Singleton pattern

From Wikipedia, the free encyclopedia

In software engineering, the **singleton** design pattern is used to restrict instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist. It is also considered an anti-pattern since it is often used as a euphemism for global variable. Before designing a class as a singleton, it is wise to consider whether it would be enough to design a normal class and just use one object.

The singleton pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made either private or protected. Note the distinction between a simple static instance of a class and a singleton. Although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation.

The classic solution to this problem is to use mutual exclusion on the class that indicates that the object is being instantiated.

Contents

Class diagram

Image:Singleton sangeet.png

Java example implementation

A **correct** threadsafe Java programming language lazy-loaded solution known as the "Initialization On Demand Holder" idiom suggested by Bill Pugh follows:

```
public class Singleton {
  // Private constructor suppresses generation of a (public) default constructor
  private Singleton() {}

  private static class SingletonHolder {
    private static Singleton instance = new Singleton();
  }

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

Another Java example implementation

An **incorrect** Java programming language lazy-loaded solution known as the "Initialization On Demand Holder" idiom suggested by Pankaj Jaiswal. Double-checked locking is not guaranteed to work [1]:

```
public class Singleton {
   private static Singleton test1;

  // Private constructor suppresses generation of a (public) default constructor
  private Singleton() {
  }

  public static Singleton getInstance() {
    if(test1 == null) {
       // Not guaranteed to work
       synchronized (Singleton.class) {
          if(test1 == null) {
             test1 = new Singleton();
          }
       }
    }
    return test1;
  }
}
```

C++ example implementation

A possible C++ solution using Curiously Recurring Template Pattern (also known as Meyers singleton) where singleton is a static local object (note: this solution is not thread-safe and is designed to give an idea of how sin work rather than a solution usable in large-scale software project).

```
template<typename T> class Singleton
{

  public:
    static T& Instance()
    {
        static T theSingleInstance; //assumes T has a default constructor
        return theSingleInstance;
    }
};
```

```
class OnlyOne : public Singleton<OnlyOne>
{
    //..rest of interface defined here
};
```

Thread-safe C++ example implementation using POSIX threads

A common design pattern for thread safety with the singleton class is to use double-checked locking. However, due to the ability of optimising compilers (and CPUs!) to re-order instructions, and the absence of any consideration being given to multiple threads of execution in the language standard, double-checked locking is *intrinsically prone to failure in C++*. There is no model—other than runtime libraries (e.g. POSIX threads, designed to provide concurrency primitives)—that can provide the necessary execution order.[1] (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf#search=%22meyers%20double%20checked%20loc . Future C++ versions may include threads as a language standard[2] (http://www.ddj.com/dept/cpp/184401518) , but this is supposition at the time of writing.

By adding a mutex to the singleton class, a thread-safe implementation may be obtained.

Define two helper classes, `mutex` and `mutex_locker`:

```
class mutex
{
public:
   mutex()
      {
      pthread_mutex_init(&m,0);
      }

   void lock()
      {
      pthread_mutex_lock(&m);
      }

   void unlock()
      {
      pthread_mutex_unlock(&m);
      }

private:
      pthread_mutex_t       m;
};
```

```
class mutex; // forward declaration

class mutex_locker
{
public:
   mutex_locker(mutex &mx) // we are designed to be called under the RAII paradigm
      {
      m=&mx;
      m->lock();
      }

   ~mutex_locker()
      {
      m->unlock();
      }

private:
      mutex         *m;
};
```

We then redefine the singleton class as follows:

```
class mutex; // forward declaration

class singleton
{
public:
   static singleton* instance();

protected:
   singleton();

private:
   static singleton        *inst;
   static mutex            m;
};
```

Implementation of the singleton class:

```
singleton::singleton()
{
   ... // do whatever initialisation is necessary
}

singleton* singleton::instance()
{
   mutex_locker lock(m);

   if(inst==0)
      inst=new singleton;

   return inst;
}
```

Runtime declaration of the static members:

```
singleton *singleton::inst=0;
mutex singleton::m;
```

Note the use of the `mutex_locker` class in the `singleton::instance()` function. The `mutex_lock` being used as an RAII object, also known as scoped lock. `mutex_locker`'s constructor aquires the lock; its destructor releases it. This guarantees that the mutex lock will be relinquished even if an exception is thrown du the execution of `singleton::instance()`, since the language specification pledges that the destructors o automatically (i.e. stack) allocated objects are invoked during stack unwind.

An obvious concern is the cost of acquiring and releasing the mutex every time `singleton::instance()` called. There are several approaches available for ameliorating this problem. Firstly, code should be profiled to the cost of the synchronisation operation is significant (premature optimisation is the root of all evil—Hoare, K Modern multi-threaded operating systems have been designed with special care given to the efficiency of concu primitives (a few hundred nanoseconds to a few microseconds per operation is not atypical). It may well be tha cost of locking and unlocking the mutex is trivial at runtime. Secondly, the nature of the singleton class is the representation of a long-lived, process-unique object that should only be created once. Pushing its first invocati early in the lifetime of a process obviates the need for locking (a compile-time flag to disable the mutex might b useful here; clearly for a single-threaded application the mutex is completely redundant). Likewise, if the concu model of the application is the master/slave paradigm, the singleton can be initialised in the main thread before threads are created. Lastly, an `instance` of the (mutex-aware) singleton can be acquired at thread startup (in model, using the `pthread_once()` function) and assigned to thread-local storage.

C# example implementation

```
public class Singleton
{
     private static readonly Singleton instance = new Singleton();

     private Singleton() {}

     public static Singleton Instance
     {
          get { return instance; }
     }
}
```

C# example implementation using Generics

This example is thread safe with lazy initialization

```
/// <summary>
/// Generic class implements singleton pattern.
/// </summary>
/// <typeparam name="T">
/// Reference type. Important: Must have private constructor (not public).
/// </typeparam>
public class Singleton<T> where T : class
{
  Singleton() { }

  public static T Instance
  {
    get { return SingletonCreator.instance; }
  }

  class SingletonCreator
  {
    static SingletonCreator() { }

    internal static readonly T instance = typeof(T).InvokeMember(typeof(T).Name,
                          BindingFlags.CreateInstance |
                          BindingFlags.Instance |
                          BindingFlags.NonPublic,
                          null, null, null) as T;
  }
}
```

REALbasic example implementation

An example REALbasic solution uses a "Shared" method (available only in REALbasic 2006r1 or greater) to provide the instance. Note that this is thread-safe because REALbasic uses a cooperative threading model (instead of a preemptive one).

```
Class Singleton
  Protected Sub Constructor()
     // Initialization code defined here
  End Sub
  Shared Function Instance() as Singleton
     static s as new Singleton
     return s
  End Function
End Class
```

In a prototype-based programming language, where objects but not classes are used, a "singleton" simply refers to an object without copies or that is not used as the prototype for any other object.

Singleton (LoadBalancer) defines an Instance operation that lets clients access its unique instance. Instance is a class operation responsible for creating and maintaining its own unique instance. Ensure a class has only one instance and provide a global point of access to it.

ActionScript 3 Example Implementation

```
package
   public final class Singleton {
       private static var instance:Singleton = new Singleton();

       public function Singleton() {
           if (instance) throw new Error("Singleton and can only be accessed through Singleton.getInst
       }

       public static function getInstance():Singleton {
           return instance;
       }
   }
}
```

Ruby example implementation

Thread safe

```
class SingleObj
  include Singleton
    # only one instance of this class can be created
end
```

Not thread safe

```
class SingleObj
  private_class_method :new
  @@single = nil
  def Singleton.create
    @@single = new unless @@single
    @@single
  end
end
```

Python Borg pattern

According to influential Python programmer Alex Martelli, *The Singleton design pattern (DP) has a catchy na*
the wrong focus—on identity rather than on state. The Borg design pattern has all instances share state instead
rough consensus in the Python community is that sharing state among instances is more elegant, at least in Pyth
than is caching creation of identical instances on class initialization. Coding shared state is nearly transparent:

```
class Borg:
    __shared_state = {}
    def __init__(self):
        self.__dict__ = self.__shared_state
    # and whatever else you want in your class -- that's all!
```

Example of usage with the factory method pattern

The singleton pattern is often used in conjunction with the factory method pattern to create a system-wide reso
whose specific type is not known to the code that uses it. An example of using these two patterns together is th
Abstract Windowing Toolkit (AWT).

`java.awt.Toolkit` (`http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Toolkit.html`) is an abstract c
binds the various AWT components to particular native toolkit implementations. The `Toolkit` class has a

`Toolkit.getDefaultToolkit()` (`http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Toolkit.html#getDefaultToolkit`) factory method that returns the platform-specific subclass of `Toolkit`. The `Toolkit` object is a singleton because the AWT needs only a single object to perform the binding and the object is relatively expensive to create. The toolkit methods must be implemented in an object and not as static methods of a class because the specific implementation is not known by the platform-independent components. The name of the specific `Toolkit` subclass used is specified by the "`awt.toolkit`" environment property accessed through `System.getProperties()` (`http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#getProperties()`).

The binding performed by the toolkit allows, for example, the backing implementation of a `java.awt.Window` (`http://java.sun.com/j2se/1.5.0/docs/api/java/awt/Window.html`) to bound to the platform-specific `java.awt.peer.WindowPeer` implementation. Neither the `Window` class nor the application using the window needs to be aware of which platform-specific subclass of the peer is used.

See also

- Initialization on Demand Holder Idiom
- Double-checked locking
- Multiton pattern

Footnotes

1. ^ David Geary. Simply Singleton (http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html) . *JavaWorld*. Retrieved on 2006-10-02.
2. ^ Alex Martelli. Singleton? We don't need no stinkin' singleton: the Borg design pattern (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/66531) . *ASPN Python Cookbook*. Retrieved on 2006-09-07.

References

- "C++ and the Perils of Double-Checked Locking" (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf#search=%22meyers%20double%20checked% Meyers, Scott and Alexandrescu, Andrei, September 2004.
- "The Boost.Threads Library" (http://www.ddj.com/dept/cpp/184401518) Kempf, B., Dr. Dobb's Portal, April 2003.

External links

- A Pattern Enforcing Compiler™ (http://pec.dev.java.net/) that enforces the Singleton pattern amongst other patterns
- Description from the Portland Pattern Repository (http://c2.com/cgi/wiki?SingletonPattern)
- Description (http://home.earthlink.net/~huston2/dp/singleton.html) by Vince Huston
- Implementing the Singleton Pattern in C# (http://www.yoda.arachsys.com/csharp/singleton.html) by Jon Skeet
- A Threadsafe C++ Template Singleton Pattern for Windows Platforms (http://www.opbarnes.com/blog/Programming/OPB/Snippets/Singleton.html) by O. Patrick Barnes
- Implementing the Inheritable Singleton Pattern in PHP5 (http://svn.shadanakar.org/filedetails.php?repname=onPHP&path=%2Ftrunk%2Fcore%2FBase%2FSingl
- Singleton Pattern and Thread Safety (http://www.oaklib.org/docs/oak/singleton.html)
- PHP patterns (http://www.php.net/manual/en/language.oop5.patterns.php)
- Singleton Considered Stupid (http://opal.cabochon.com/~stevey/blog-rants/singleton-stupid.html)
- Article "Double-checked locking and the Singleton pattern (http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j) " by Peter Haggar
- Article "Use your singletons wisely (http://www-106.ibm.com/developerworks/library/co-single.html) " b Rainsberger
- Article "Simply Singleton (http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.htm David Geary
- Article "Description of Singleton (http://www.dofactory.com/Patterns/PatternSingleton.aspx) " by Arun

Design patterns in *Design Patterns*

Creational: Abstract factory • Builder • Factory • Prototype • **Singleton**

Structural: Adapter • Bridge • Composite • Decorator • Façade • Flyweight • Proxy

Behavorial: Chain of responsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • S • Template method • Visitor

Categories: Software design patterns | Articles with example Java code | Articles with example C++ code | Arti with example C Sharp code | Articles with example REALbasic code | Articles with example ActionScript code Articles with example Ruby code | Articles with example Python code

---