Creational Design Patterns

Creational Design Patterns

- A creational design pattern deals with object creation and initialization, providing guidance about which objects are created for a given situation. These design patterns are used to increase flexibility and to reuse existing code.
- **Factory Method**: Creates objects with a common interface and lets a class defer instantiation to subclasses.
- **Abstract Factory**: Creates a family of related objects.
- **Builder**: A step-by-step pattern for creating complex objects, separating construction and representation.
- **Prototype**: Supports the copying of existing objects without code becoming dependent on classes.
- **Singleton**: Restricts object creation for a class to only one instance.

Singleton Design Pattern

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

- Intent
- To restrict the instantiation of a class to a single instance.
- To provide global access to that instance.

- (Why Singleton?)
- Some scenarios where having only one instance is essential include:

- Logging Service: Only one logger should be responsible for writing logs to avoid conflicts.
- Configuration Settings: A centralized access point ensures consistency.
- Database Connections: Managing a single connection pool for performance optimization.

- Features
- Private Constructor: Prevents creating objects from outside the class.

- Static Instance: Ensures the same instance is returned every time.

- Lazy Initialization: The instance is created only when it's needed.

- Pros
- Controlled access to the sole instance.
- Reduces memory footprint by preventing multiple instances.
- Particularly useful for resource management (e.g., logging, database connections).

- Cons
- Can be difficult to unit test due to global access.
- May introduce performance issues if implemented with poor synchronization in multi-threaded environments.
- Breaks the Single Responsibility Principle if it handles too much.

- Real-life analogy:

- “Imagine a government with only one president at a time. Regardless of who tries to become the president, there can be only one.”

- Relate this to Singleton: "In software, we sometimes need a class to have only one instance across the application—just like there's only one president."

- Class Diagram:

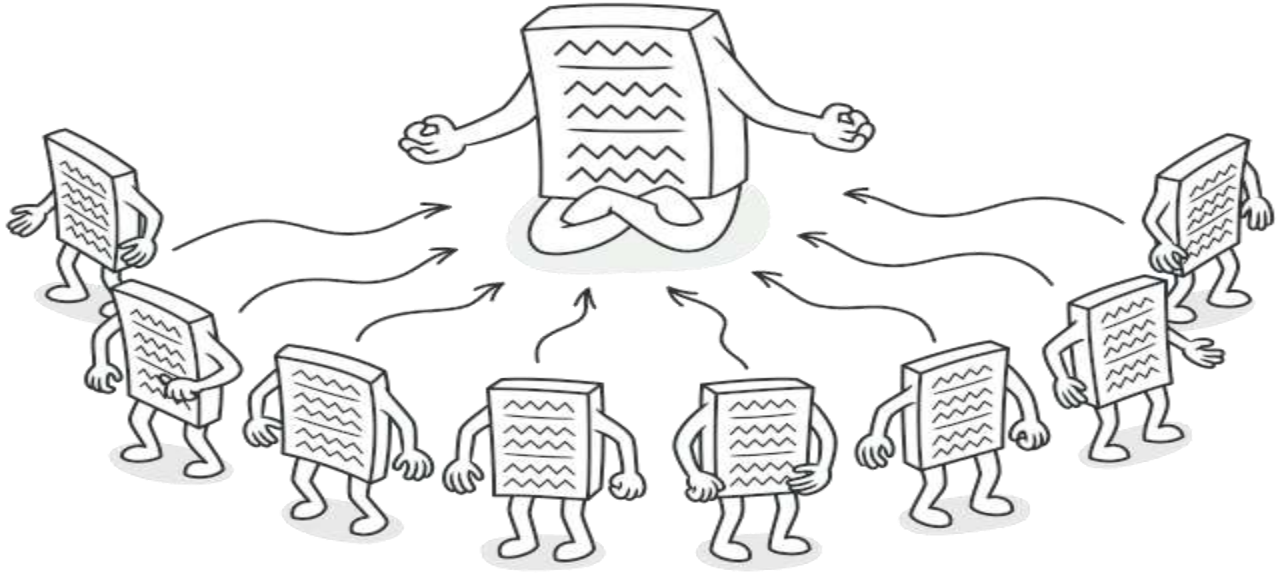
- Private constructor (to prevent external instantiation).

- Static method: getInstance().

- Static attribute: instance.

Singleton Design Pattern

- **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. **Singleton** is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.



- Problem

1. **Ensure that a class has just a single instance.** Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.
2. Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.
3. Note that this behavior is impossible to implement with a regular constructor since a constructor call **must** always return a new object by design.

- **2.Provide a global access point to that instance.** Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app.
- Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

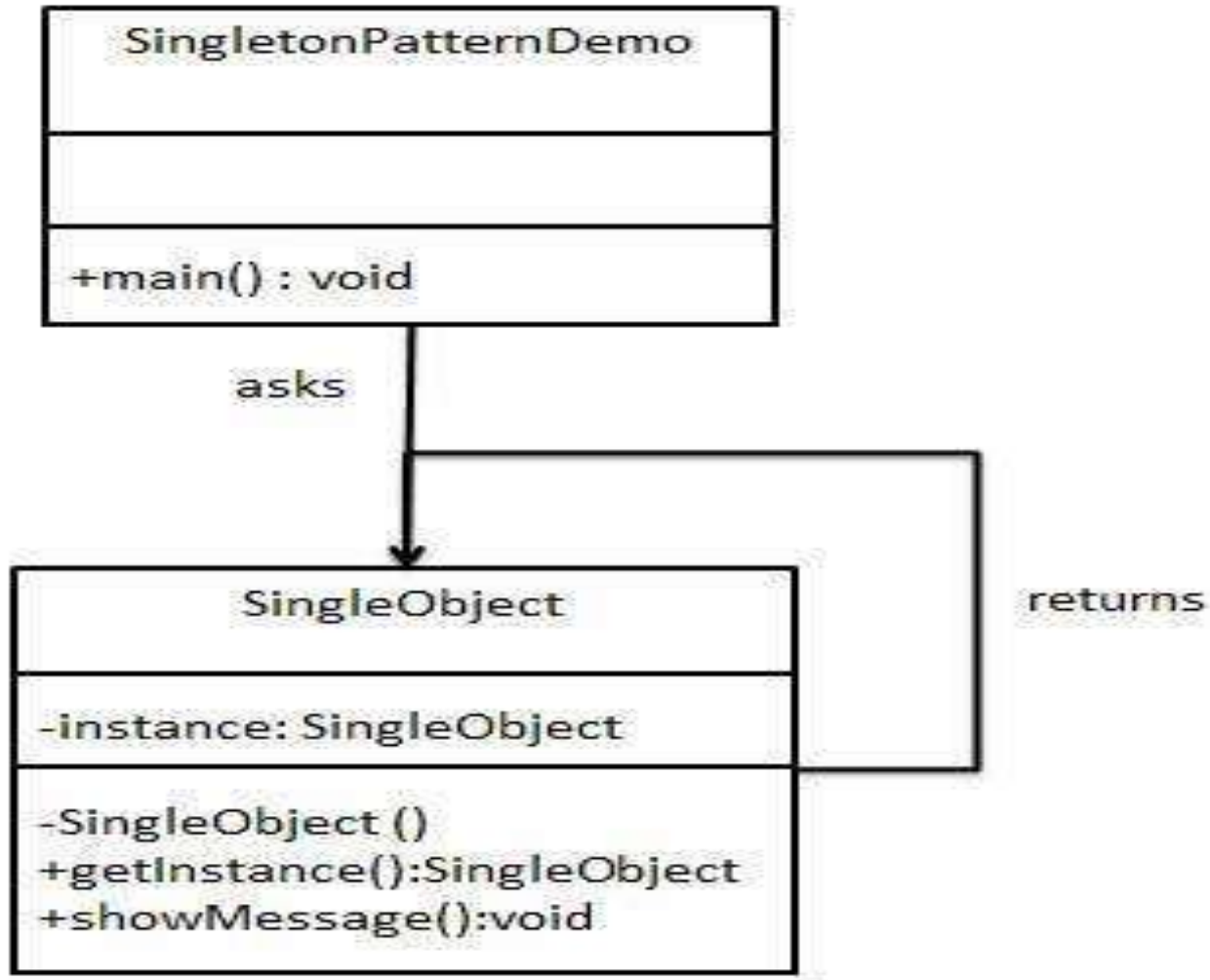
- Solution
- All implementations of the Singleton have these two steps in common:

- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor. This method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.
- If your code has access to the Singleton class, then it’s able to call the Singleton’s static method. So whenever that method is called, the same object is always returned.

- **Java Singleton Pattern Implementation**
- To implement a singleton pattern, we have different approaches, but all of them have the following common concepts.
- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for the outer world to get the instance of the singleton class.

Implementation

- We're going to create a SingleObject class.
- SingleObject class have its constructor as private and have a static instance of itself.
- SingleObject class provides a static method to get its static instance to outside world.
- SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.


SingletonPatternDemo
+main() : void
asks
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void
returns

Step 1
Create a Singleton Class.(SingleObject.java)

```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2
Get the only object from the singleton class. (*SingletonPatternDemo.java)*)

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visibl
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3
Verify the output.

```
Hello World!
```

- Applicability
- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Use the Singleton pattern when you need stricter control over global variables.

- **How to Implement**

1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.

Pros and Cons

- ✓ You can be sure that a class has only a single instance.
- ✓ You gain a global access point to that instance.
- ✓ The singleton object is initialized only when it's requested for the first time.

- ✗ Violates the *Single Responsibility Principle*. The pattern solves two problems at the time.
- ✗ The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
- ✗ The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- ✗ It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

Factory Design Pattern

- The Factory Design Pattern is a creational design pattern that provides a way to create objects without specifying the exact class of object that will be created.
- Instead of directly instantiating classes using the new keyword, you delegate the responsibility of creating instances to factory methods.

- (Why Factory Pattern?)
- Imagine you’re developing a furniture information system (like the one you're designing) where users can view various furniture items (e.g., Chair, Table, Sofa). If you instantiate objects directly, adding new furniture types would require modifying existing code.

- Instead, using a Factory Pattern, you define a general interface for creating furniture and let the subclasses specify their type. This makes the system easily extendable.

When to Use

- When the exact types and dependencies of objects to be created are not known until runtime.
- To encapsulate object creation logic in a central place.
- When you want to provide a common interface for creating objects, but want to allow subclasses to alter the type of objects created.

- Intent
- To encapsulate object creation to avoid direct instantiation.

- To allow subclasses to decide which class to instantiate.

- To promote flexibility and reusability.

- Features
- Decouples client code from specific classes - The client interacts only with interfaces or abstract classes.

- Simplifies object creation logic - Object creation code is in one place (the factory).

- Easily extendable - New classes can be added without changing existing code.

- Pros
- Encourages loose coupling.

- Centralizes the object creation process.

- Promotes code reusability and flexibility.

- Cons
- Increases complexity due to additional interfaces or classes.

- May introduce unnecessary abstraction if not used correctly.

EXAMPLE

- Let's say you are creating a notification system that sends alerts via different channels like SMS, Email, and Push Notifications.

```
// Step 1: Create a Notification interface
interface Notification {
    void notifyUser();
}

// Step 2: Concrete implementations of Notification
class SMSNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending an SMS notification");
    }
}
```

```
class EmailNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending an Email notification");
    }
}

class PushNotification implements Notification {
    @Override
    public void notifyUser() {
        System.out.println("Sending a Push notification");
    }
}
```

```
// Step 3: Create a NotificationFactory class
class NotificationFactory {
    public static Notification createNotification(String type) {
        if (type.equalsIgnoreCase("SMS")) {
            return new SMSNotification();
        } else if (type.equalsIgnoreCase("EMAIL")) {
            return new EmailNotification();
        } else if (type.equalsIgnoreCase("PUSH")) {
            return new PushNotification();
        }
        return null;
    }
}
```

```
// Step 4: Client code
public class Main {
    public static void main(String[] args) {
        Notification notification = NotificationFactory.createNotification("SMS");
        notification.notifyUser();
    }
}
```

Explanation

- NotificationFactory is the factory class responsible for creating objects of different types based on the input.
- The client (Main class) doesn't need to know the details of object creation. It simply calls NotificationFactory.createNotification().
- Open/Closed Principle: You can add new notification types without modifying existing code (just by adding a new class and updating the factory method).

Advantages

- Encapsulation: Object creation logic is isolated in a single place.
- Loose Coupling: Clients are decoupled from the concrete implementations.
- Scalability: New types can be added with minimal code change.

Disadvantages

- As the number of subclasses grows, the factory method can become complex and difficult to maintain.

- Violates Single Responsibility Principle if the factory is responsible for too many types.