The Singleton Pattern

Other Considerations

Douglas C. Schmidt
Learning Objectives in This Lesson

• Recognize how the Singleton pattern can be applied to centralize access to global resources.
• Understand the structure & functionality of the Singleton pattern.
• Know how to implement the Singleton pattern in C++.
• Be aware of other considerations when applying the Singleton pattern.
Consequences

+ Helps “declutter” class & method interfaces
Consequences

+ Reduces namespace pollution & centralizes access to global resources

See [wiki.c2.com/?GlobalVariablesConsideredHarmful](http://wiki.c2.com/?GlobalVariablesConsideredHarmful)
Consequences
+ Allows extension by subclassing

Singleton

static Instance()
SingletonOperation()
GetSingletonData()

static uniqueInstance
singletonData

Derived Singleton

If (uniqueInstance == null)
uniqueInstance = new Singleton();
return uniqueInstance;
Consequences

+ Only allocates resources for objects actually accessed at least once

See en.wikipedia.org/wiki/Lazy_initialization
Consequences

+ Alleviates problems with global variables in certain programming languages
Consequences

– Does not address all the liabilities with global variables

• In particular, increased implicit dependencies & reduced program clarity
Consequences

- Implementation may be less efficient than a global variable
  - Due to additional indirection & synchronization overhead
Consequences

- Subtle concurrency & dynamic loading traps & pitfalls
Singleton

Consequences

- [c2.com/cgi/wiki?SingletonsAreEvil](c2.com/cgi/wiki?SingletonsAreEvil) summarizes *Singleton* drawbacks
Implementation considerations

• Determine if you really must use Singleton!
  • [www.ibm.com/developerworks/webservices/library/co-single](http://www.ibm.com/developerworks/webservices/library/co-single) has good tips:
    • Will every application use this class exactly the same way? (*exactly* is the key word)
    • Will every application ever need only one instance of this class? (*ever* & *one* are the key words)
    • Should the clients of this class be unaware of the application they are part of?

It’s often possible (& desirable) to avoid using *Singleton* in your programs.
Implementation considerations

• Defining static instance method & data

class Singleton {
    private static Singleton sInst = null;
    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            sInst = result = new Singleton();
        }
        return result;
    }
    ...
}
Implementation considerations

• Avoiding concurrency hazards

```java
class Singleton {
    private static Singleton sInst = null;
    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            sInst = result = new Singleton();
        }
        return result;
    }
}
...
Implementation considerations

- Avoiding concurrency hazards

```java
class Singleton {
    private static Singleton sInst = null;
    public static Singleton instance() {
        synchronized (Singleton.class) {
            Singleton result = sInst;
            if (result == null) {
                sInst = result = new Singleton();
            }
            return result;
        }
    }
}
```

*Too much synchronization*
Implementation considerations

• Avoiding concurrency hazards

```java
class Singleton {
    private static volatile Singleton sInst = null;
    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = sInst;
                if (result == null) {
                    sInst = result =
                    new Singleton();
                }
            }
        }
        return result;
    }
    ...
}
```

Just the right amount of synchronization
Implementation considerations

• Avoiding concurrency hazards

```java
class Singleton {
    private static volatile Singleton sInst = null;

    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            synchronized (Singleton.class) {
                result = sInst;
                if (result == null) {
                    sInst = result = new Singleton();
                }
            }
        }
        return result;
    }
}
...
```

*Only synchronize when sInst is null*
**Implementation considerations**

- Avoiding concurrency hazards

```java
class Singleton {
    private static volatile Singleton sInst = null;

    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = sInst;
                if (result == null) {
                    sInst = result = new Singleton();
                }
            }
        }
        return result;
    }
    ...
}
```

*No synchronization after sInst is created*
Implementation considerations

• Avoiding concurrency hazards

```java
class Singleton {
    private static volatile Singleton sInst = null;

    public static Singleton instance() {
        Singleton result = sInst;
        if (result == null) {
            synchronized(Singleton.class) {
                result = sInst;
                if (result == null) {
                    sInst = result =
                    new Singleton();
                }
            }
        }
        return result;
    }
    ...
}
```

This solution only works in JDK5 & above.

Implementation considerations

- Avoiding concurrency hazards

```java
class Singleton {
    /** Private constructor */
    private Singleton() { }

    /**
     * SingletonHolder’s loaded on first
     * execution of Singleton.instance()
     * or first access to SingletonHolder.
     * INSTANCE, not before
     */
    private static class SingletonHolder {
        public static final Singleton
            instance_ = new Singleton();
    }

    /** Returns single instance */
    public static Singleton instance() {
        return SingletonHolder.instance_;}
}
```

This solution works in all JDKs!

en.wikipedia.org/wiki/Singleton_pattern#The_solution_of_Bill_Pugh has more.
Implementation considerations

- Deleting singletons
Implementation considerations

- Registering the singleton instance with manager

<table>
<thead>
<tr>
<th>Object Lifetime Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>init ()</td>
</tr>
<tr>
<td>fini ()</td>
</tr>
<tr>
<td>static starting_up ()</td>
</tr>
<tr>
<td>static shutting_down ()</td>
</tr>
<tr>
<td>static at_exit ()</td>
</tr>
<tr>
<td>static instance ()</td>
</tr>
<tr>
<td>at_exit_i ()</td>
</tr>
<tr>
<td>static instance_</td>
</tr>
</tbody>
</table>

www.dre.vanderbilt.edu/~schmidt/PDF/ObjMan.pdf: Singleton management
Known uses

- Unidraw’s Unidraw object
- Smalltalk-80 ChangeSet, the set of changes to code
- InterViews Session object
- ACE Singleton

```cpp
template <typename TYPE>
TYPE *ACE_Singleton<TYPE>::instance () {
    TYPE *tmp = instance_;  
    #if defined (ALPHA_MP)
        // Insert CPU-specific memory barrier
        // instruction to synchronize cache lines. 
        asm ("mb");
    #endif /* ALPHA_MP */
    // First check
    if (tmp == 0) {
        ACE_Guard<ACE_Thread_Mutex> guard (lock_);
        tmp = instance_; // Reload tmp.
        // Double check.
        if (tmp == 0) {
            tmp = new TYPE;
            #if defined (ALPHA_MP)
                // Insert a second CPU-specific memory
                // barrier instruction.
                asm ("mb");
            #endif /* ALPHA_MP */
            instance_ = tmp;
        }
    }
    return tmp;
}
```

Known uses

- Unidraw’s Unidraw object
- Smalltalk-80 ChangeSet, the set of changes to code
- InterViews Session object
- ACE Singleton
- The Java AWT Desktop `getDesktop()` method

### `getDesktop`

```java
public static Desktop getDesktop()
```

Returns the Desktop instance of the current browser context. On some platforms the Desktop API may not be supported; use the `isDesktopSupported()` method to determine if the current desktop is supported.

**Returns:**
the Desktop instance of the current browser context

**Throws:**
HeadlessException - if `GraphicsEnvironment.isHeadless()` returns true

UnsupportedOperationException - if this class is not supported on the current platform

**See Also:**
`isDesktopSupported()`, `GraphicsEnvironment.isHeadless()`

See [docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop](https://docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop)
Summary of the Singleton Pattern

- **Singleton** simplifies access to global resources in the expression tree processing app.

Singleton is the “go-to” of patterns, so apply it with care.