Singleton

Intent

- Ensure a class only has one instance, and provide a global point of access to it.

Motivation

- It's important for some classes to have exactly one instance.

1. Although there can be many printers in a system, there should be only one printer spooler.
2. There should be only one file system and one window manager.
3. A digital filter will have one A/D converter.
4. An accounting system will be dedicated to serving one company.

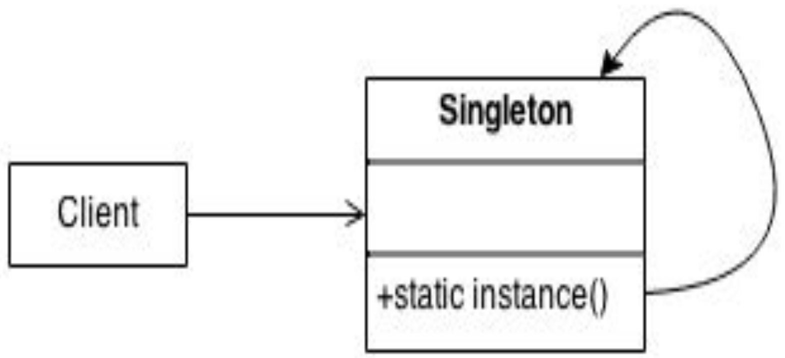
How to implement

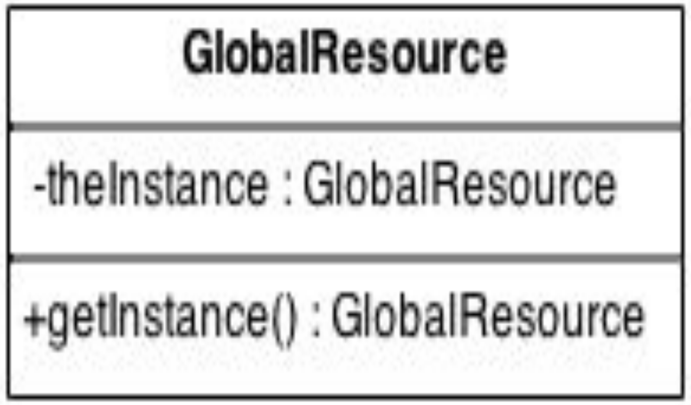
- A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.
- A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Applicability

- Use the Singleton pattern when
  - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure


Client
Singleton
+static instance()


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

Example

- One of the implementers of the *Logger* interface, the *FileLogger* class, logs incoming messages to the file *log.txt*.
- Having a singleton is helpful when there is only one physical instance of what the object represents. This is true in case of the *FileLogger* because there is only one physical log file. In an application, when different client objects try to log messages to the file, there could potentially be multiple instances of the *FileLogger* class in use by each of the client objects.
- This could lead to different issues due to the concurrent access to the same file by different objects.

Sample Code

```
package com.myjava.constructors;

public class MySingleTon {

    private static MySingleTon myObj;
    /**
     * Create private constructor
     */
    private MySingleTon(){

    }
/**
 * Create a static method to get instance.
 */
public static MySingleTon getInstance(){
    if(myObj == null){
        myObj = new MySingleTon();
    }
    return myObj;
}

public void getSomeThing(){
    // do something here
    System.out.println("I am here....");
}

public static void main(String a[]){
    MySingleTon st = MySingleTon.getInstance();
    st.getSomeThing();
}
}
```

Hints

- Make the Constructor Private
- Static Public Interface to Access an Instance

Related patterns

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- State objects are often Singletons.

Nton

- Means the application is provided with N-instances of a particular class.
- Each call to **getInstance** method is served with one of the available instances in round robin fashion.
- The number of instances required by the application is configurable.

Behavior and Advantages

- The number of instances required by the application should be configurable.
- The class should have a private constructor to avoid creation of external instances.
- It should have static synchronized method which returns one of the available instances in round robin fashion (Thread synchronized has to be taken care to avoid creation of duplicate instances)
- In case of service based Nton's, this provides a way of load balancing by delegating the responsibility of handling the request to one of the available and identical services.

Sample Code

```
import java.util.ArrayList;
import java.util.List;
public class Nton {
   private static final int NUM_OF_INSTANCES = 5;
   private static final List<Nton> instanceList = new Array
   List<Nton>();
   private static int instanceRequestCount = 0;
   private int instance Num;
   private Nton() {
   }
```

Sample Code

```
public static synchronized Nton getInstance() {
  Nton instance = null;
  if (instanceList.size() == NUM_OF_INSTANCES) {

  instance = instanceList.get(instanceRequestCount % NUM_O
  F_INSTANCES);}
  else
    instance = new Nton();
        instance.instanceNum = instanceList.size() + 1;
        instanceList.add(instance);
  }
  instanceRequestCount ++;
  return instance;
}
```

Sample Code

```
  public int getInstanceNum() {
    return instanceNum;
  }
  public String toString() {
    return "Nton : " + getInstanceNum();
  }
}
public class Test {
  public static void main (String [] args) {
    for (int i = 0 ; i < 20 ; i ++) {
            System.out.println(Nton.getInstance());
        }
  }
}
```