

Singleton Pattern

One-of-a-Kind Objects

Singleton

- For objects we need only one of: thread pools, caches, dialog boxes, objects for preferences or configuration objects, ...
- Instantiating more than one of these (for some) could lead to incorrect program behavior, overuse of resources, or inconsistent results.
- Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- Singleton pattern is a time-tested convention for ensuring only one object is instantiated.

Questions

- How to create an instance of MyObject?
- Can another object create a MyObject?
- What if we don't have a public class?
- What about a public class with a private constructor?

Private Constructor

```
public class MyClass {  
    private MyClass() {  
    }  
}
```

- In this case MyClass cannot be instantiated outside of MyClass because it has a private constructor.
- So, how can we create an instance of MyClass?
- From where can the MyClass constructor be called?

Consider the general construct...

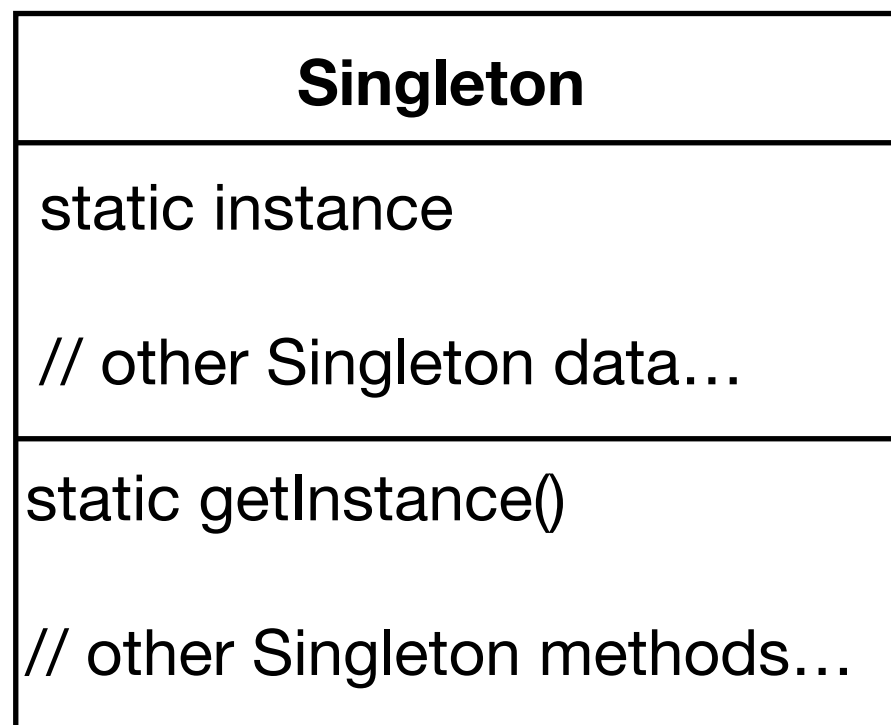
```
public class MyClass {  
    private MyClass() {  
    }  
  
    public static MyClass getInstance() {  
    }  
}
```

- `getInstance()` is static. Do you need an instance of `MyClass`?

Mostly complete

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

UML Class Diagram



Name of class

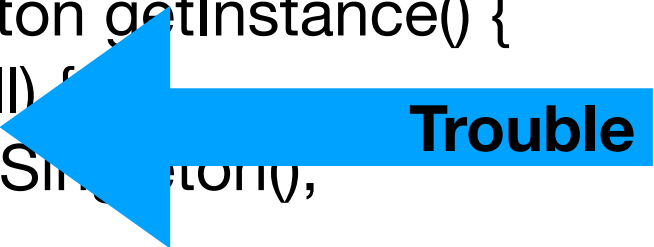
Class attributes

Class methods

Is it thread-safe?

- Consider two threads executing method *getInstance()* and the value of variable *instance*.

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



Solution?

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

Every thread has to wait its turn before it can enter the method.

Expensive!

Synchronization is needed only the first time through getInstance()

Can we improve multithreading?

- Options
 - If getInstance() performance is not critical, do nothing.
 - Eager instantiation
 - Initialization-on-demand holder
 - Double-check locking
 - Reduce use of synchronization in getInstance()
 - Improves performance

Do nothing

Eager instantiation

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
  
    private Singleton() {...}  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
    ...  
}
```

Initialization-on-demand Holder

- A lazy-loaded singleton
- Only used when constructor cannot fail

```
public class Singleton {  
  
    private Singleton() {...}  
  
    private static class LazyHolder {  
        static final Singleton INSTANCE = new Singleton();  
    }  
  
    public static Singleton getInstance() {  
        return LazyHolder.INSTANCE;  
    }  
}
```

Double-check Locking

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