collection and traversal responsibilities separate.
5. **Support for Multiple Iterations**: Enables multiple iterators to operate on the same collection simultaneously.

Example 1: Custom String Iterator

```java
// Iterator Interface
public interface Iterator<T> {
    boolean hasNext();
    T next();
```

```
}

// Collection Interface
public interface IterableCollection<T> {
    Iterator<T> createIterator();
}

// Concrete Collection
public class StringCollection implements IterableCollection<String> {
    private String[] items;
    private int size;

    public StringCollection(int capacity) {
        items = new String[capacity];
        size = 0;
    }

    public void add(String item) {
        if (size < items.length) {
            items[size++] = item;
        }
    }

    @Override
    public Iterator<String> createIterator() {
        return new StringIterator(items, size);
    }
}

// Concrete Iterator
public class StringIterator implements Iterator<String> {
    private String[] items;
    private int position;
    private int size;

    public StringIterator(String[] items, int size) {
        this.items = items;
        this.size = size;
        this.position = 0;
    }

    @Override
    public boolean hasNext() {
        return position < size;
    }

    @Override
    public String next() {
        return items[position++];
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        StringCollection collection = new StringCollection(5);
        collection.add("One");
        collection.add("Two");
        collection.add("Three");

        Iterator<String> iterator = collection.createIterator();

        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Example 2: Number Collection Iterator

```
// Concrete Collection
public class NumberCollection {
    private List<Integer> numbers = new ArrayList<>();

    public void add(int number) {
        numbers.add(number);
    }

    public Iterator<Integer> getIterator() {
        return numbers.iterator();
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        NumberCollection collection = new NumberCollection();
        collection.add(10);
        collection.add(20);
        collection.add(30);

        Iterator<Integer> iterator = collection.getIterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Example 3: Custom List with Reverse Iterator

```
// Concrete Collection with Reverse Iterator
public class CustomList<T> {
    private List<T> items = new ArrayList<>();

    public void add(T item) {
        items.add(item);
    }

    public Iterator<T> iterator() {
        return items.iterator();
    }

    public Iterator<T> reverseIterator() {
        return new Iterator<>() {
            private int index = items.size() - 1;

            @Override
            public boolean hasNext() {
                return index >= 0;
            }

            @Override
            public T next() {
                return items.get(index--);
            }
        };
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        CustomList<String> list = new CustomList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        System.out.println("Forward Iteration:");
        Iterator<String> forward = list.iterator();
        while (forward.hasNext()) {
            System.out.println(forward.next());
        }

        System.out.println("Reverse Iteration:");
        Iterator<String> reverse = list.reverseIterator();
        while (reverse.hasNext()) {
            System.out.println(reverse.next());
        }
    }
}
```

Exercises for Iterator Design Pattern

1. **Design a Playlist Iterator**
   Create a playlist class where you can iterate over songs in the order they were added and in reverse.

2. **Binary Tree Iterator**
   Implement an iterator to traverse a binary tree using in-order, pre-order, or post-order traversal.

3. **Shopping Cart Iterator**
   Develop a shopping cart class where you can iterate through items to calculate the total cost.

4. **Filter Iterator**
   Create an iterator that filters items based on a condition (e.g., iterate only through even numbers in a list).

5. **Custom Iterator for a Matrix**
   Build an iterator to traverse a 2D matrix row by row or column by column.

Java Classes That Use Iterator Design Pattern

1. **`java.util.Iterator`**

- **Description**: The `Iterator` interface is widely used in Java collections like `ArrayList`, `HashSet`, and `HashMap`.
- **Snippet**:

```
import java.util.*;
```

```java
public class IteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

2. **`java.util.Enumeration`**

- **Description**: The `Enumeration` interface is an older version of the iterator used in legacy collections like `Vector`.
- **Snippet**:

```java
import java.util.*;

public class EnumerationExample {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.add("One");
        vector.add("Two");
        vector.add("Three");

        Enumeration<String> enumeration = vector.elements();
        while (enumeration.hasMoreElements()) {
            System.out.println(enumeration.nextElement());
        }
    }
}
```

3. **`java.util.Spliterator`**

- **Description**: The `Spliterator` is used for parallel iteration over elements in a collection.
- **Snippet**:

```java
import java.util.*;
```

```
public class SpliteratorExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("Alpha", "Beta", "Gamma");

        Spliterator<String> spliterator = list.spliterator();
        spliterator.forEachRemaining(System.out::println);
    }
}
```

Mediator Design Pattern

The **Mediator Design Pattern** defines an object that encapsulates how a set of objects interact, promoting loose coupling by preventing objects from referencing each other directly. Instead, communication is centralized through a mediator object.

Key Benefits of the Mediator Design Pattern

1. **Loose Coupling**: Reduces direct dependencies between interacting objects.
2. **Centralized Communication**: Simplifies the communication logic by consolidating it into a single mediator.
3. **Enhanced Maintenance**: Adding or modifying interactions is easier since changes are confined to the mediator.
4. **Scalability**: Supports addition of new objects without affecting existing ones.
5. **Improved Readability**: Organizes communication, making the system easier to understand.

Example 1: Chat Room

```
// Mediator Interface
public interface ChatMediator {
    void sendMessage(String message, User user);
    void addUser(User user);
}

// Concrete Mediator
public class ChatRoom implements ChatMediator {
    private List<User> users = new ArrayList<>();

    @Override
    public void sendMessage(String message, User user) {
        for (User u : users) {
```

```java
            if (u != user) {
                u.receive(message);
            }
        }
    }

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

// Colleague
public abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }

    public abstract void send(String message);

    public abstract void receive(String message);
}

// Concrete Colleague
public class ConcreteUser extends User {
    public ConcreteUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }

    @Override
    public void send(String message) {
        System.out.println(this.name + ": Sending Message = " + message);
        mediator.sendMessage(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println(this.name + ": Received Message = " + message);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        ChatMediator mediator = new ChatRoom();

        User user1 = new ConcreteUser(mediator, "Alice");
        User user2 = new ConcreteUser(mediator, "Bob");
        User user3 = new ConcreteUser(mediator, "Charlie");

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

        user1.send("Hello, everyone!");
    }
}
```

Example 2: Traffic Light Controller

```java
// Mediator Interface
public interface TrafficMediator {
    void notify(String event, TrafficLight trafficLight);
}

// Concrete Mediator
public class TrafficController implements TrafficMediator {
    private TrafficLight northSouth;
    private TrafficLight eastWest;

    public void setNorthSouth(TrafficLight northSouth) {
        this.northSouth = northSouth;
    }

    public void setEastWest(TrafficLight eastWest) {
        this.eastWest = eastWest;
    }

    @Override
    public void notify(String event, TrafficLight trafficLight) {
        if (trafficLight == northSouth) {
            eastWest.setGreen(false);
        } else if (trafficLight == eastWest) {
            northSouth.setGreen(false);
        }
    }
}

// Colleague
```

```java
public class TrafficLight {
    private TrafficMediator mediator;
    private boolean isGreen;

    public TrafficLight(TrafficMediator mediator) {
        this.mediator = mediator;
    }

    public void setGreen(boolean isGreen) {
        this.isGreen = isGreen;
        if (isGreen) {
            mediator.notify("green", this);
        }
    }

    public boolean isGreen() {
        return isGreen;
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        TrafficController mediator = new TrafficController();
        TrafficLight northSouth = new TrafficLight(mediator);
        TrafficLight eastWest = new TrafficLight(mediator);

        mediator.setNorthSouth(northSouth);
        mediator.setEastWest(eastWest);

        northSouth.setGreen(true);
        System.out.println("North-South is green: " + northSouth.isGreen());
        System.out.println("East-West is green: " + eastWest.isGreen());
    }
}
```

Example 3: Flight Control System

```java
// Mediator Interface
public interface FlightControlMediator {
    void requestLanding(Aircraft aircraft);
    void notifyLandingCompleted(Aircraft aircraft);
}

// Concrete Mediator
```

```java
public class AirTrafficControl implements FlightControlMediator {
    private Queue<Aircraft> landingQueue = new LinkedList<>();

    @Override
    public void requestLanding(Aircraft aircraft) {
        landingQueue.add(aircraft);
        System.out.println(aircraft.getName() + " is requesting landing.");
        if (landingQueue.size() == 1) {
            aircraft.land();
        }
    }

    @Override
    public void notifyLandingCompleted(Aircraft aircraft) {
        landingQueue.poll();
        System.out.println(aircraft.getName() + " has landed.");
        if (!landingQueue.isEmpty()) {
            landingQueue.peek().land();
        }
    }
}

// Colleague
public class Aircraft {
    private String name;
    private FlightControlMediator mediator;

    public Aircraft(String name, FlightControlMediator mediator) {
        this.name = name;
        this.mediator = mediator;
    }

    public String getName() {
        return name;
    }

    public void requestLanding() {
        mediator.requestLanding(this);
    }

    public void land() {
        System.out.println(name + " is landing...");
        mediator.notifyLandingCompleted(this);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        FlightControlMediator mediator = new AirTrafficControl();

        Aircraft aircraft1 = new Aircraft("Flight A1", mediator);
        Aircraft aircraft2 = new Aircraft("Flight B2", mediator);

        aircraft1.requestLanding();
        aircraft2.requestLanding();
    }
}
```

Exercises for Mediator Design Pattern

1. **Smart Home System**
   Implement a mediator to manage interactions between devices like lights, thermostat, and security system in a smart home.

2. **Notification Center**
   Create a notification system where different apps (email, SMS, push notifications) communicate through a mediator.

3. **Online Auction Platform**
   Design an auction system where the mediator handles bids from multiple participants.

4. **Library Management System**
   Implement a system where a mediator manages interactions between books, borrowers, and librarians.

5. **Job Portal System**
   Create a job portal where the mediator connects employers and job seekers.

Java Classes That Use Mediator Design Pattern

1. `java.util.Timer`

- **Description**: The `Timer` class acts as a mediator between tasks that need to be executed at specific times.
- **Snippet**:

```java
import java.util.Timer;
import java.util.TimerTask;
```

```java
public class TimerMediatorExample {
    public static void main(String[] args) {
        Timer timer = new Timer();

        timer.schedule(new TimerTask() {
            @Override
            public void run() {
                System.out.println("Task executed!");
            }
        }, 1000);

        System.out.println("Task scheduled.");
    }
}
```

2. **`java.util.concurrent.ExecutorService`**

- **Description**: Acts as a mediator for managing thread execution.
- **Snippet**:

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorServiceMediatorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        executor.execute(() -> System.out.println("Task 1 executed"));
        executor.execute(() -> System.out.println("Task 2 executed"));

        executor.shutdown();
    }
}
```

3. **`java.awt.Component`**

- **Description**: GUI components in Java use the mediator pattern to communicate through the `Container` or `LayoutManager` .
- **Snippet**:

```java
import javax.swing.*;
import java.awt.*;
```

```java
public class SwingMediatorExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Mediator Pattern");
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());

        JButton button = new JButton("Click Me");
        JTextField textField = new JTextField(20);

        button.addActionListener(e -> textField.setText("Button clicked!"));

        panel.add(button);
        panel.add(textField);

        frame.add(panel);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Template Method Design Pattern

The **Template Design Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This allows subclasses to redefine certain steps of the algorithm without changing the algorithm's structure. It helps to define the overall workflow while leaving the specifics to the subclass.

Key Benefits of the Template Design Pattern

1. **Code Reusability**: Common steps of the algorithm can be reused across subclasses.
2. **Separation of Concerns**: The algorithm structure is maintained in a superclass, while subclasses handle specific steps.
3. **Flexibility**: Subclasses can change specific steps without modifying the overall algorithm.
4. **Maintainability**: Modifications to common logic need to be made only once in the base class.
5. **Extensibility**: It is easier to add new subclasses with different implementations of specific steps.

Example 1: Cooking Recipe

```java
// Abstract Class (Template)
public abstract class CookingRecipe {
    // Template Method
    public void cook() {
        prepareIngredients();
        cookFood();
        serve();
    }

    // Abstract Methods to be implemented by subclasses
    protected abstract void prepareIngredients();
    protected abstract void cookFood();

    // Concrete Method
    protected void serve() {
        System.out.println("Serve the food.");
    }
}

// Concrete Class for Cooking Pasta
public class PastaRecipe extends CookingRecipe {
    @Override
    protected void prepareIngredients() {
        System.out.println("Boiling pasta and preparing sauce.");
    }

    @Override
    protected void cookFood() {
        System.out.println("Cooking pasta with sauce.");
    }
}

// Concrete Class for Cooking Rice
public class RiceRecipe extends CookingRecipe {
    @Override
    protected void prepareIngredients() {
        System.out.println("Soaking rice and preparing vegetables.");
    }

    @Override
    protected void cookFood() {
        System.out.println("Cooking rice with vegetables.");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        CookingRecipe pasta = new PastaRecipe();
        CookingRecipe rice = new RiceRecipe();
        System.out.println("Pasta Recipe:");
        pasta.cook();

        System.out.println("\nRice Recipe:");
        rice.cook();
    }
}
```

Example 2: Data Processing Pipeline

```
// Abstract Class (Template)
public abstract class DataProcessor {
    // Template Method
    public final void processData() {
        loadData();
        processDataStep();
        saveData();
    }

    // Abstract Methods to be implemented by subclasses
    protected abstract void loadData();
    protected abstract void processDataStep();
    protected abstract void saveData();
}

// Concrete Class for Processing JSON Data
public class JSONDataProcessor extends DataProcessor {
    @Override
    protected void loadData() {
        System.out.println("Loading data from JSON file.");
    }

    @Override
    protected void processDataStep() {
        System.out.println("Processing data from JSON.");
    }

    @Override
    protected void saveData() {
        System.out.println("Saving processed data to JSON file.");
    }
}

// Concrete Class for Processing CSV Data
public class CSVDataProcessor extends DataProcessor {
```

```java
    @Override
    protected void loadData() {
        System.out.println("Loading data from CSV file.");
    }

    @Override
    protected void processDataStep() {
        System.out.println("Processing data from CSV.");
    }

    @Override
    protected void saveData() {
        System.out.println("Saving processed data to CSV file.");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        DataProcessor jsonProcessor = new JSONDataProcessor();
        DataProcessor csvProcessor = new CSVDataProcessor();

        System.out.println("Processing JSON Data:");
        jsonProcessor.processData();

        System.out.println("\nProcessing CSV Data:");
        csvProcessor.processData();
    }
}
```

Example 3: Game Simulation

```java
// Abstract Class (Template)
public abstract class Game {
    // Template Method
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }

    // Abstract Methods to be implemented by subclasses
    protected abstract void initialize();
    protected abstract void startPlay();
    protected abstract void endPlay();
```

```java
}

// Concrete Class for Football Game
class Football extends Game {
    @Override
    protected void initialize() {
        System.out.println("Initializing football game.");
    }

    @Override
    protected void startPlay() {
        System.out.println("Starting football game.");
    }

    @Override
    protected void endPlay() {
        System.out.println("Ending football game.");
    }
}

// Concrete Class for Basketball Game
public class Basketball extends Game {
    @Override
    protected void initialize() {
        System.out.println("Initializing basketball game.");
    }

    @Override
    protected void startPlay() {
        System.out.println("Starting basketball game.");
    }

    @Override
    protected void endPlay() {
        System.out.println("Ending basketball game.");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Game football = new Football();
        Game basketball = new Basketball();

        System.out.println("Football Game:");
        football.play();

        System.out.println("\nBasketball Game:");
        basketball.play();
    }
}
```

Exercises for Template Design Pattern

1. **File Compression Process**
   Create an abstract class for compressing files, with methods like `loadFile()`, `compress()`, and `saveFile()`. Implement concrete subclasses for different file formats (e.g., ZIP, TAR).

2. **Customer Order Processing**
   Design a system for processing orders in an e-commerce application. Create an abstract method for order validation, payment, and shipping. Implement concrete classes for different order types (e.g., physical and digital).

3. **Document Printing**
   Create a class hierarchy for printing documents. Include abstract methods like `openDocument()`, `printDocument()`, and `closeDocument()`. Implement subclasses for PDF and Word document printing.

4. **Task Execution Pipeline**
   Implement a task execution pipeline where common steps are shared (e.g., `initialize()`, `execute()`, and `finish()`), and concrete subclasses define the specifics of each task.

5. **Report Generation Process**
   Create an abstract report generation class with methods like `prepareData()`, `generateReport()`, and `sendReport()`. Implement subclasses for generating different types of reports (e.g., sales, inventory).

Java Classes That Use Template Design Pattern

1. `java.io.InputStream` (Abstract Class)

The `InputStream` class in Java is an example of a template method pattern, as it defines a `read()` method that specifies the algorithm for reading data from an input source, leaving the details to its subclasses like `FileInputStream`, `BufferedInputStream`, etc.

```
public abstract class InputStream {
    public int read() throws IOException {
        return read(new byte[1024]);
    }

    public int read(byte[] b) throws IOException {
        // Default implementation
        return -1;
    }
}

public class FileInputStream extends InputStream {
    public int read(byte[] b) throws IOException {
        // Implementation to read data from a file
        return 0;
    }
}
```

2. `java.util.AbstractList` (Abstract Class)

The `AbstractList` class in the Java Collections Framework implements the template method pattern. It provides the `size()` method as the template method, and subclasses like `ArrayList` or `LinkedList` implement the specific list behaviors.

```
public abstract class AbstractList<E> implements List<E> {
    public int size() {
        // Template method that calls other abstract methods
        return listSize();
    }

    protected abstract int listSize();
}

public class ArrayList<E> extends AbstractList<E> {
    protected int listSize() {
        return 10; // For example
    }
}
```

3. `javax.swing.AbstractButton` (Abstract Class)

The `AbstractButton` class in Swing is another example where the template method pattern is used. It defines the overall structure for handling button events but leaves the details of button actions to its subclasses like `JButton`.

```
public abstract class AbstractButton extends JComponent {
    public void doClick() {
```

```java
        // Template method
        if (isEnabled()) {
            clickAction();
        }
    }

    protected abstract void clickAction();
}

public class JButton extends AbstractButton {
    protected void clickAction() {
        System.out.println("Button clicked!");
    }
}
```