Mẫu thiết kế Singleton

Nguyễn Khắc Huy

Thành phần tĩnh

□ Member variables

```
class PhanSo {
        int tuSo;
        int mauSo;
public:
        PhanSo(int tu, int mau)
}
PhanSo::PhanSo(int tu, int mau) {
    this->tuSo = tu;
    this->mauSo = mau;
}
void main() {
        PhanSo a(1,2); // what do have have?
        PhanSo b(3,5); // what do have have?
}
```

Thành phần tĩnh

- Static variable

```
class PhanSo {
        int tuSo;
        int mauSo;
        int count = 0;
public:
        PhanSo(int tu, int mau);
}
PhanSo::PhanSo(int tu, int mau) {
    this->tuSo = tu;
    this->mauSo = mau;
    count++;
}
```

```
void main() {
    PhanSo a(1, 2); // ??
    PhanSo b(3, 5); // ??
}
```

Thành phần tĩnh

- Static variable

```
class PhanSo {
        int tuSo;
        int mauSo;
        static int count;
public:
        PhanSo(int tu, int mau, int count);
}
PhanSo::PhanSo(int tu, int mau) {
    this->tuSo = tu;
    this->mauSo = mau;
    count++;
}
```

```
PhanSo::count = 0;
void main() {
    PhanSo a(1, 2); // ??
    PhanSo b(3, 5); // ??
}
```

Mẫu thiết kế là gì?

- Design pattern

Sometimes it is important to have only one instance of a class

- One printer spooler (though many printers)
- One file system
- One window manager

The challenge:

- Ensure that a certain class has only **one** instance
- Provide global access to it

Question:

How can you prevent *any* instances of a class from being created?

Question:

How can you prevent *any* instances of a class from being created?

1. **Don't write it…**
2. **?**

Moving towards a solution…

1. Restrict the ability to construct more than one instance of a Singleton class

2. Make the class responsible for keeping track of its one and onl instance

3. Provide a global (static) access method

Solution 1: *Eager Instantiation* using static initialization

```
public class Singleton {

  private static Singleton uniqueInstance = new Singleton();


  // private constructor cannot be called;
  // no instances can be created.
  private Singleton() {
  }

  // return the same instance to all callers of this method
  public static Singleton getInstance() {
        return uniqueInstance;
  }
}
```

Important Concepts

1. Why is the constructor private?

   Any client that tries to instantiate the **`Singleton`** class directly will get an error at compile time.

   - ✓ This ensures that nobody other than the **`Singleton`** class and instantiate the object of this class.

2. How can you access the single instance of the **`Singleton`** class?

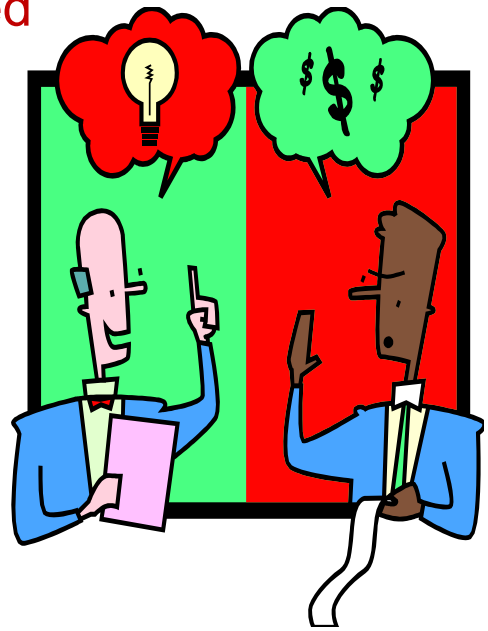
   `getInstance``()` is a static method. We can use the class name to reference the method.

   - ✓ **`Singleton.getInstance()`**

   - ✓ This is how we can provide global access to the single instance of the **`Singleton`** class.

Eager instantiation has pros and cons

- Static initialization is guaranteed to be thread-safe
- Global/static objects are created whether they are used or not, as soon as the application is loaded.
- Creating a Singleton object can be resource intensive and you may not want to create it until you need it.
- We might not have all the information to create an object at static initialization time.
  - Example: the kind of file/database/resource that you are connecting to is usually not known at startup



Solution 2: A Lazy Singleton

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

static variable to hold the single instance of the class.

```
      private Singleton() {
            // construction code goes here.
      }
```

Only this class can instantiate itself.

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

Provides a way to instantiate the class and also provides global access to it.

Lazy initialization

The `Singleton` class on the previous slide uses *lazy initialization*

- The `uniqueInstance` is not created and stored in the `getInstance()` method *until it is first accessed.*

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

Lazy Singletons can have issues with Multithreading

Thread1	Thread2	Value of uniqueInstance
Singleton.getInstance()		null
	Singleton.getInstance()	null
if (uniqueInstance == null){		null
	if (uniqueInstance == null) {	null
uniqueInstance = new Singleton()		Object 1
return uniqueInstance;		Object 1
	uniqueInstance = new Singleton()	Object 2
	return uniqueInstance;	Object 2

Another alternative for lazy initialization

```
public class Singleton {
        private static volatile Singleton uniqueInstance =
null;


        private Singleton() {
                // construction code goes here.
        }

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

Using the Java keyword *synchronized* forces every thread to wait its turn before it can enter the method. This prevents two threads from entering the method at the same time.

The Cost of Synchronization

- *Synchronization is expensive.* Synchronized methods can take 100 times longer to execute than unsynchronized methods
  - This can affect performance of your program if `getInstance()` is called frequently

- Synchronization is relevant only the first time though the method, so why force repeated usage?
  - Once we have set the `uniqueInstance` to an appropriate instance, then there is no need to further synchronize the method.

Lazy Initialization Solution 2: *Double Checked Locking*

```
public class Singleton {
   private volatile static Singleton uniqueInstance = null;
      private Singleton() {
            // construction code goes here.
      }

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

volatile ensures that the JVM always has an up-to-date value of uniqueInstance

Enter a *synchronized block* only when there is no instance.

We need to synchronize *only the first time through.*

Double-checked locking solves multithreading problems

Thread1	Thread2	Value of uniqueInstance
Singleton.getInstance()		null
	Singleton.getInstance()	null
if (uniqueInstance == null)		null
	if (uniqueInstance == null)	null

synchronized (Singleton.class) {
if (uniqueInstance == null)
uniqueInstance = new Singleton()
}
		Object 1

return uniqueInstance;		Object 1

	synchronized (Singleton.class) {
	If (uniqueInstance == null)
	// will evaluate to false!!!
		Object 1

	return uniqueInstance;	Object 1

The Cost of Double-checked locking

Code is more complex

- Additional complexity = decreased maintainability
- Is it worth it?

One final possible approach

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

      private Singleton() {
            // do nothing here.
      }
}
```

Public static variable holds the single instance of the class.

What's wrong with this approach using an eager "global" variable?

Summary

Lazy versus Eager instantiation

- With a Lazy Singleton, you do not have to create an instance until you need it.
- With Eager initialization, the object is created at load time, before it is actually needed
  - ✓And whether or not it is actually ever accessed

A Singleton class encapsulates its sole instance.

- Singleton provides only one instance of a class and global access.
- Public global variables provide only global access but cannot ensure one and only one unique instance.

Bài tập

- Bài tập 9.1:

Trong trò chơi tung xúc xắc, người chơi khi bắt đầu trò chơi sẽ nhận 5000$. Trước khi bắt đầu tung xúc xắc, người chơi sẽ đặt cược một số tiền nhất định nhỏ hơn số dư mình có, đồng thời đưa ra lời dự đoán về kết quả chẵn hoặc lẻ. Dự đoán xong, người chơi bắt đầu tung xúc xắc. Kết quả xúc xắc đúng với dự đoán người chơi sẽ được x2 số tiền đã cược. Ngược lại người chơi mất số tiền đã cược.

Yêu cầu:

Xây dựng lớp đối tượng Random áp dụng mẫu Singleton cho phép phát sinh ra số nguyên ngẫu nhiên trong khoảng giới hạn cho trước.

Viết chương trình tung xúc xắc 1-6 mặt cho một người chơi.

Reference

- Course Advanced Programming Topics – Matthew Hertz, Canisius College.
- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.