Singleton Design Pattern

Dr. Abbas Rasoolzadegan

Singleton Pattern

- **Intent**:

Ensure a class only has one instance, and provide a global point of access to it.

- **Classification**: Object Creational.

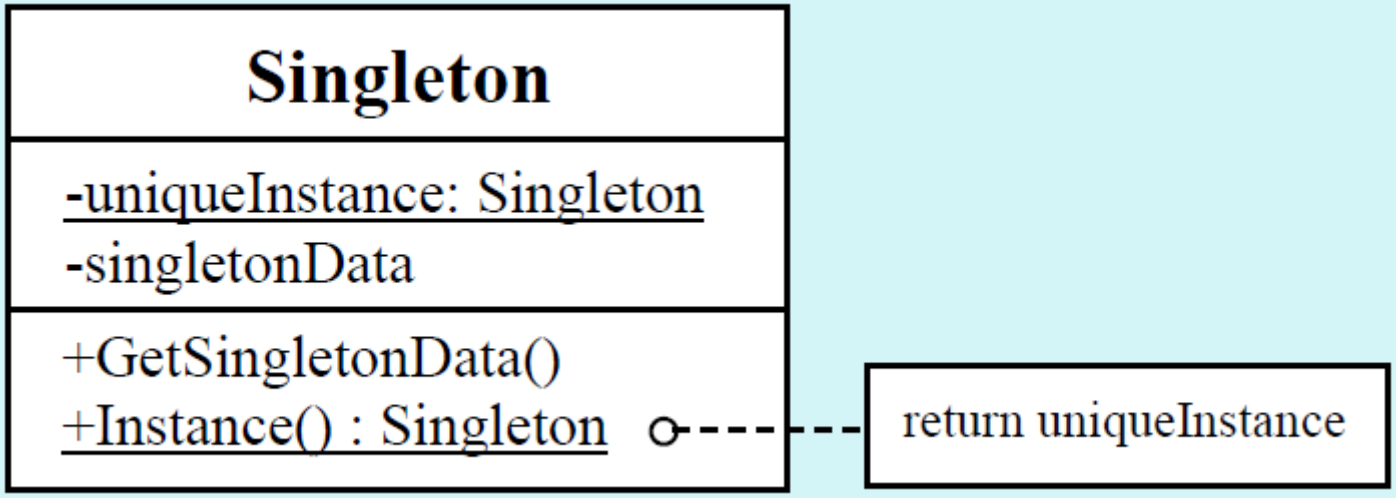
Motivation

- 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.
- We want to ensure that additional instances of the class can not be created.
- The solution is to make the class itself responsible for keeping track of its sole instance and provide a way to access the instance.

Applicability

Use the Singleton pattern when:

- There must be exactly one instance of a class, and it must be accessible to clients from a wellknown 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.


Singleton
-uniqueInstance: Singleton
-singletonData
+GetSingletonData()
+Instance() : Singleton
return uniqueInstance

Structure

Consequences

The Singleton pattern has several benefits:

- Controlled access to sole instance.
- Permits a variable number of instances.

Implementation

- OK, so how do we implement the Singleton pattern?
- We'll use a static method to allow clients to get a reference to the single instance and we'll use a private constructor!

```
/**Class Singleton is an implementation of a class that only
allows one instantiation.*/
public class Singleton {
      // The private reference to the one and only instance.
      private static Singleton uniqueInstance = null;
      // An instance attribute.
      private int data = 0;
```

Implementation (cont.)

```
    /**Returns a reference to the single instance.
      * Creates the instance if it does not yet exist.
      * (This is called lazy instantiation.)*/
    public static Singleton instance() {
        if (uniqueInstance == null) uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    /**The Singleton Constructor.
      * Note that it is private!
      * No client can instantiate a Singleton object!*/
    private Singleton() {}
    // Accessors and mutators here!
}
```

Implementation (cont.)

- **Here's a test program:**

```
public class TestSingleton {
    public static void main(String args[]) {
        // Get a reference to the single instance of Singleton.
        Singleton s = Singleton.instance();
        // Set the data value.
        s.setData(34);
        System.out.println("First reference: " + s);
        System.out.println("Singleton data value is: " + s.getData());
        // Get another reference to the Singleton. Is it the same object?
        s = null;
        s = Singleton.instance();
        system.out.println("\nSecond reference: " + s);
        system.out.println("Singleton data value is: " + s.getData());}
}
```

Implementation (cont.)

- **And the test program output:**

```
First reference: Singleton@1cc810
Singleton data value is: 34

Second reference: Singleton@1cc810
Singleton data value is: 34
```

Implementation (cont.)

- Note that the singleton instance is only created when needed. This is called *lazy instantiation*.
- Thought experiment: What if two threads concurrently invoke the instance() method? Any problems?
- Yup! Two instances of the singleton class could be created!
- How could we prevent this? Several methods:
  - Method 1: Make the instance() synchronized. Synchronization is expensive, however, and is really only needed the first time the unique instance is created.
  - Method 2: Do an eager instantiation of the instance rather than a lazy instantiation.

Implementation (cont.)

Method 1: Synchronization (Thread-safe) - Java Sample Code:

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

Implementation (cont.)

```
class Singleton {
    private static Singleton instance;
    private Singleton() {
        System.out.println("Singleton(): Initializing Instance");}
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    System.out.println("getInstance():First time
                    getInstance was invoked!");
                    instance = new Singleton();}}}
        return instance;
    }
    public void doSomething() {
        System.out.println("doSomething(): Singleton does something!");
    }
}
```

Implementation (cont.)

.net Sample Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace DesignPattern_SingleTon {
    public sealed class SingleTon {
        private static volatile SingleTon Instance;
        private static object _lock = new object();
        private SingleTon() { }
        public static SingleTon Getinstance() {
            if (Instance == null) {
                lock (_lock) {
                    if (Instance == null)
                        Instance = new SingleTon();} }
            return Instance; }
        public String Name; }
}
```

Implementation (cont.)

- Method 2: Here is Singleton with eager instantiation.
- We'll create the singleton instance in a static initializer. This is guaranteed to be thread safe.

```
/**Class Singleton is an implementation of a class that
   * only allows one instantiation.*/
public class Singleton {
    // The private reference to the one and only instance.
    // Let's eagerly instantiate it here.
    private static Singleton uniqueInstance = new Singleton();
    // An instance attribute.
    private int data = 0;
```

Implementation (cont.)

```
    /** Returns a reference to the single instance.*/
    public static Singleton instance() {
        return uniqueInstance;
    }
    /**The Singleton Constructor.
     * Note that it is private!
     * No client can instantiate a Singleton object!*/
     private Singleton() {}
     // Accessors and mutators here!
}
```

Singleton With Subclassing

- What if we want to be able to subclass Singleton and have the single instance be a subclass instance?
- For example, suppose MazeFactory had subclasses EnchantedMazeFactory and AgentMazeFactory. We want to instantiate just one factory, either an EnchantedMazeFactor or an AgentMazeFactory.
- How could we do this? Several methods:
  - Method 1: Have the static instance() method of MazeFactory determine the particular subclass instance to instantiate. This could be done via an argument or environment variable. The constructors of the subclasses can not be private in this case, and thus clients *could* instantiate other instances of the subclasses.
  - Method 2: Have each subclass provide a static instance() method. Now the subclass onstructors can be private.

Singleton With Subclassing: Method 1

- Method 1: Have the MazeFactory instance() method determine the subclass to instantiate.

```
/** Class MazeFactory is an implementation of a class that
  * only allows one instantiation of a subclass.*/
public abstract class MazeFactory {
    // The private reference to the one and only instance.
    private static MazeFactory uniqueInstance = null;
    // The MazeFactory constructor.
    // If you have a default constructor, it can not be private
    // here!
    protected MazeFactory() {}
```

Singleton With Subclassing: Method 1 (cont.)

```
    // Return a reference to the single instance.
    // If instance not yet created, create "enchanted" as default.
    public static MazeFactory instance() {
        if (uniqueInstance == null) return instance("enchanted");
        else return uniqueInstance;}
    // Create the instance using the specified String name.
    public static MazeFactory instance(String name) {
        if (uniqueInstance == null)
            if (name.equals("enchanted"))
                uniqueInstance = new EnchantedMazeFactory();
            else if (name.equals("agent"))
                uniqueInstance = new AgentMazeFactory();
        return uniqueInstance;}
}
```

Singleton With Subclassing: Method 1 (cont.)

- Client code to create factory the first time:

```
MazeFactory factory = MazeFactory.instance("enchanted");
```

- Client code to access the factory:

```
MazeFactory factory = MazeFactory.instance();
```

- Note that the constructors of EnchantedMazeFactory and AgentMazeFactory can not be private, since MazeFactory must be able to instantiate them. Thus, clients could potentially instantiate other instances of these subclasses.

Singleton With Subclassing: Method 1 (cont.)

- The instance(String) methods violates the Open-Closed Principle, since it must be modified for each new MazeFactory subclass.
- We could use Java class names as the argument to the instance(String) method, yielding simpler code:

```
public static MazeFactory instance(String name) {
    if (uniqueInstance == null)
        uniqueInstance = Class.forName(name).newInstance();
    return uniqueInstance;
}
```

Singleton With Subclassing: Method 2

- Method 2: Have each subclass provide a static instance method().

```
/**Class MazeFactory is an implementation of a class that only
allows one instantiation of a subclass. This version requires its
subclasses to provide an implementation of a static instance()
method.*/
public abstract class MazeFactory {
    // The protected reference to the one and only instance.
    protected static MazeFactory uniqueInstance = null;
    // The MazeFactory constructor.
    // If you have a default constructor, it can not be private
    // here!
    protected MazeFactory() {}
    // Return a reference to the single instance.
    public static MazeFactory instance() {
        return uniqueInstance;}
}
```

Singleton With Subclassing: Method 2 (cont.)

```
/**Class EnchantedMazeFactory is an implementation of a class
that only allows one instantiation.*/

public class EnchantedMazeFactory extends MazeFactory {

    // Return a reference to the single instance.

    public static MazeFactory instance() {

        if (uniqueInstance == null)

            uniqueInstance = new EnchantedMazeFactory();

        return uniqueInstance;

    }

    // Private subclass constructor!!

    private EnchantedMazeFactory() {}

}
```

Singleton With Subclassing: Method 2 (cont.)

- Client code to create factory the first time:

```
MazeFactory factory = EnchantedMazeFactory.instance();
```

- Client code to access the factory:

```
MazeFactory factory = MazeFactory.instance();
```

- Note that now the constructors of the subclasses are private. Only one subclass instance can be created!
- Also note that the client can get a null reference if it invokes `MazeFactory.instance()` before the unique subclass instance is first created.
- Finally, note that uniqueInstance is now protected!

Singleton With Subclassing: Method 3

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

Singleton With Subclassing: Method 3 (cont.)

```
Singleton* Singleton::Instance() {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup
        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
static MySingleton theSingleton;
```

Multiton Design Pattern

Java Sample Code

```
public class FooMultiton {
      private static final Map<Object, FooMultiton> instances=new
      HashMap<Object,FooMultiton>();
      private FooMultiton() {// no explicit implementation}
      public static synchronized FooMultiton getInstance(Object key) {
          // Our "per key" singleton
          FooMultiton instance = instances.get(key);
          if (instance == null) {
              // Lazily create instance
              instance = new FooMultiton();  // Add it to map
              instances.put(key, instance);}
          return instance;
      }
     // other fields and methods ...
}
```

.net Sample Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections.Concurrent;
namespace DesignPatternMultiton {
    public sealed class Camera {
        private static volatile ConcurrentDictionary<int, Camera> D_instance = new
        ConcurrentDictionary<int, Camera>();
        private static object _SyncLock= new object();
        private Camera() { }
        public static Camera GetInstance(int in_key) {
            if (!D_instance.ContainsKey(in_key)) {
                lock (_SyncLock) {
                    if (!D_instance.ContainsKey(in_key))
                        D_instance.GetOrAdd(in_key, new Camera());}}
            return D_instance[in_key];}}
}
```