# **Singleton Method Design Pattern in Java**

Last Updated : 07 Nov, 2023

In object-oriented programming, a java singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Java Singleton classes, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

**Remember the key points while defining a class as a singleton class that is while designing a singleton class:**

1. Make a constructor private.
2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

## **Purpose of Singleton Class**

The primary purpose of a java Singleton class is to restrict the limit of the number of object creations to only one. This often ensures that there is access control to resources, for example, socket or database connection.

Memory space wastage does not occur with the use of the singleton class because it restricts instance creation. As the object creation will take place only once instead of creating it each time a new request is made.

We can use this single object repeatedly as per the requirements. This is the reason why multi-threaded and database applications mostly make use of the Singleton pattern in Java for caching, logging, thread pooling, configuration settings, and much more.

For example, there is a license with us, and we have only one database connection or suppose our JDBC driver does not allow us to do multithreading, then the Singleton class comes into the picture and makes sure that at a time, only a single connection or a single thread can access the connection.

## **How to Design/Create a Singleton Class in Java?**

To create a singleton class, we must follow the steps, given below:

**1.** Ensure that only one instance of the class exists.

**2.** Provide global access to that instance by

* Declaring all constructors of the class to be private.
* Providing a static method that returns a reference to the instance. The lazy initialization concept is used to write the static methods.
* The instance is stored as a private static variable.

Example of singleton classes is **Runtime class, Action Servlet, and Service Locator**. Private constructors and factory methods are also an example of the singleton class.

## **Difference between Normal Class and Singleton Class**

We can distinguish a Singleton class from the usual classes with respect to the process of instantiating the object of the class. To instantiate a normal class, we use a java constructor. On the other hand, to instantiate a singleton class, we use the getInstance() method.

The other difference is that a normal class vanishes at the end of the lifecycle of the application while the singleton class does not destroy with the completion of an application.

### **Forms of Singleton Class Pattern**

There are two forms of singleton design patterns, which are:

* **Early Instantiation:** The object creation takes place at the load time.
* **Lazy Instantiation:** The object creation is done according to the requirement.

**// Java program implementing Singleton class**

**// with using getInstance() method**

**// Class 1**

**// Helper class**

**class Singleton {**

**// Static variable reference of single\_instance**

**// of type Singleton**

**private static Singleton single\_instance = null;**

**// Declaring a variable of type String**

**public String s;**

**// Constructor**

**// Here we will be creating private constructor**

**// restricted to this class itself**

**private Singleton()**

**{**

**s = "Hello I am a string part of Singleton class";**

**}**

**// Static method**

**// Static method to create instance of Singleton class**

**public static synchronized Singleton getInstance()**

**{**

**if (single\_instance == null)**

**single\_instance = new Singleton();**

**return single\_instance;**

**}**

**}**

**// Class 2**

**// Main class**

**class GFG {**

**// Main driver method**

**public static void main(String args[])**

**{**

**// Instantiating Singleton class with variable x**

**Singleton x = Singleton.getInstance();**

**// Instantiating Singleton class with variable y**

**Singleton y = Singleton.getInstance();**

**// Instantiating Singleton class with variable z**

**Singleton z = Singleton.getInstance();**

**// Printing the hash code for above variable as**

**// declared**

**System.out.println("Hashcode of x is "**

**+ x.hashCode());**

**System.out.println("Hashcode of y is "**

**+ y.hashCode());**

**System.out.println("Hashcode of z is "**

**+ z.hashCode());**

**// Condition check**

**if (x == y && y == z) {**

**// Print statement**

**System.out.println(**

**"Three objects point to the same memory location on the heap i.e, to the same object");**

**}**

**else {**

**// Print statement**

**System.out.println(**

**"Three objects DO NOT point to the same memory location on the heap");**

**}**

**}**

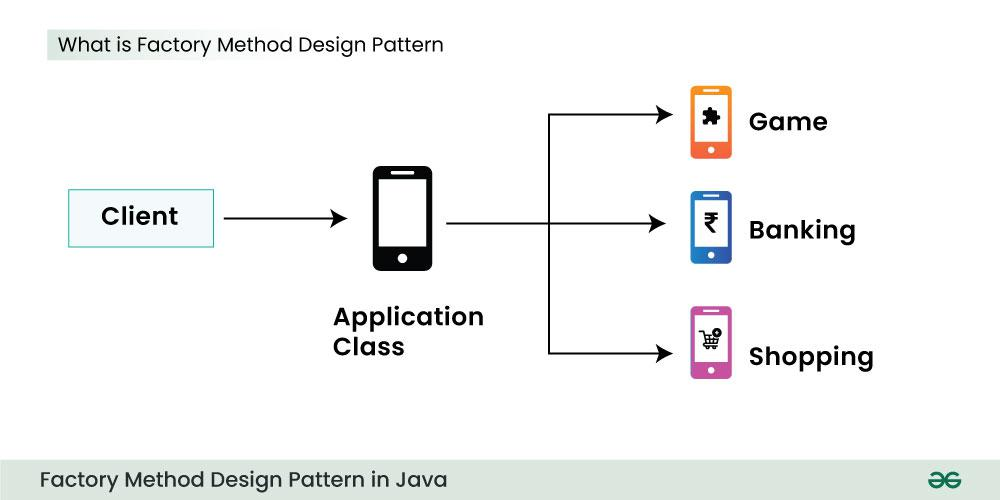
**}**

# **Factory method design pattern in Java**

**It is a** [**creational design pattern**](https://www.geeksforgeeks.org/creational-design-pattern/) **that talks about the creation of an object. The factory design pattern says to define an interface ( A java interface or an abstract class) for creating the object and let the subclasses decide which class to instantiate.**

## **What is the Factory Method Design Pattern in Java?**

***Factory Method Design Pattern define an interface for creating an object, but let subclass decide which class to instantiate. Factory Method lets a class defer instantiation to subclass.***

****

**Below is the explanation of the above image:**

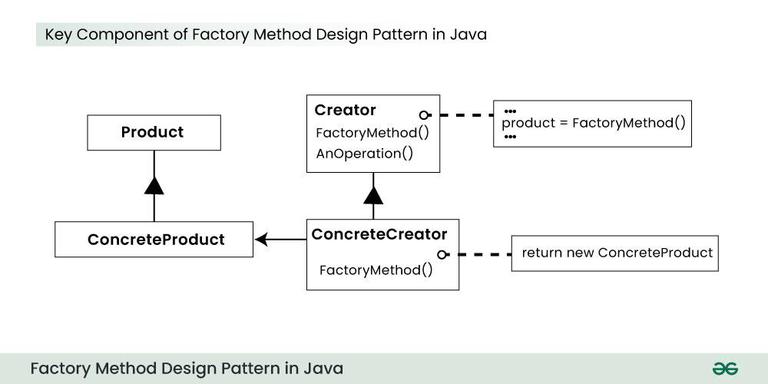
* **The factory method in the interface lets a class defer the instantiation to one or more concrete subclasses.**
* **Since these design patterns talk about the instantiation of an object they come under the category of creational design pattern.**
* **If we notice the name Factory method, that means there is a method which is a factory, and in general, factories are involved with creational stuff and here with this, an object is being created.**
* **It is one of the best ways to create an object where object creation logic is hidden from the client. Now Let’s look at the implementation.**

## **When to use Factory Method Design Pattern in Java?**

**Factory method design pattern can be used in java in following cases:**

* **A class cannot predict the type of objects it needs to create.**
* **A class wants its subclasses to specify the objects it creates.**
* **Classes delegate responsibility to one of multiple helper subclasses, and you aim to keep the information about which helper subclass is the delegate within a specific scope or location.**

## **Key Components of Factory Method Design Pattern**

****

### **Product**

* **It’s an abstract class or interface that defines the common operations for the objects that the factory will create.**
* **Concrete Products are the actual classes that implement the Product interface, each representing a specific type of object to be created.**

### **Creator**

* **It’s an abstract class or interface that declares the factory method.**
* **This method is responsible for creating Product objects, but it delegates the actual creation to subclasses.**

### **Concrete Creators**

* **These are subclasses of the Creator that implement the factory method.**
* **They decide which specific Concrete Product to create, often based on input parameters or configuration.**

### **Factory Method**

* **It’s a method defined in the Creator class that is responsible for creating Product objects.**
* **It’s typically declared as abstract in the Creator and implemented in the Concrete Creators.**

## **Factory Method Design Pattern Example in Java**

### **Problem Statement**

***You are developing a software system for an e-commerce platform that deals with various types of products. Each product category (e.g., electronics, clothing, books) requires specific handling during creation. However, you want to decouple the client code from the concrete product creation logic to enhance flexibility and maintainability. Additionally, you want to allow for easy extension by adding new product types in the future without modifying existing code.***

### **Solution using Abstract Class**

**The above problem can be solved using Factory Method Design Pattern:**

* **Java**

**// Abstract Product Class**

**abstract class Product {**

**public abstract void display();**

**}**

**// Concrete Products**

**class ConcreteProductA extends Product {**

**@Override**

**public void display() {**

**System.out.println("This is Concrete Product A.");**

**}**

**}**

**class ConcreteProductB extends Product {**

**@Override**

**public void display() {**

**System.out.println("This is Concrete Product B.");**

**}**

**}**

**// Creator Abstract Class**

**abstract class Creator {**

**public abstract Product factoryMethod();**

**}**

**// Concrete Creators**

**class ConcreteCreatorA extends Creator {**

**@Override**

**public Product factoryMethod() {**

**return new ConcreteProductA();**

**}**

**}**

**class ConcreteCreatorB extends Creator {**

**@Override**

**public Product factoryMethod() {**

**return new ConcreteProductB();**

**}**

**}**

**// Client Code**

**public class FactoryMethodExample {**

**public static void main(String[] args) {**

**Creator creatorA = new ConcreteCreatorA();**

**Product productA = creatorA.factoryMethod();**

**productA.display();**

**Creator creatorB = new ConcreteCreatorB();**

**Product productB = creatorB.factoryMethod();**

**productB.display();**

**}**

**}**

**—-----------------------------------------**

**// Product Interface**

**interface Product {**

**void display();**

**}**

**// Concrete Products**

**class ConcreteProductA implements Product {**

**@Override**

**public void display() {**

**System.out.println("This is Concrete Product A.");**

**}**

**}**

**class ConcreteProductB implements Product {**

**@Override**

**public void display() {**

**System.out.println("This is Concrete Product B.");**

**}**

**}**

**// Factory Interface**

**interface Factory {**

**Product factoryMethod();**

**}**

**// Concrete Factories**

**class ConcreteFactoryA implements Factory {**

**@Override**

**public Product factoryMethod() {**

**return new ConcreteProductA();**

**}**

**}**

**class ConcreteFactoryB implements Factory {**

**@Override**

**public Product factoryMethod() {**

**return new ConcreteProductB();**

**}**

**}**

**// Client Code**

**public class FactoryMethodExample {**

**public static void main(String[] args) {**

**Factory factoryA = new ConcreteFactoryA();**

**Product productA = factoryA.factoryMethod();**

**productA.display();**

**Factory factoryB = new ConcreteFactoryB();**

**Product productB = factoryB.factoryMethod();**

**productB.display();**

**}**

**}**

## **Use Cases of the Factory Method Design Pattern in Java**

**Here are some common applications of the Factory Method Design pattern in Java:**

* **Creational Frameworks:**
  + **JDBC (Java Database Connectivity) uses factories extensively for creating connections, statements, and result sets. Dependency injection frameworks like Spring and Guice rely heavily on factories to create and manage beans.**
* **GUI Toolkits:**
  + **Swing and JavaFX use factories to create UI components like buttons, text fields, and labels, allowing for customization and flexibility in UI design.**
* **Logging Frameworks:**
  + **Logging frameworks like Log4j and Logback use factories to create loggers with different configurations, enabling control over logging levels and output destinations.**
* **Serialization and Deserialization:**
  + **Object serialization frameworks often use factories to create objects from serialized data, supporting different serialization formats and versioning.**
* **Plugin Systems:**
  + **Plugin-based systems often use factories to load and create plugin instances dynamically, allowing for extensibility and customization.**
* **Game Development:**
  + **Game engines often use factories to create different types of game objects, characters, and levels, promoting code organization and flexibility.**
* **Web Development:**
  + **Web frameworks sometimes use factories to create view components, controllers, and services, enabling modularity and testability in web applications.**

## **Advantages of Factory Method Design Pattern in Java**

**The advantages of Factory Method Design Pattern in Java are:**

* **Decoupling: It separates object creation logic from the client code that uses those objects. This makes the code more flexible and maintainable because changes to the creation process don’t require modifications to client code.**
* **Extensibility: It’s easy to introduce new product types without changing the client code. You simply need to create a new Concrete Creator subclass and implement the factory method to produce the new product.**
* **Testability: It simplifies unit testing by allowing you to mock or stub out product creation during tests. You can test different product implementations in isolation without relying on actual object creation.**
* **Code Reusability: The factory method can be reused in different parts of the application where object creation is needed. This promotes centralizing and reusing object creation logic.**
* **Encapsulation: It hides the concrete product classes from the client code, making the code less dependent on specific implementations. This improves maintainability and reduces coupling.**

## **Disadvantages of Factory Method Design Pattern in Java**

**The disavantages of Factory Method Design Pattern in Java are:**

* **Increased Complexity: It introduces additional classes and interfaces, adding a layer of abstraction that can make the code more complex to understand and maintain, especially for those unfamiliar with the pattern.**
* **Overhead: The use of polymorphism and dynamic binding can slightly impact performance, although this is often negligible in most applications.**
* **Tight Coupling Within Product Hierarchies: Concrete Creators are still tightly coupled to their corresponding Concrete Products. Changes to one often necessitate changes to the other.**
* **Dependency on Concrete Subclasses: The client code still depends on the abstract Creator class, requiring knowledge of its concrete subclasses to make correct factory method calls.**
* **Potential for Overuse: It’s important to use the Factory Method pattern judiciously to avoid over-engineering the application. Simple object creation can often be handled directly without the need for a factory.**
* **Testing Challenges: Testing the factory logic itself can be more complex.**