Mastering Object Creation: A Deep Dive into Creational Design Patterns in Java

Rizwanul Haque · Follow

Published in Stackademic · 9 min read · Apr 29, 2024

Creational design patterns in Java are a category of design patterns that deal with object creation mechanisms, aiming to provide flexible and robust solutions to various instantiation problems. These patterns encapsulate the object creation process, hide the complexities involved, and promote code reuse and maintainability. In this article, we'll explore the most commonly used creational design patterns in Java, their implementation details, and their respective use cases.

Types of Creational Design Patterns

1. Singleton Pattern
2. Factory Method Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

1. Singleton Pattern

The Singleton design pattern is one of the simplest creational design patterns in Java. It ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system.

Implementation of Singleton Pattern

In Java, there are several ways to implement the Singleton pattern. Here, I'll demonstrate two commonly used approaches: the Eager Initialization and Lazy Initialization.

1. Eager Initialization

In the eager initialization approach, the singleton instance is created at the time of class loading. It ensures that the instance is always available, but it may consume memory even if the instance is not needed.

```
public class EagerSingleton {
    // Static variable to hold the single instance of the class
    private static final EagerSingleton instance = new EagerSingleton();

    // Private constructor to prevent instantiation from outside
    private EagerSingleton() {}

    // Static method to get the singleton instance
    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

2. Lazy Initialization

In the lazy initialization approach, the singleton instance is created only when it is needed for the first time. This approach saves memory because the instance is not created until the getInstance() method is called.

```
public class LazySingleton {
    // Private static variable to hold the single instance of the class
    private static LazySingleton instance;

    // Private constructor to prevent instantiation from outside
```

```
    private LazySingleton() {}

    // Static method to get the singleton instance
    public static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

Usage of Singleton Pattern

Singleton pattern can be used in various scenarios where only one instance of a class is required, such as:

- Logging Classes: Ensuring that only one logger instance exists throughout the application.
- Database Connection: Managing a single connection object to interact with a database.
- Configuration Settings: Storing and accessing global configuration settings.

Example Usage

Let's illustrate the usage of the Singleton pattern with a simple example:

```java
public class SingletonDemo {
    public static void main(String[] args) {
        // Get the singleton instance of EagerSingleton
        EagerSingleton eagerInstance = EagerSingleton.getInstance();

        // Get the singleton instance of LazySingleton
        LazySingleton lazyInstance = LazySingleton.getInstance();
    }
}
```

Benefits of Singleton Pattern

- Controlled Access: Singleton pattern provides a global point of access to the single instance, ensuring that the object is easily accessible throughout the application.
- Memory Optimization: Lazy initialization helps in conserving memory by creating the instance only when it is required.
- Thread Safety: If implemented correctly, Singleton pattern ensures thread safety in a multi-threaded environment.

Factory Method Pattern

The Factory Method pattern in Java is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by decoupling the client code from the actual object creation logic, thus making it easier to extend and maintain the codebase.

Implementation of Factory Method Pattern

Let's illustrate the Factory Method pattern with a simple example of creating different types of shapes.

1. Define the Shape Interface

First, we define the `Shape` interface representing various shapes.

```
public interface Shape {
    void draw();
}
```

2. Implement Concrete Shape Classes

Next, we implement concrete classes that implement the `Shape` interface.

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

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

public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Square");
    }
}
```

3. Define the Shape Factory Interface

Create a `ShapeFactory` interface with a method for creating shapes.

```
public interface ShapeFactory {
    Shape createShape();
```

```
}
```

4. Implement Concrete Shape Factory Classes

Now, we implement concrete factories for each type of shape.

```
public class CircleFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Circle();
    }
}

public class RectangleFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Rectangle();
    }
}

public class SquareFactory implements ShapeFactory {
    @Override
    public Shape createShape() {
        return new Square();
    }
}
```

Usage of Factory Method Pattern

Now, let's see how we can use the Factory Method pattern to create different types of shapes without exposing the object creation logic to the client code.

```
public class FactoryMethodDemo {
    public static void main(String[] args) {
        ShapeFactory circleFactory = new CircleFactory();
        Shape circle = circleFactory.createShape();
        circle.draw();

        ShapeFactory rectangleFactory = new RectangleFactory();
        Shape rectangle = rectangleFactory.createShape();
        rectangle.draw();

        ShapeFactory squareFactory = new SquareFactory();
        Shape square = squareFactory.createShape();
        square.draw();
    }
}
```

Benefits of Factory Method Pattern

- Loose Coupling: The client code is decoupled from the actual object creation logic, allowing for easier maintenance and extension.
- Encapsulation: The creation of objects is encapsulated within factory classes, hiding the implementation details from clients.

- Extensibility: It's easy to introduce new types of objects by adding new factory classes without modifying existing code.

Abstract Factory Pattern

The Abstract Factory design pattern in Java is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It is an extension of the Factory Method pattern, where a factory interface is responsible for creating multiple related objects.

Implementation of Abstract Factory Pattern

Let's illustrate the Abstract Factory pattern with an example of creating different types of shapes with their corresponding colors.

1. Define the Shape and Color Interfaces

First, we define interfaces representing shapes and colors.

```
public interface Shape {
    void draw();
}

public interface Color {
```

```
    void fill();
}
```

2. Implement Concrete Shape Classes

Next, we implement concrete classes for different shapes.

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

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

public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Square");
    }
}
```

3. Implement Concrete Color Classes

Implement concrete classes for different colors.

```
public class Red implements Color {
    @Override
    public void fill() {
        System.out.println("Filling with Red color");
    }
}

public class Green implements Color {
    @Override
    public void fill() {
        System.out.println("Filling with Green color");
    }
}

public class Blue implements Color {
    @Override
    public void fill() {
        System.out.println("Filling with Blue color");
    }
}
```

4. Define the Abstract Factory Interface

Create interfaces for Shape and Color factories.

```
public interface AbstractFactory {
    Shape createShape();
    Color createColor();
}
```

5. Implement Concrete Factories

Implement concrete factories for creating shapes and colors.

```
public class ShapeFactory implements AbstractFactory {
    @Override
    public Shape createShape() {
        return new Circle();
    }

    @Override
    public Color createColor() {
        return new Red();
    }
}

public class ColorFactory implements AbstractFactory {
    @Override
    public Shape createShape() {
        return new Rectangle();
    }

    @Override
    public Color createColor() {
```

```
        return new Blue();
    }
}
```

Usage of Abstract Factory Pattern

Now, let's see how we can use the Abstract Factory pattern to create families of related objects.

```
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        AbstractFactory shapeFactory = new ShapeFactory();
        Shape circle = shapeFactory.createShape();
        circle.draw();
        Color red = shapeFactory.createColor();
        red.fill();

        AbstractFactory colorFactory = new ColorFactory();
        Shape rectangle = colorFactory.createShape();
        rectangle.draw();
        Color blue = colorFactory.createColor();
        blue.fill();
    }
}
```

Benefits of Abstract Factory Pattern

- Encapsulation: The creation of families of related objects is encapsulated within factory classes, hiding the implementation details from clients.
- Flexibility: It's easy to switch between different families of objects by using different factory implementations.
- Consistency: The Abstract Factory pattern ensures that the created objects are compatible and belong to the same family.

Builder Pattern

The Builder pattern is a creational design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations. This pattern is particularly useful when dealing with objects that have numerous optional parameters or configurations.

Implementation of Builder Pattern

Let's delve into the details of implementing the Builder pattern:

1. Define the Computer Class

```java
public class Computer {
    private String processor;
    private String graphicsCard;
    private int ram;
    private int storageCapacity;
    // Other properties...

    // Private constructor to prevent direct instantiation
    private Computer(ComputerBuilder builder) {
        this.processor = builder.processor;
        this.graphicsCard = builder.graphicsCard;
        this.ram = builder.ram;
        this.storageCapacity = builder.storageCapacity;
        // Set other properties...
    }

    // Getters for properties
    // Optional: Implement additional methods as needed
}
```

2. Create the ComputerBuilder Class

```java
public class Computer {
    // Properties...

    // Private constructor...

    public static class ComputerBuilder {
        private String processor;
```

```
        private String graphicsCard;
        private int ram;
        private int storageCapacity;
        // Other properties...

        public ComputerBuilder() {
            // Initialize default values if needed
        }

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

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

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

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

        // Methods for setting other properties...

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

3. Build a Custom Computer Using the ComputerBuilder

```
public class BuilderPatternDemo {
    public static void main(String[] args) {
        // Create a builder object
        Computer.ComputerBuilder builder = new Computer.ComputerBuilder();

        // Build a custom computer
        Computer computer = builder
            .processor("Intel Core i7")
            .graphicsCard("NVIDIA GeForce RTX 3080")
            .ram(16)
            .storageCapacity(512)
            // Set other properties...
            .build();
    }
}
```

Real-world Use Case

In a real-world scenario, the Builder pattern can be used to build custom computer systems tailored to individual user requirements. Users can select various components such as processor, graphics card, RAM, storage capacity, and additional peripherals to create a computer system that meets their specific needs.

Prototype Pattern

The Prototype pattern is a creational design pattern that enables the creation of new objects by copying an existing object, known as the prototype. This pattern promotes efficiency by avoiding the overhead of creating objects from scratch and is particularly useful when creating objects is expensive or resource-intensive.

Implementation of Prototype Pattern

Let's delve into the details of implementing the Prototype pattern:

1. Define the Document Prototype Interface

First, we define an interface representing the prototype for documents. This interface will declare the common behavior for document prototypes.

```
public abstract class DocumentPrototype implements Cloneable {
    // Optional: Declare common properties and methods...

    // Abstract method to be implemented by concrete prototype classes for cloni
    @Override
```

```
    public abstract DocumentPrototype clone() throws CloneNotSupportedException;
}
```

The `DocumentPrototype` abstract class serves as the base for concrete document prototypes. It extends `Cloneable` to indicate that objects implementing this interface can be cloned. Concrete document prototype classes will override the `clone()` method to define their cloning behavior.

2. Implement Concrete Document Prototype Classes

Next, we implement concrete classes that represent specific types of documents and extend the `DocumentPrototype` class. These classes provide their own cloning logic.

```
public class UserDocument extends DocumentPrototype {
    private String username;
    // Other properties...

    // Constructor to initialize username and other properties.
    public UserDocument(String username) {
        this.username = username;
        // Initialize other properties...
    }

    // Override clone method to provide cloning behavior for UserDocument.
```

```
    @Override
    public DocumentPrototype clone() throws CloneNotSupportedException {
        return (UserDocument) super.clone();
    }
}

public class ReportDocument extends DocumentPrototype {
    private int reportId;
    // Other properties...

    // Constructor to initialize reportId and other properties.
    public ReportDocument(int reportId) {
        this.reportId = reportId;
        // Initialize other properties...
    }

    // Override clone method to provide cloning behavior for ReportDocument.
    @Override
    public DocumentPrototype clone() throws CloneNotSupportedException {
        return (ReportDocument) super.clone();
    }
}
```

Concrete document prototype classes like `UserDocument` and `ReportDocument` extend the `DocumentPrototype` abstract class and provide their own implementation for the `clone()` method. This method returns a clone of the current object using the `super.clone()` method, ensuring a shallow copy of the object.

3. Create and Clone Document Prototype Objects

Now, we can create prototype objects and clone them to create new instances.

```
public class PrototypeDemo {
    public static void main(String[] args) {
        try {
            // Create prototype objects
            UserDocument userDocumentPrototype = new UserDocument("JohnDoe");
            ReportDocument reportDocumentPrototype = new ReportDocument(123);

            // Clone prototype objects to create new instances
            UserDocument userDocumentClone = (UserDocument) userDocumentPrototyp
            ReportDocument reportDocumentClone = (ReportDocument) reportDocument

            // Optionally modify cloned objects if needed
            userDocumentClone.setUsername("JaneDoe");
            reportDocumentClone.setReportId(456);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

In the `PrototypeDemo` class, we demonstrate the usage of the Prototype pattern by creating prototype objects (`userDocumentPrototype` and `reportDocumentPrototype`) and cloning them to create new instances (`userDocumentClone` and `reportDocumentClone`). We catch the `CloneNotSupportedException` in case cloning is not supported for a particular object.

Conclusion

Creational design patterns in Java offer solutions to common problems related to object creation. By understanding the principles behind each pattern and their respective use cases, developers can effectively design and implement flexible, reusable, and maintainable Java applications.

Stackademic 🎓

Thank you for reading until the end. Before you go:

- Please consider **clapping** and **following** the writer! 👏
- Follow us X | LinkedIn | YouTube | Discord
- Visit our other platforms: In Plain English | CoFeed | Venture | Cubed
- More content at Stackademic.com

Java Design Patterns Creational Design Pattern

Written by Rizwanul Haque

Follow

1.1K Followers · Writer for Stackademic

Lead Android Developer | Passionate about Coding | Sharing Insights 🚀 | Let's explore together! 📱💻 #AndroidDev #Kotlin #Tech

More from Rizwanul Haque and Stackademic

Rizwanul Haque in Stackademic

Important Coroutine Interview Questions for Experienced Andro...

Coroutines have revolutionized asynchronous programming in Kotlin, offering a powerful...

Feb 13 1K 3

Dylan Cooper in Stackademic

Google Python Team Entirely Laid Off, Flutter Team Also “Facing the...

Google’s good news and bad news came very suddenly.

May 2 1.5K 26

Walid LARABI in Stackademic

Top 20 AI buzzwords in 2024

Rizwanul Haque

SSL Pinning in Android Apps for Enhanced Security

Explore AI's world! Understand 20 buzzwords easily with simple explanations and visuals.

Feb 19 1.5K 15

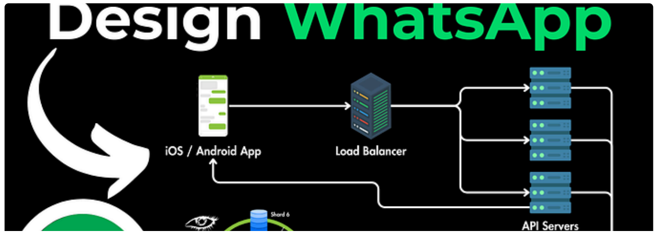
Introduction:

Dec 20, 2023 835 2

See all from Rizwanul Haque

See all from Stackademic

Recommended from Medium



Medium Search Write

Hayk Simonyan in Level Up Coding

Renan Schmitt in JavaJams

System Design Interview: Design WhatsApp

Prepare for system design interviews with this guide to designing a WhatsApp-like…

Jun 18 944 6

Spring Boot: Handling a REST Endpoint That Queries More Data…

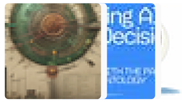
If you are a developer and have not faced this issue yet, it is still worth reading, as at some…

Jun 24 176 3

Lists

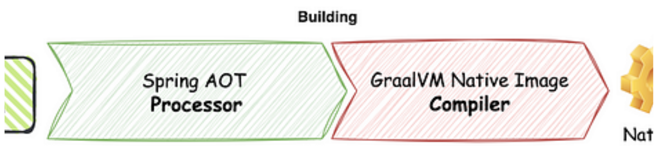
General Coding Knowledge

20 stories · 1360 saves



data science and AI

40 stories · 197 saves



Saeed Zarinfam in ITNEXT

10 Spring Boot Performance Best Practices

Vasko Jovanoski

Say Goodbye to Repetition: Building a Common Library in…

Making Spring Boot applications performant and resource-efficient

Jun 24 172 3

Learn to create a shared library in Spring Boot, enhancing code reusability, simplifyin...

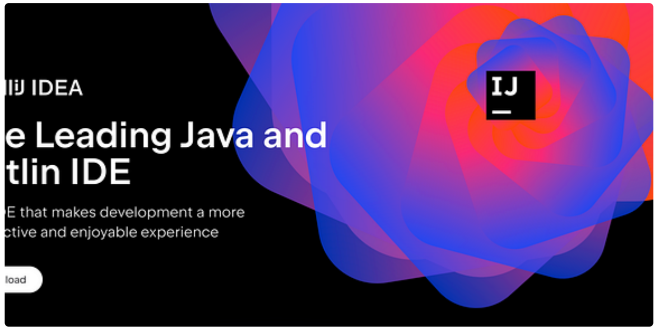
Jun 25 244 7

Jan Kammerath

Why Tech Workers Are Fleeing Germany — A Reality Check

Over the past months and years I have seen a number of friends and colleagues leave...

Jun 23 4.4K 134



Gumparthy Pavan Kumar

Top 5 useful Free IntelliJ Plugins to improve your journey as a Java...

Intellij IDE

Jun 8 38

See more recommendations