Design Patterns

The Singleton Design Pattern

The Issue:

Singleton

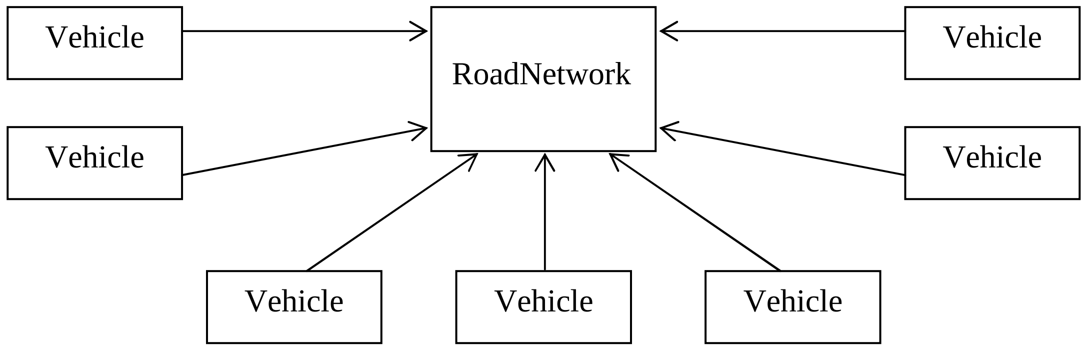
There are times when we create a class that we will only ever want to have *one* instance of. Examples of classes like this are those designed to handle database connections, logging, data caches, etc. Having multiple objects created in these situations is more than inconvenient, it can cause your program to run incorrectly and possibly crash.

It is often left up to the programmer to insure that only one instance is ever created. This is problematic because it is rare that a single programmer is the only one that will ever work on a piece of code for as long as it exists. There is no guarantee that some other programmer won’t inadvertently instantiate another instance of the class.

An important consideration in implementing this pattern is how to make this single instance easily accessible by many other objects- ideally without creating a global variable in which to store it.

The Issue (cont.):

Examine the following diagram.

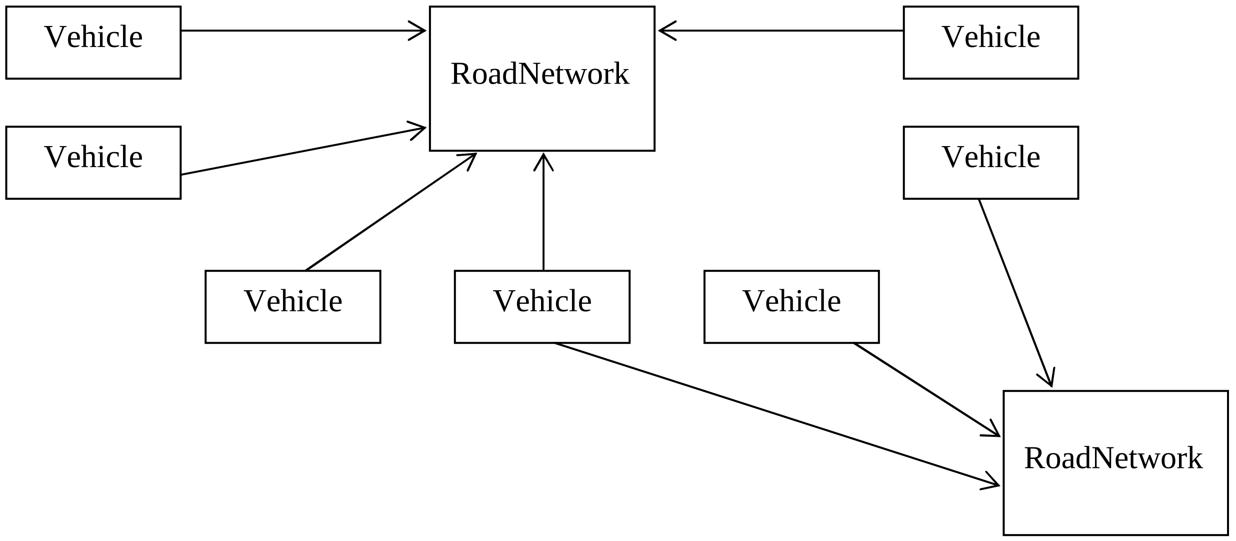


This shows a class relationship where multiple *Vehicle* objects access a single *RoadNetwork* object.

The *RoadNetwork* object contains all info related to elements of the road network.

The Issue (cont.):

Examine the following diagram.



If a second *RoadNetwork* object were inadvertently created, then 2 copies of the same *RoadNetwork* would exist, possibly in different states. Additionally, some *Vehicle* objects would refer to one *RoadNetwork* object; other *Vehicle* objects would refer the other *RoadNetwork* object.

The Solution:

The Singleton Pattern

Using the Singleton design pattern we can avoid this problem.

The Singleton design pattern is designed to restrict instantiation of a class to one object. The singleton pattern is implemented by creating a class with a method that creates a new instance of the object only if one does not already exist. If one does exist, it returns a reference to the object that already exists. If not, a new instance is created and a reference to that new object is returned.
To make sure that the object cannot be instantiated any other way the constructor is made either private or protected.

The 2 key points that are supported in the Singleton pattern are:

- Create a class that can only have one instance and allow the class to manage the one instance
- Provide a global access point to the instance

The Solution (cont.):

The following describes a basic Java implementation of the Singleton design pattern.

```
1  public final class Singleton
2  {
3     private static Singleton ourInstance;
4
5     public static Singleton getInstance() throws Exception
6     {
7           if (ourInstance == null)
8                 ourInstance = new Singleton();
9
10          return ourInstance;
11    }
12
13    private Singleton()
14    {
15    }
16
17    // Other methods here
18 }
```

- **Line 1** declares class “Singleton” as final, so that subclasses cannot be created that could provide multiple instantiations.
- **Line 3** declares a static reference to a Singleton object and invokes the private constructor.
- **Lines 5 through 11** declare a static accessor to the one Singleton object. If the “ourInstance” reference is “null”, the “ourInstance” reference is set to an instance of the Singleton class – this will be the only Singleton object instantiated. If the “ourInstance” reference is not “null” (indicating that it has already been instantiated) no new instance will be created. Ultimately, the “ourInstance” reference is returned.
- **Lines 13 through 15** declare a private constructor – only the Singleton class itself can instantiate a Singleton object using this constructor.

The Solution (cont.):

The Singleton object is accessed as follows:

```
Singleton.getInstance().someMethod();

int intVar = Singleton.getInstance().getSomeInteger();
```

The Singleton operates as follows:

First Call:

```
public static Singleton getInstance() throws Exception
{
   if (ourInstance == null)           // Here, it IS null.
      ourInstance = new Singleton();  // So we create the one instance
   return ourInstance;                // And return it!
}
```

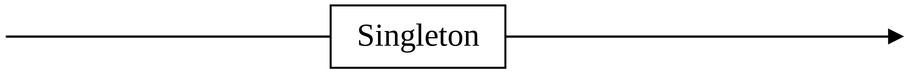
Second and Subsequent Calls:

```
public static Singleton getInstance() throws Exception
{
   if (ourInstance == null)           // Here, it is NOT null, so we DO NOT create a new instance
      ourInstance = new Singleton();
   return ourInstance;                // Just return the existing instance!
}
```

The Solution (cont.):

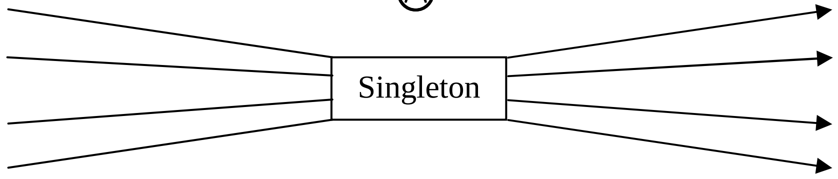
This works extremely well - in a single threaded environment.

☺



In a multithreaded environment however, things can break down.

☹



Recall that one of the main points of the Singleton pattern is:

- Create a class that can only have one instance

In a multithreaded environment, if 2 (or more) threads execute the static “getInstance()” method of the Singleton class at nearly the same time, it is possible to instantiate multiple instances of the Singleton class.

The Solution (cont.):

This scenario highlights why the basic Singleton implementation is not “thread safe”.

Two or more threads can (and eventually will) execute the static “getInstance()” method at nearly the same time, before the Singleton instance has been created. When this happens, as is shown below, you can end up with 2 *different* instances of the Singleton class. This violates the main point of the Singleton pattern - Create a class that can only have one instance.

Time ↓

Thread 1	Thread 2	Value of “ourInstance”

`public static Singleton getInstance()`
`throws Exception`
`{`
		`null`

	`public static Singleton getInstance()`
	`throws Exception`
	`{`
		`null`

`if (ourInstance == null)`		`null`
	`if (ourInstance == null)`	`null`

`ourInstance = new Singleton();`
`return ourInstance;`
`}`
		`Singleton Object #1`

	`ourInstance = new Singleton();`
	`return ourInstance;`
	`}`
		`Singleton Object #2`

The Solution (cont.):

There are several coding techniques that can solve this problem, making the Singleton pattern safe in a multithreaded environment.

Technique #1: Synchronize the "getInstance()" method:

```
public final class Singleton
{
   private static Singleton ourInstance;

   public static synchronized Singleton getInstance() throws Exception
   {
         if (ourInstance == null)
               ourInstance = new Singleton();

         return ourInstance;
   }

   private Singleton()
   {
   }

   // Other methods here
}
```

By adding the "synchronized" keyword to the "getInstance()" method, we require all threads to wait their turn before they can enter the method. Only one thread can execute the "getInstance()" method at one time.

However….

Synchronization can be an expensive option. The only time synchronization is really needed is the *first* time this method is called.

In subsequent calls, multithreading is no longer an issue because the instance has already been created.

This technique will require a thread to wait for some other thread to finish executing the "getInstance()" method *every* time it is called. This is unnecessary and will reduce the performance benefits of multithreading.

In the worst case, this can reduce performance by a factor of 100!

The Solution (cont.):

Technique #2: “Eager” instance initialization:

In the previous examples, we created the one Singleton instance *only* when we needed it, and not before. This technique is called “lazy” initialization and is a common technique used to avoid performing work that might never be needed.

If the Singleton instance will *always* be created in your application, you can perform an “eager” initialization, as shown below:

```
public final class Singleton
{
   private static Singleton ourInstance = new Singleton();

   public static Singleton getInstance() throws Exception
   {
         return ourInstance;
   }

   private Singleton()
   {
   }

   // Other methods here
}
```

This technique removes the synchronization issue present in the previous techniques completely by creating the one Singleton instance when the class is loaded by the JVM before any thread can ever access the “ourInstance” variable.

There *are* a couple of drawbacks to this technique:

- The Singleton constructor cannot throw any exceptions.
- You may not pass any parameters to the Singleton’s constructor. (In some advanced Singleton implementations, parameters might be needed by the Singleton constructor)

The Solution (cont.):

Technique #3: Double-Checked Locking:

In this technique, we do not synchronize the entire “getInstance()” method, we create a synchronized block.

We first check to see if the one Singleton instance has been created yet. If is has not, we *then* synchronize and create the instance – otherwise no synchronization is done.

```java
public final class Singleton
{
   private volatile static Singleton ourInstance;

   public static Singleton getInstance()
   {
      if (ourInstance == null)
      {
         synchronized (Singleton.class)
         {
            if (ourInstance == null) // Double-Check!
               ourInstance = new Singleton();
         }
      }
      return ourInstance;
   }

   private Singleton()
   {
   }

   // Other methods here
}
```

For variables marked with the “volatile” keyword, threads will be required to access the value of “ourInstance” from ***main*** *memory, rather than access cached variable values in local (thread) memory.*

The Solution (cont.):

The second key point of the Singleton pattern is: “Provide a global access point to the instance”

This is accomplished by invoking the static “getInstance()” method. Remember that “static” methods are considered “class” methods and can be invoked via the class (no instances needed):

ClassName.staticMethod()

In the context of the Singleton pattern, this means that we can invoke the static “getInstance()” method via a class without needing a reference to an instance. This can be done anywhere, removing the need to maintain some global variable to refer to the Singleton instance:

Singleton.getInstance()

Since the static “getInstance()” method returns a reference to the one Singleton instance, we can append to that call any method accessible to that instance:

Singleton.getInstance().someMethod();

boolean results = Singleton.getInstance().someOtherMethod(someInt);

$$\underbrace{\textit{Singleton.getInstance()}}_{\substack{\textit{Returns a reference to}\\ \textit{the Singleton instance}}}\underbrace{\textit{.someMethod();}}_{\substack{\textit{Invokes a method on}\\ \textit{that instance}}}$$

The Solution (cont.):

Random examples of Singleton usages:

- `Logger.getInstance().logDebug(“DEBUG: Value of ‘speed’ attribute: “ + getSpeed());`
- `DatabaseManager.getInstance().connect(uname, psswd, url);`
- `VehicleController.getInstance().findVehicle(id);`

-
  ```
  if (WarehouseManager.getInstance().hasItem(itemId))
  {
       […]
  }
  ```
- `ConnectionManager.getInstance().disconnect();`
- `int terrainType = Terrain.getInstance().getGroundCover(x,y);`
- `GUIController.getInstance().displayTotal(sumValue);`
- `MusicTrack mt = PlayList.getInstance().getRandomTrack();`

Singleton
Design Pattern

Summary:

- The Singleton pattern ensures that you can only ever have one instance of a class.
- Singleton classes manage their own single instance.
- The Singleton pattern provides global access to the one instance.
- Singleton pattern implementations make use of a private constructor and a static accessor method combined with a static variable.
- Implementations of the Singleton pattern can be tailored to single or multi-threaded applications.
- Multi-Threaded Singleton Techniques:
  - Synchronize the “getInstance()” method
  - Eager instance initialization
  - Double-Checked Locking