第一节课

面向的对象的六大原则

1. **单一职责原则**

   一个类只有一个引起它发生变化的原因。

2. **开闭原则**

   对扩展开放，对修改封闭。

3. **里氏替换原则**

   所有引用父类的地方都可以透明的使用其子类的对象。

4. **依赖倒置原则**

   模块间的依赖通过抽象发生，实现类之间的依赖通过接口或抽象类。

5. **接口隔离原则**

   类间的依赖关系应该建立在最小的接口上。

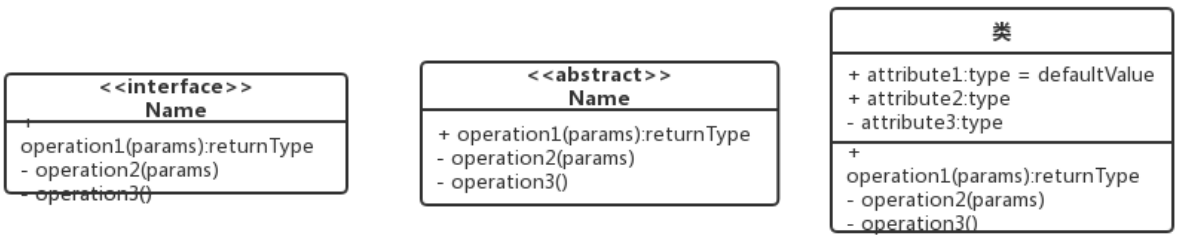
6. **迪米特原则**

   一个对象应该对其他对象有最少的了解。

UML图

[UML类图中箭头和线条的含义和用法]

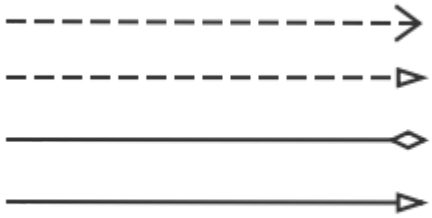
- **类图的表示方法**



- **访问权限修饰符**

  > - public `+`
  > - protected `#`
  > - package `~`
  > - private `-`

- **关系标识符**

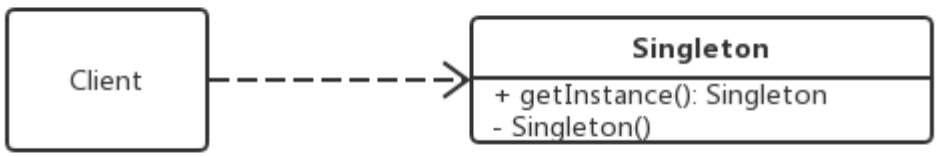


依赖	实现	聚合	继承
虚线箭头	虚线空心三角形	实线空心菱形	实线空心三角形
A离不开B	实现一个接口	A包含B，但B不是A的一部分	父子关系

常用设计模式

单例模式

零、单例模式UML类图



- Client 高层客户端
- Singleton 单例类

一、静态属性加载时机:

`.java` --> `.class` --> `类加载`

类的生命周期：`加载` --> `验证` --> `准备` --> `解析` --> `初始化` --> `使用` --> `卸载`

准备阶段： `static`变量设置初始值，使用的内存在方法区进行分配，真正的赋值到初始化阶段才会执行，`final`则在准备阶段赋值

初始化阶段： 使用`new`关键字实例化对象、读取或设置一个类的静态字段、调用一个类的静态方法

二、饿汉模式

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

    public static Singleton getInstance() {
        return ourInstance;
    }

    private Singleton() {
    }
}
```

由上面可知，饿汉模式的单例，在类加载的准备阶段已经实例化了对象

三、**懒汉模式**

```
public class LazySingleton {
    public static LazySingleton lazySingleton;

    private LazySingleton() {
    }

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

懒汉模式的单例，在初始化的时候才开始分配内存

四、**Double Check Lock (DCL)模式**

```
public class DCLSingleton {
    private static volatile DCLSingleton dclSingleton = null;
    private DCLSingleton() {
    }

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

`volatile`作用：

- 此变量对所有线程的可见性，一条线程修改了变量的值，其他线程可以立即得知
- 禁止指令重排序优化

```
Singleton instance = new Singleton()
    //1-2-3
    //1-3-2 DCL失效
```

- 给Singleton的实例分配内存
- 调用Singleton的构造函数，初始化成员字段
- instance变量指向Singleton实例在内存中的地址（instance不为空）

五、**静态内部类单例模式**

静态内部类加载

```
public class StaticInnerClassSingleton {

    private StaticInnerClassSingleton() {
    }

    public static StaticInnerClassSingleton getInstance() {
        return SingletonHolder.sInstance;
    }

    private static class SingletonHolder {
        private static final StaticInnerClassSingleton sInstance = new
StaticInnerClassSingleton();
    }
}
```

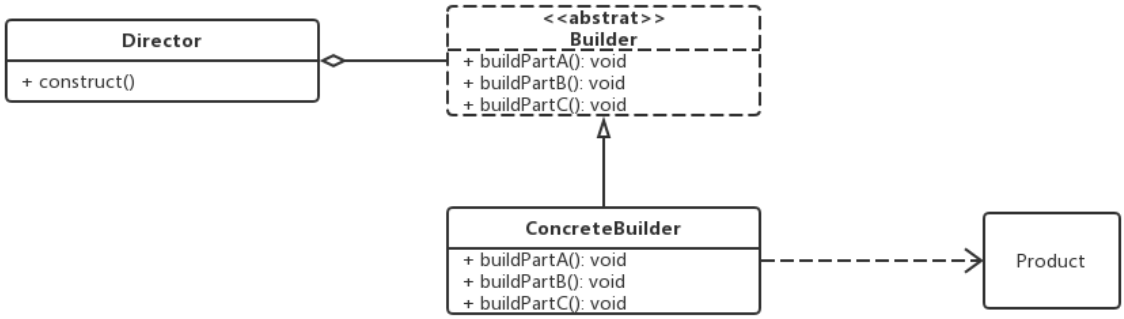
六、**其他** 枚举单例 使用容器实现单例

七、**单例模式总结**

- 构造函数为private
- 通过一个静态方法返回单例类对象
- 单例的对象有且只有一个，保证线程安全
- 反序列化时不会重新构建对象

Builder模式

零、**UML类图**



- Product 产品类
- Builder 抽象Builder类，规范产品的组建
- ConcreteBuilder 具体的Builder类
- Director 统一组装过程

一、链式调用

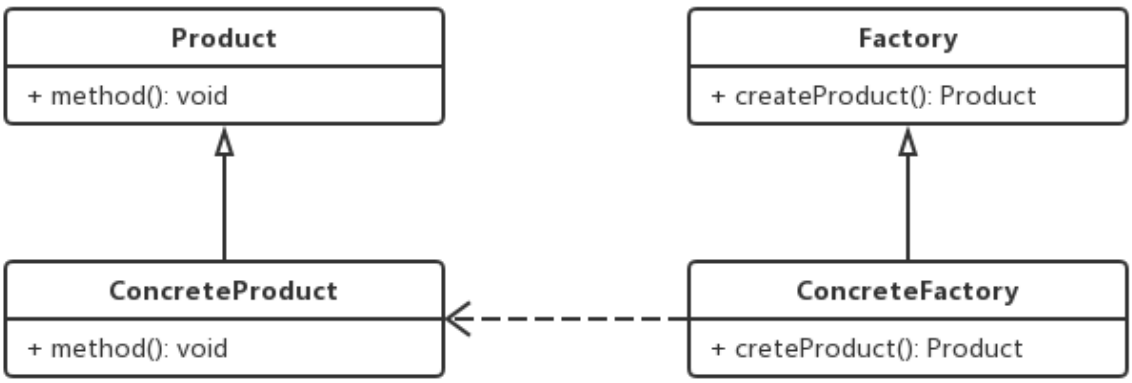
在每个`set`方法里面返回当前对象的实例 `return this`。

例子：**Android**里面的`Notification` `AlertDialog`

二、

工厂方法模式

> 定义一个用于创建对象的接口，让子类决定实例化哪个类

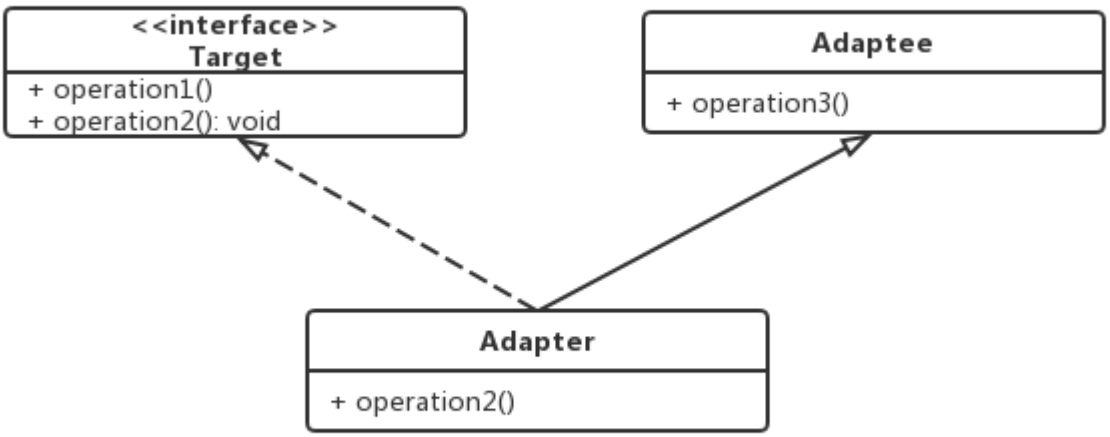


适配器模式

> 把一个类的接口变换成客户端期待的另外一种接口，使原本因接口不匹配而无法工作的两个类一起工作

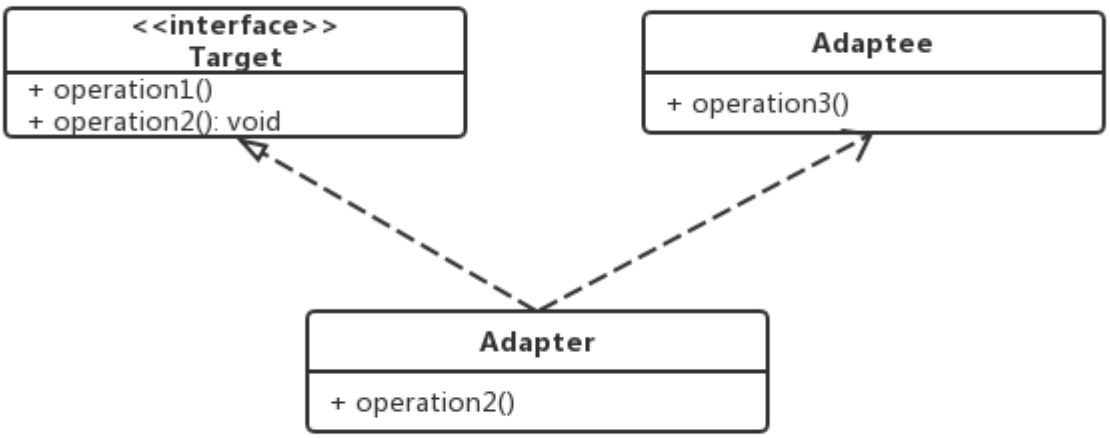
1. **类适配器**

> 通过实现Target接口以及继承Adaptee类来实现接口转换



2. **对象适配器**

> 把适配类的API转换成为目标类的API



让一只鸡发出鸭子的叫声

其他模式

1. 责任链模式，Android事件分发
2. 观察者模式，广播，消息机制
3. 代理模式，AIDL
4. ……

Java8新特性

java8新特性

接口的默认方法与静态方法

1. **之前的接口**
   - 接口里的方法自动为`public`、域为`public static final`
   - 实现该接口的时候，要实现所有的方法
   - 不能实现方法
2. **静态方法**

```
public interface A {

    static int add(int a, int b) {
        return a + b;
    }
}
```

3. **默认方法**

   为接口方法提供一个默认实现，使用`default`修饰符标记

   实现该接口的时候，默认方法可选。

Lambda表达式

参数、箭头、表达式

1. **函数式接口**

   只有一个抽象方法的接口，当需要这种接口的对象时，可以提供`lambda`表达式

2. **方法引用**

   有现成的方法可以完成其他代码的功能