**Chapter 5 Notes – Singleton Pattern**

**Overview**

1. The intent of the singleton pattern is to ensure that a class has only one instance and provide a global point of access to it. A generalized class diagram is shown on the right.
2. Implementation of the classic singleton using lazy instantiation:

**public** **class** Singleton {

 **private** **static** Singleton *instance* = **null**;

 **private** Singleton() {

 // Exists only to prevents instantiation.

 }

 **public** **static** Singleton getInstance() {

 **if**(*instance* == **null**) {

 *instance* = **new** Singleton();

 }

 **return** *instance*;

 }

 **public** String doSomething() {

 **return** "I'm a Singleton";

 }

}

Using the singleton:

Singleton singleton = Singleton.*getInstance*();

System.***out***.println(singleton.doSomething());

**Singleton in Java API**

1. The *Runtime* class is a singleton. Every Java application has a single instance of the *Runtime* class that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the *getRuntime* method.

Runtime runtime = Runtime.*getRuntime*();

// run any executable, in this case notepad

**try** {

 runtime.exec( "notepad.exe" );

} **catch** (IOException e) {

 e.printStackTrace();

}

System.***out***.println( "num processors=" + runtime.availableProcessors() );

System.***out***.println( "avail memory=" + runtime.freeMemory() );

1. The *Desktop* class is a singleton and allows a Java application to launch associated applications registered on the native desktop to handle a URI or a file. The *getDesktop* method returns the instance. Sample code is shown below (the *File* and *URI* instances must be surrounded with try/catch):

Desktop desktop = Desktop.*getDesktop*();

// Open outlook, can also specify a URI (to, from, etc)

String msg = "mailto:davegibson2@gmail.com?subject=Launched%20From%20Java";

URI uri = URI.*create*(msg);

desktop.mail(uri);

// Opens this file in notepad

desktop.edit( **new** File("src\\desktop\\stuff.txt") );

// Opens word with this file.

desktop.edit( **new** File("src\\desktop\\stuff.docx") );

// opens default browser with this page

desktop.browse( **new** URI("http://www.valdosta.edu") );

// prints from notepad

desktop.print( **new** File("src\\desktop\\stuff.txt") );

// prints from word.

desktop.print( **new** File("src\\desktop\\stuff.docx") );

1. As we saw earlier, *Toolkit* is an abstract factory. *Toolkit* is also a singleton. The static method, *getDefaultToolkit* returns a concrete toolkit as a singleton*.*



**Singleton Issues**

1. Although rare, the Singleton, as written is not thread safe. If a thread is preempted at Line 2 before the assignment is made, the *instance* variable will still be null, and another thread can subsequently enter the if block. In that case, two distinct singleton instances will be created.

1: if(instance == null) {

2: instance = new Singleton();

3: }

1. A solution is to use double-checked locking.

**public** **static** Singleton getInstance() {

 **if**(*singleton* == **null**) {

 **synchronized**(Singleton.**class**) {

 **if**(*singleton* == **null**) {

 *singleton* = **new** Singleton();

 }

 }

 }

 **return** *singleton*;

}

However, double-checked locking has its own issues:

<https://en.wikipedia.org/wiki/Double-checked_locking#Usage_in_Java>

1. Another solution to the thread problem is to simply use an eagerly created instance:

**public** **class** Singleton {

 **private** **static** Singleton *singleton* = **new** Singleton();

 **protected** Singleton() {}

 **public** **static** Singleton getInstance() {

 **return** *singleton*;

 }

}

1. A better way to use the eagerly created singleton approach is to define the *Singleton* as a static inner class and employ the enclosing class (*SingletonWrapper*) to retrieve the instance. This has the benefit that inner classes are not loaded until they are referenced. Thus, perhaps we shouldn’t say “eagerly created.”

**public** **class** SingletonWrapper {

 **private** **static** **class** Singleton {

 **public** **static** **final** Singleton ***singleton*** = **new** Singleton();

 **public** String doSomething() {

 **return** "I'm a singleton";

 }

 }

 **public** **static** Singleton getInstance() {

 **return** Singleton.***singleton***;

 }

 **public** **static** **void** main(String[] args) {

 Singleton s = SingletonWrapper.*getInstance*();

 System.***out***.println(s.doSomething());

 }

}

1. Another problem with Singletons is that different applications can be using different class loaders. In such a case, each application will obtain a different (unique) singleton. The only solution to this is to specify the class loader to be the one that loaded the singleton originally. An incredibly thorough read in singleton can be found here:

[https://www.securecoding.cert.org/confluence/display/java/MSC07-J.+Prevent+multiple+instantiations+of+singleton+objects](https://www.securecoding.cert.org/confluence/display/java/MSC07-J.%2BPrevent%2Bmultiple%2Binstantiations%2Bof%2Bsingleton%2Bobjects)

1. This blog post, “Why Singletons are Evil” is a good read. In summary:
2. Singleton, as any global variable, creates hidden dependencies in your code. Non-static objects are passed to methods and thus the dependency is explicit. With singleton you have to inspect the code to see where it is used.
3. Violation of the “Single Responsibility Principle.” A class should not care whether it is a singleton; it should only be concerned with its business responsibilities.
4. Classes that use the singleton are tightly coupled with the singleton and are harder to test.
5. Singletons carry state with them as long as the program runs. This can hide bugs when testing as the state is carried from test to test.

<https://blogs.msdn.microsoft.com/scottdensmore/2004/05/25/why-singletons-are-evil/>

**Real-World Singleton Examples**

1. Apache Tomcat is an open source servlet container (application server) which provides a "pure Java" HTTP web server environment for Java code to run. It utilizes a StringManager singleton registry to process error messages. Error messages are cataloged by the package that contains a class. It contains a getManager(packageName) method that returns a singleton from which error information can be obtained.

Source: <http://onjava.com/onjava/2003/08/27/singleton.html>

1. Another real example is a component used for Credit Card Validation:

<http://software.topcoder.com/catalog/c_component.jsp?comp=10515357&ver=1>

**Homework**

1. Consider a class, *LogData* that stores the date, and time that a resource was accessed. Suppose you need a *Logger* class that stores instances of *LogData* at various places in a software system. For example, in certain methods of the system you want to create an instance of *LogData* and have the *Logger* store it. The *Logger* class should also provide a list of *LogData* that has been stored. Write the *Logger* class as a singleton and provide code showing how it is used. For the purposes of this problem, simply define the *LogData* class like this:

**import** java.util.GregorianCalendar;

**public** **class** LogData {

 **private** GregorianCalendar cal;

 **public** LogData(GregorianCalendar cal) {

 **this**.cal = cal;

 }

 @Override

 **public** String toString() {

 // Short Date Format

 String date = String.*format*("%tm/%td/%ty",cal,cal,cal);

 // Short Time Format

 String time = String.*format*("%tl:%tM:%tS %tp",cal,cal,cal,cal);

 **return** date + " " + time;

 }

 **public** **static** **void** main( String[] args ) **throws** InterruptedException {

 LogData logData1 = **new** LogData(**new** GregorianCalendar());

 Thread.*sleep*(2000);

 LogData logData2 = **new** LogData(**new** GregorianCalendar());

 System.***out***.println(logData1);

 System.***out***.println(logData2);

 }

}