

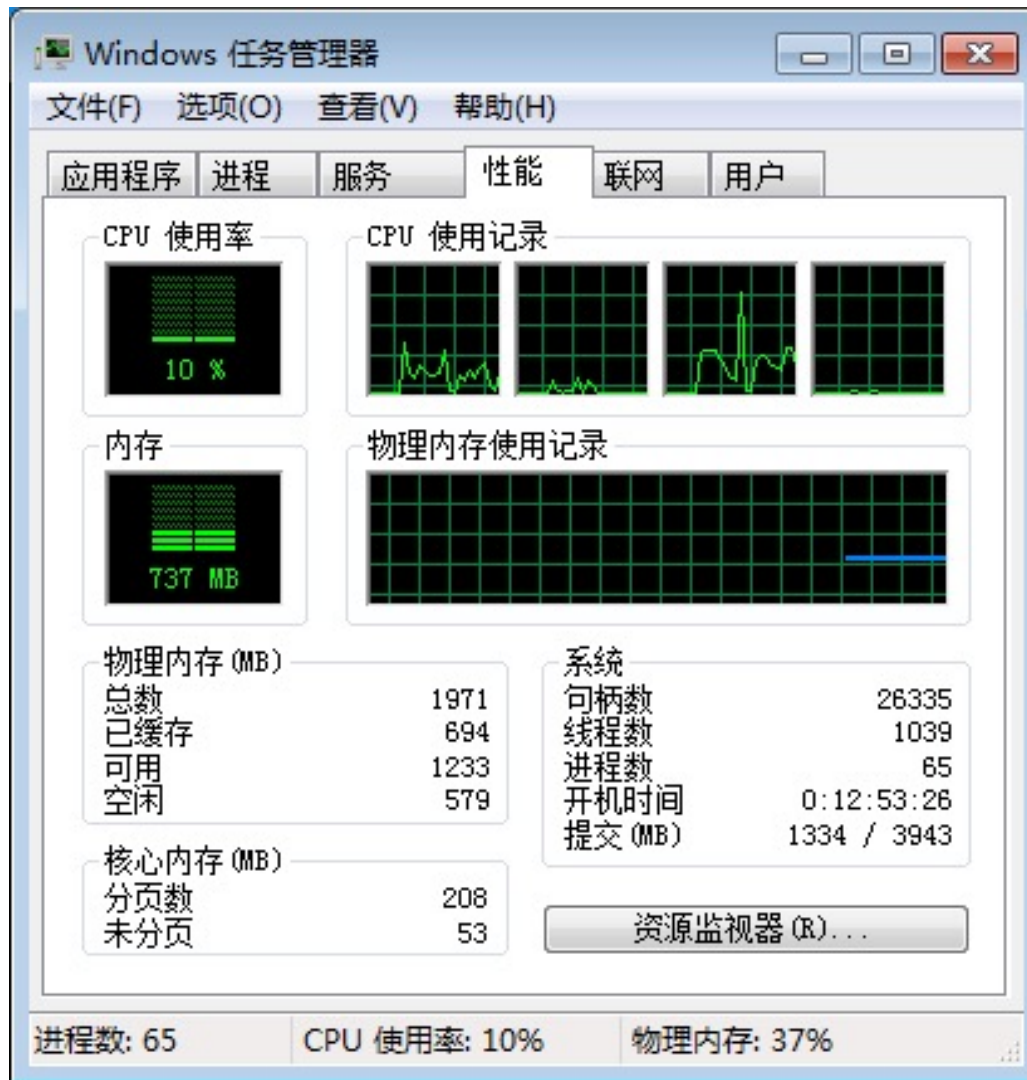


南京大學

# 设计模式-创建型模式(三)

## Design Pattern-Creational Pattern (3)

# 单例模式概述



Windows任务管理器

在正常情况下只能打开  
唯一一个任务管理器！

# 单例模式概述

- 如何保证一个类只有一个实例并且这个实例易于被访问？
  - (1) 全局变量：可以确保对象随时都可以被访问，但不能防止创建多个对象
  - (2) 让类自身负责创建和保存它的唯一实例，并保证不能创建其他实例，它还提供一个访问该实例的方法

单例模式

# 单例模式概述

- 单例模式的定义

单例模式：确保一个类**只有一个实例**，并提供一个**全局访问点**来访问这个唯一实例。

**Singleton Pattern:** Ensure a class has **only one instance**, and provide **a global point** of access to it.

- 对象创建型模式



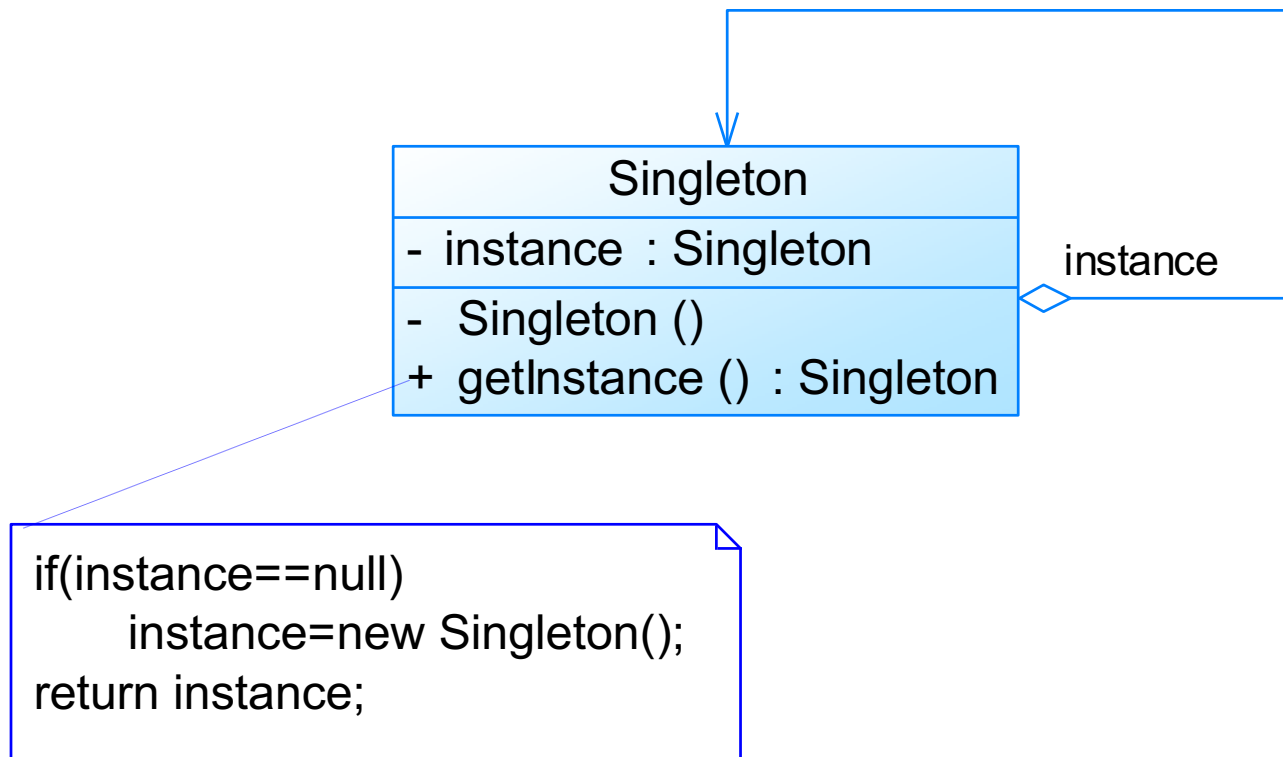
**Only one!**

# 单例模式概述

- 单例模式的定义
  - 要点：
    - 某个类只能有一个实例
    - 必须自行创建这个实例
    - 必须自行向整个系统提供这个实例

# 单例模式的结构与实现

- 单例模式的结构



# 单例模式的结构与实现

- 单例模式的结构
  - 单例模式只包含一个单例角色：
    - Singleton（单例）

# 单例模式的结构与实现

- 单例模式的实现
  - 私有构造函数

```
public class Singleton {  
    private static Singleton instance=null; //静态私有成员变量  
  
    //私有构造函数  
    private Singleton() {  
    }  
  
    //静态公有工厂方法，返回唯一实例  
    public static Singleton getInstance() {  
        if(instance==null)  
            instance=new Singleton();  
        return instance;  
    }  
}
```



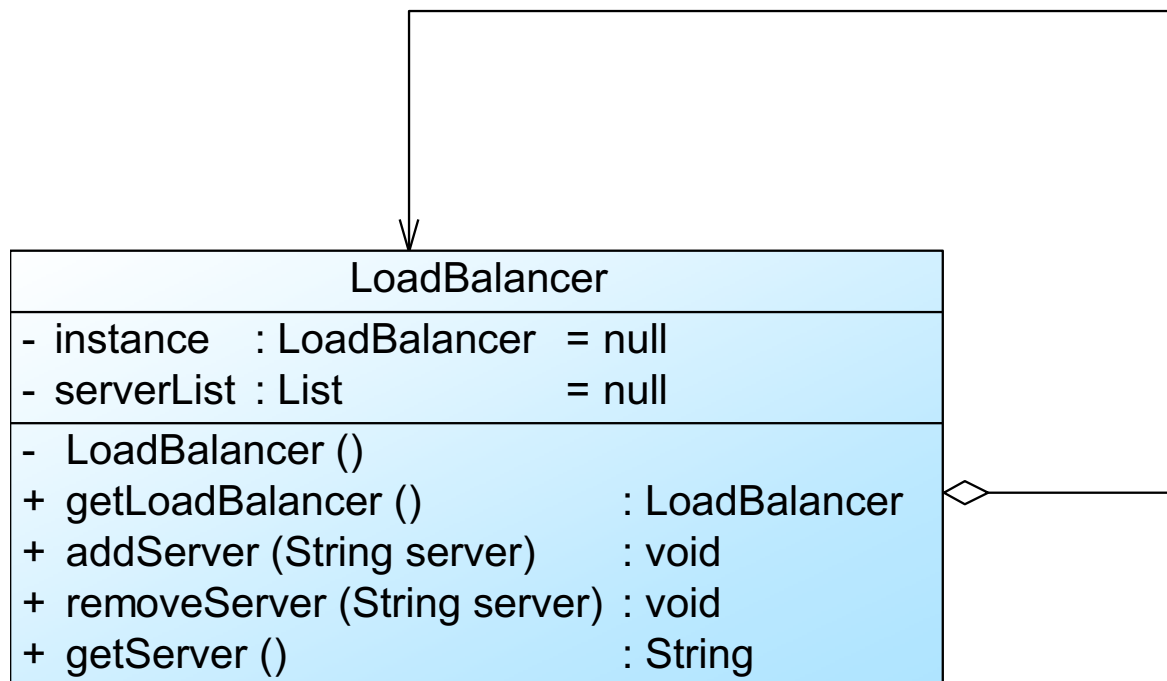
# 单例模式的应用实例

- 实例说明

某软件公司承接了一个服务器负载均衡(Load Balance)软件的开发工作, 该软件运行在一台负载均衡服务器上, 可以将并发访问和数据流量分发到服务器集群中的多台设备上, 进行并发处理, 提高了系统的整体处理能力, 缩短了响应时间。由于集群中的服务器需要动态删减, 且客户端请求需要统一分发, 因此需要确保负载均衡器的唯一性, 只能有一个负载均衡器来负责服务器的管理和请求的分发, 否则将会带来服务器状态的不一致以及请求分配冲突等问题。如何确保负载均衡器的唯一性是该软件成功的关键, 试使用单例模式设计服务器负载均衡器。

# 单例模式的应用实例

- 实例类图



服务器负载均衡器结构图

# 单例模式的应用实例

- 实例代码

- (1) LoadBalancer : 负载均衡器类, 充当单例角色
- (2) Client : 客户端测试类

演示.....

Code (designpatterns.singleton)

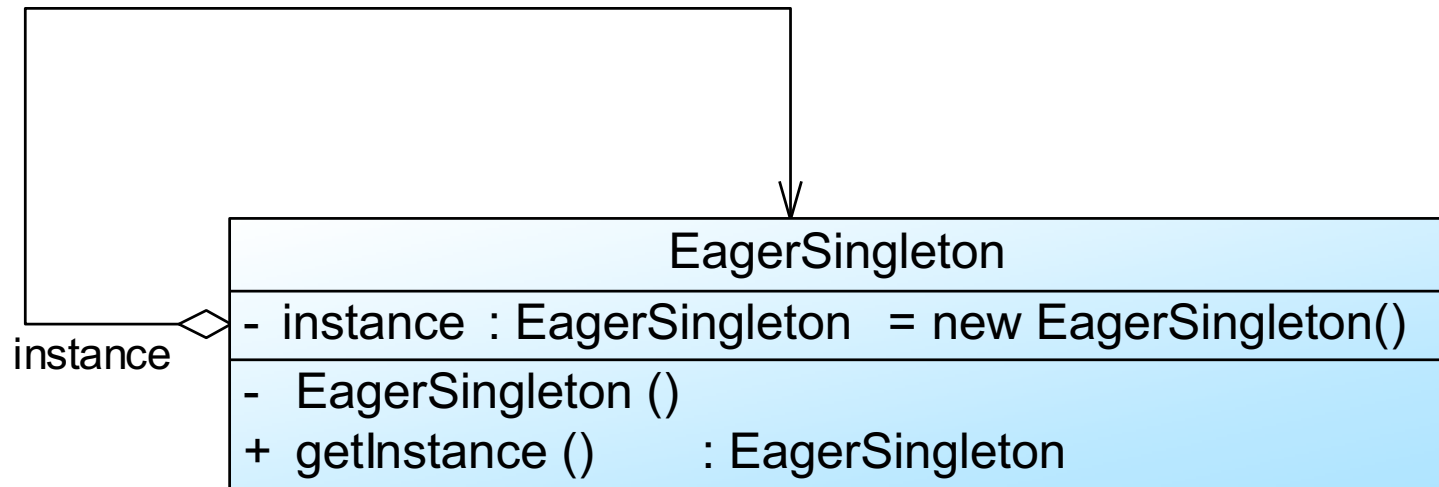
# 单例模式的应用实例

- 结果及分析

```
//判断服务器负载均衡器是否相同
if (balancer1 == balancer2 && balancer2 == balancer3 && balancer3 ==
balancer4) {
    System.out.println("服务器负载均衡器具有唯一性！");
}
```

# 饿汉式单例与懒汉式单例

- 饿汉式单例类
  - 饿汉式单例类(Eager Singleton)



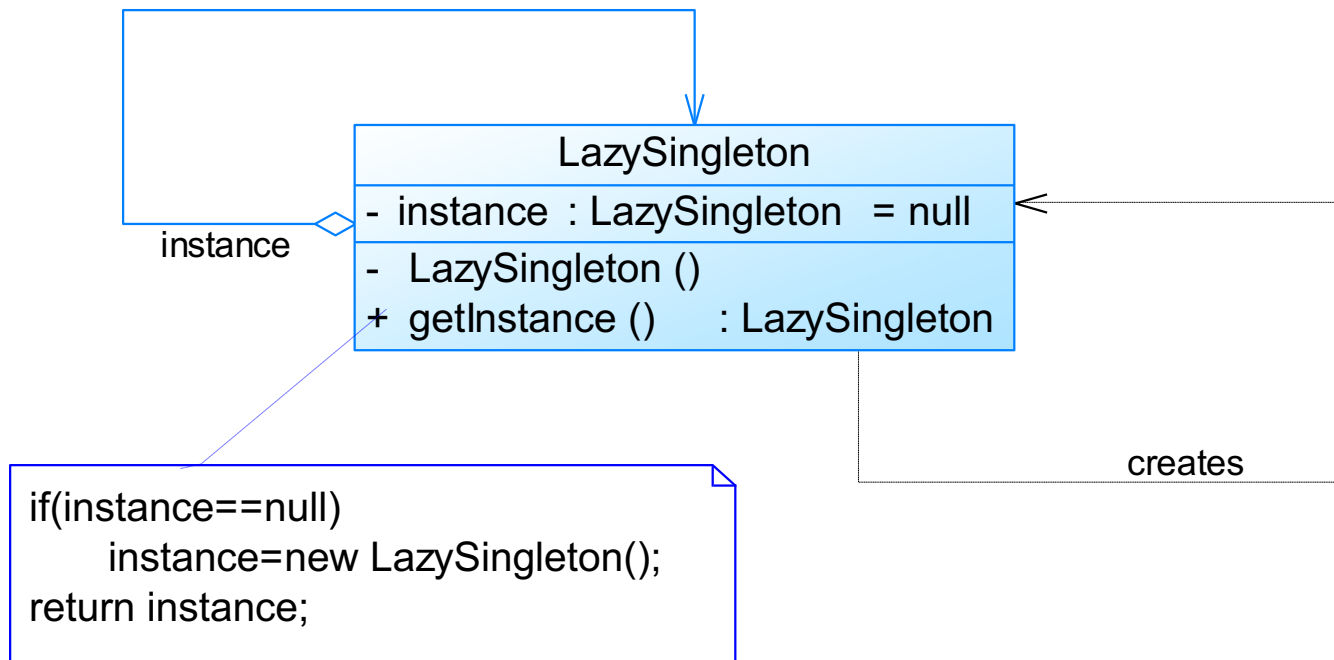
# 饿汉式单例与懒汉式单例

- 饿汉式单例类
  - 饿汉式单例类(Eager Singleton)

```
public class EagerSingleton {  
    private static final EagerSingleton instance = new EagerSingleton();  
    private EagerSingleton() { }  
  
    public static EagerSingleton getInstance() {  
        return instance;  
    }  
}
```

# 饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
  - 懒汉式单例类(Lazy Singleton)



# 饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
  - 延迟加载

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
  
    private LazySingleton() {}  
  
    public static LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

多个线程同时访问将导致  
创建多个单例对象！怎么  
办？



需要较长时间

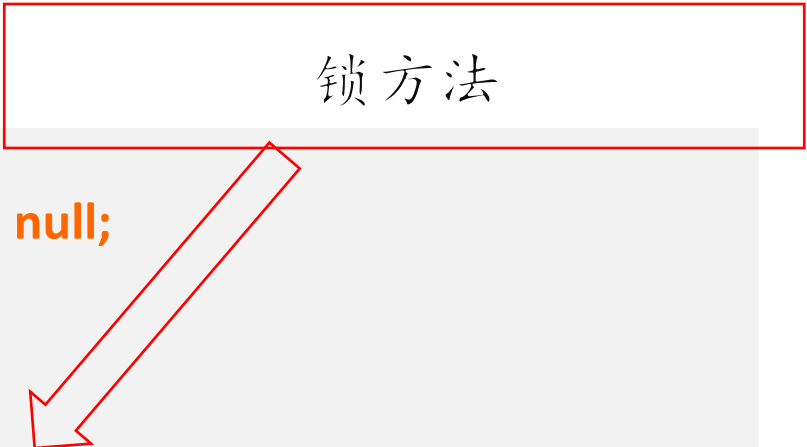


# 饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
  - 延迟加载

锁方法

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    synchronized public static LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```



# 饿汉式单例与懒汉式单例

- 懒汉式单例类与双重检查锁定
  - 延迟加载

锁代码段

```
.....  
public static LazySingleton getInstance() {  
    if (instance == null) {  
        synchronized (LazySingleton.class) {  
            instance = new LazySingleton();  
        }  
    }  
    return instance;  
}  
.....
```

# 饿汉式单例与懒汉式单例

```
public class LazySingleton {  
    private volatile static LazySingleton instance = null;  
  
    private LazySingleton() { }  
  
    public static LazySingleton getInstance() {  
        //第一重判断  
        if (instance == null) {  
            //锁定代码块  
            synchronized (LazySingleton.class) {  
                //第二重判断  
                if (instance == null) {  
                    instance = new LazySingleton(); //创建单例实例  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Double-Check Locking

双重检查锁定



# 饿汉式单例与懒汉式单例

- 饿汉式单例类与懒汉式单例类的比较
  - **饿汉式单例类**：无须考虑多个线程同时访问的问题；调用速度和反应时间优于懒汉式单例；资源利用效率不及懒汉式单例；系统加载时间可能会比较长
  - **懒汉式单例类**：实现了延迟加载；必须处理好多个线程同时访问的问题；需通过双重检查锁定等机制进行控制，将导致系统性能受到一定影响

# 饿汉式单例与懒汉式单例

- 使用静态内部类实现单例模式

```
//Initialization on Demand Holder
```

```
public class Singleton {  
    private Singleton() {  
    }  
}
```

```
//静态内部类
```

```
private static class HolderClass {  
    private final static Singleton instance = new Singleton();  
}
```

```
public static Singleton getInstance() {  
    return HolderClass.instance;  
}
```

```
public static void main(String args[]) {  
    Singleton s1, s2;  
    s1 = Singleton.getInstance();  
    s2 = Singleton.getInstance();  
    System.out.println(s1==s2);  
}
```

当getInstance()被调用，  
HolderClass被加载，静态  
对象instance真正被创建

# 单例模式的优缺点与适用环境

- 模式优点
  - 提供了对唯一实例的受控访问
  - 可以节约系统资源，提高系统的性能
  - 允许可变数目的实例（多例类）

# 单例模式的优缺点与适用环境

- 模式缺点
  - 扩展困难（缺少抽象层）
  - 单例类的职责过重、一定程度上违背单一职责原则
  - 由于自动垃圾回收机制，可能会导致共享的单例对象的状态丢失

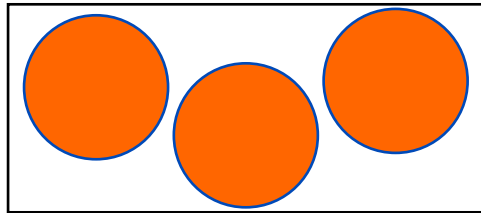
# 单例模式的优缺点与适用环境

- 模式适用环境
  - 系统只需要一个实例对象，或者因为资源消耗太大而只允许创建一个对象
  - 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例



# 思考

- 设计一个多例类(Multiton)，用户可以自行指定实例个数。



Multiton
- array : Multiton[]
- Multiton ()
+ getInstance () : Multiton
+ random () : int

- 目标
  - 另外一种获取对象单一性的方法
- 动机
  - 无论创建了多少Monostate的实例，它们都表现得像一个对象一样，甚至把当前的所有实例都销毁或者解除职责，也不会丢失数据。
- 解决方式/实现
  - 将所有变量都设置为静态的

# 例子

```
public class Monostate{  
    private static int itsX=0;  
    public Monostate(){}  
    public void setX(int x){ itsX = x;}  
    public int getX() {return itsX;}  
}
```

Monostate
<u>-state</u>
+setState(in s) +getState()

- 优点

- 透明性：使用Monostate对象和使用常规对象没有什么区别，使用者不需要知道对象是monostate
- 可派生性：派生类都是Monostate
- 多态性：由于方法不是静态的，所以可以在派生类中override。因此不同的派生类可以基于同样的静态变量表现出不同的行为

- 缺点

- 不可转换性：不能透过派生把常规类转换成Monostate类
- 内存占用：即使从未使用Monostate，它的变量也要占用内存空间

- Singleton模式使用时由构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制
- Monostate模式只是简单地把对象的所有变量变成静态的。
- 如果希望透过派生去约束一个现存类，并且不介意它的调用者都必须调用instance()方法来获取访问权，那么Singleton是最合适的。
- 如果希望类的单一性本质对使用者透明，或者希望使用单一对象的多态派生对象，那么Monostate是最适合的。

# 总结：Creational Patterns

30

- Simple Factory
  - 本质：由一个工厂对象决定创建出哪一种产品类的实例
- Factory Method
  - 本质：用一个virtual method完成创建过程
- Abstract Factory
  - 一个product族的factory method构成了一个factory接口
- Prototype
  - 通过product原型来构造product，Clone + prototype manager
- Builder
  - 通过一个构造算法和builder接口把构造过程与客户隔离开
- Singleton
  - 单实例类型。由构造函数，一个静态变量，以及一个静态方法对实例化进行控制和限制

- 创建型模式抽象了实例化的过程
  - 类创建模式使用继承改变被实例化的类
  - 对象创建模式将实例化委派给其它对象
- 创建型模式中不断出现的主题
  - 封装了系统所使用的具体类的信息
  - 隐藏了如何创建和组成这些具体类的实例
- 了解每一种模式的实质
  - 具体实现的时候可能会有变化情况，或者扩展，或者退化

# 总结：Creational Patterns

32

- Factory Method是基础，Abstract Factory是它的扩展
- Factory Method、Abstract Factory、Prototype都涉及到类层次结构中对象的创建过程，有所取舍
  - Prototype需要Prototype Manager
  - Factory Method需要依附一个Creator类
  - Abstract Factory需要一个平行的类层次
  - 根据应用的其他需求，以及语言提供的便利来决定使用哪种模式



# 总结：Creational Patterns

33

- Builder往往适合于特定的结构需要，它所针对的product比较复杂
- Singleton有比较强烈的物理意义，可以用在许多细微的地方，不一定与类层次关联
- 这些patterns都很常见，有时需要结合两种或者多种模式完成系统中对象的构造过程

- java.lang.Math类和java.lang.StrictMath类是否是单例模式？
- 单例模式课上讲了三种实现方式，分别是饿汉式，懒汉式，以及静态内部类实现方式：
  - 1. 对于懒汉式实现方式，请列举可以保证线程安全的两种方法；
  - 2. 请比较三种实现方式的差异；
  - 3. 请分析上述三种实现方式下，单例实例进入内存空间的时间点。
- 请尝试设计实现一个多例类(Multiton)，用户可以自行指定实例个数，给出实现代码以及设计思路。

提交作业到教学立方（4月14号24点截止）