Singleton Pattern

This lesson discusses how the Singleton pattern enforces only a single instance of a class to ever get produced and exist throughout an application's lifetime.

We'll cover the following

What is it ?

Singleton pattern as the name suggests is used to create one and only instance of a class. There are several examples where only a single instance of a class should exist and the constraint be enforced. Caches, thread pools, registries are examples of objects that should only have a single instance.

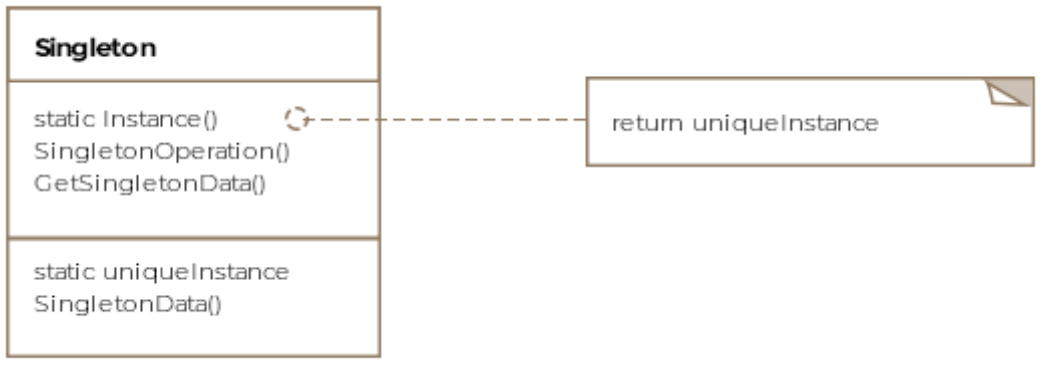
Its trivial to new-up an object of a class but how do we ensure that only one object ever gets created? The answer is to make the constructor private of the class we intend to define as singleton. That way, only the members of the class can access the private constructor and no one else.

Formally the Singleton pattern is defined as ***ensuring that only a single instance of a class exists and a global point of access to it exists***.

Class Diagram

The class diagram consists of only a single entity

- **Singleton**



Class Diagram

Example

As an example, let's say we want to model the American President's official aircraft called "Airforce One" in our software. There can only be one instance of Airforce One and a singleton class is the best suited representation.

Below is the code for our singleton class

```
public class AirforceOne {

    // The sole instance of the class
    private static AirforceOne onlyInstance;
```

```
    // Make the constructor private so its only accessible to
    // members of the class.
    private AirforceOne() {
    }

    public void fly() {
        System.out.println("Airforce one is flying...");
    }

    // Create a static method for object creation
    public static AirforceOne getInstance() {

        // Only instantiate the object when needed.
        if (onlyInstance == null) {
            onlyInstance = new AirforceOne();
        }

        return onlyInstance;
    }
}

public class Client {

    public void main() {
        AirforceOne airforceOne = AirforceOne.getInstance();
        airforceOne.fly();
    }
}
```

Multithreading and Singleton

The above code will work hunky dory as long as the application is single threaded. As soon as multiple threads start using the class, there's a potential that multiple objects get created. Here's one example scenario:

- Thread **A** calls the method `getInstance` and finds the `onlyInstance` to be null but before it can actually new-up the instance it gets context switched out.
- Now thread **B** comes along and calls the `getInstance` method and goes on to new-up the instance and returns the `AirforceOne` object.
- When thread **A** is scheduled again, is when the mischief begins. The thread was already past the if null condition check and will proceed to new-up another object of `AirforceOne` and assign it to `onlyInstance`. Now there are two different `AirforceOne` objects out in the wild, one with thread A and one with thread B.

There are two trivial ways to fix this race condition.

- One is to add `synchronized` to the `getInstance()` method.

```
synchronized public static AirforceOne getInstance()
```

- The other is to undertake static initialization of the instance, which is guaranteed to be thread-safe.

```
    // The sole instance of the class
    private static AirforceOne onlyInstance = new AirforceOne();
```

The problem with the above approaches is that synchronization is expensive and static initialization creates the object even if it's not used in a particular run of the application. If the object creation is expensive then static intialization can cost us performance.

Double-Checked Locking

The next evolution of our singleton pattern would be to synchronize only when the object is created for the first time and if its already created, then we don't attempt to synchronize the accessing threads. This pattern has a name called **"double-checked locking"**.

```
public class AirforceOneWithDoubleCheckedLocking {

    // The sole instance of the class. Note its marked volatile
    private volatile static AirforceOneWithDoubleCheckedLocking onlyInstance;
```

```
// Make the constructor private so its only accessible to
// members of the class.
private AirforceOneWithDoubleCheckedLocking() {
}

public void fly() {
    System.out.println("Airforce one is flying...");
}

// Create a static method for object creation
synchronized public static AirforceOneWithDoubleCheckedLocking getInstance() {

    // Only instantiate the object when needed.
    if (onlyInstance == null) {
        // Note how we are synchronizing on the class object
        synchronized (AirforceOneWithDoubleCheckedLocking.class) {
            if (onlyInstance == null) {
                onlyInstance = new AirforceOneWithDoubleCheckedLocking();
            }
        }
    }

    return onlyInstance;
    }
}
```

The above solution marks the singleton instance volatile however the JVM `volatile` implementation for Java versions 1.4 will not work correctly for double checked locking and you'll need to use another way to create your singletons.

The *double checked locking* is now considered an antipattern and its utility has largely passed away as JVM startup times have sped up over the years.

Other Examples

In the Java API we have the following singletons:

- java.lang.Runtime
- java.awt.Desktop

Caveats

- Its possible to subclass a singleton class by making the constructor protected instead of private. It might be suitable under some circumstances. An approach taken in these scenarios is to create a **register of singletons** of the subclasses and the `getInstance` method can take in a parameter or use an environment variable to return the desired singleton. The registry maintains a mapping of string names to singleton objects.

← Back
Builder Pattern

Next →
Prototype Pattern

Completed