THE SINGLETON PATTERN

The Singleton pattern is grouped with the other Creational patterns, although it is to some extent a “non-creational” pattern. There are any number of cases in programming where 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.

The easiest way to make a class that can have only one instance is to 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;
```

The problem is how to find out whether creating an instance was successful or not, since 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, however, because there is nothing to keep you from creating many instances of such non-functional classes and forgetting to check for this error condition.

A better way is to 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);
   }
}
```

Note that other than calling its parent classes through the `super()`method, this new exception type doesn’t do anything in particular. 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 of PrintSpooler.

Throwing the Exception

Let’s write the skeleton of our PrintSpooler class; we’ll omit all of the printing methods and just concentrate on correctly implementing the Singleton pattern:

```
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
   }
}
```

Creating an Instance of the Class

Now that we’ve created our simple Singleton pattern in the PrintSpooler class, let’s see how we use it. Remember that we must enclose every method that may throw an exception in a `try - catch` block.

```
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());}
   }
}
```

Then, if we execute this program, we get the following results:

```
Opening one spooler
printer opened
Opening two spoolers
Only one spooler allowed
```

where the last line indicates than an exception was thrown as expected. You will find the complete source of this program on the example CD-ROM as singleSpooler.java.

Static Classes as Singleton Patterns

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 use the same approach to a Singleton pattern, making it a *final* class. You can't create *any* instance of classes like Math, and can only call the static methods directly in the existing final class.

```
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");
   }
}
```

One advantage of the final class approach is that you don’t have to wrap things in awkward try blocks. The disadvantage is that if you would like to drop the restrictions of Singleton status, this is easier to do in the exception style class structure. We’d have a lot of reprogramming to do to make the static approach allow multiple instances.

Creating Singleton Using a Static Method

Another approach, suggested by *Design Patterns*, is to create Singletons using a static method to issue and keep track of instances. To prevent instantiating the class more than once, we 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;
   }
}
```

One major advantage to this approach is that you don't have to worry about exception handling if the singleton already exists-- you simply get a null return from the Instance method:

```
 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");
```

And, 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();
```

Finding the Singletons in a Large Program

In a large, complex program it may not be simple to discover where in the code a Singleton has been instantiated. Remember that in Java, global variables do not really exist, so you can't save these Singletons conveniently in a single place.

One solution is to create such singletons at the beginning of the program and pass them as arguments to the major classes that might need to use them.

```
pr1 = iSpooler.Instance();
Customers cust = new Customers(pr1);
```

A more elaborate solution could be to create a registry of all the Singleton classes in the program and make the registry generally available. Each time a Singleton instantiates itself, it notes that in the Registry. Then any part of the program can ask for the instance of any singleton using an identifying string and get back that instance variable.

The disadvantage of the registry approach is that type checking may be reduced, since the table of singletons in the registry probably keeps all of the singletons as Objects, for example in a Hashtable object. And, of course, the registry itself is probably a Singleton and must be passed to all parts of the program using the constructor or various set functions.

Other Consequences of the Singleton Pattern

1. It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
2. You can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.