单例模式 你是我的唯一

单例模式应用场景举例：

“曾经沧海难为水，除却巫山不是云”，这句话用现在的语言解释就是“你是我的唯一”。GG 和 MM 都是初次恋爱，都把对方视为自己此生的唯一。而且 GG 和 MM 都在不断的向对方学习，不断的完善自己。GG 和 MM 的甜蜜和幸福很快就轰动了整个院系。男生一般都拿 GG 的女朋友教育自己的女朋友说别人怎么怎么样，而女生也经常拿 MM 的男朋友说男生该如何如何做。而且，年级辅导员还在年级会上表扬了 GG 和 MM，说男生都应该想 MM 的男朋友学习，女生都应该向 GG 的女朋友学习！呵呵，很显然，大家都知道，辅导员说 GG 的女朋友就是指 MM，而说 MM 的男朋友时就是指 GG。

单例模式解释：

GoF 对单例模式(Singleton Pattern)的定义是：保证一个类、只有一个实例存在，同时提供能对该实例加以访问的全局访问方法。

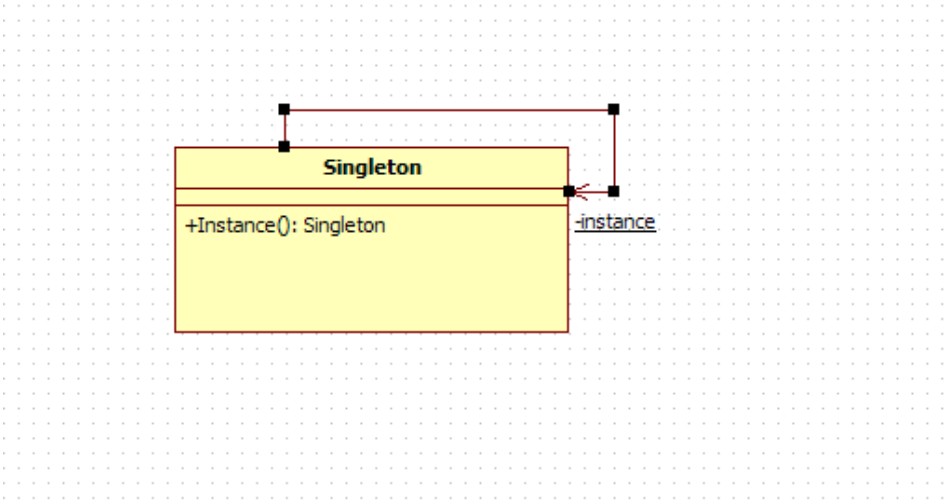
单例模式是一种对象创建型模式，使用单例模式，可以保证为一个类只生成唯一的实例对象。也就是说，在整个程序空间中，该类只存在一个实例对象。

单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

英文定义为：Ensure a class only has one instance, and provide a global point of access to it.

单例模式的 UML 图：

单例模式比较的单纯，其 UML 图如下所示：



单例模式深入分析：

单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。

单例模式适合于一个类只有一个实例的情况，比如窗口管理器，打印缓冲池和文件系统，它们都是原型的例子。典型的情况是，那些对象的类型被遍及一个软件系统的不同对象访问，

因此需要一个全局的访问指针，这便是众所周知的单例模式的应用。当然这只有在你确信你不再需要任何多于一个的实例的情况下

在计算机系统中，需要管理的资源包括软件外部资源，譬如每台计算机可以有若干个打印机，但只能有一个 Printer Spooler， 以避免两个打印作业同时输出到打印机中。每台计算机可以有若干传真卡，但是只应该有一个软件负责管理传真卡，以避免出现两份传真作业同时传到传真卡中的情况。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

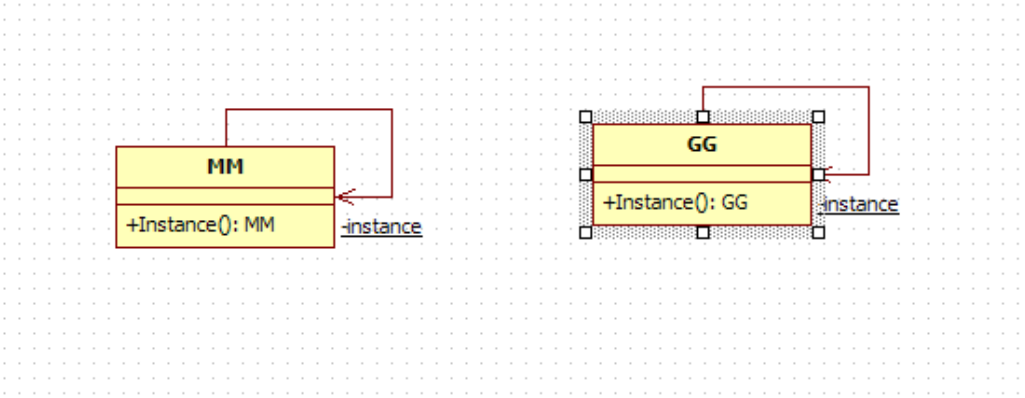
需要管理的资源也包括软件内部资源，譬如，大多数的软件都有一个（甚至多个）属性（properties）文件存放系统配置。这样的系统应当由一个对象来管理一个属性文件。

需要管理的软件内部资源也包括譬如负责记录网站来访人数的部件，记录软件系统内部事件、出错信息的部件，或是对系统的表现进行检查的部件等。这些部件都必须集中管理。

单例模式使用场景分析及代码实现：

在上面的使用场景中，无论是谁叫 GG 的女朋友，大家都知道只的是 MM；而相应的，无论是谁说 MM 的男朋友，大家都知道是 GG。GG 和 MM 分别都是对方单例 O(∩_∩)O 哈哈~

UML 模型图如下所示：



笔者在这里以 MM 的男朋友 GG 为例进行单例模式的说明。

GG 单例模式的第一个版本，采用的是“饿汉式”，也就是当类加载进来的就立即实例化 GG 对象，但是这种方式比较的消耗计算机资源。具体实现代码如下：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG单例模式的第一个版本 为"饿汉式"
 */
public class GGVersionOne {
    //在类被加载进入内存的时候就创建单一的GG对象
    public static final GGVersionOne gGVersionOne = new GGVersionOne();
    //名称属性
    private String name;
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionOne() {
    }

    //提供一个全局的静态方法
    public static GGVersionOne getGG() {
        return gGVersionOne;
    }
}
```

GG 单例模式的第二个版本：“懒汉式”，在单线程下能够非常好的工作，但是在多线程下存在线程安全问题，具体代码如下：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG单例模式的第二个版本 采用"懒汉式" 在需要使用的时候才实例化GG
 */
public class GGVersionTwo {
    //GG的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionTwo gGVersionTwo;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionTwo() {
    }
```

```
    //提供一个全局的静态方法
    public static GGVersionTwo getGG() {
        if(gGVersionTwo == null) {
            gGVersionTwo = new GGVersionTwo();
        }
        return gGVersionTwo;
    }
}
```

GG 单例模式的第三个版本，为解决多线程问题，采用了对函数进行同步的方式，但是比较浪费资源，因为每次都要进行同步检查，而实际中真正需要检查只是第一次实例化的时候，具体代码如下所示：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG单例模式的第三个版本 对函数进行同步
 */
public class GGVersionThree {
    //GG的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionThree gGVersionThree;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionThree() {
    }

    //提供一个全局的静态方法，使用同步方法
    public static synchronized GGVersionThree getGG() {
        if(gGVersionThree == null) {
            gGVersionThree = new GGVersionThree();
        }
        return gGVersionThree;
    }
}
```

GG 单例模式第四个版本，既解决了“懒汉式的”多线程问题，又解决了资源浪费的现象，看上去是一种不错的选择，具体代码如下所示：

```
package com.diermeng.designPattern.Singleton;
/*
 * GG单例模式的第四个版本，既解决了“懒汉式的”多线程问题，又解决了资源浪费的现象，看上去是一种不错的选择
 */
public class GGVersionFour {
    //GG的姓名
    private String name;
    //对单例本身引用的名称
    private static GGVersionFour gGVersionFour;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    //构造函数私有化
    private GGVersionFour() {
    }

    //提供一个全局的静态方法
    public static GGVersionFour getGG() {
        if(gGVersionFour == null) {
            synchronized (GGVersionFour.class) {
                if(gGVersionFour == null) {
                    gGVersionFour = new GGVersionFour();
                }
            }

        }
        return gGVersionFour;
    }
}
```

最后我们建立测试客户端测试一下版本四：

```
package com.diermeng.designPattern.Singleton.client;
import com.diermeng.designPattern.Singleton.GGVersionFour;

/*
 * 测试客户端
 */
public class SingletonTest {
    public static void main(String[] args) {
        //实例化
        GGVersionFour gG1 = GGVersionFour.getGG();
        GGVersionFour gG2 = GGVersionFour.getGG();
        //设值
        gG1.setName("GGAlias");
        gG2.setName("GG");

        System.out.println(gG1.getName());
        System.out.println(gG2.getName());


    }
}
```

输出的结果如下：

```
GG
GG
```

单例模式的优缺点分析：

优点：客户端使用单例模式类的实例的时候，只需要调用一个单一的方法即可生成一个唯一的实例，有利于节约资源。

缺点：首先单例模式很难实现序列化，这就导致采用单例模式的类很难被持久化，当然也很难通过网络传输；其次由于单例采用静态方法，无法在继承结构中使用。最后如果在分布式集群的环境中存在多个 Java 虚拟机的情况下，具体确定哪个单例在运行也是很困难的事情。

单例模式的实际应用简介：

单例模式一般会出现在以下情况下：

在多个线程之间，比如 servlet 环境，共享同一个资源或者操作同一个对象

在整个程序空间使用全局变量，共享资源

大规模系统中，为了性能的考虑，需要节省对象的创建时间等等。

温馨提示：

细心的读者可能会发现，笔者在写单例模式的双重检查方式的使用了“看上去是一种不错的选择”之语，之所以样说，是因为：Java 的线程工作顺序是不确定的，这就会导致在多线程的情况没有实例化就使用的现象，进而导致程序崩溃。不过双重检查在 C 语言中并没有问题。因为大师说：双重检查对 Java 语言并不是成立的。尽管如此，双重检查仍然不失为解决多线程情况下单例模式的一种理想的方案。