Singleton Pattern

What is it?

- If you need to make sure that there can be one and only one instance of a class.
  - For example, your system can have only one window manager or print spooler, or a single point of access to a database engine.

Approach 1

- Embed a static variable inside the class that we set on the first instance and check for each time we enter the constructor.
  - A static variable is one for which there is only one instance, no matter how many instances there are of the class.
    - static boolean instance_flag = false;

Disadvantage of Approach 1

- How to find out whether creating an instance was successful or not
  - Remember constructors do not return values.
- One way would be to call a method that checks for the success of creation, and which simply returns some value derived from the static variable.
  - This is inelegant and prone to error
    - because there is nothing to keep you from creating many instances of such non-functional classes and forgetting to check for this error condition.

Approach 2

- Create a class that throws an Exception when it is instantiated more than once.
  - Let' s create our own exception class for this case

```
class SingletonException extends RuntimeException
{
   //new exception type for singleton classes
   public SingletonException()
   {
      super();
   }
//-----------------------------------------------
   public SingletonException(String s)
   {
      super(s);
   }
}
```

Why a New Exception?

- This new exception type doesn' t do anything in particular
  - other than calling its parent classes through the super() method,.
- However, it is convenient to have our own named exception type so that the compiler will warn us of the type of exception we must catch when we attempt to create an instance

Lets Implement the Class

```
class PrintSpooler
{
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean
              instance_flag=false; //true if 1 instance

   public PrintSpooler() throws SingletonException
   {
   if (instance_flag)
      throw new SingletonException("Only one spooler allowed");
   else
      instance_flag = true;   //set flag for 1 instance
      System.out.println("spooler opened");
   }
   //-------------------------------------------
   public void finalize()
   {
      instance_flag = false;     //clear if destroyed
   }
}
```

Lets Use it!

```
public class singleSpooler
{
   static public void main(String argv[])
   {
      PrintSpooler pr1, pr2;

     //open one spooler--this should always work
      System.out.println("Opening one spooler");
      try{
      pr1 = new PrintSpooler();
      }
      catch (SingletonException e)
      {System.out.println(e.getMessage());}

      //try to open another spooler --should fail
      System.out.println("Opening two spoolers");
     try{
     pr2 = new PrintSpooler();
     }
     catch (SingletonException e)
     {System.out.println(e.getMessage());}
   }
}
```

Disadvantage of Approach 2

- Must enclose every method that may throw an exception in a try - catch block.
- And the exception-based solution is not really “transparent”

Approach 3

- There already is a kind of Singleton class in the standard Java class libraries: the Math class.
  - This is a class that is declared final and all methods are declared static, meaning that the class cannot be extended.
  - The purpose of the Math class is to wrap a number of common mathematical functions such as sin and log in a class-like structure, since the Java language does not support functions that are not methods in a class.
- You can’t create any instance of classes like Math, and can only call the static methods directly in the existing final class.
- You can use the same approach to a Singleton pattern, making it a final class.

Approach 3

```
final class PrintSpooler
{
 //a static class implementation of Singleton pattern
 static public void print(String s)
 {
 System.out.println(s);
 }
}
//==============================
public class staticPrint
{
   public static void main(String argv[])
   {
      Printer.print("here it is");
   }
}
```

Disadvantage of Approach 3

- Difficult to drop the restrictions of Singleton status
  - a lot of reprogramming to do to make the static approach allow multiple instances
- this is easier to do in the exception style class structure

Approach 4

- Create Singletons using a static method to issue and keep track of instances.
- To prevent instantiating the class more than once, make the constructor private so an instance can only be created from within the static method of the class

```
class iSpooler
{
   //this is a prototype for a printer-spooler class
   //such that only one instance can ever exist
   static boolean instance_flag = false; //true if 1 instance

  //the constructor is privatized-
  //but need not have any content
   private iSpooler()    {  }
//static Instance method returns one instance or null
   static public iSpooler Instance()
   {
      if (! instance_flag)
      {
         instance_flag = true;
         return new iSpooler();   //only callable from within
      }
      else
         return null;      //return no further instances
   }
   //--------------------------------------------
   public void finalize()
   {
      instance_flag = false;
   }
}
```

Approach 4

- Don't have to worry about exception handling if the singleton already exists-- you simply get a null return from the Instance method

Lets Use it!

```
 iSpooler pr1, pr2;
//open one spooler--this should always work
System.out.println("Opening one spooler");
pr1 = iSpooler.Instance();
if(pr1 != null)
   System.out.println("got 1 spooler");
//try to open another spooler --should fail
System.out.println("Opening two spoolers");

pr2 = iSpooler.Instance();
if(pr2 == null)
   System.out.println("no instance available");
```

Compile-time Error Checking

- should you try to create instances of the iSpooler class directly, this will fail at compile time because the constructor has been declared as private.

```
//fails at compile time because constructor is privatized
 iSpooler pr3 = new iSpooler();
```

Approach 5

```
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;
  }
}
```

Dropping the Singleton Requirement

- Suddenly, we decide that we can allow “n” instances of the (previously) Singleton object
- How would you adapt each of the previous five approaches?
  - Which implementation is “maintainable”?