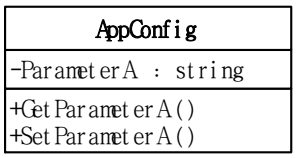
实验一 单例模式的应用

1 实验目的

1) 掌握单例模式（Singleton）的特点
2) 分析具体问题，使用单例模式进行设计。

2 实验内容和要求

很多应用项目都有配置文件，这些配置文件里面定义一些应用需要的参数数据。



通常客户端使用这个类是通过 new 一个 AppConfig 的实例来得到一个操作配置文件内容的对象。如果在系统运行中，有很多地方都需要使用配置文件的内容，系统中会同时存在多份配置文件的内容，这会严重浪费**内存资源**。事实上，对于 AppConfig 类，在运行期间，**只需要一个对象实例**就够了。那么应该怎么实现呢？用 C#或 Java 控制台应用程序实现该单例模式。绘制该模式的 UML 图。

3 UML 图

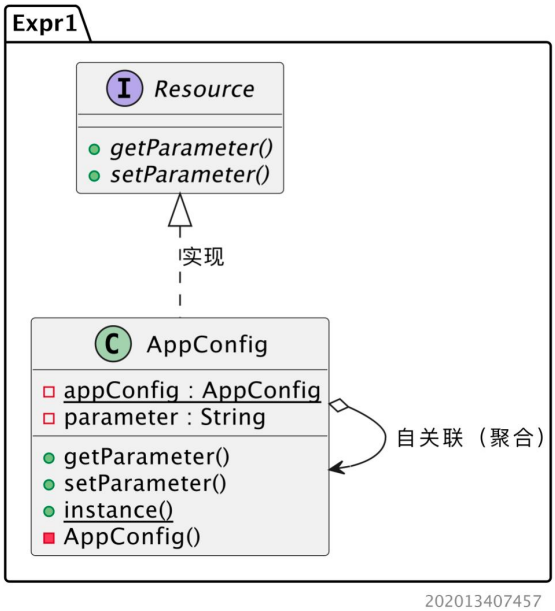


图 1.3.1 单例模式类图

4 代码实现

```
public class SingletonPattern {
    public static <T> void show(Resource<T> r) {
        T val = r.getParameter();
        System.out.println(val);
    }

    public static <T> void put(Resource<T> r, T val) {
        r.setParameter(val);
    }

    public static void main(String[] args) {
        System.out.println("Inside main()");
        Resource<String> r = AppConfig.instance();
        Resource<String> r2 = AppConfig.instance();
        show(r);
        put(r2, "customized config");
        show(r);
    }
}
```

```
public interface Resource<T> {
    T getParameter();
    void setParameter(T t);
}
```

```
public class AppConfig implements Resource<String> {
    private static AppConfig appConfig = new AppConfig();
    private String parameter = "default config";

    private AppConfig() {
        System.out.println("AppConfig()");
    }

    public static AppConfig instance() {
        return appConfig;
    }
```

```
//承上
@Override
    public synchronized String getParameter() {
        return parameter;
    }

    @Override
    public synchronized void setParameter(String
parameter) {
        this.parameter = parameter;
    }
}
```

测试结果

```
Inside main()
AppConfig()
default config
customized config
```

5 要点总结

1. 构造函数应该是**私有**的，以保证其不会在别处创建，破坏单例要求；
2. 获取对象函数 instance()应是**公有静态**函数，符合单例模式的思想；
3. 同时，自关联的对象应该是静态成员变量，由于 JVM 的工作方式，饿汉式单例类可以保证在类实例化时同时初始化成员变量；
4. 这样的方式是线程安全的，由于多线程可能同时对共享资源进行读取，所以 get 和 set 函数应当冠以 synchronized。

实验二 工厂模式的应用

1 实验目的

1) 掌握工厂模式（Factory）的特点
2) 分析具体问题，使用工厂模式进行设计。

2 实验内容和要求

有一个 OEM 制造商代理做 HP 笔记本电脑(Laptop), 后来该制造商得到了更多的品牌笔记本电脑的订单 Acer，Lenovo，Dell，该 OEM 商发现，如果一次同时做很多个牌子的笔记本，有些不利于管理。利用工厂模式改善设计，用 C#或 Java 控制台应用程序实现该 OEM 制造商的工厂模式。绘制该模式的 UML 图。

3 UML 图

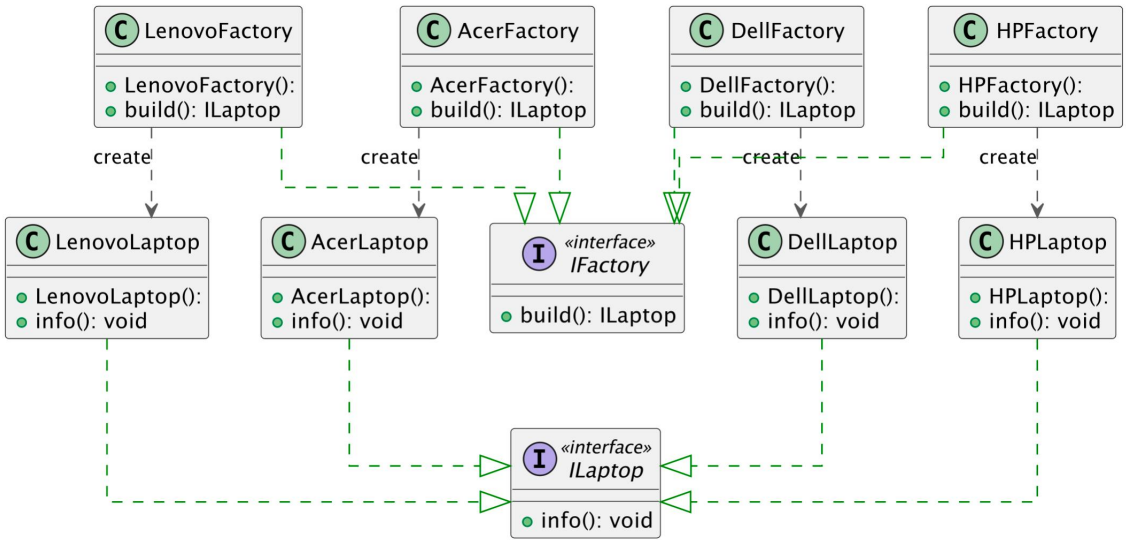


图 2.3.1 单例模式类图

4 代码实现

```
public interface ILaptop {
    void info();
}

public interface IFactory {
     ILaptop build();
}
```

```java
public class LenovoLaptop implements ILaptop {
    @Override
    public void info() {
        System.out.println("Lenovo Laptop");
    }
}

public class HPLaptop implements ILaptop {
    @Override
    public void info() {
        System.out.println("HP Laptop");
    }
}

public class DellLaptop implements ILaptop {
    @Override
    public void info() {
        System.out.println("Dell Laptop");
    }
}

public class AcerLaptop implements ILaptop {
    @Override
    public void info() {
        System.out.println("Acer Laptop");
    }
}

public class LenovoFactory implements IFactory{
    @Override
    public ILaptop build(){
        System.out.println("Lenovo 工厂生产 Lenovo 笔记本");
        return new LenovoLaptop();
    }
}

public class HPFactory implements IFactory{
    @Override
    public ILaptop build(){
        System.out.println("HP 工厂生产 HP 笔记本");
        return new HPLaptop();
    }
}
```

```java
public class DellFactory implements IFactory {
    @Override
    public ILaptop build(){
        System.out.println("Dell 工厂生产 Dell 笔记本");
        return new DellLaptop();
    }
}

public class AcerFactory implements IFactory {
    @Override
    public ILaptop build(){
        System.out.println("Acer 工厂生产 Acer 笔记本");
        return new AcerLaptop();
    }
}

public class Main {
    public static void main(String[] args) {
        IFactory factory = new LenovoFactory();
        ILaptop laptop=factory.build();
        laptop.info();
    }
}
```

测试结果

```
Lenovo 工厂生产 Lenovo 笔记本
Lenovo Laptop
```

5 要点总结

1. 接口编程，以满足开放封闭原则，当添加新的品牌时，无需修改已有代码；
2. 委托工厂封装对象创建的逻辑，符合单一职责原则。