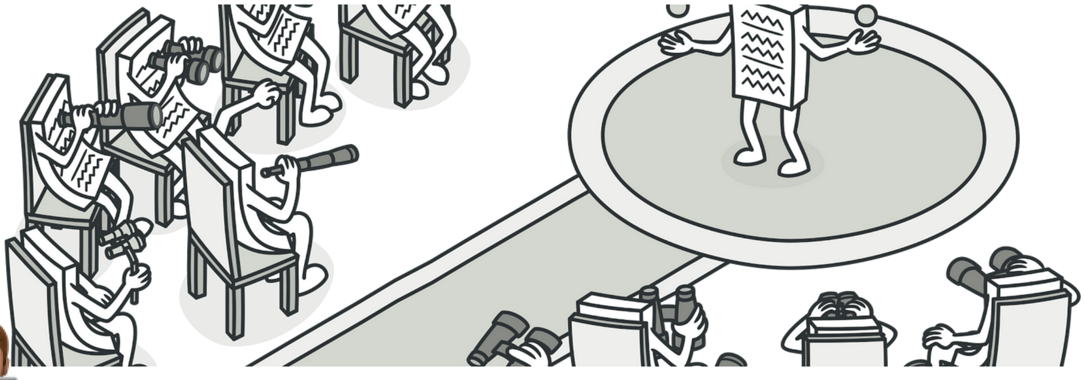
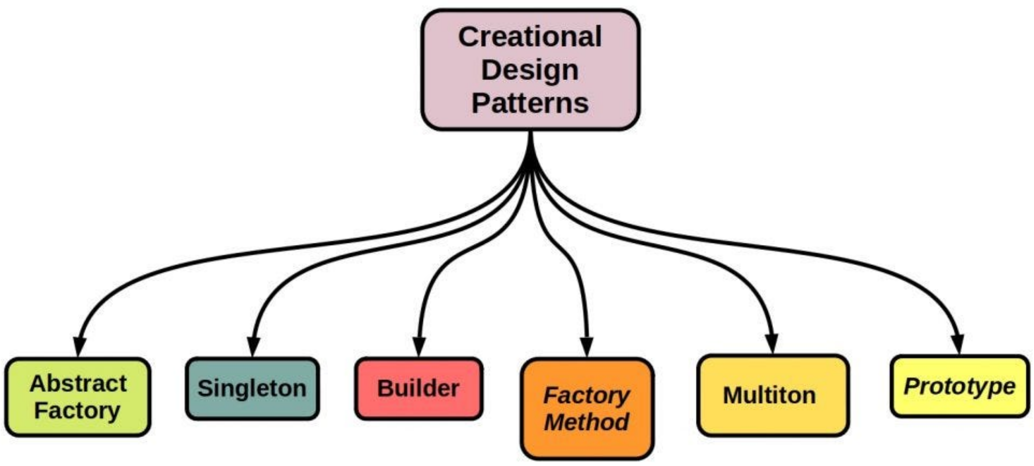


Design Patterns: Creational

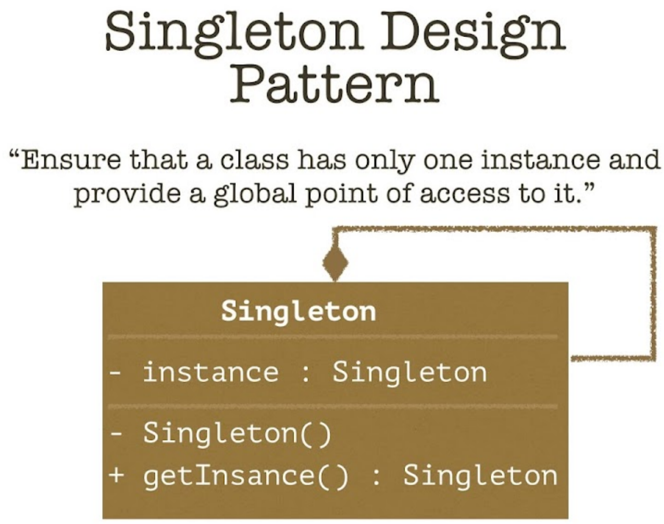
▼ Creational Patterns:

- Creational patterns focus on the **process of object creation**, ensuring that objects are created in a way that suits the system's design requirements.
- These patterns allow developers to manage and control how objects are instantiated, giving flexibility to change the instantiation process without altering existing code.



What is a Singleton Pattern?

- The Singleton Pattern ensures that a class has only **one instance** and provides a global point of access to that instance.
- This is particularly useful in scenarios where exactly one object is needed to coordinate actions across the system.



Why do we need a Singleton?

- Some applications require only a single instance of a class to control access to resources.
    - **Examples:**
        - Logger classes (global logging for the entire application).
        - Configuration managers (central management of app configuration).
        - Database connection pools (ensure only one connection pool exists).

Benefits of Singleton Pattern

- **Controlled access** to the sole instance.
- Rather than creating multiple objects, a single instance manages everything.
- All parts of the system use the same instance, ensuring uniform behaviour across the application.

Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

https://refactoring.guru/design-patterns/singleton

Singletons in Java | Baeldung

See how to implement the Singleton Design Pattern in plain Java.

https://www.baeldung.com/java-singleton

▼ Basic Implementation of Singleton Pattern

Here's a simple implementation of the Singleton pattern in Java:

```
public class Logger {

    // Step 1: Create a private static instance of the class
    private static Logger instance;

    // Step 2: Private constructor to prevent instantiation
    private Logger() {}

    // Step 3: Public method to provide global access to the instance
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    // Example method
    public void logMessage(String message) {
        System.out.println("Log: " + message);
```

```
    }
}
```

Key Points about the Basic Implementation

- **Private Constructor**: Prevents instantiation from outside the class.
- **Static Instance**: Ensures a single instance across the entire application.
- **Lazy Initialisation**: The instance is created only when it's needed (first time `getInstance()` is called).

▼ Thread-Safe Singleton Implementation

In a multi-threaded environment, multiple threads could try to instantiate the Singleton at the same time. To prevent this, we need to make the Singleton thread-safe.

1. **Synchronized Method** → One simple approach is to synchronise the `getInstance` method, but this can lead to performance issues.

```
public class ThreadSafeLogger {

    private static ThreadSafeLogger instance;

    private ThreadSafeLogger() {}

    public static synchronized ThreadSafeLogger
getInstance() {
        if (instance == null) {
            instance = new ThreadSafeLogger();
        }
        return instance;
    }

    public void logMessage(String message) {
        System.out.println("Log: " + message);
    }
}
```

2. **Double-Checked Locking → A** more efficient thread-safe approach using double-checked locking.

```
public class EfficientThreadSafeLogger {

    private static volatile EfficientThreadSafeL
ogger instance;

    private EfficientThreadSafeLogger() {}

    public static EfficientThreadSafeLogger getI
nstance() {
        if (instance == null) {
            synchronized (EfficientThreadSafeLog
ger.class) {
                if (instance == null) {
                    instance = new EfficientThre
adSafeLogger();
                }
            }
        }
        return instance;
    }

    public void logMessage(String message) {
        System.out.println("Log: " + message);
    }
}
```

Common Pitfalls in Singleton

- **Global State**: Singleton can introduce global state, making it harder to isolate components during testing.
- **Testing Challenges**: It's hard to mock or substitute the singleton class in unit tests, unless dependency injection or mock frameworks are used.
- **Tight Coupling**: Singleton can lead to tight coupling between classes, reducing flexibility and increasing dependency

management complexity.

▼ Eager Initialisation vs. Lazy Initialisation

Eager Initialisation: Singleton instance is created at the time of class loading.

```
public class EagerLogger {

    // Step 1: Initialize the instance at class loa
d time
    private static final EagerLogger instance = new
EagerLogger();

    private EagerLogger() {}

    public static EagerLogger getInstance() {
        return instance;
    }

    public void logMessage(String message) {
        System.out.println("Log: " + message);
    }
}
```

Lazy Initialisation: Singleton instance is created when it's actually needed, as shown in the previous examples.

```
public static void main(String[] args) {
        Logger logger = Logger.getInstance();
        logger.logMessage("Singleton pattern in act
ion!"); // Output: Log: Singleton pattern in actio
n!
    }
```

Which one to use?

- **Eager Initialisation**: Use when the instance is lightweight and expected to be used frequently.

- **Lazy Initialisation**: Use when the instance might not always be needed and can be created on demand.

Common Pitfalls of Singleton:

- **Global State**
  - A **Singleton** can inadvertently introduce **global state** into the application.
  - Global state refers to variables or data that are accessible throughout the entire application.
  - While Singleton ensures that only one instance of a class exists, it also means that every part of the program shares that one instance.
  - If that instance contains mutable data, it can lead to unintended consequences when different parts of the system change that state.

  Example Scenario:

  Imagine we have a Singleton `ConfigManager` that holds application-wide configuration settings.

```
public class ConfigManager {
    private static ConfigManager instance;
    private String setting;

    private ConfigManager() {}

    public static ConfigManager getInstance() {
        if (instance == null) {
            instance = new ConfigManager();
        }
        return instance;
    }

    public void setSetting(String setting) {
        this.setting = setting;
    }

    public String getSetting() {
```

```
        return setting;
    }
}
```

Since all parts of the program use the same instance of `ConfigManager`, a change in one part can unexpectedly affect other parts of the system.

Testing Example:

```
public class ConfigManagerTest {

    @Test
    void testGlobalStateIssue() {
        ConfigManager configManager = ConfigManager.getInstance();

        configManager.setSetting("Development");

        // In a different part of the program, another test runs
        ConfigManager anotherReference = ConfigManager.getInstance();
        anotherReference.setSetting("Production");

        // Original reference has now changed unexpectedly
        assertEquals("Production", configManager.getSetting());
    }
}
```

Here, the shared instance leads to a **global state** issue, where modifying the setting in one place affects all other places. This makes it difficult to predict the system's behaviour.

- **Testing Challenges**

- Testing Singleton classes is tricky because of their **global nature**.
- Since Singleton classes control their instantiation, it becomes hard to substitute them with **mock objects** or **different instances** in unit tests.
- It can also interfere with **test isolation**.

Example Scenario:

Imagine a Singleton `DatabaseConnection` that connects to a database.

```
public class DatabaseConnection {
    private static DatabaseConnection instance;

    private DatabaseConnection() {
        // Expensive connection setup
    }

    public static DatabaseConnection getInstance
() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public String query(String sql) {
        // Database query implementation
        return "Result";
    }
}
```

When running tests, we might want to **mock** the database connection or use a **different instance** for isolation, but Singleton makes this challenging.

Test Challenge:

```
public class DatabaseConnectionTest {

    @Test
    void testQuery() {
        DatabaseConnection dbConn = DatabaseConn
ection.getInstance();
        // Hard to isolate or mock this connecti
on in a unit test
        String result = dbConn.query("SELECT * F
ROM users");
        assertEquals("Result", result);
    }

    @Test
    void testWithMock() {
        DatabaseConnection mockConn = Mockito.mo
ck(DatabaseConnection.class);
        Mockito.when(mockConn.query("SELECT * FR
OM users")).thenReturn("Mocked Result");
        // But there's no easy way to inject thi
s mock into the Singleton structure
    }
}
```

Solution:

This can be mitigated by using **dependency injection** (DI) frameworks or testing libraries that allow **mocking singletons** (like `Mockito` with `PowerMock`). Alternatively, refactoring to avoid a Singleton can also resolve this issue.

- **Tight Coupling**
  - Singleton can create **tight coupling** between classes.
  - When multiple classes depend on a Singleton, it becomes harder to change the Singleton's implementation or switch to a different pattern.

- Over time, this can lead to **spaghetti code** and a **rigid architecture**.

Example Scenario:

Let's assume we have multiple classes relying on a `Logger` Singleton. As the system grows, they become tightly coupled to the specific Singleton implementation.

```
public class Logger {
    private static Logger instance;

    private Logger() {}

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println(message);
    }
}

// Multiple classes relying on Logger Singleton
public class ServiceA {
    public void performAction() {
        Logger.getInstance().log("ServiceA is pe
rforming an action");
    }
}

public class ServiceB {
    public void performAction() {
        Logger.getInstance().log("ServiceB is pe
rforming an action");
```

```
    }
}
```

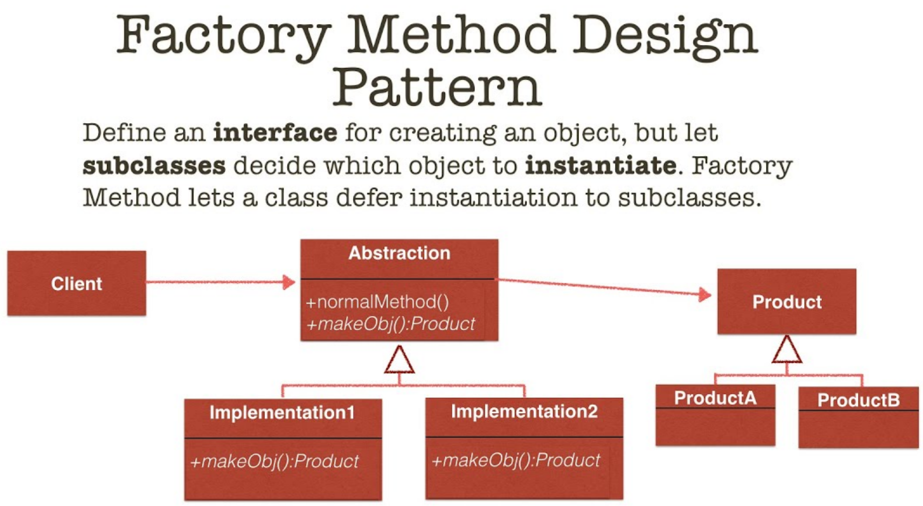
If we want to replace `Logger` with a different logging framework, we'd have to refactor **all classes** that rely on `Logger.getInstance()`, which introduces **tight coupling**.

Testing Example:

```
public class LoggerTest {

    @Test
    void testLoggerWithMultipleServices() {
        Logger logger = Logger.getInstance();

        // Logger instance used in multiple plac
es can create coupling issues
        ServiceA serviceA = new ServiceA();
        ServiceB serviceB = new ServiceB();

        serviceA.performAction(); // Relies on t
he same Logger
        serviceB.performAction(); // Relies on t
he same Logger
    }
}
```

▼ What is the Factory Method Pattern?

- The **Factory Method Pattern** defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
- The essence of the pattern is that **object creation is deferred to a specialised method**, often called a **factory method**.
    - **Problem**: You have a class that needs to create objects, but you want to delegate the responsibility of deciding which class to instantiate.
    - **Solution**: Use the **Factory Method Pattern**, where the object creation is delegated to subclasses or a specific factory class.



The Factory Design Pattern in Java | Baeldung

Explore the factory design pattern.

https://www.baeldung.com/java-factory-pattern

Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of

https://refactoring.guru/design-patterns/factory-method

Example Scenario

- Imagine you are building a logistics system.
- Depending on whether you are handling **land** or **sea transportation**, you will need to instantiate different kinds of vehicles, such as **trucks** or **ships**.
- In a standard scenario, you might use `new` to create these objects, but this approach would make your code less flexible if new vehicle types are introduced later.

Step-by-Step Example

Step 1: Define the Product Interface (Common Interface for Products)

You define a common interface for the types of objects you want to create.

```
public interface Transport {
    void deliver();
}
```

Step 2: Concrete Products (Specific Object Types)

You create concrete classes that implement the common interface, such as `Truck` and `Ship`.

```
public class Truck implements Transport {
    @Override
    public void deliver() {
        System.out.println("Delivering by land i
n a truck");
    }
}

public class Ship implements Transport {
    @Override
    public void deliver() {
        System.out.println("Delivering by sea in
a ship");
    }
}
```

Step 3: Factory Interface or Abstract Class

Now, define an abstract class (or an interface) that declares the factory method responsible for creating objects of type `Transport`.

```
public abstract class Logistics {
    // The Factory Method
    public abstract Transport createTransport();
```

```
    // Other methods using the product created b
y the factory method
    public void planDelivery() {
        Transport transport = createTransport();
        transport.deliver();
    }
}
```

Step 4: Concrete Factories (Classes that decide which product to create)

Concrete factory classes will override the factory method to decide which `Transport` to create.

```
public class RoadLogistics extends Logistics {
    @Override
    public Transport createTransport() {
        return new Truck();  // Concrete Product
(Truck)
    }
}

public class SeaLogistics extends Logistics {
    @Override
    public Transport createTransport() {
        return new Ship();  // Concrete Product
(Ship)
    }
}
```

Step 5: Client Code

The client code calls the factory method but doesn't need to know the exact class of the object that will be created.

```
public class LogisticsApp {
    public static void main(String[] args) {
        // Choosing the type of logistics dynami
cally
        Logistics logistics = new RoadLogistics
```

```
();
        logistics.planDelivery();  // Output: De
livering by land in a truck

        logistics = new SeaLogistics();
        logistics.planDelivery();  // Output: De
livering by sea in a ship
    }
}
```

Why Use the Factory Method Pattern?

- The factory method separates the process of creating an object from the client code that uses it — This allows you to introduce new types of products without modifying existing code.
- If new product types are introduced (e.g., **AirLogistics**), they can be handled by creating a new concrete class without modifying the existing code.
- It gives flexibility in object creation while ensuring the client remains decoupled from specific product implementations.

Common Pitfalls of Factory Method:

- **Over-complication**
  - The Factory Method Pattern introduces abstraction by creating additional classes (factory and product classes) to decouple object creation. However, if your application only requires a small number of product variations, this extra complexity may become burdensome rather than beneficial.
  - **Over-complication** occurs when the Factory Method Pattern introduces too much overhead for a problem that could be solved with simpler constructs, like constructors or static methods.

  Example:

  Consider a scenario where you're building a system that only deals with **two vehicle types**: `Car` and `Bike`.

  - If you apply the Factory Method Pattern here, you'll need:

- A `Vehicle` interface.
- A `Car` class implementing `Vehicle`.
- A `Bike` class implementing `Vehicle`.
- A `VehicleFactory` abstract class or interface.
- A `CarFactory` and `BikeFactory` that inherit from `VehicleFactory`.

- While this is technically correct, the amount of boilerplate code introduced far outweighs the benefit of using the Factory Method Pattern.
- For two types of vehicles, it might be better to use a **simple constructor** or a **static method** rather than adding unnecessary layers of abstraction.

```
// Example: Overcomplicated Factory for Two Vehicle Types
interface Vehicle {
    void move();
}

class Car implements Vehicle {
    @Override
    public void move() {
        System.out.println("Car is moving");
    }
}

class Bike implements Vehicle {
    @Override
    public void move() {
        System.out.println("Bike is moving");
    }
}

abstract class VehicleFactory {
    public abstract Vehicle createVehicle();
}
```

```
class CarFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
}

class BikeFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bike();
    }
}
```

In this case, simply using direct instantiation would be far more efficient:

```
// Simpler Code
Vehicle car = new Car();
Vehicle bike = new Bike();
```

- **Violation of the Open/Closed Principle**
  - The **Open/Closed Principle** (OCP) suggests that classes should be **open for extension but closed for modification**. This means that when you add new functionality (e.g., adding a new product type), you should **extend** existing classes rather than modifying them.
  - However, in some cases, the Factory Method Pattern can lead to violations of this principle if you find yourself constantly modifying existing factory logic to accommodate new products.

  Example:

  Suppose your logistics system initially only supports `Truck` and `Ship`.

  - Later, you need to introduce `Plane` and `Train`.

- If the factory classes or factory methods have to be modified repeatedly to accommodate these new vehicle types, you are violating OCP by constantly updating the same code.

  ```
  // Violating OCP by Modifying Factory
  class Logistics {
      public Transport createTransport(String t
  ype) {
          if (type.equals("Truck")) {
              return new Truck();
          } else if (type.equals("Ship")) {
              return new Ship();
          } else if (type.equals("Plane")) {
              return new Plane();  // Modifying
  the factory logic
          } else if (type.equals("Train")) {
              return new Train();  // Modifying
  again for Train
          } else {
              throw new IllegalArgumentExceptio
  n("Invalid transport type");
          }
      }
  }
  ```

  Each time you add a new type of vehicle, you modify the `createTransport` method. This violates the **Open/Closed Principle** because instead of **extending** the code with new subclasses, you're constantly **modifying** the original logic.

Solution:

To solve this, you can structure your code so that new vehicle types can be **extended** without modifying existing factory logic. This can be achieved by creating a separate factory for each new type of vehicle or by using an **Abstract Factory**.

```
// Extending Without Modifying Existing Code (OC
P Compliant)
abstract class TransportFactory {
```

```
    public abstract Transport createTransport();
}

class TruckFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Truck();
    }
}

class PlaneFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Plane();
    }
}

class TrainFactory extends TransportFactory {
    @Override
    public Transport createTransport() {
        return new Train();
    }
}
```

Now, adding a new type of vehicle (e.g., **Plane**) doesn't require modifying existing classes. You just need to create a new factory that extends `TransportFactory`, keeping the rest of the code intact.