The Singleton pattern

There can only be one!

Sometimes you want only one object

There are times when you design a system and realize that some of the objects should really only exist in one “canonical” (the one true) instance.

In Java’s API, we have a class called Runtime. It is supposed to desctribe the (one and only) runtime environment in which the JVM executes.

The designers decided it was important that there was only one representation (object) of the runtime system.

What can we do to ensure this requirement?

The Singleton design pattern

The Singleton pattern exists for exactly this reason. It is a solution to the problem of ensuring that a class has only one instance, and that it is globally accessible (all code must share this one and only instance).

There exists a few variants of the solution to this problem as we will see.

How to go about creating a Singleton

- private static variable of the same type as the class
- public static getInstance() method returning the variable
- Possibly, do initialization of the variable in the getInstace() method on the first call of the method
- Make all constructors private

Example Singleton class

```
public class Highlander{
  private static Highlander instance;
  private Highlander(){} // prevent instantiation
  public static Highlander getInstance(){
    if(instance == null){ // only first time this is true
      instance = new Highlander();
    }
    return instance;
  }
  // instance methods
}
// Client code:
Highlander.getInstance().someMethod();
```

Should we use Singletons everywhere we can?

For a discussion on good and bad use of Singletons, please see

https://www.ibm.com/developerworks/library/co-single/

(for instance (no pun intended))

As with most design patterns or anything related to programming, there are tons of hate-groups online, as well as tons of evangelists. Use your own judgement to follow whichever group you feel is right.

Disclaimer: Your judgement will differ as your experience grows ;-)

The Singleton example was lazily initialized

```
public static Highlander getInstance(){
    if(instance == null){ // only first time this is true
      instance = new Highlander();
    }
    return instance;
  }
```

Only when the first caller requests an instance, it is created.

A statically initialized version

```
public class Highlander{
  private static Highlander instance = new Highlander();
  // prevent instantiation
  private Highlander(){}
  // public access! Create the instance on first call
  public static Highlander getInstance(){
    return instance;
  }
  // instance methods...
}
// Client code:
Highlander.getInstance().someMethod();
```

A Singleton without a factory method

```
public class Highlander{
  public final static Hightlande INSTANCE=new Highlander();
  // prevent instantiation
  private HighLander(){}
  // instance methods...
}

// Client code:
Highlander.INSTANCE.someMethod();
```

Using an enum as a Singleton (modern stuff!)

```
public enum Highlander{
  INSTANCE;
  // instance methods...
}
//Client code:
Highlander.INSTANCE.someMethod();

// Bonus feature: Enums are thread-safe :)
// This new feature is available since... 2005
```