COMP2511

Creational Pattern:

Singleton Pattern

Prepared by

Dr. Ashesh Mahidadia

Creational Patterns

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

- Factory Method
  - provides an interface for creating objects in a superclass,
    but allows subclasses to alter the type of objects that will be created.
- Abstract Factory
  - let users produce families of related objects
    without specifying their concrete classes.
- Singleton
  - Let users ensure that a class has only one instance,
    while providing a global access point to this instance.

Singleton Pattern

Singleton Pattern

Intent: 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.

Problem: A client wants to,

- ensure that a class has just a single instance, and
- provide a global access point to that instance

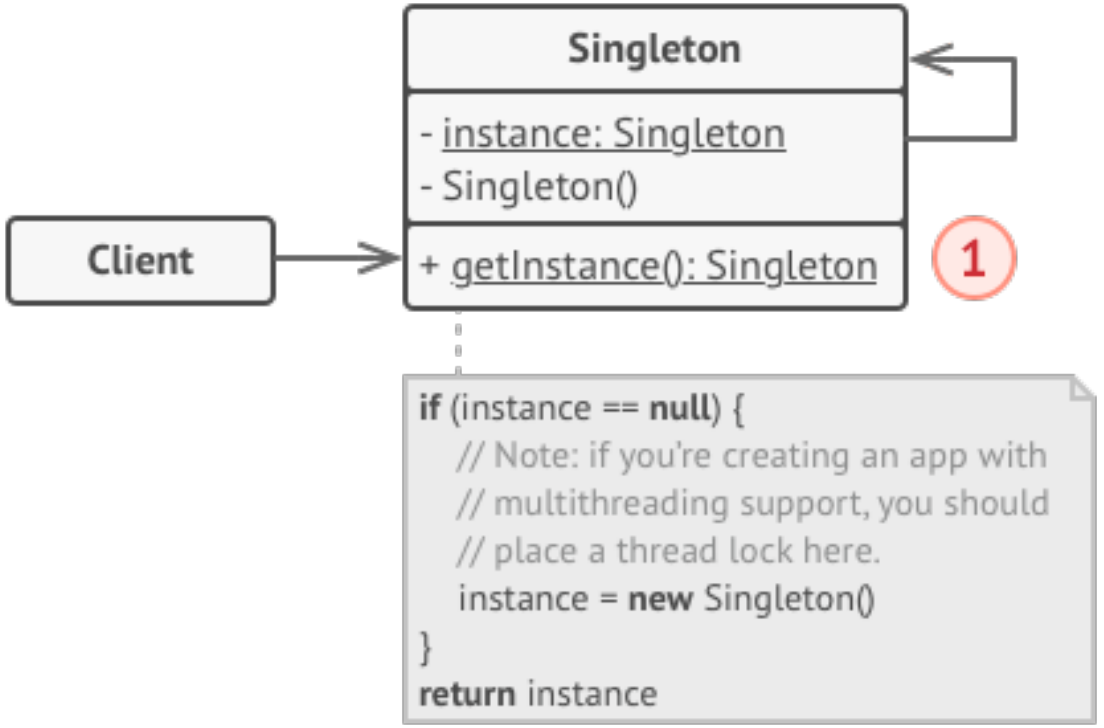
Solution:

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. Under the hood, 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.
- Whenever Singleton's static method is called, the same object is always returned.

Singleton: Structure

- The Singleton class declares the static method *getInstance* (1) that returns the same instance of its own class.
- The Singleton's constructor should be hidden from the client code.
- Calling the *getInstance* (1) method should be the only way of getting the Singleton object.



Singleton: How to Implement

- Add a private static field to the class for storing the singleton instance.
- Declare a public static creation method for getting the singleton instance.
- 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.
- 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.
- In a client, call singleton's static creation method to access the object.

Example in **Java (MUST read)**:

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

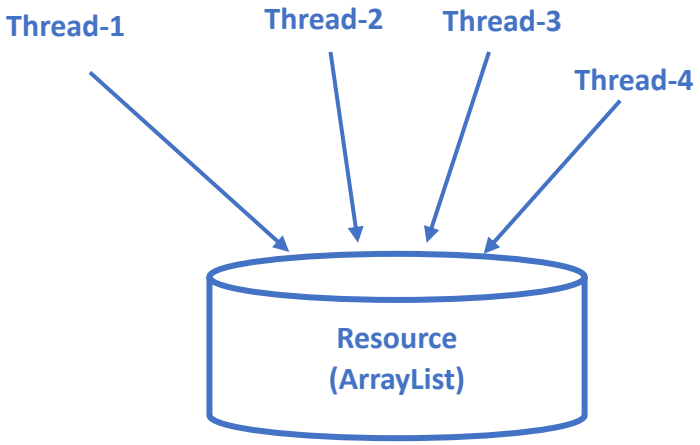
Singleton Pattern

For more information, read:

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

Introduction to Concurrency

- Several modern languages, including Java, allow for concurrent execution of multiple threads.
- To make the most of today's multi-core hardware, we must create applications that employ multiple threads.
- Therefore, having a fundamental understanding of concurrency is essential!
- *Thread safe*: **many threads** can access the **same resources** without exposing incorrect behaviour or causing unpredictable outcomes.
- Unfortunately, many Java libraries lack thread safety. Examples are ArrayList, StringBuilde, etc.



Thread Safety

- **Time slicing** in Java refers to the process of allocating time to threads.
- The order in which the threads run is **uncertain**. It is also unpredictable how many statements of one thread run before some of the other thread's statements run.
- Two threads modifying the same object (data) may operate in parallel.

```java
public class Account {
        private balance int = 500;

        ...
        ...

        public void withdraw(int amt){
                int old_balance = balance;
                ...
                // checking ... takes 500 mil secs
                ...
                balance = old_balance - amt;
                ...
        }

        ...
        ...

}
```

Let's create an object for the account number 1234,

```java
Account a1 = new Account(1234);
```

If **two** threads **thread-1** and **thread-2** call the following method 100 mil secs apart (for example from two browser windows), both may be successful, even if the balance is only $500!

```java
a1.withdraw(400);
```

A possible solution using synchronized

- A **synchronized** method acquires the lock of the object or class at the start, executes the method, and then releases the lock at the end.
- The use of ***synchronized*** key word allows **only one thread** to execute the method, avoiding concurrency issues.
- To make the most efficient use of the several CPUs available, a portion of the code (under synchronized) that accesses a shared resource must be **kept to a minimum**.
- We can also *synchronized* a set of statements, however it is a good practice to synchronized a method.
- Java offers **thread-safe collection wrappers**, using static methods. For example, `Collections.synchronizedList(list)`
- `Java.util.concurrent` package contains collections that are suitable and optimized for multiple threads.

```java
public class Account {
        private balance int = 500;

        ...
        ...

        public synchronized void withdraw(int amt){
                int old_balance = balance;
                ...
                // checking ... takes 500 mil secs
                ...
                balance = old_balance - amt;
                ...
        }

        ...
        ...
}
```

Need to avoid ….

- When two or more threads are stuck waiting for each other indefinitely, the condition is referred to as a **deadlock**.

- When two or more threads are caught in an endless cycle of reacting to one another, this is known as a **livelock**.

- **Starvation** occurs when one or more threads are unable to progress due to another "greedy" thread.

A lot more to concurrency ..

- There is much more to concurrency than what we have briefly addressed here; however, it is beyond the scope of this course.

An example of Singleton pattern using synchronized

Demo: Let's see what happens when we use synchronized and when we don't.

End