Singleton (单例模式, Creational Pattern)

Kai SHI

Why Do We Need "One of A Kind (独一无二的)" Object?

- There are many objects we only need one of:
  - thread pools,
  - caches,
  - dialog boxes,
  - objects that handle preferences and registry (注册表) settings,
  - objects used for logging,
  - objects that act as device drivers to devices like printers and graphics cards.

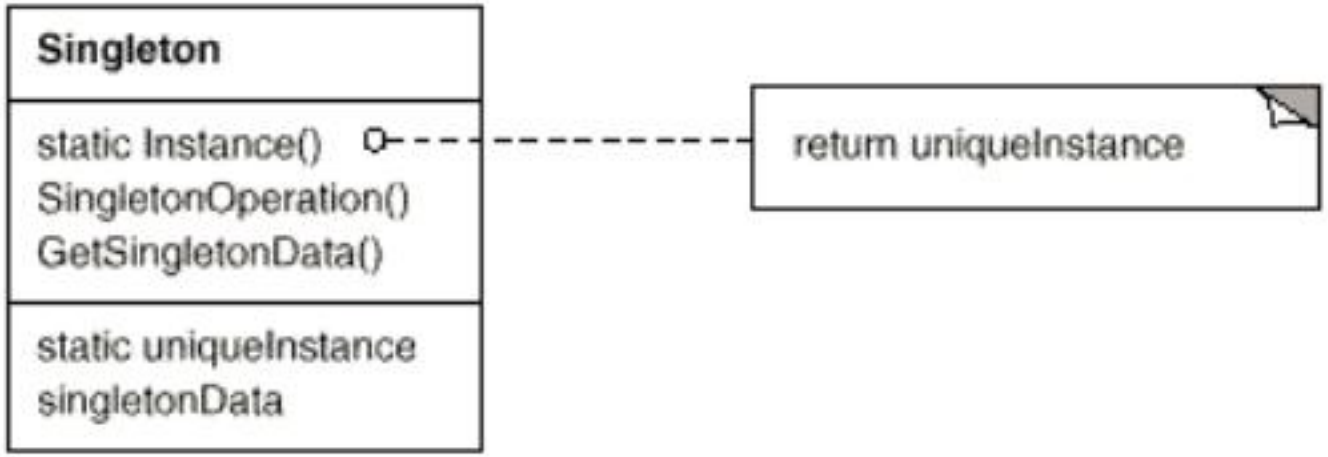
The Singleton Pattern

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it
- Motivation
  - Sometimes we want just a single instance of a class to exist in the system. For example, we want just one window manager. Or just one factory for a family of products.
  - We need to have that one instance easily accessible.
  - And we want to ensure that additional instances of the class can not be created.

Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure



Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance.
  - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences: Benefits

- Controlled access to sole instance.
- Permits a variable number of instances. The pattern makes it easy to allow more than one instance of the Singleton class. You can use the same approach to control the number of instances that the application uses.
- More flexible than static class (class with all static properties and methods).
  - Static class must be stateless; Singleton could be stateful

Singleton Implementation

- Use a static method to allow clients to get a reference to the single instance.
- Use a private constructor.

Implementation: Eager Singleton

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

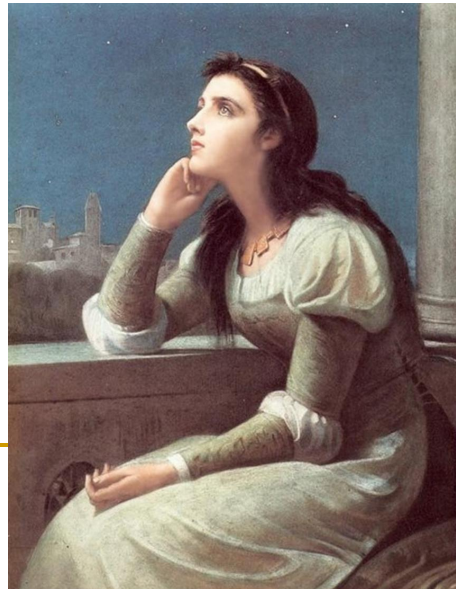
We've already got an instance, so just return it.

Implementation: Lazy Singleton: What's the problem with this code?

```
public final class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```



Implementation: Lazy Singleton: Not Thread-Safe

```
public final class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

Multiple Singleton objects may be obtained during multithreaded access (that is, each thread has its own singleton object)

Implementation: Lazy Singleton: Thread safety: synchronized getInstance()

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

But after the singleton is created, it still needs to be synchronized every time the singleton is obtained! low efficiency.

Implementation: Lazy Singleton: Thread safety: "double-checked locking"

volatile reminds the compiler that the variables may change at any time, so every time the program stores or reads this variable, it reads data directly from the variable address. If there is no volatile keyword, the compiler may optimize reading and storage, and may temporarily use the value in the register. If this variable is updated by another program, there will be inconsistencies.

```
public class Singleton {
    private volatile* static Singleton uniqueInstance;

    private Singleton() {}

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

Each class has a unique corresponding Class object. ".class" returns the runtime class of an object. That Class object is the object that is **locked** by **static synchronized methods** of the represented class.

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.