**Singleton and Reflection Issues in Java**

1. **Public Constructor: Advantages and Disadvantages**

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method, which is simply a static method that returns an instance of the class.

**A public constructor** allows creating multiple instances of a class, but sometimes this is not desired. One alternative is static factory methods, which have advantages:

**1.Named Methods:** Unlike constructors, factory methods can have meaningful names, making code more readable.

**Example:**

// Constructor approach (less readable)  
BigInteger prime = new BigInteger(32, 100, new Random());  
  
// Factory method (more readable)  
BigInteger prime = BigInteger.probablePrime(32, new Random());

**2.One application of this flexibility is that an API can return objects without making their lasses public**.

**3.Reusability:** They can return existing instances, optimizing memory usage.

**3.Encapsulation:** They are not required to create a new object each time they’re invoked. They can control instance creation logic, unlike direct constructor calls. This allows immutable classes (Item 17) to use pre-constructed instances, or to cache instances as they’re constructed, and dispense (distribute) them repeatedly to avoid creating unnecessary duplicate objects. For example, the Boolean.valueOf(boolean) method illustrates this technique: it never creates an object. This technique is similar to the Flyweight pattern:

***The Flyweight design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. It aims to reduce the number of objects created and to decrease memory footprint, which is particularly useful when dealing with a large number of similar objects.***

The ability of static factory methods to return the same object from repeated invocations allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be **instance-controlled**.

**4.Return subclass Type:** They can return an object of any subtype of their return type unlike constructors. Unlike constructors, static factory methods can return any subtype of their declared return type. This provides greater flexibility in choosing the actual class of the returned object.

**5.By this method you can hide implementation classes:** Another advantage is that static factory methods can return instances without exposing their implementation classes. This makes the API more compact and easier to understand. For example, in the Java Collections Framework, there are 45 different utility implementations (e.g., unmodifiable collections, synchronized collections). Instead of making all these classes public, they are exposed via static factory methods in java.util.Collections.This reduces:

* **API bulk** – fewer public classes to manage.
* **Conceptual complexity** – programmers only need to understand the interface, not its implementations.
* **Dependency on implementation details** – users rely on the interface rather than a specific class.

## *Static Factory Methods Promote (extend/Upgraded) Interface-Based Design*

Since these methods typically return an interface type rather than a concrete (بدنه متد) implementation, they encourage best practices like coding to interfaces instead of implementations (see Item 64 in Effective Java).

* ***Changes in Java 8 & 9***
* **Before Java 8:** Interfaces couldn't have static methods, so companion utility classes (like Collections) were necessary.
* **Java 8:** Introduced static methods inside interfaces, so in some cases, a separate utility class is no longer needed.
* **Java 9:** Introduced private static methods in interfaces, but static fields and static inner classes in interfaces still need to be public.

## Static Methods & Overriding

**Static Methods Cannot Be Overridden But can be hidden**

Static methods belong to the class, not the instance. So, if a subclass defines a static method with the same name as a superclass, it hides the method rather than overriding it.

**Example: Static Method Hiding**

class Parent {  
 static void show() {  
 System.*out*.println("Static method in Parent");  
 }  
}  
  
class Child extends Parent {  
 static void show() { // This is hiding, not overriding  
 System.*out*.println("Static method in Child");  
 }  
}  
  
public class Test {  
 public static void main(String[] args) {  
 Parent p = new Child();  
 p.*show*(); // Output: Static method in Parent  
 }  
}

The method call p.show() invokes the Parent class’s version of show(), even though p is a Child instance.

**Static Methods Can Return a Subtype (Covariant Return Type)**

Even though static methods cannot be overridden, a subclass can define a static method with the same name and return a subtype.

**Example: Returning a Subtype**

class Parent {  
 static Parent getInstance() {  
 return new Parent();  
 }  
}  
  
class Child extends Parent {  
 static Child getInstance() { // Returns a subtype  
 return new Child();  
 }  
}  
  
public class Test {  
 public static void main(String[] args) {  
 Parent p = Parent.*getInstance*();  
 System.*out*.println(p.getClass().getSimpleName()); // Output: Parent  
  
 Child c = Child.*getInstance*();  
 System.*out*.println(c.getClass().getSimpleName()); // Output: Child  
  
 Parent cp = Child.*getInstance*();  
 System.*out*.println(cp.getClass().getSimpleName()); // Output: Parent  
 }  
}

**This is not overriding but method hiding with a covariant return type.**

# Static factory methods

**Static factory methods offer more flexibility than constructors. They allow returning a subtype and hiding implementation details. They reduce API complexity and promote interface-based programming. Static methods cannot be overridden but can be hidden. A static method in a subclass can return a subtype (covariant return).**

**One key advantage of static factory methods is that the class of the returned object can change depending on the input parameters. Moreover, the returned object's class can also change between different releases, without breaking client code. Example:**

**EnumSet** A great example of this technique is the EnumSet class (Item 36 in *Effective Java*). EnumSet has no public constructors; instead, it relies exclusively on static factory methods to create instances.

In the OpenJDK implementation, EnumSet has two internal implementations, and the static factory method chooses between them based on the number of elements in the enum:

* If the enum has 64 or fewer elements (which is the case for most enums), the factory method returns an instance of RegularEnumSet.
  + This implementation uses a single long value to store elements efficiently.
* If the enum has more than 64 elements, the factory method returns an instance of JumboEnumSet.
  + This implementation uses a long array to store elements.

public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {  
 Enum<?>[] universe = getUniverse(elementType);  
 if (universe == null)  
 throw new ClassCastException(elementType + " not an enum");  
  
 if (universe.length <= 64)  
 return new RegularEnumSet<>(elementType, universe);  
 else  
 return new JumboEnumSet<>(elementType, universe);  
}

**Benefits of This Approach:**

* **Encapsulation of Implementation Details:** Clients do not need to know about RegularEnumSet or JumboEnumSet—they only interact with EnumSet.
* **Flexibility for Future Changes:** If RegularEnumSet no longer provides a performance benefit, it can be removed in future versions without affecting client code. Similarly, new implementations can be introduced in the future to improve performance, without requiring changes from the users of EnumSet.

**Static factories also mean that the class of the returned object need not exist when the class containing the method is written.** This is a key concept in service provider frameworks, such as Java Database Connectivity (JDBC). In this situation we use the Factory pattern.

## What is a Service Provider Framework?

A service provider framework allows different providers to implement a service while keeping clients decoupled from the actual implementations. This means that a client does not need to know which specific implementation it is using.

**Main Components of a Service Provider Framework:**

* **Service Interface:** Defines the contract for the service that providers must implement.

public interface PaymentService {  
 void processPayment(double amount);  
}  
public class PayPalPaymentService implements PaymentService {  
 @Override  
 public void processPayment(double amount) {  
 System.*out*.println("Processing PayPal payment of $" + amount);  
 }  
}  
  
// Stripe Implementation  
public class StripePaymentService implements PaymentService {  
 @Override  
 public void processPayment(double amount) {  
 System.*out*.println("Processing Stripe payment of $" + amount);  
 }  
}

* **Provider Registration API:** Allows service providers to register their services.

public interface PaymentServiceProvider {  
 PaymentService createService();  
}  
  
public class PayPalServiceProvider implements PaymentServiceProvider {  
 @Override  
 public PaymentService createService() {  
 return new PayPalPaymentService();  
 }  
}  
  
// Stripe Service Provider  
public class StripeServiceProvider implements PaymentServiceProvider {  
 @Override  
 public PaymentService createService() {  
 return new StripePaymentService();  
 }  
}

* **Service Access API:** Clients use this to obtain service instances. This API acts as a flexible static factory method, enabling clients to:
* Specify criteria for selecting an implementation.
* Get a default implementation if no criteria are provided.
* Cycle through all available implementations.

import java.util.HashMap;  
import java.util.Map;  
  
public class PaymentServiceFactory {  
 private static final Map<String, PaymentServiceProvider> *providers* = new HashMap<>();  
  
 // Register providers  
 public static void registerProvider(String name, PaymentServiceProvider provider) {  
 *providers*.put(name, provider);  
 }  
  
 // Get a service instance  
 public static PaymentService getPaymentService(String name) {  
 PaymentServiceProvider provider = *providers*.get(name);  
 if (provider == null) {  
 throw new IllegalArgumentException("No provider registered with name: " + name);  
 }  
 return provider.createService();  
 }  
}

* *Optional Component: Service Provider Interface (SPI):* ***Some frameworks include an SPI, which describes a factory object responsible for creating instances of the service interface. If an SPI is not provided, implementations must be instantiated reflectively (see Item 65 in Effective Java).***

## Factory Pattern

A factory pattern is useful when you need a controlled way to create instances of a class:

public class PersonFactory {  
 public static Person createPerson(String name, int age) {  
 return new Person(name, age);  
 }  
}  
class Person {  
 private final String name;  
 private final int age;  
  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}

Factory pattern provides abstraction and improves maintainability.

**Example: JDBC as a Service Provider Framework**

The JDBC API follows this structure:

* **Service Interface:** → Connection
* **Provider Registration API:** → DriverManager.registerDriver
* **Service Access API:** → DriverManager.getConnection
* **Service Provider Interface:** → Driver

**Variations of Service Provider Frameworks:**

* The service access API can return an enhanced version of the service interface, rather than the one provided by service providers. This follows the Bridge Pattern (Design Patterns, Gamma et al., 1995).
* Dependency injection frameworks (see Item 5 in *Effective Java*) can be seen as advanced service providers.

**Modern Alternative: ServiceLoader**

Since Java 6, the platform includes java.util.ServiceLoader, a built-in service provider framework. Instead of writing custom service provider frameworks, developers should prefer using ServiceLoader whenever possible (see Item 59 in *Effective Java*).

* JDBC does not use ServiceLoader because it was designed before Java 6.
* ServiceLoader is a standard mechanism in Java for implementing the Service Provider Framework. It was introduced in Java 6 and improved in Java 9 to support modules. It allows automatic discovery and loading of service providers without requiring manual registration in code.

**What Problem Does ServiceLoader Solve?**

In the traditional approach (like the code in your document), we register service providers manually:

* Define the Service Interface.
* Implement Different Payment Services.
* Register Service Providers.

To enable ServiceLoader to find service providers, create a configuration file in:

* **File Path:** META-INF/services/com.example.PaymentService
* **File Content:** Include each service implementation class package address.

**Use ServiceLoader to Dynamically Load Services in a factory:**

import java.util.ServiceLoader;  
  
public class PaymentApplication {  
 public static void main(String[] args) {  
 // Load all available PaymentService implementations  
 ServiceLoader<PaymentService> services = ServiceLoader.*load*(PaymentService.class);  
  
 for (PaymentService service : services) {  
 service.processPayment(100.0);  
 }  
 }  
}  
import java.util.HashMap;  
import java.util.Map;  
  
public class PaymentServiceFactory {  
 private static final Map<String, PaymentService> *services* = new HashMap<>();  
  
 // Load registered services  
 public PaymentServiceFactory() {  
 ServiceLoader<PaymentService> servicesList = ServiceLoader.*load*(PaymentService.class);  
  
 for (PaymentService service : servicesList) {  
 *services*.put(PaymentService.class.getName(), service);  
 }  
 }  
  
 // Get a service instance  
 public static PaymentService getPaymentService(String name) {  
 PaymentService service = *services*.get(name);  
 if (service == null) {  
 throw new IllegalArgumentException("No provider registered with name: " + name);  
 }  
 return service;  
 }  
}

## Creating Singleton Pattern by Static Factory Method

public class Singleton {  
 private static Singleton *singleton*;  
   
  
 private Singleton() {  
 } // ❌ چون private هست، کسی نمی‌تونه از این کلاس ارث‌بری کنه!  
  
 public static Singleton getInstance() {  
 if(*singleton*==null){  
 *singleton*= new Singleton();  
  
 }  
 return *singleton*;  
 }  
}

### **Singleton and Subclassing Issue**

One major issue with implementing Singleton using a traditional class is that **it prevents subclassing**. Since the constructor is private, you cannot extend the class.However, **inner classes** can still extend it because they have access to the private constructor. For example:

public class Singleton {  
 private static Singleton *singleton*;  
  
 private Singleton() {  
 }  
  
 public static Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
 }  
  
 // Inner subclass (allowed because it has access to private constructor)  
 class SubSingleton extends Singleton {  
 }  
}

If you want to **allow subclassing** while still implementing the **Singleton pattern**, you can design your singleton class in a way that allows controlled inheritance. There are a few approaches to achieve this:

### **1. Protected Constructor**

By changing the constructor from private to protected, you allow subclasses to extend the singleton class, but still prevent direct instantiation from outside the class hierarchy.

public class Singleton {  
 private static Singleton *singleton*;  
 // 🔓 Protected constructor allows subclassing  
 protected Singleton() {  
 }  
 public static Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
 }  
}  
public class SubSingleton extends Singleton {  
 private static SubSingleton *subSingleton*;  
 private SubSingleton() {  
 super();  
 }  
 public static SubSingleton getInstance() {  
 if (*subSingleton* == null) {  
 *subSingleton* = new SubSingleton();  
 }  
 return *subSingleton*;  
 }  
}

**Drawback:**  
Each subclass would have its own singleton instance. If you want a single instance across a hierarchy, check the next solution.

### **2. Singleton per Subclass (Using Reflection and Generics)**

If you want **one singleton per subclass** without rewriting the getInstance() method every time, you can use **generics** and **reflection**:

import java.util.Map;  
import java.util.concurrent.ConcurrentHashMap;  
  
public abstract class Singleton {  
 private static final Map<Class<?>, Object> *instances* = new ConcurrentHashMap<>();  
 private static final Map<Class<?>, Boolean> *isBeingCreated* = new ConcurrentHashMap<>();  
  
 protected Singleton() {  
 Class<?> clazz = this.getClass();  
 // ✅ Check if the constructor was called from getInstance() method  
 if (!Boolean.*TRUE*.equals(*isBeingCreated*.get(clazz))) {  
 throw new RuntimeException("Reflection attack detected: Use getInstance() instead.");  
 }  
 }  
  
 @SuppressWarnings("unchecked")  
 public static <T> T getInstance(Class<T> clazz) {  
 return (T) *instances*.computeIfAbsent(clazz, c -> {  
 try {  
 *isBeingCreated*.put(c, true); // ✅ Allow constructor access only during instance creation  
 var constructor = c.getDeclaredConstructor();  
 constructor.setAccessible(true);  
 return constructor.newInstance();  
 } catch (Exception e) {  
 throw new RuntimeException("Cannot create singleton instance for: " + c.getName(), e);  
 } finally {  
 *isBeingCreated*.remove(c); // 🧹 Clean up after creation  
 }  
 });  
 }  
}  
public class SubSingletonA extends Singleton {  
 private SubSingletonA() {  
 super();  
 }  
  
 public static SubSingletonA getInstance() {  
 return Singleton.*getInstance*(SubSingletonA.class);  
 }  
}  
  
public class SubSingletonB extends Singleton {  
 private SubSingletonB() {  
 super();  
 }  
  
 public static SubSingletonB getInstance() {  
 return Singleton.*getInstance*(SubSingletonB.class);  
 }  
}

**How Does This Apply to Your Code?**

In your current Service Provider Framework code, you are using static factory methods inside PaymentServiceFactory:

public static PaymentService getPaymentService(String name) {  
 PaymentServiceProvider provider = providers.get(name);  
 if (provider == null) {  
 throw new IllegalArgumentException("No provider registered with name: " + name);  
 }  
 return provider.createService();  
}

Since this method returns an instance of PaymentService and there is no public constructor in the actual implementations (PayPalPaymentService or StripePaymentService), it means that clients cannot subclass these classes.

**Why is This Considered a "Blessing in Disguise"?**

* **Encourages Composition Over Inheritance:** Instead of subclassing PayPalPaymentService, developers should use it via composition, meaning they should wrap or delegate functionality rather than extend it.
* **Ensures Immutability (if needed):** If a class does not allow subclassing, it becomes easier to make immutable objects, which is useful for thread safety and functional programming.

**What if You Want to Allow Subclassing?**

* Provide public or protected constructors in PayPalPaymentService and StripePaymentService.
* Modify the factory to allow custom extensions, e.g., by providing a way to register new subclasses.

📌 **Should You Allow Subclassing in Your Service Provider Framework?**

* If you want flexibility for third-party extensions, allow subclassing.
* If you want strict control over implementations, keep using static factory methods without public constructors.

✅ **Solution:**

* If subclassing is required, Singleton might not be the best choice.
* Instead, use Dependency Injection to inject the instance where needed, avoiding the need for subclassing.

## Reflection and Serialization Breaking Singleton

Even if a class has a private constructor, reflection can bypass it:

class Person {  
 private final String name;  
 private final int age;  
  
 private Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public static Person createPerson(String name, int age) {  
 return new Person(name, age);  
 }  
}  
import java.lang.reflect.Constructor;  
  
public class ReflectionAttack {  
 public static void main(String[] args) throws Exception {  
 Constructor<Person> constructor = Person.class.getDeclaredConstructor(String.class, int.class);  
 constructor.setAccessible(true); // Breaks encapsulation  
 Person hackedPerson = constructor.newInstance("Hacker", 99);  
  
 hackedPerson.introduce(); // Hi, I'm Hacker and I'm 99 years old.  
 }  
}

✅ **Problem:** Even though the constructor is private, reflection allows unauthorized instance creation.

## Preventing Reflection Attacks

To prevent this, modify the Person class:

class Person {  
 private static boolean *instanceCreated* = false;  
 private final String name;  
 private final int age;  
  
 private Person(String name, int age) {  
 if (*instanceCreated != null*) {  
 throw new RuntimeException("Reflection attack detected! Only one instance allowed.");  
 }  
 *instanceCreated* = true;  
 this.name = name;  
 this.age = age;  
 }  
  
 public static Person createPerson(String name, int age) {  
 return new Person(name, age);  
 }  
}

✅ Now, even Reflection cannot create multiple instances.

## Why Can't a Singleton Be Serialized Normally?

When a class is Singleton, it means there should be only one instance throughout the application. However, if this class is Serializable, a serious problem may occur.

🚨 **The Main Problem:**

When we serialize and then deserialize a Singleton, a new instance of the class might be created, breaking the Singleton pattern!

import java.io.\*;  
  
class Singleton implements Serializable {  
 private static final long *serialVersionUID* = 1L; // For version control  
  
 private static final Singleton *INSTANCE* = new Singleton();  
  
 private Singleton() {} // 🔒 Prevents new instance creation  
  
 public static Singleton getInstance() {  
 return *INSTANCE*;  
 }  
}

Testing the Serialization Issue

import java.io.\*;  
  
public class SingletonSerializationTest {  
 public static void main(String[] args) {  
 try {  
 Singleton instance1 = Singleton.getInstance();  
  
 // 🚀 Serialize the Singleton  
 ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("singleton.ser"));  
 out.writeObject(instance1);  
 out.close();  
  
 // 🚀 Deserialize the Singleton  
 ObjectInputStream in = new ObjectInputStream(new FileInputStream("singleton.ser"));  
 Singleton instance2 = (Singleton) in.readObject();  
 in.close();  
  
 // 🤯 Check if both instances are the same  
 System.*out*.println("Instance 1 hashcode: " + instance1.hashCode());  
 System.*out*.println("Instance 2 hashcode: " + instance2.hashCode());  
 System.*out*.println("Are they same? " + (instance1 == instance2));  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Problematic Output:

Instance 1 hashcode: 12345678  
Instance 2 hashcode: 87654321  
Are they same? false ❌

Solution: Implement readResolve() to Prevent New Instances

import java.io.\*;  
  
class Singleton implements Serializable {  
 private static final long *serialVersionUID* = 1L;  
  
 private static final Singleton *INSTANCE* = new Singleton();  
  
 private Singleton() {}  
  
 public static Singleton getInstance() {  
 return *INSTANCE*;  
 }  
  
 // 🚀 Prevents new instance creation after Deserialization  
 protected Object readResolve() {  
 return *INSTANCE*;  
 }  
}

**Why Does readResolve() Fix the Issue?**

This method is called after deserialization. Instead of returning a new instance, it returns the original Singleton instance. Now, even after deserialization, there is only one instance of the Singleton!

**Output After Fix:**

Instance 1 hashcode: 12345678  
Instance 2 hashcode: 12345678  
Are they same? true ✅

## Using Enum for Singleton (Best Approach)

Enums completely prevent Reflection attacks:

public enum PersonSingleton {  
 *INSTANCE*;  
  
 private final String name = "Singleton Person";  
 private final int age = 40;  
  
 public void introduce() {  
 System.*out*.println("Hi, I'm " + name + " and I'm " + age + " years old.");  
 }  
}

✅ **Advantages:**

* JVM guarantees a single instance.
* Prevents Reflection and Serialization attacks.
* Thread-safe without extra code.

## Singleton Factory Pattern (Not Singleton Person)

If we only want a singleton factory, but allow multiple Person instances:

public enum PersonFactory {  
 *INSTANCE*;  
  
 public Person createPerson(String name, int age) {  
 return new Person(name, age);  
 }  
}

✅ Allows controlled creation of instances without enforcing a singleton Person

## Conclusion

* **If Person should be a singleton:** Use enum (Best Practice).
* **If Person should allow multiple instances but controlled:** Use a singleton factory.
* **Prevent Reflection attacks** by throwing exceptions in the constructor.
* **If you want to use singleton with serialization,** use the readResolve method override to return the current static instance.
* **Singleton is not ideal if subclassing is needed**—consider Dependency Injection instead.

It is harder for programmers to find Static factory methods compared to constructors, making it difficult to figure out how to instantiate a class that uses static factories. While the Javadoc tool may help highlight these methods in the future, one way to improve this issue is by clearly documenting static factories and using common naming conventions. Examples of such conventions include:

* **from**: for type-conversion methods (e.g., Date.from(instant)).
* **of**: for aggregation methods (e.g., EnumSet.of(JACK, QUEEN, KING)).
* **valueOf**: a more verbose alternative to from and of (e.g., BigInteger.valueOf(Integer.MAX\_VALUE)).
* **instance** or **getInstance**: to return an instance that’s described by parameters (e.g., StackWalker.getInstance(options)).
* **create** or **newInstance**: guarantees a new instance (e.g., Array.newInstance(classObject, arrayLen)).
* **getType**: used for methods in different classes (e.g., Files.getFileStore(path)).
* **newType**: like newInstance, but for methods in other classes (e.g., Files.newBufferedReader(path)).
* **type**: a concise alternative to getType and newType (e.g., Collections.list(legacyLitany)).

In conclusion, while both static factory methods and public constructors have their place, static factories are often preferable. Understanding their benefits can help developers choose between them effectively.