Design Patterns

Week 1 : Introduction and Singleton

Adapted from Jon Simon's (jonathan_simon@yahoo.com)

What is a Design Pattern?

- A problem that someone has already solved.
- A model or design to use as a guide

- More formally: “A proven solution to a common problem in a specified context."

Real World Examples

- Blueprint for a house
- Manufacturing

Why Study Design Patterns?

- Provides software developers a toolkit for handling problems that have already been solved.
- Provides a vocabulary that can be used amongst software developers.
  - The Pattern Name itself helps establish a vocabulary
- Helps you *think* about how to solve a software problem.

The Gang of Four

- "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

- Defines a Catalog of different design patterns.

- Three different types
  - *Creational* – "creating objects in a manner suitable for the situation"
  - *Structural* – "ease the design by identifying a simple way to realize relationships between entities"
  - *Behavioral* – "common communication patterns between objects"

The Gang of Four: Pattern Catalog

Creational

- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Prototype**
- **Singleton**

Structural

- **Adapter**
- **Bridge**
- **Composite**
- **Decorator**
- **Façade**
- **Flyweight**
- **Proxy**

Behavioral

- **Chain of Responsibility**
- **Command**
- **Interpreter**
- **Iterator**
- **Mediator**
- **Memento**
- **Observer**
- **State**
- **Strategy**
- **Template Method**
- **Visitor**

Patterns in blue will be discussed in class.

The Book

- “Head First Design Patterns”
  - Eric Freeman & Elisabeth Freeman
  - Available for download from SlideShare Web site
- Book is based on the Gang of Four design patterns.
- Easier to read.
- Examples are fun, but not necessarily “real world”.

Example: Logger

What is wrong with this code?

```
public class Logger
    {
        public Logger() { }

        public void LogMessage() {
            //Open File "log.txt"
            //Write Message
            //Close File
        }
    }
```

Example: Logger (cont)

- Since there is an external Shared Resource (“log.txt”), we want to closely control how we communicate with it.
- We shouldn’t create an object of the Logger class every time we want to access this Shared Resource. Is there any reason for that?
- We need ONE.

Singleton

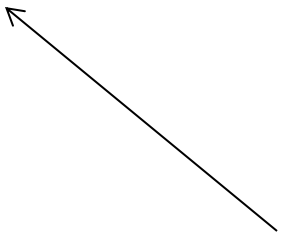
- GoF Definition: “The Singleton Pattern ensures a class has only *one instance*, and provides a global point of access to it.”

- Best Uses
  - Logging
  - Caches
  - Registry Settings
  - Access External Resources
    - Printer
    - Device Driver
    - Database

Logger – as a Singleton

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

     private static Logger uniqueInstance;

    public static Logger getInstance()
    {

      if (uniqueInstance == null)
             uniqueInstance = new Logger();

      return uniqueInstance;

    }
}
```

See pg 173 in book



Note the parameterless constructor

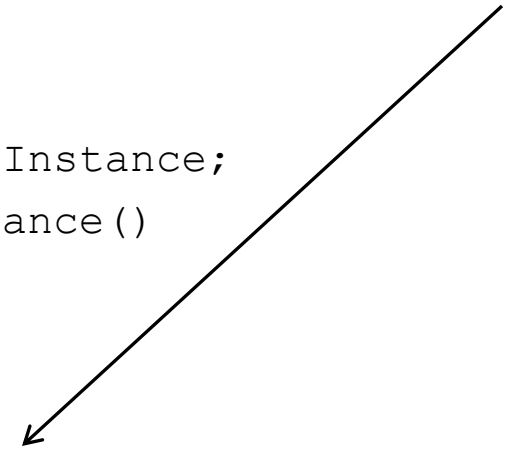
Lazy Instantiation

- Objects are only created, when it is needed
- Helps control that we've created the Singleton just once.
- If it is resource intensive to set up, we want to do it once.

Threading

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

   private static Singleton uniqueInstance;
   public static Singleton getInstance()
   {

     if (uniqueInstance == null)
            uniqueInstance = new Singleton();

         return uniqueInstance;

   }
}
```

What would happen if two different threads accessed this line at the same time?



Option #1: Simple Locking

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

   private static Singleton uniqueInstance;

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

         return uniqueInstance;
   }
}
```

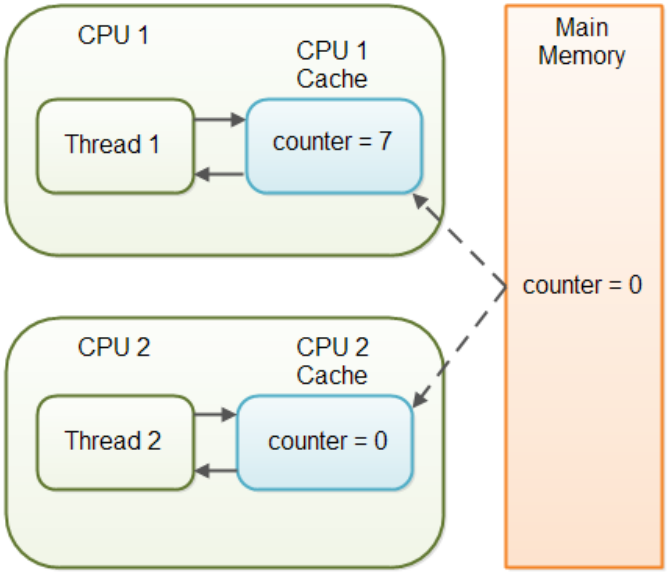
Option #2 – Double-Checked Locking

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

   private volatile static Singleton uniqueInstance;

   public static Singleton getInstance()
   {
        if (uniqueInstance == null) {           // single checked
            synchronized(Singleton.class) {
                if (uniqueInstance == null)     // double checked
                        uniqueInstance = new Singleton();
             }
         }

         return uniqueInstance;
   }
}
```

pg 182

volatile Variable

- Used to mark a Java variable as “being stored in main memory”
- Every read/write of a volatile variable is directly from/to main memory, not from/to the cache
- Guarantees visibility of changes to variables across threads



Option #3: “Eager” Initialization

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

   private static Singleton uniqueInstance = new Singleton()

   public static Singleton getInstance()
   {
         return uniqueInstance;
   }

}
```

pg 181

Runtime guarantees that this is thread-safe

1. Instance is created the first time any member of the class is referenced.
2. Good to use if the application always creates; and if little overhead to create.

SUMMARY

- ***Pattern Name*** – Singleton
- ***Problem*** – Ensures one instance of an object and global access to it.
- ***Solution***
  - Hide the constructor
  - Use static method to return one instance of the object
- ***Consequences***
  - Lazy Instantiation
  - Threading