Singleton pattern

From Wikipedia, the free encyclopedia

In software engineering, the **singleton pattern** is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

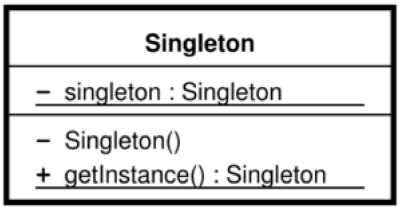
There are some who are critical of the singleton pattern and consider it to be an anti-pattern in that it is frequently used in scenarios where it is not beneficial, introduces unnecessary restrictions in situations where a sole instance of a class is not actually required, and introduces global state into an application.[1][2][3]

Contents

Common uses

- The abstract factory, builder, and prototype patterns can use Singletons in their implementation.
- Facade objects are often singletons because only one Facade object is required.
- State objects are often singletons.
- Singletons are often preferred to global variables because:
  - They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.[4]
  - They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

UML



Implementation

Implementation of a singleton pattern must satisfy the single instance and global access principles. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects. The singleton pattern is implemented by creating a class with a method that creates a new instance of the class 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 private. 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**.

Example

The solutions given here in Java are all thread-safe, but differ in supported language versions and lazy-loading characteristics. Since Java 5.0, the easiest way to create a Singleton is the enum type approach, as shown at the end of the section.

Lazy initialization

This method uses double-checked locking, which should not be used prior to J2SE 5.0, as it is vulnerable to subtle bugs.[5] The problem is that an out-of-order write may allow the `instance` reference to be returned before the `Singleton` constructor is executed.[6]

```
public final class SingletonDemo {
    private static volatile SingletonDemo instance;
    private SingletonDemo() { }

    public static SingletonDemo getInstance() {
        if (instance == null ) {
            synchronized (SingletonDemo.class) {
                if (instance == null) {
                    instance = new SingletonDemo();
                }
            }
        }

        return instance;
    }
}
```

An alternate simpler and cleaner version may be used at the expense of potentially lower concurrency in a multithreaded environment:

```
public final class SingletonDemo {
    private static SingletonDemo instance = null;
    private SingletonDemo() { }

    public static synchronized SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }
```

```
        return instance;
    }
}
```

Eager initialization

If the program will always need an instance, or if the cost of creating the instance is not too large in terms of time/resources, the programmer can switch to eager initialization, which always creates an instance:

```
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

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

This method has a number of advantages:

- The static initializer is run when the class is initialized, after class loading but before the class is used by any thread.
- There is no need to synchronize the `getInstance()` method, meaning all threads will see the same instance and no (expensive) locking is required.
- The `final` keyword means that the instance cannot be redefined, ensuring that one (and only one) instance ever exists.

Static block initialization

Some authors refer to a similar solution allowing some pre-processing (e.g. for error-checking).[7] In this sense, the traditional approach could be seen as a particular case of this one, as the class loader would do exactly the same processing.

```
public final class Singleton {
    private static final Singleton instance;

    static {
        try {
            instance = new Singleton();
        } catch (Exception e) {
            throw new RuntimeException("Darn, an error occurred!", e);
        }
    }

    public static Singleton getInstance() {
        return instance;
    }

    private Singleton() {
        // ...
    }
}
```

Initialization-on-demand holder idiom

University of Maryland Computer Science researcher Bill Pugh has written about the code issues underlying the Singleton pattern when implemented in Java.[8] Pugh's efforts on the "Double-checked locking" idiom led to changes in the Java memory model in Java 5 and to what is generally regarded as the standard method to

implement Singletons in Java. The technique known as the initialization-on-demand holder idiom is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines.

The nested 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 final class Singleton {
        // Private constructor. Prevents instantiation from other classes.
        private Singleton() { }

        /**
         * Initializes singleton.
         *
         * {@link SingletonHolder} is loaded on the first execution of {@link Singleton#getInstance()} or the
         * {@link SingletonHolder#INSTANCE}, not before.
         */
        private static class SingletonHolder {
                private static final Singleton INSTANCE = new Singleton();
        }

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

Alternatively, the inner class SingletonHolder can also be substituted by implementing a Property which provides also access to the static final/read-only class members. Just like the lazy object in C#, whenever the Singleton.INSTANCE property is called, this singleton is instantiated for the very first time.

The enum way

In the second edition of his book *Effective Java*, Joshua Bloch claims that "a single-element enum type is the best way to implement a singleton"[9] for any language that supports enums, like Java. The use of an enum is very easy to implement and has no drawbacks regarding serializable objects, which have to be circumvented in the other ways.

```
public enum Singleton {
    INSTANCE;
    public void execute (String arg) {
        // Perform operation here
    }
}
```

The public method can be written to take any desired types of arguments; a single String argument is used here as an example.

This approach implements the singleton by taking advantage of Java's guarantee that any enum value is instantiated only once in a Java program. Since Java enum values are globally accessible, so is the singleton, initialized lazily by the class loader. The drawback is that the enum type is somewhat inflexible.

Prototype-based singleton

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. Example in Io:

```
Foo := Object clone
Foo clone := Foo
```

Example of use with the abstract factory pattern

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

`java.awt.Toolkit (https://docs.oracle.com/javase/8/docs/api/java/awt/Toolkit.html)` is an abstract class that binds the various AWT components to particular native toolkit implementations. The `Toolkit` class has a `Toolkit.getDefaultToolkit() (https://docs.oracle.com/javase/8/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() (https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#getProperties())`.

The binding performed by the toolkit allows, for example, the backing implementation of a `java.awt.Window (https://docs.oracle.com/javase/8/docs/api/java/awt/Window.html)` to bind 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.

References

1. Scott Densmore. Why singletons are evil (http://blogs.msdn.com/scottdensmore/archive/2004/05/25/140827.aspx), May 2004
2. Steve Yegge. Singletons considered stupid (http://steve.yegge.googlepages.com/singleton-considered-stupid), September 2004
3. Clean Code Talks - Global State and Singletons (http://googletesting.blogspot.com/2008/11/clean-code-talks-global-state-and.html)
4. Gamma, E, Helm, R, Johnson, R, Vlissides, J: "Design Patterns", page 128. Addison-Wesley, 1995
5. "The "Double-Checked Locking is Broken" Declaration". University of Maryland. 21 April 2016.
6. Haggar, Peter (1 May 2002). "Double-checked locking and the Singleton pattern". IBM.
7. Coffey, Neil (November 16, 2008). "JavaMex tutorials". Retrieved April 10, 2012.
8. Pugh, Bill (November 16, 2008). "The Java Memory Model". Retrieved April 27, 2009.
9. Joshua Bloch: *Effective Java* 2nd edition, ISBN 978-0-321-35668-0, 2008, p. 18

- "C++ and the Perils of Double-Checked Locking" (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf#search=%22meyers%20double%20checked%20locking%22) 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

- Complete article "Singleton Design Pattern techniques (http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/)"
- 4 different ways to implement singleton in Java "Ways to implement singleton in Java (http://www.javaexperience.com/design-patterns-singleton-design-pattern/)"
- Book extract: Implementing the Singleton Pattern in C# (http://csharpindepth.com/Articles/General/Singleton.aspx) by Jon Skeet

The Wikibook *Computer Science/Design Patterns* has a page on the topic of: ***Singleton implementations in various languages***

- Singleton at Microsoft patterns & practices Developer Center (http://msdn.microsoft.com/en-us/library/ms998426.aspx)
- Ruby standard library documentation for Singleton (http://ruby-doc.org/stdlib/libdoc/singleton/rdoc/index.html)
- IBM article "Double-checked locking and the Singleton pattern (http://www-128.ibm.com/developerworks/java/library/j-dcl.html?loc=j)" by Peter Haggar
- IBM article "Use your singletons wisely (http://www.ibm.com/developerworks/library/co-single/)" by J. B. Rainsberger
- Javaworld article "Simply Singleton (http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html)" by David Geary
- Google article "Why Singletons Are Controversial (http://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial)"
- Google Singleton Detector (http://code.google.com/p/google-singleton-detector/) (analyzes Java bytecode to detect singletons)

Wikimedia Commons has media related to ***Singleton pattern***.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=720553408"

Categories: Software design patterns | Anti-patterns | Software optimization

---