Creational design patterns - Singleton

Key terms

Design patterns

> A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Creational design patterns

> Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Singleton

> The singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Singleton

Problem

- `Shared resource` - Imagine you have a class that is responsible for managing the database connection. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple database connections, which is not what you want. Similarly, there can be a class that is responsible for managing the logging mechanism. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple log files, which is not what you want.
- `Single access point` - Applications often require configuration. For example, you might want to configure the database connection parameters. You want to make sure that only one instance of this

class exists in your application. A configuration class should have a single access point to the configuration parameters. If you create multiple instances of this class, you will end up with multiple configuration files.

Solution

Singleton pattern 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. To implement the Singleton patter, the following steps are required:

- `Constructor hiding` - The constructor of the singleton class should be private or protected. This will prevent other classes from instantiating the singleton class.
- `Global access point` - The singleton class should provide a global access point to get the instance of the singleton class. This global access point should be static and should return the same instance of the singleton class every time it is called. If the instance does not exist, it should create the instance and then return it.

Simple singleton

The first step is to hide the constructor by making it private. This will prevent other classes from instantiating the singleton class.

```java
public class Database {
    private Database() {
    }
}
```

The above code restricts the instantiation of the Database class. Now, we need to provide a global access point to get the instance of the Database class. We can do this by creating a static method that returns the instance of the Database class. If the instance does not exist, it should create the instance and then return it.

```java
public class Database {
    private static Database instance = new Database();

    private Database() {
    }

    public static Database getInstance() {
        return instance;
    }
}
```

To implement the getInstance() method, we need to create a static variable of the Database class. This variable will hold the instance of the Database class. We will initialize this variable to null. The getInstance() method will check if the instance variable is null. If it is null, it will create a new instance of the Database class and assign it to the instance variable. Finally, it will return the instance variable. This is known as lazy initialization.

```java
public class Database {
    private static Database instance = null;

    private Database() {
    }

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

Thread-safe singleton

The above code is not thread-safe. If two threads call the getInstance() method at the same time, both threads will check if the instance variable is null. Both threads will find that the instance variable is null. Both threads will create a new instance of the Database class. This will result in two instances of the Database class. To make the above code thread-safe, we can make the getInstance() method synchronized.

```java
public class Database {
    private static Database instance = null;

    private Database() {
    }

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

Double-checked locking

The above code is thread-safe. However, it is not efficient. If two threads call the getInstance() method at the same time, both threads will check if the instance variable is null. Both threads will find that the instance variable is null. Both threads will wait for the lock to be released. Once the lock is released, one thread will create a new instance of the Database class. The other thread will wait for the lock to be released. Once the lock is released, it will create a new instance of the Database class. This will result in two instances of the Database class. To make the above code efficient, we can use double-checked locking.

```java
public class Database {
    private static Database instance = null;

    private Database() {
    }

    public static Database getInstance() {
        if (instance == null) {
            synchronized (Database.class) {
                if (instance == null) {
                    instance = new Database();
                }
            }
        }
        return instance;
    }
}
```

Summary

- The singleton pattern 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.
- Use cases of singleton pattern
  - Shared resource like database connection, logging mechanism
  - Object that should be instantiated only once like configuration object
- Hide the constructor of the singleton class by making it private so that other classes cannot instantiate the singleton class.
- Add a static method that returns the instance of the singleton class. If the instance does not exist, it should create the instance and then return it.
- Thread safety
  - Make the `getInstance()` method synchronized.
  - Use double-checked locking.

Reading list

- Singleton pattern - Different implementations