

**A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design**. Design patterns are not finished designs that can be transformed directly into code. Instead, they are templates or blueprints that can be used to solve specific design issues, making code more robust, maintainable, and scalable.

**Components of a Design Pattern**

1. **Pattern Name**: A descriptive name that conveys the core idea of the pattern succinctly.
2. **Problem**: A description of the problem that the pattern aims to solve, including the context and conditions under which it is applicable.
3. **Solution**: The general arrangement of objects and classes that solve the problem. This part of the pattern provides a blueprint for implementation.
4. **Consequences**: The results and trade-offs of applying the pattern. This includes the benefits and potential drawbacks.

**Categories of Design Patterns**

1. **Creational Patterns**: Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
   * **Examples**: Singleton, Factory Method, Abstract Factory, Builder, Prototype.
2. **Structural Patterns**: Deal with object composition or structure, ensuring **that if one part of a system changes, the entire system doesn’t need to change along with it.**
   * **Examples**: Adapter, Composite, Decorator, Facade, Proxy, Flyweight, Bridge.
3. **Behavioral Patterns**: Deal with communication between objects, managing algorithms, relationships, and responsibilities between them.
   * Examples: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

**Benefits of Using Design Patterns**

* **Improved Communication**: Design patterns provide a common vocabulary, making it easier for team members to communicate about design decisions.
* **Proven Solutions**: They offer tested solutions that have been used in many scenarios, reducing the likelihood of design flaws.
* **Efficiency**: They save time by providing solutions that developers can adapt instead of creating from scratch.
* **Maintainability**: They promote better code organization, making it easier to maintain and extend.

**Creational design patterns**

deal with object creation mechanisms, optimizing the process to be adaptable and suitable for various situations. These patterns aim to make a system independent of how its objects are created, composed, and represented. Here are some common creational design patterns:

* + - * 1. Singleton Design Pattern

The Singleton design pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is particularly useful in cases where exactly one instance of a class is needed to coordinate actions across the system. Here, we'll discuss different implementations of the Singleton pattern in Java, including lazy initialization, eager initialization, thread-safe Singleton, and using an enum.

### 1. Lazy Initialization Singleton

**Purpose**: The instance is created only when it is requested for the first time, which helps in saving resources.

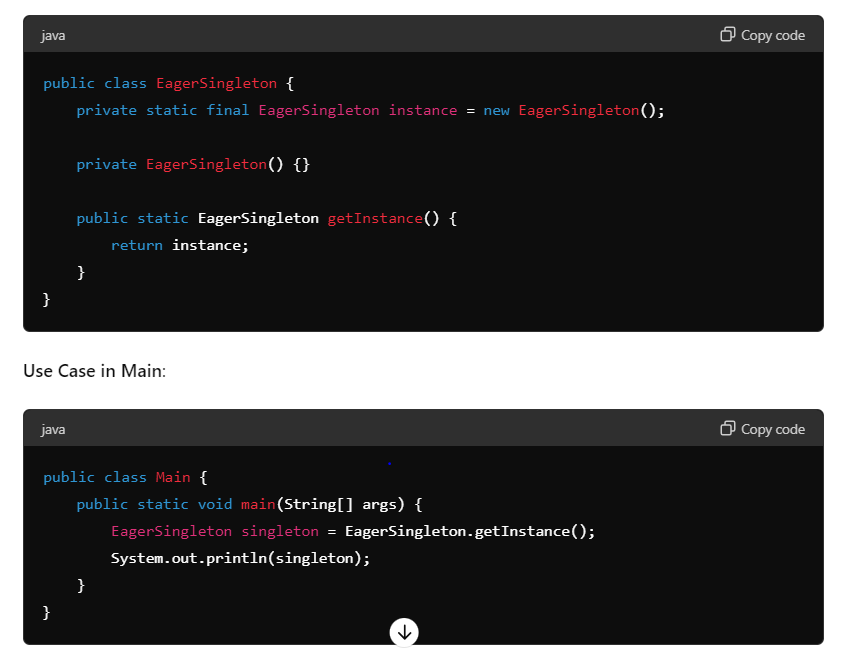
**Example**:



### 2. Eager Initialization Singleton

**Purpose**: The instance is created at the time of class loading. This method is easy to implement but may lead to resource wastage if the instance is never requested.

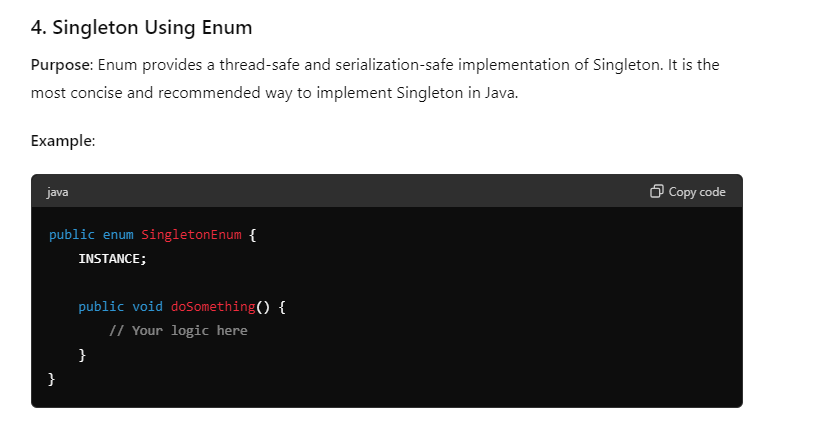
**Example**:



### 3-. Thread-Safe Singleton (Double-Checked Locking)

**Purpose**: Ensures that the instance is created only once even in a multithreaded environment. This method checks the instance twice to ensure that it is created only once.





**Use Cases in a Project Application**

* **Configuration Manager**: A Singleton can be used to manage configuration settings in an application, ensuring that there is a single point of access and consistent configuration values throughout the application.
* **Logger**: A Singleton can be used for logging purposes, ensuring that all parts of the application use the same logging instance.
* **Cache**: Singleton can be used to manage a cache, ensuring that all parts of the application access the same cached data.
* **Database Connection Pool**: Managing a pool of database connections where a Singleton ensures that the connection pool is globally accessible.

Role of volatile

The volatile keyword in Java is used to ensure visibility of changes to variables across threads. When it comes to implementing a thread-safe Singleton pattern, the volatile keyword plays a crucial role in conjunction with the double-checked locking technique. Here's why and how it is used:

**The Role of volatile**

1. **Visibility**: When a variable is declared as volatile, it ensures that any thread reading the field will see the most recently written value. Without volatile, a thread could cache the value of the instance variable, leading to stale data being read.
2. **Ordering**: The volatile keyword prevents the JVM from reordering instructions in a way that could cause unexpected behavior in a concurrent environment. It creates a happens-before relationship, ensuring that the construction of the Singleton instance is fully completed before any thread reads the instance variable.

**Double-Checked Locking with volatile**

In a Singleton implementation, the double-checked locking technique is used to reduce the overhead of acquiring a lock by first checking if the instance is already created without synchronization. Only if the instance is null, it synchronizes and checks again before creating the instance.

**Example Implementation**

Here's how you implement a thread-safe Singleton with double-checked locking and volatile:

java

Copy code

public class ThreadSafeSingleton {

private static volatile ThreadSafeSingleton instance;

// Private constructor to prevent instantiation

private ThreadSafeSingleton() {}

public static ThreadSafeSingleton getInstance() {

if (instance == null) { // First check (no locking)

synchronized (ThreadSafeSingleton.class) {

if (instance == null) { // Second check (with locking)

instance = new ThreadSafeSingleton();

}

}

}

return instance;

}

}

**Why Use volatile**

Without volatile, the following issues could arise:

* **Partial Construction Visibility**: Without volatile, it's possible for the instance reference to be visible to other threads before the constructor of ThreadSafeSingleton has finished executing, leading to a partially constructed object being accessed.
* **Instruction Reordering**: The Java Memory Model allows the JVM and the CPU to reorder instructions for performance optimization. Without volatile, there is no guarantee that the new ThreadSafeSingleton instance is fully constructed before the reference is assigned to the instance variable.

**Detailed Example**

Consider the following scenario without volatile:

1. **Thread A** enters the getInstance() method and sees instance is null.
2. **Thread A** enters the synchronized block and starts creating a new ThreadSafeSingleton instance.
3. **Thread B** enters the getInstance() method and sees instance is still null (since Thread A hasn't finished constructing the object).
4. **Thread B** also enters the synchronized block and starts creating another ThreadSafeSingleton instance.

By marking the instance variable as volatile, you ensure that:

* The write to the instance variable in Thread A is visible to Thread B.
* The construction of the ThreadSafeSingleton object happens-before the assignment to the instance variable.

**Summary**

The volatile keyword ensures that the instance variable is correctly published to other threads and prevents the JVM from reordering instructions in a way that could lead to the Singleton being partially constructed when accessed by other threads. This is crucial for maintaining the Singleton property in a multithreaded environment.

Different ways to break singleton and solution with code

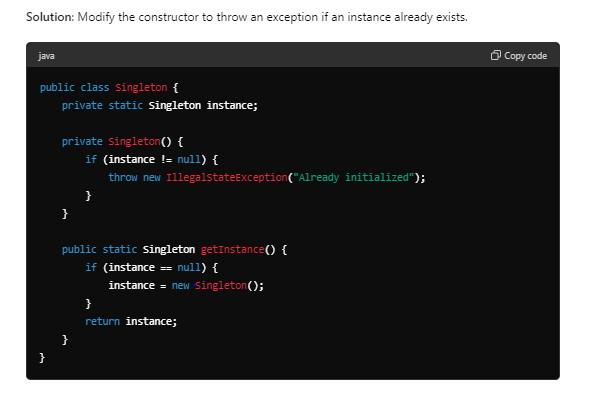
The Singleton design pattern can be broken in several ways in Java, despite efforts to make it robust. Here are common ways Singleton can be broken and their respective solutions:

### 1. ****Reflection****

**Problem**: Reflection can be used to invoke the private constructor of a Singleton class, leading to the creation of multiple instances.

**Example**:



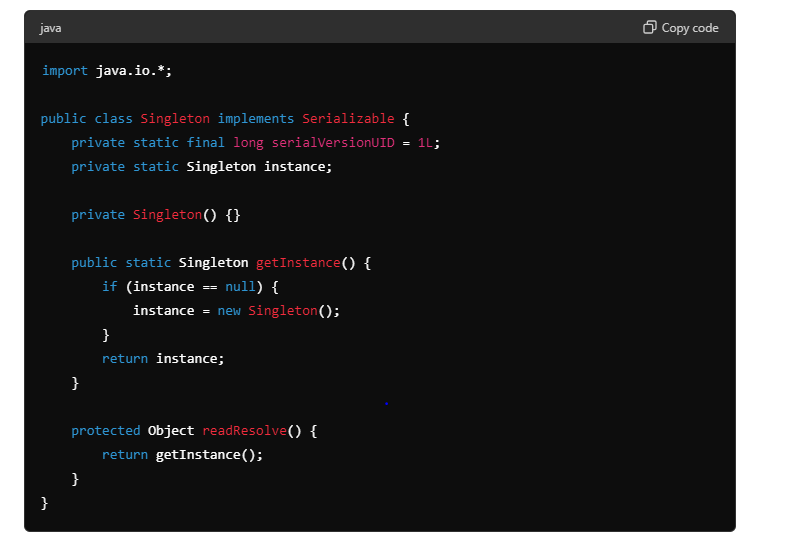


### 2. ****Serialization****

**Problem**: During serialization and deserialization, a new instance of the Singleton class can be created.



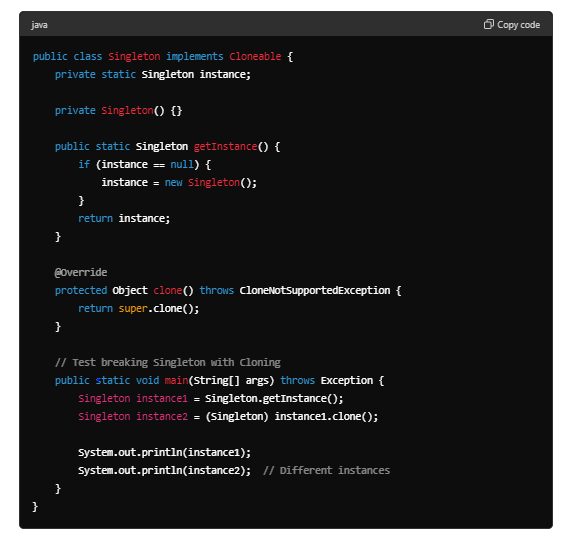
**Solution**: Implement the readResolve method to return the same instance.



### 3. ****Cloning****

**Problem**: The clone method can be used to create a copy of the Singleton instance.

**Example**:





2:- Factory Design Pattern

**A factory pattern is a creational pattern that abstracts or hides the object creation process.**

It hides the complex details of object creation and it is responsible for creating the object and giving them to the classes or program

**Example**

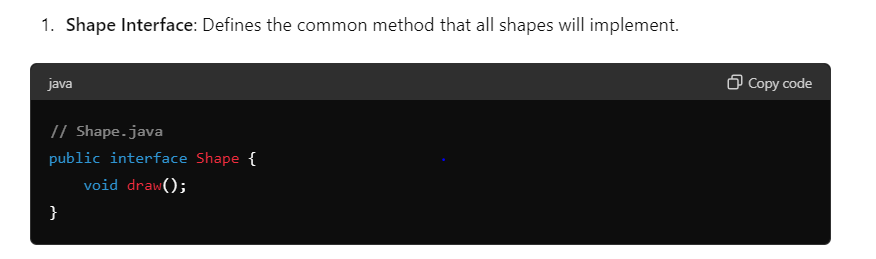
1- When you think of factory you can think of a car factory a chocolate factory or a toy factory.A car factory is responsible for manufacturing the cars .A car dealer need not worry about how the car is manufactured. He simply asks the car factory to deliver him some cars. The car factory is responsible for manufacturing them and delivering them to the dealer.

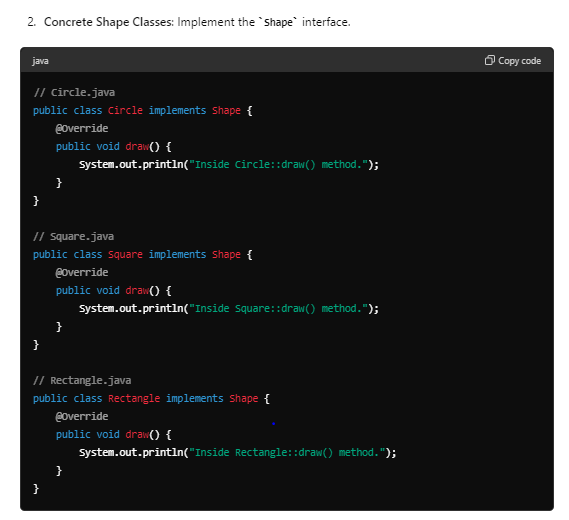
2-Similarly the Chocolate Factory delivers different types of chocolates based on what the chocolate store asks them to deliver.

3: An example from the JDBC space when we use different databases like Oracle, MySql, SqlServer. In Java we use something called JDBC driver to connect to a database.Driver is an interface in the JDBC API and the implementation for this driver is provided by different vendors. The responsibility of the driver is to connect to a particular database and execute sql statements against it.To get a connection we need not remember each and every driver in how it works. For example if we want to connect to Oracle you need not deal with the Oracle driver and same for other databases.You simply use DriverManager which is a class in the JDBC API to get connection and you pass in a connection string which is specific to a particular database where it is different for mysql, it's different for sqlserver.The driver manager acts as a factory and it will return the connection for us by using the appropriate driver for Oracle it will use Oracle driver for my sql it will use the my sql driver and so on.It hides all the details of finding a driver and creating a connection against the database for us. So give it a string it will give you a connection. So here the get connection is a static method on the factory class.We need not create the instance of the driver manager to create objects of type connection.We simply use the static method on the factory class and get the object we need.

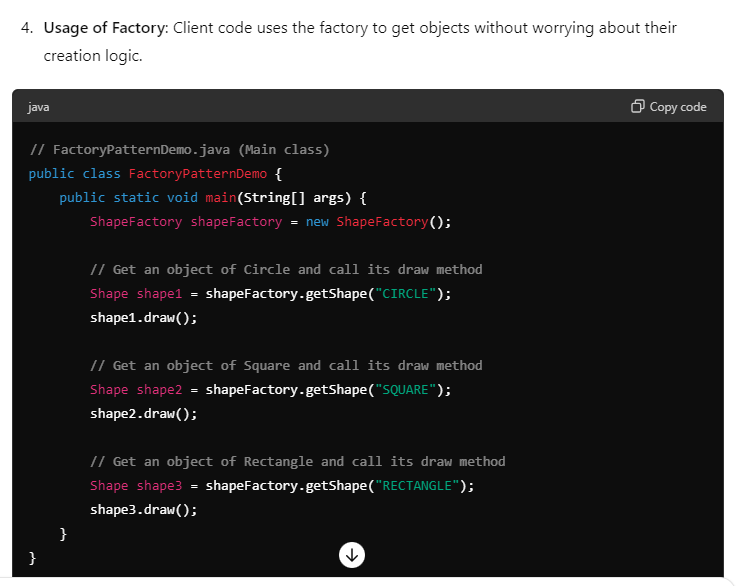
**Example Explanation**

Let's illustrate the Factory pattern with a simple example of a Shape factory that creates different types of shapes (Circle, Square, Rectangle).









**Explanation:**

* **Shape Interface**: Defines a common interface draw() that all concrete shape classes implement.
* **Concrete Shape Classes**: Implement the Shape interface with their specific implementations of the draw() method.
* **Shape Factory**: Contains a factory method getShape(String shapeType) that creates and returns instances of different shapes based on the shapeType parameter provided.
* **FactoryPatternDemo**: Demonstrates how client code can use the ShapeFactory to get instances of shapes without knowing the specific implementation details of each shape class.

**Benefits of Factory Pattern:**

* **Encapsulation**: The creation of objects is encapsulated within the factory, abstracting the client code from object creation logic.
* **Flexibility**: Easily add new types of shapes by extending the factory without modifying existing client code.
* **Decoupling**: Clients depend on the factory interface rather than concrete implementations, promoting loose coupling and easier maintenance.

The Factory pattern is widely used in scenarios where object creation needs to be centralized, dynamically determined, or when the client code should be unaware of the specific classes being instantiated.

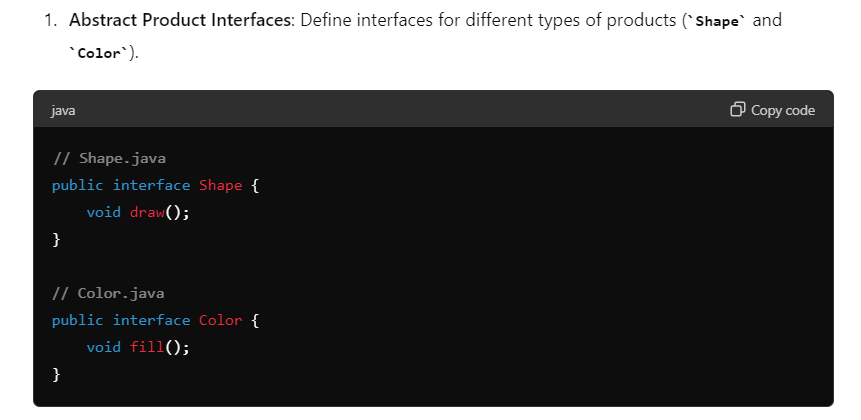
3:- Abstract factory

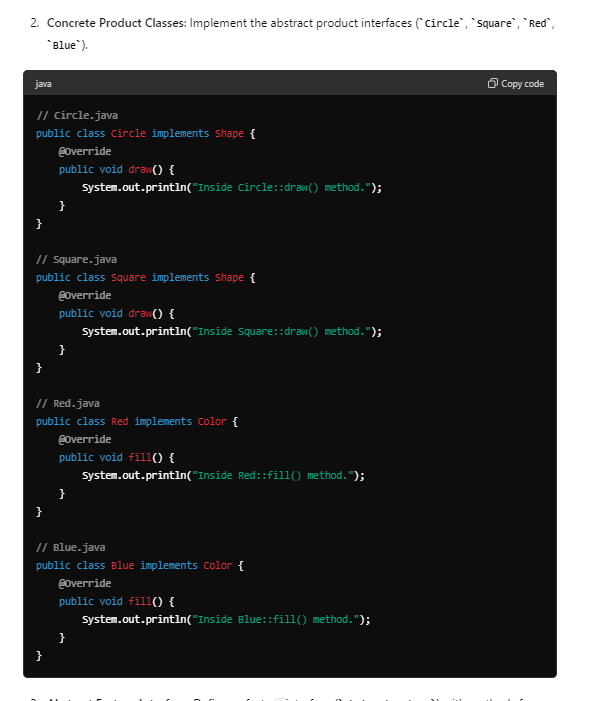
It is Factory of factories

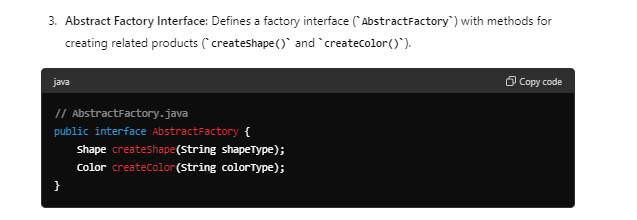
 Now that you have mastered the Factory design pattern learning and implementing the Abstract Factory pattern will be quite easy because an abstract factory is a factory of factories.That is a factory pattern was hiding the details of object creation and factory of factories or **an abstract factory.hides the creation of the factory itself.**

### Example Explanation

Let's illustrate the Abstract Factory pattern with an example of creating different types of shapes (Shape) and their respective colors (Color). We'll define an abstract factory (AbstractFactory) and concrete factories (ShapeFactory and ColorFactory) to create these objects.

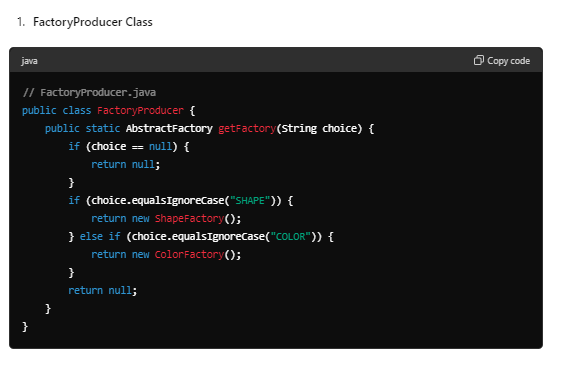






**4- Concrete Factory Classes**: Implement the abstract factory interface to create families of related products (ShapeFactory and ColorFactory).







**Explanation:**

* **Abstract Product Interfaces (Shape, Color)**: Define interfaces for related products (Shape and Color).
* **Concrete Product Classes (Circle, Square, Red, Blue)**: Implement the interfaces to create specific products.
* **Abstract Factory Interface (AbstractFactory)**: Declares methods for creating abstract products (createShape() and createColor()).
* **Concrete Factory Classes (ShapeFactory, ColorFactory)**: Implement the abstract factory interface to create families of related products (ShapeFactory creates shapes, ColorFactory creates colors).
* **Usage (AbstractFactoryPatternDemo)**: Client code uses the abstract factory (FactoryProducer) to obtain a specific factory (ShapeFactory or ColorFactory), then uses that factory to create related products (Shape or Color).

**Use Cases in a Project**

The Abstract Factory pattern is useful in projects where:

* **Cross-platform UI**: You need to create UI components (buttons, menus) that are platform-specific (Windows, macOS) but need to ensure they are consistent within the platform.
* **Database Access**: You have different database implementations (MySQL, PostgreSQL) and need to create data access objects (DAOs) that are specific to each database but provide a common interface.
* **Logging Frameworks**: You want to create loggers (file logger, console logger) that are specific to different environments (development, production) but provide a consistent logging interface.

By using the Abstract Factory pattern, you can create families of related or dependent objects in a way that is modular, easy to extend, and hides the implementation details from the client code, promoting flexibility and maintainability in your application.

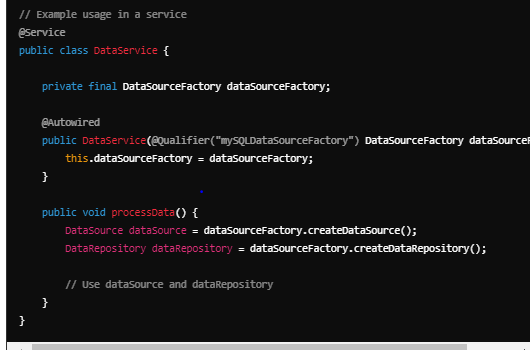
Wrt springboot

In Spring Boot and Java applications, the Abstract Factory design pattern finds use in scenarios where you need to create families of related or dependent objects. This pattern is particularly useful when you want to provide a way to create sets of related objects (such as different families of DAOs, services, or components) without specifying their concrete classes directly. Here are some common use cases where the Abstract Factory pattern can be applied effectively:

### 1. ****Database Access with Multiple Data Sources****

In Spring Boot applications, you might need to connect to multiple databases (e.g., MySQL and PostgreSQL) and create data access objects (DAOs) that are specific to each database type. The Abstract Factory pattern helps in creating families of related DAOs without tightly coupling them to specific database implementations.





4 - Builder Pattern

The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions

**Requirement :**

First will create a problem then solution

Let say we are creating HttpClient class that is responsible for making http calls. And this class can be used by other class which want to make http calls

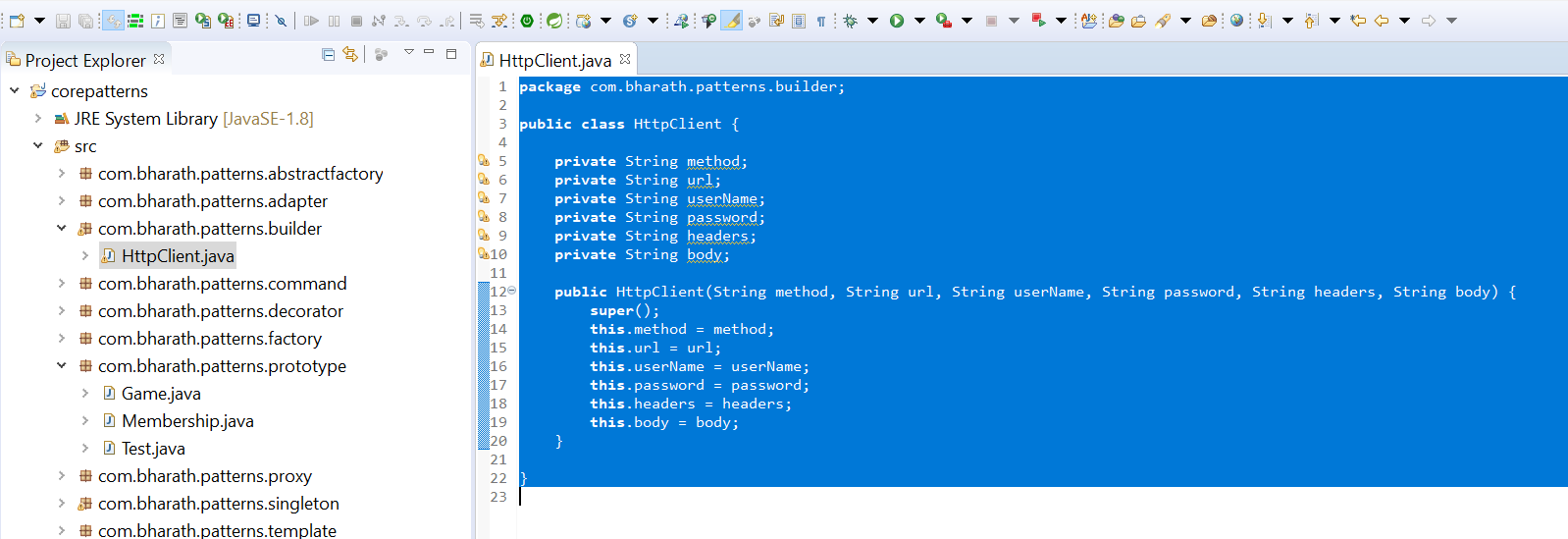
In http client we can pass method ,url,username,pw but some application don’t want to pass username and pw then we can pass as null.

But for other developer this will become tough what null stands for that is where builder pattern come in picture.

**So once we implement builder pattern the code is much more readable and configurable as well and they can skip other parameter by not**

**Invoking the method**

**The Problem**



Graphical user interface, text, application, chat or text message

Description automatically generated

Note :--

If any developer look this code , he will get confuse what this null correspond to. This is not very readable code that is where builder pattern come in picture. Builder pattern give us option what we want and code is more readable.

**Code walk through**

We need a builder class that will make it easy for end client to create an instance of the object just by configuring setting the properties they need and leave the rest

**HttpClientBuilder :---**

will have copy of all the properties the actual class has of which instance we are going to create

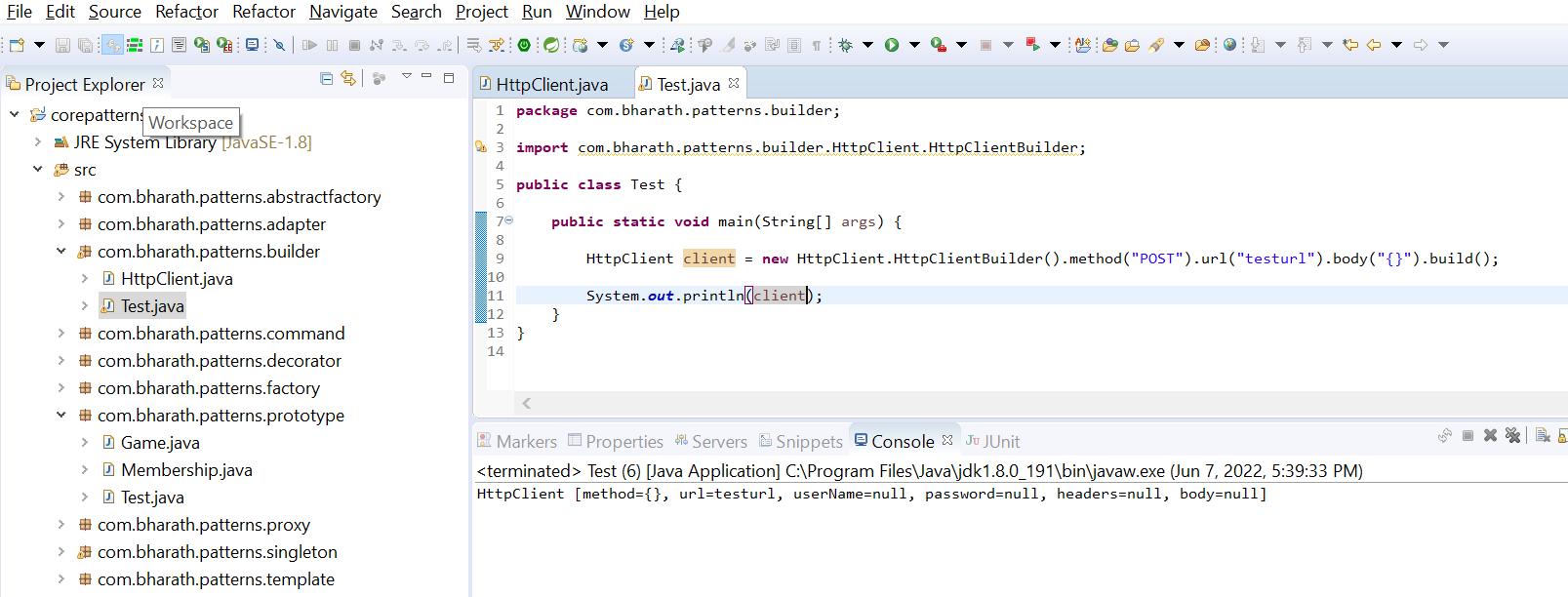
And for each of the property we are going to create a method and each of the method will return instance of the same class

.build : instantiation of httpclient happen which all the properties they want to set

Graphical user interface

Description automatically generated

Test



Now the code is much more readable

**Use case of builder in springboot application**

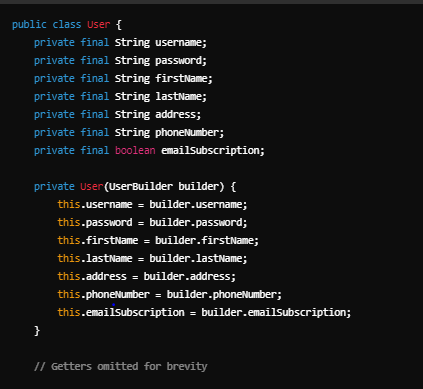
In real-world Java projects, the Builder pattern is commonly used in various scenarios where you need to create complex objects with optional attributes, maintain immutability, and improve code readability. Here are some practical use cases where the Builder pattern is applied effectively within Java applications:

**1. Configuring and Creating Objects with Optional Parameters**

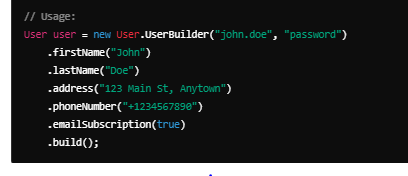
Java applications often require creating objects with many optional parameters. Instead of using telescoping constructors or multiple setters, the Builder pattern provides a cleaner and more maintainable approach.

**Example:**

* **Scenario**: Configuring a User object with optional attributes like address, phone number, and email preferences.

****

****

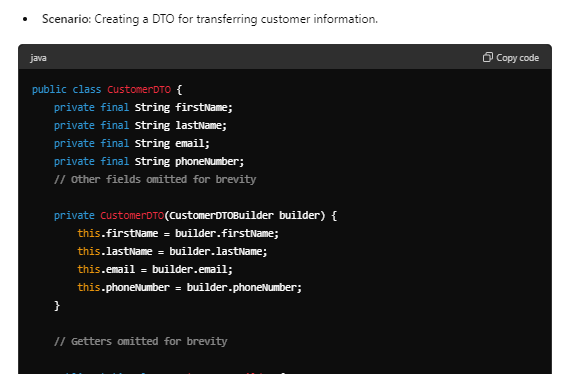
****

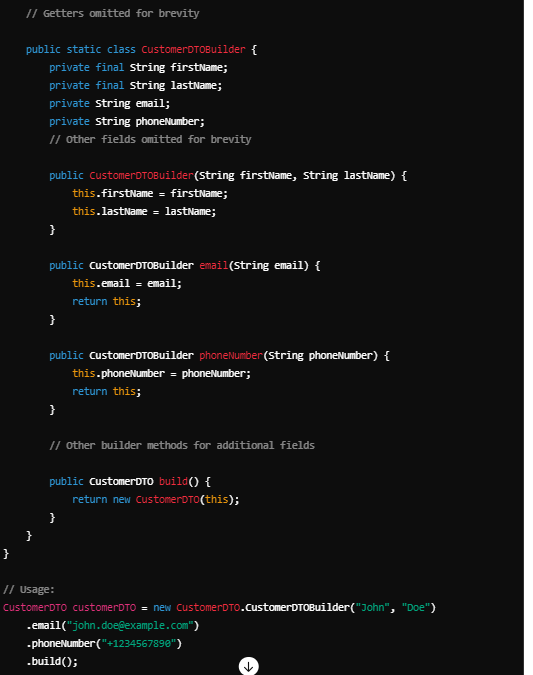
**2. Data Transfer Objects (DTOs) and Serialization**

DTOs are used to transfer data between different layers of an application or between applications. Using the Builder pattern for DTOs allows you to construct them with only the necessary data fields and avoid unnecessary setters.

**Example:**

* **Scenario**: Creating a DTO for transferring customer information.

****

****

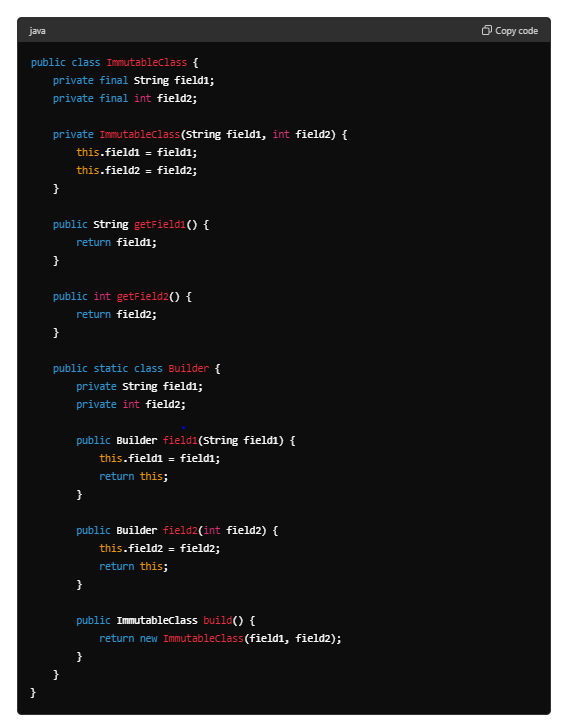
### 3. ****Creating Immutable Objects****

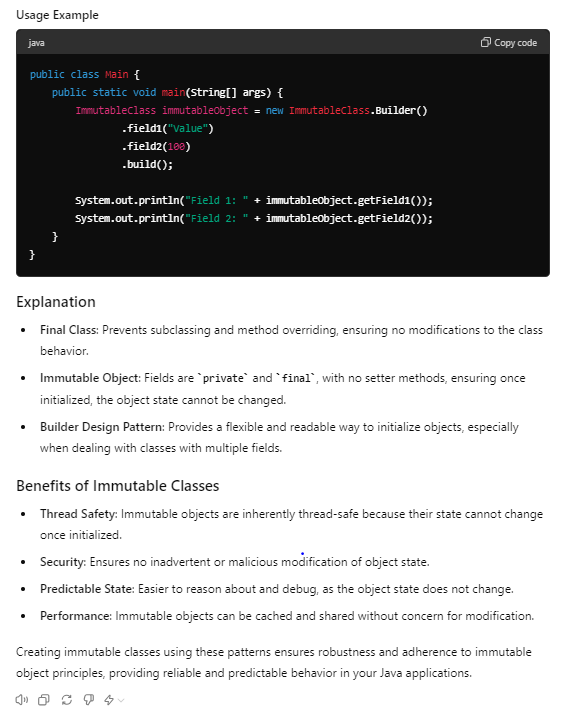
Builders are useful for creating immutable objects where all attributes are set during construction and cannot be changed afterward. This ensures thread safety and prevents unintended modifications.

#### . ****Builder Design Pattern****

* Use a builder to construct instances of the immutable class, allowing flexible initialization while maintaining immutability.

**Builder Class**

****

****

**5:-** Prototype design pattern

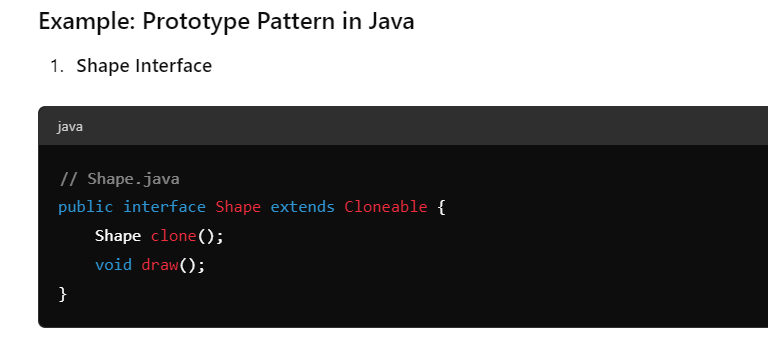
It is creational design pattern that uses an existing object to create new object

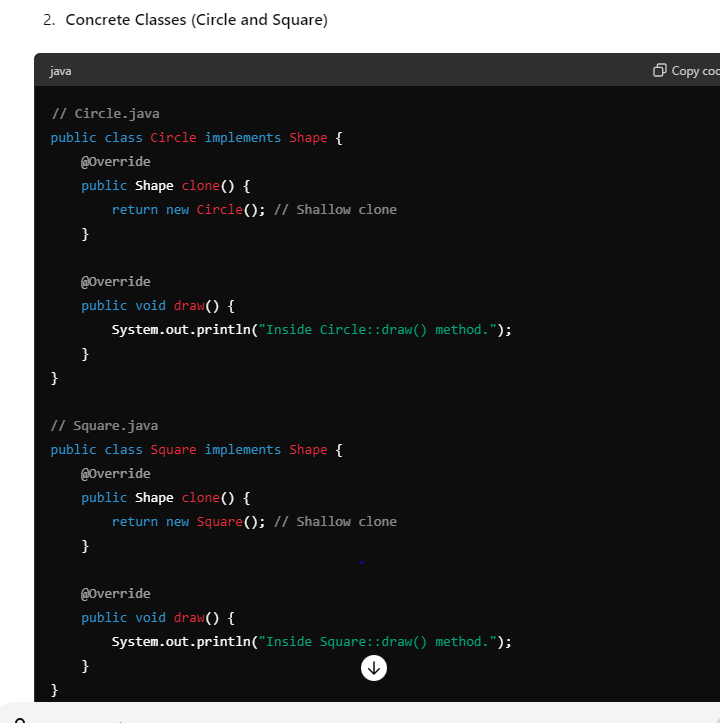
Example :

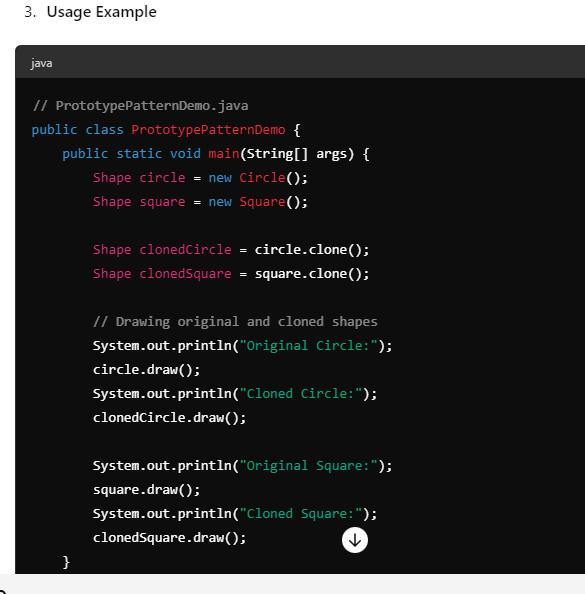
Suppose many gamers login the instead of initializing everything again a prototype should be ready

Prototype is a creational design pattern that allows cloning objects, even complex ones, without coupling to their specific classes. All prototype classes should have a common interface that makes it possible to copy objects even if their concrete classes are unknown.

Certainly! Here's a simple example of the Prototype design pattern in Java using a basic Shape interface and its concrete implementations (Circle and Square).

****

****

****

### Explanation

* **Shape Interface**: Defines the clone() method for cloning shapes and draw() method for drawing shapes.
* **Concrete Classes (Circle, Square)**: Implement the Shape interface and provide their own implementations of clone() for shallow cloning and draw() for drawing the shape.
* **Usage Example (PrototypePatternDemo)**: Demonstrates how to use the Prototype pattern to clone and draw Circle and Square shapes.

### Benefits of Prototype Pattern

* **Object Creation**: Avoids repeated object creation by cloning instances, especially when the initialization process is complex or resource-intensive.
* **Flexibility**: Enables dynamic creation of new objects with different configurations at runtime by cloning prototypes.
* **Performance**: Improves performance compared to creating new objects from scratch, especially in scenarios where object creation is expensive.

### When to Use Prototype Pattern

* **Object Initialization**: Use when creating new objects is costly and can be optimized by copying existing instances.
* **Customization**: Allows creating variations of objects with different configurations without subclassing or using complex initialization logic.

In this Java example, the Prototype pattern provides a simple yet effective way to manage and create instances of objects (Circle, Square) by cloning prototypes. This promotes code reusability, improves performance, and simplifies object creation in scenarios where object initialization is non-trivial.