Design Pattern :

Design patterns are standard solutions to common problems in software design. They provide templates for structuring code in effective, reusable, and maintainable ways. Design patterns are generally categorized into the following types: creational, structural, behavioral, and architectural patterns.

**1. Creational Patterns**

These patterns deal with object creation mechanisms, simplifying how objects are instantiated and ensuring your code is flexible and decoupled. They handle the process of object creation, abstracting the instantiation logic.

**Singleton Design Pattern in Java**

The **Singleton pattern** ensures that a class has **only one instance** and provides a **global access point** to it.

**📌 Key Characteristics of Singleton Pattern**

* ✔ **Single Instance:** Only one object of the class exists in memory.
* ✔ **Global Access:** Provides a static method to get the instance.
* ✔ **Lazy or Eager Initialization:** Object is created **only when needed** (lazy) or **at class loading** (eager).
* ✔ **Thread Safety:** Ensures instance is created safely in **multithreaded environments**.

**Real-World Examples of Singleton Design Pattern**

* ✔ Database Connection Pool – Manages database connections through a single shared instance.
* ✔ Logger – Centralized logging through a single logger instance.
* ✔ Configuration Manager – Provides global access to application configuration settings.
* ✔ Thread Pool – Controls access to a shared pool of threads.
* ✔ Cache – Maintains a single in-memory store for frequently used data.
* ✔ Task Manager (OS) – Single instance monitors and manages system processes.
* ✔ Spring Boot Beans – Default scope for Spring beans is singleton, ensuring one shared

The volatile keyword in Java is used to ensure visibility of changes to a variable across multiple threads. When a variable is declared volatile, it guarantees that:

* Any write to the volatile variable is immediately flushed to main memory.
* Any read of the volatile variable always fetches the latest value directly from main memory, not from a thread’s local cache.

**Key Steps for a Thread-Safe Singleton**

1. Private Constructor
   * Prevents external instantiation.
2. Static Instance Variable
   * Holds the single instance.
3. Global Access Method
   * A static method (commonly getInstance()) that controls object instantiation and access.

private static volatile Singleton instance;

public static Singleton getInstance() {

if (instance == null) {

synchronized(Singleton.class) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

Static Inner Class (Bill Pugh Singleton)

**private** **static** **class** Holder {

**private** **static** **final** Singleton INSTANCE = **new** Singleton();

}

**public** **static** Singleton getInstance() {

**return** Holder.INSTANCE;

}

* Pros: Thread-safe, lazy-loaded, no synchronization required. Recommended for most cases.

Factory Design pattern

We have many class and creation object logic is not directly given to user rather we provide something like interface

Factory Design Pattern 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.

* It abstracts the object creation process.
* Helps in creating objects without exposing the instantiation logic to the client.
* The client only interacts with the common interface, not the concrete classes.
* You **delegate object creation** to the factory instead of manually doing new Pizza() yourself.

**Think of it like a restaurant 🍽**

* You (the Client / FoodApp)  
  You want some food, but you don’t want to cook it yourself.  
  You also don’t care *how* it’s made, you just want the end product.
* Menu (the Product interface — Food)  
  The menu just tells you “we have Pizza, Burger, Pasta,” but does not explain the cooking steps.  
  All foods share some basic actions, like prepare().
* Chef for each food type (Concrete Creators — PizzaFactory, BurgerFactory, PastaFactory)
  + PizzaFactory knows how to make a Pizza.
  + BurgerFactory knows how to make a Burger.
  + PastaFactory knows how to make Pasta.
* Order counter (Creator interface — FoodFactory)  
  The restaurant promises: “Whatever food you order, we’ll give you something implementing Food.”  
  But *how* it’s made depends on the chef you asked for.

**Without Factory Method ❌**

You (FoodApp) would be cooking directly:

java

Food food = **new** Pizza(); *// hard-coded*

If you later want a Burger, you must change your code.

**With Factory Method ✅**

You just say:

java

FoodFactory factory = **new** PizzaFactory(); *// pick a chef*

Food food = factory.createFood(); *// chef makes the food*

food.prepare(); *// enjoy!*

If tomorrow you want to add Sandwich, you just create:

java

**class** SandwichFactory **implements** FoodFactory {

**public** Food createFood() { **return** **new** Sandwich(); }

}

And you do not change your main application — you just plug in the new factory.

**Why it’s called**Factory Method**:**

* Because the method that creates the object (createFood()) is in a specialized “factory” class.
* You delegate object creation to the factory instead of manually doing new Pizza() yourself.
* This keeps your app flexible and loosely coupled — it can handle new products easily.

**Builder Design Pattern:**

The Builder pattern is a creational design pattern that helps in constructing complex objects step by step. Instead of creating a complex object all at once (typically via a large constructor), the Builder pattern separates the construction process from the final representation. This makes it easier and more flexible to create different versions or configurations of an object

A screen shot of a computer

AI-generated content may be incorrect.

**Example analogy: Making a pizza**

Instead of having one giant constructor to specify all pizza toppings and options, you build your pizza step by step:

* Choose crust type
* Add sauce
* Add cheese
* Add toppings
* Bake

Each step is a method call on the builder, which finally produces the pizza object.

. The Goal of This Code

You’re using the Builder Design Pattern to create a Computer object step-by-step without having to create large, messy constructors with too many parameters, especially when some parameters are optional (like graphicsCard and bluetooth).

2. The Computer Class

**public** **class** Computer {

**private** **final** String HDD;

**private** **final** String RAM;

**private** **final** String graphicsCard;

**private** **final** String bluetooth;

* All fields are final → immutability (once created, the object’s state cannot change).
* Required parameters: HDD and RAM.
* Optional parameters: graphicsCard and bluetooth.
* The constructor is private:
* java
* **private** Computer(String HDD, String RAM, String graphicsCard, String bluetooth) { ... }
* Meaning you cannot create a Computer directly with new Computer(...) from outside the class — you must use the builder.

4. Creating Objects with the Builder

In TestBuilderPattern:

java

Computer gamingComputer = **new** Computer.ComputerBuilder("1 TB", "16 GB")

.enableGraphicsCard("ROG")

.enableBluetooth("SONY")

.build();

Here’s what happens in order:

1. new Computer.ComputerBuilder("1 TB", "16 GB") → required params set.
2. .enableGraphicsCard("ROG") → optional param set.
3. .enableBluetooth("SONY") → optional param set.
4. .build() → creates a final immutable Computer object with all these values.

[Devaraj-Umapathi/BuilderDesignPattern/src/WithBuilder/Computer.java at Builder-DesignPattern · Devaraj-Umapathi/Devaraj-Umapathi](https://github.com/Devaraj-Umapathi/Devaraj-Umapathi/blob/Builder-DesignPattern/BuilderDesignPattern/src/WithBuilder/Computer.java)