Singleton Pattern

Chapter Content

InterBit
Training & Consulting Ltd.

- The Singleton Pattern – Overview
- Lazily-initialized Singleton
- Double-checked-Locking
- Static-Methods Singleton
- More Considerations
- Discussion
- Exercise

Overview

InterBit
Training & Consulting Ltd.

- Used when there should always be **at most one single instance of a class.**
- **Common code pattern:**
  - Embed a single static instance inside the class.
  - Make sure the constructor cannot be called more then once (e.g., by making it private).
- Variations of the singleton pattern:

Lazily-initialized Singleton

InterBit Training & Consulting Ltd.

```
// A singleton (program should use a single printSpooler):
class PrintSpooler {
      // single instance:
      private static PrintSpooler instance;

      // Private Constructor:
      private PrintSpooler() {...}


      // getInstance method:
      public static synchronized PrintSpooler getInstance(){
            if (instance==null) {
                instance=new PrintSpooler();
             }
            return instance;
      }
}
```

Usage:

```
PrintSpooler  ps= PrintSpooler.getInstace();
ps.printDocument(...);
```

Double-checked-Locking

InterBit Training & Consulting Ltd.

- Sadly, when using lazy initialization, we pay the **very expensive** price of synchronization.
- DCL was a **failed** attempt to reduce synchronization.

```
// UNSAFE !  May return incomplete instance, since optimizer may choose to
// first assign address to instance, and only then invoke constructor body
// See: http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-double_p.html

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

Double-checked-Locking - Safe

InterBit Training & Consulting Ltd.

- Correct implementation in **any** Java version:

```
/**
 * The inner class is referenced no earlier (and therefore loaded
 * no earlier by the class loader) than the moment that getInstance()
 * is called. Thus, this   solution is thread-safe without requiring
 * special language constructs (i.e. volatile or synchronized).
 */
public class Singleton {
  // Private constructor prevents instantiation from other classes
  private Singleton() {}
  /**
   * SingletonHolder is loaded on the first execution of Singleton.getInstance()
   * or the first access to SingletonHolder.INSTANCE, not before.
   */
  private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

Double-checked-Locking JDK 1.5

InterBit Training & Consulting Ltd.

- Correct implementation only with Java 1.5 and later:

```
/**
 * JDK5 and later extends the semantics for volatile so that the system will
 * not allow a write of a volatile to be reordered with respect to any
 * previous read or write, and a read of a volatile cannot be reordered
 * with respect to any following read or write. The Double-Checked Locking
 * idiom can be made to work by declaring the helper field to be volatile.
 *
 * This does not work under JDK4 and earlier.
 */
 Public class Foo {
     // this is our singleton
     private volatile Helper helper = null;

     public Helper getHelper() {
         if (helper == null) {
             synchronized(this) {
                 if (helper == null)
                    helper = new Helper();
             }
         }
         return helper;
     }
 }
```

Static-Methods Singleton

```
final class Math {
  private Math() {} // never called

  public static double sin(double x) { ...}
  public static double cos(double x) { ...}

}
```

- Constructor is private, to disallow instantiation.
- Class is final, so as to disallow inheritance.
- **java.lang.Math** follows this design pattern (a.k.a Utility Pattern).

More Considerations

- **N-ton:** you may decide on any fixed number of instances (not necessarily 1).
- **Locating:** how will other classes gain access to the singleton instance ?
  - Call *YourSingletonClass.getInstance().*
  - Receive it as a parameter (see next slide).
  - Program may have some centralized registry of singleton instances.

More Considerations

- **In large complex programs** it may not be simple to discover where a Singleton was instantiated.
- **One Solution:** create such singletons at the beginning of the program and pass them as arguments to the major classes that might need to use them:

```
...
pr1 = iSpooler.Instance();
Customers cust = new Customers(pr1);
...
```

More Considerations

- **Another Solution:** Create a Registry - When a Singleton instantiates, it notes that in the Registry. Anyone, anywhere, can ask for an instance

- **But**: Type checking may be reduced. The table of singletons in the registry *probably* keeps all of the singletons as Objects.

- **Plus...** The registry itself may be a Singleton

Discussion

InterBit
Training & Consulting Ltd.

- **Singleton is the object-oriented replacement for a global variable ...**
- What are the advantages & disadvantages of using a singleton (especially with the getInstance() approach) ?

Think, for example: It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated...

SUMMARY OF CREATIONAL DP'S

Factory is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory.

Abstract Factory is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes.

Builder assembles a number of objects to make a new object, based on the data with which it is presented. (Sometimes, uses a Factory itself).

SUMMARY OF CREATIONAL DP'S

(...Continued)

Prototype copies or clones an existing class rather than creating a new instance when creating new instances is more expensive.

Singleton is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that