Lesson9
完善类的设计_2

主讲老师：申雪萍

主要内容

- 对象向上映射和向下映射
- Object类
- 操作符“==”与对象的equals()方法
- 内部类
- 匿名类
- **最终方法、最终类**
- 复杂软件的管理（package和import的使用）
- 高内聚、低耦合

讨论：（overview）

- 1、继承带来了哪些好处？
  2、继承是否破坏了类的封装性？

面向对象的三大特征之一：*继承真的很好*（overview）

一. 继承避免了公用代码的重复开发，减少代码的冗余，提高程序的复用性；

二. 支持多态（通过向上映射），提高程序的可扩展性；

三. 继承是类实现可重用性和可扩充性的关键持征。在继承关系下类之间组成网状的层次结构。

四. 通过继承增强一致性，从而减少模块间的接口和界面。

五. 通过向上映射，让我们体验到多态的益处。

继承是否破坏了类的封装性？ （overview）

- 是的。继承破坏了封装性，换句话说，子类依赖于父类的实现细节。**继承很容易改变父类实现的细节(所以父类中能写成final尽量写成final)，**即使父类整体没有问题，也有可能因为子类细节实现不当，而破坏父类的约束。
- 其实这是一个平衡关系，不是绝对关系，一定程度的封装和一定程度的继承，可以提高开发效率，继承破坏了封装，但是有时继承是必须的，为了继承牺牲一定的封装是允许的。不能绝对的为了封装，就不去继承。

最终方法，最终类的**必要性**

- 因为继承破坏了封装性，换句话说，子类依赖于父类的实现细节。**继承很容易改变父类实现的细节，**即使父类整体没有问题，也有可能因为子类细节实现不当，而破坏父类的约束。
- **所以父类中能写成final尽量写成final**

最终类

- 如果一个类没有必要再派生子类，通常可以用final关键字修饰，表明它是一个最终类。

```
1. Public final class Math{}
2. Public final class String{}
```

最终方法

- 最终方法：用关键字final修饰的方法称为最终方法。

- 最终方法既不能被覆盖，也不能被重载，它是一个最终方法，其方法的定义永远不能改变

final方法和final类

- final类中的方法可以不声明为final方法，但实际上final类中的方法都是隐式的final方法
- final修饰的方法不一定要存在于final类中。
- **定义类头时，abstract和final不能同时使用**
- 访问权限为private的方法默认为final的

常量

一. 常量：在程序运行过程中，其值不变的量。

二. Java中的常量使用关键字final修饰。

三. final既可以修饰简单数据类型，也可以修饰复合数据类型。

四. final常量可以在声明的同时赋初值，也可以在构造函数中

五. 复合数据类型常量可以是Java类库定义的复合数据类型，也可以是用户自定义的复合数据类型

常量声明格式

格式：final 数据类型 常量名=值

例如：final double PI=3.1415926;

final String str=“Hello World”;

注意事项

一. 简单数据类型常量其值一旦确定，就不能被改变。

二. 复合数据类型常量指的是引用不能被改变，而其具体的值是可以改变的。

三. 常量既可以是局部常量，也可以是类常量和实例常量。如果是类常量，在数据类型前加static修饰（由所有对象共享）。如果是实例常量，就不加static修饰。

四. 常量名一般大写，多个单词之间用下划线连接。

示例代码

```
class PersonA {
    String name;
    String sex;
    int age;
    final static double PAI = 3.1415926; // 静态常量
    final double ID; // 常量，表示每一个人的id不同，但一旦赋值又是不能变化的

    public PersonA(String n, String s, int a, int id) {
        name = n;
        sex = s;
        age = a;
        ID = id; // 构造函数中赋值
    }

    public String toString() {
        String s =
    "姓名：" + name + "," + "性别：" + sex + "," + "年龄：" + age;
        return s;
    }
}
```

示例代码

```
public class FinalTest {
    public static void main(String args[]) {
        final double PAI = 3.1415926; // 局部常量

        // final既可以修饰简单数据类型，也可以修饰符和数据类型
        final PersonA p1 = new PersonA("Tom", "M", 23, 001);
        PersonA p2 = new PersonA("Mary", "F", 20, 002);
        System.out.println("final p1:" + p1.toString());

        // p1=p2 //对final对象重新赋值会产生编译错误
        // 以下对final对象中的成员变量，重新赋指是可以的
        p1.name = p2.name;
        p1.sex = p2.sex;
        p1.age = p2.age;
        System.out.println("final p1:" + p1.toString());
    }
}
```

小结：使用最终方法，最终类增强程序的鲁棒性

- 将方法或者类声明为final型可以有效防止他人覆写该函数，或者继承于该类。**但是或许更重要的是，这么做可以“关闭”动态绑定。**

- 或者说，这么做便是告诉编译器：动态绑定是不需要的。于是编译器可以产生效率较佳的程序代码。

思考题

```
package com.buaa.edu;
public class EduBackground {

    String primarySchool;
    String secondarySchool;
    String juniorHSchool;
    String seniorHSchool;
    String university;

    public EduBackground() {

    }
}
```

```java
package com.buaa.edu;
public class Person {
    private String name;
    private int age;
    private String gender;
    private final EduBackground edu = new EduBackground();
    public Person() {
    }
    // final修饰局部变量、修饰成员方法、修饰方法的参数
    // 修饰局部变量时，局部变量的值不能改变
    public void finalLocal() {
        final int i;
        final EduBackground edu = new EduBackground();
        i = 1;
        System.out.println("finalLocal: i = " + i);
    }
    // 修饰方法的参数时(简单数据类型)，参数i不能被修改
    public void finalArgs(final int i) {
        // i = 3;
        System.out.println("finalArgs: i = " + i);
    }
    // 修饰方法的参数时(复合数据类型)，不能指向新的位置
    public void finalArgs(final EduBackground edu) {
        // edu = new EduBackground();
        System.out.println("finalArgs: edu");
    }

    // 修饰成员方法时，成员方法不能被子类重写
    public final void finalMethod() {
        int i = 2;
        System.out.println("finalMethod: i = " + i);
    }

    private final void priFinalMethod() {
        System.out.println("Person:priFinalMethod");
    }

    public static void main(String[] args) {
        Person per = new Person();
        Student stu = new Student();
        Person per1 = stu;

        per.priFinalMethod();
        stu.priFinalMethod();
        per1.priFinalMethod();
    }
}
```

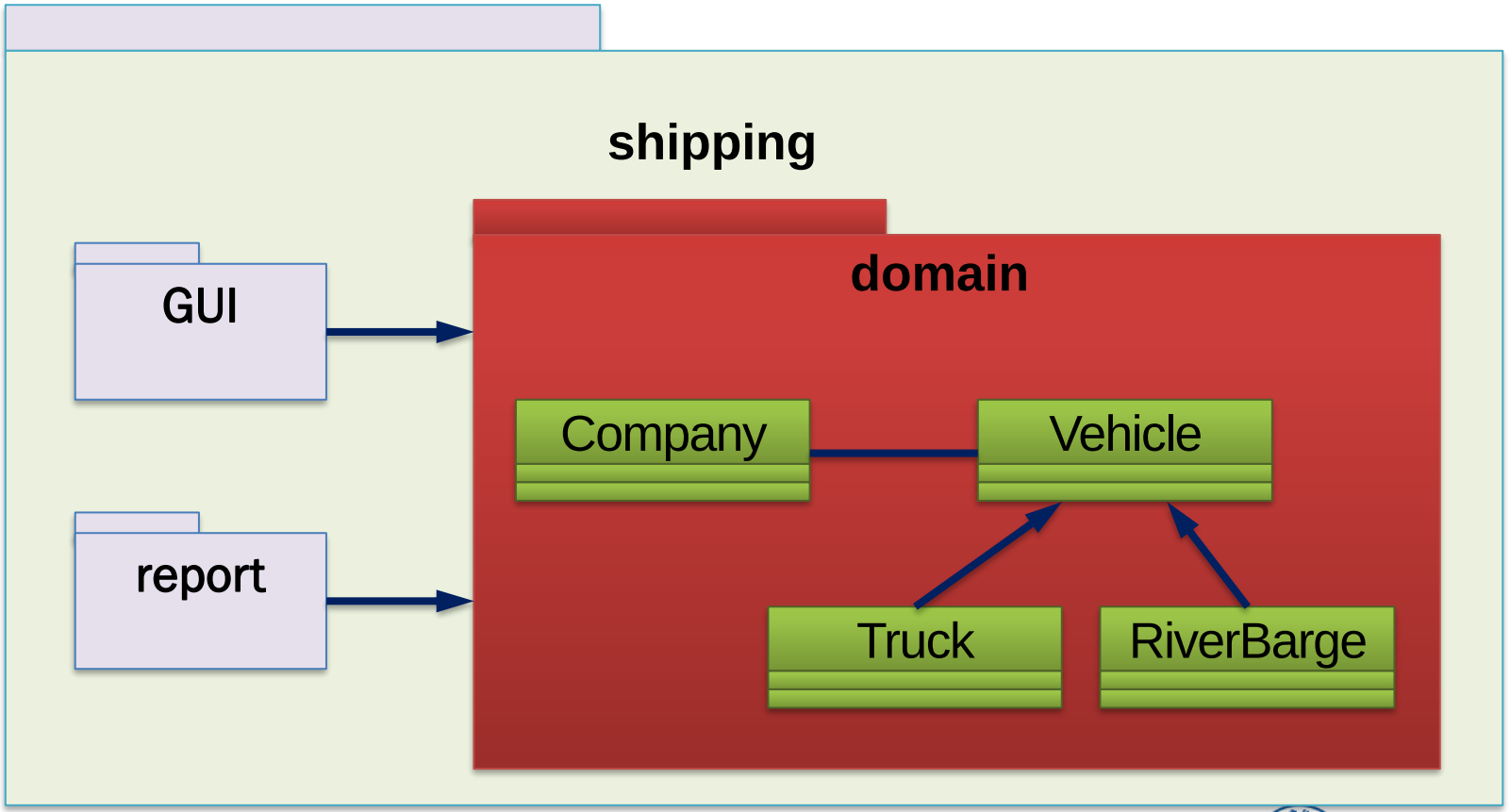
```java
package com.buaa.edu;
public class Student extends Person {
    private final int stuNumber;
    private int score;
    private static final int BAN_JI=20210001;
    public Student() {
        stuNumber=(int)Math.random()*500;
        score=(int)Math.random()*100;
    }
//子类不能重写父类被final修饰的方法
//  public final void finalMethod() {
//          int i = 2;
//          System.out.println("finalMethod: i = " + i);
//  }
    public final void priFinalMethod() {
        System.out.println("Student:priFinalMethod");
    }
}
```

主要内容

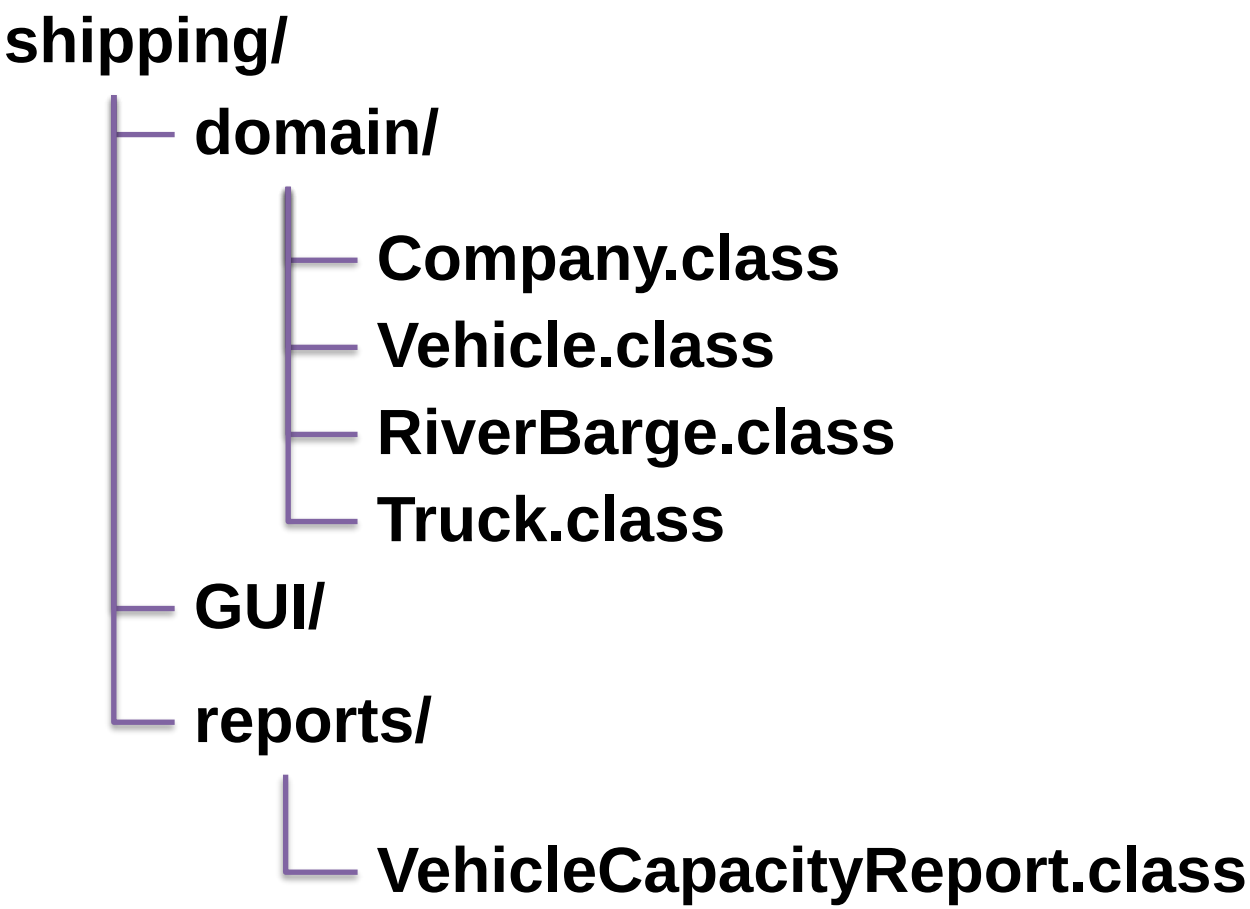
- 对象向上映射和向下映射（Upcasting和downcasting）
- 操作符“==”与对象的equals()方法
- 内部类
- 匿名类
- 最终方法、最终类
- **复杂软件的管理（package和import的使用）**
- 高内聚、低耦合
- 代码复用

软件(software)和包(package)

- 包帮我们管理大型的软件系统
- 包可以包含类、接口和子包



软件(software)和包(package)



Java包的理解

- Java通过关键字package创建包
- 包（package）是类和接口的集合
- Java用包来管理名称空间。包消除了不同组的类中潜在的类名冲突问题

Java包的理解

- Java里的每一个类都属于一个特定的包
- “包是类的容器”
- 包是有层次的

java.awt.Color

Java包的理解

- 包定义语句必须是Java源文件中的第一条可执行语句
- 在默认的情况下，系统会为每一个 .java 源文件创建一个无名包
- 无名包不能被其他包所引用，为解决这个问题，需创建有名子的包

包的引用

- 包的引用是通过关键字 import 实现的

import 包名[.下一级包名[…]].引用的类名;

```
import mylib.Person;
```

import 包名[.下一级包名[…]].*;

```
import mylib.*;
```

直接使用名称:

```
public class Student extends mylib.Person
```

包的引用

- Java 中 的 类 库 被 包 含 在 一 个 叫 做 java/javax 的包里
- 默 认 的 情 况 下 ， Java 类 只 能 访 问 到 在 java.lang的类和接口
- 要是用来自其他包里的类，可以通过包的名字来引用，或者可以把它们导入到源文件里

Java的系统程序包

Java 常用类库	
API 包	功能
java.lang	包含Java语言的核心类库
java.util	提供各种实用工具类
java.io	标准输入/输出类
java.net	实现Java网络功能的类库

Java的系统程序包

Java 常用类库	
java.awt	包含了Java图形界面中常用的类和接口
java.applet	提供对通用Applet支持的类和接口
java.security	支持Java程序安全性的类和接口

小结

- 从用户的角度看，Java源程序中的类分为两种
  - 系统定义的类，即Java类库
  - 用户自己定义的类
  - 作为软件工程师，我们首先应该了解系统定义的类或者开源类能否支持，如果不能再用户自己定义的类
- **Java的类可组织在包（package）中**

主要内容

- 对象向上映射和向下映射（Upcasting和downcasting）
- 操作符“==”与对象的equals()方法
- 内部类
- 匿名类
- 最终方法、最终类
- **复杂软件的管理（package和import的使用）**
- **高内聚、低耦合**
- **代码复用**

模块，耦合，内聚

- 模块就是从逻辑上将系统分解为更细微的部分, 分而治之, 复杂问题拆解为若干简单问题, 逐个解决。**模块的粒度可大可小, 可以是函数, 类, 功能块等等**。

- 耦合主要描述模块之间的关系

- 内聚主要描述模块内部

低耦合

- 低耦合：是指软件系统中，模块与模块之间的直接依赖程度低。
- 模块之间存在依赖,模块独立性越差，耦合越强,导致一个模块的改动可能会影响到其他模块 。比如模块A直接操作了模块B中数据, 则视为强耦合, **若A只是通过数据与模块B交互, 则视为弱耦合**。
- **好处：**独立的模块便于扩展, 维护, 写单元测试, 如果模块之间重重依赖, 会极大降低开发效率，代码可维护性差。

高内聚

- **高内聚**：系统存在AB两个模块儿进行交互，如果修改了A模块儿不影响B模块的工作，那么认为A模块儿有足够的内聚。
- 一个模块应当尽可能独立完成某个功能。模块内部的元素, 关联性越强, 则内聚越高, 模块单一性更强。
- 如果有各种场景需要被引入到当前模块, 代码质量将变得非常脆弱, 这种情况建议拆分为多个模块。
- **危害**：低内聚的模块代码, 不管是维护, 扩展还是重构都相当麻烦, 难以下手。

接口设计原则

- 很多设计模式, 框架都是基于高内聚低耦合进行设计，好的接口应当满足设计模式六大原则：
  - 单一职责原则: 一个类只负责一个功能领域中的相应职责。
  - 开闭原则: 一个软件实体应当对扩展开放，对修改关闭。
  - 里氏代换原则: 所有引用基类（父类）的地方必须能透明地使用其子类的对象。

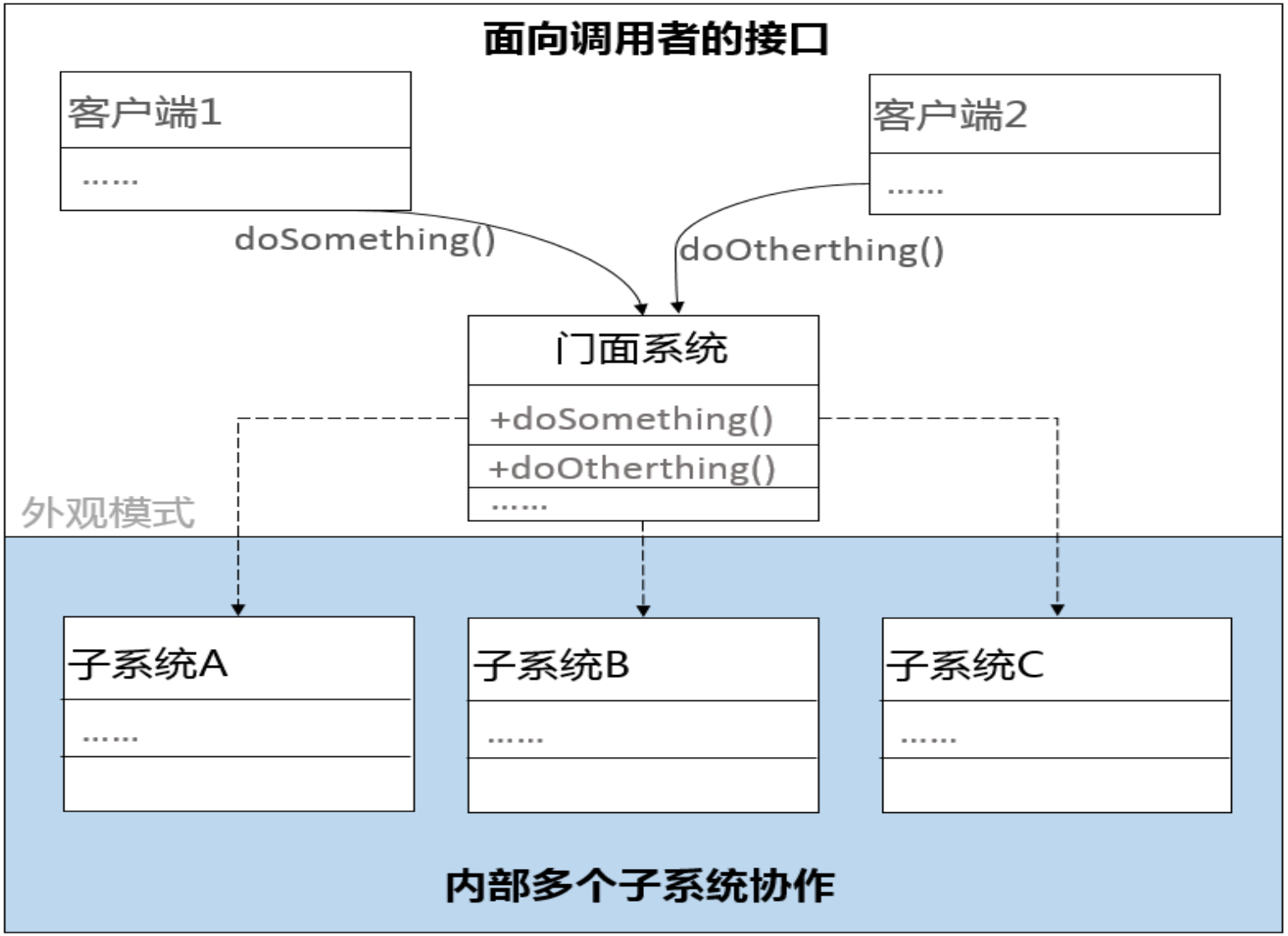
接口设计原则

- 依赖倒转原则: 抽象不应该依赖于细节, 细节应当依赖于抽象。 换言之, 要针对接口编程, 而不是针对实现编程。
- 接口隔离原则: 使用多个专门的接口, 而不使用单一的总接口, 即客户端不应该依赖那些它不需要的接口。
- 迪米特法则: 一个软件实体应当尽可能少地与其他实体发生相互作用。
  - 例如外观模式, 对外暴露统一接口。

外观模式（Facade）

- 为系统中多个子系统提供一致的对外调用，对客户端隐藏子系统细节，降低其与子系统的耦合。

外观模式（Facade）


面向调用者的接口
客户端1
.....
客户端2
.....
doSomething()
doOtherthing()
门面系统
+doSomething()
+doOtherthing()
.....
外观模式
子系统A
.....
子系统B
.....
子系统C
.....
内部多个子系统协作

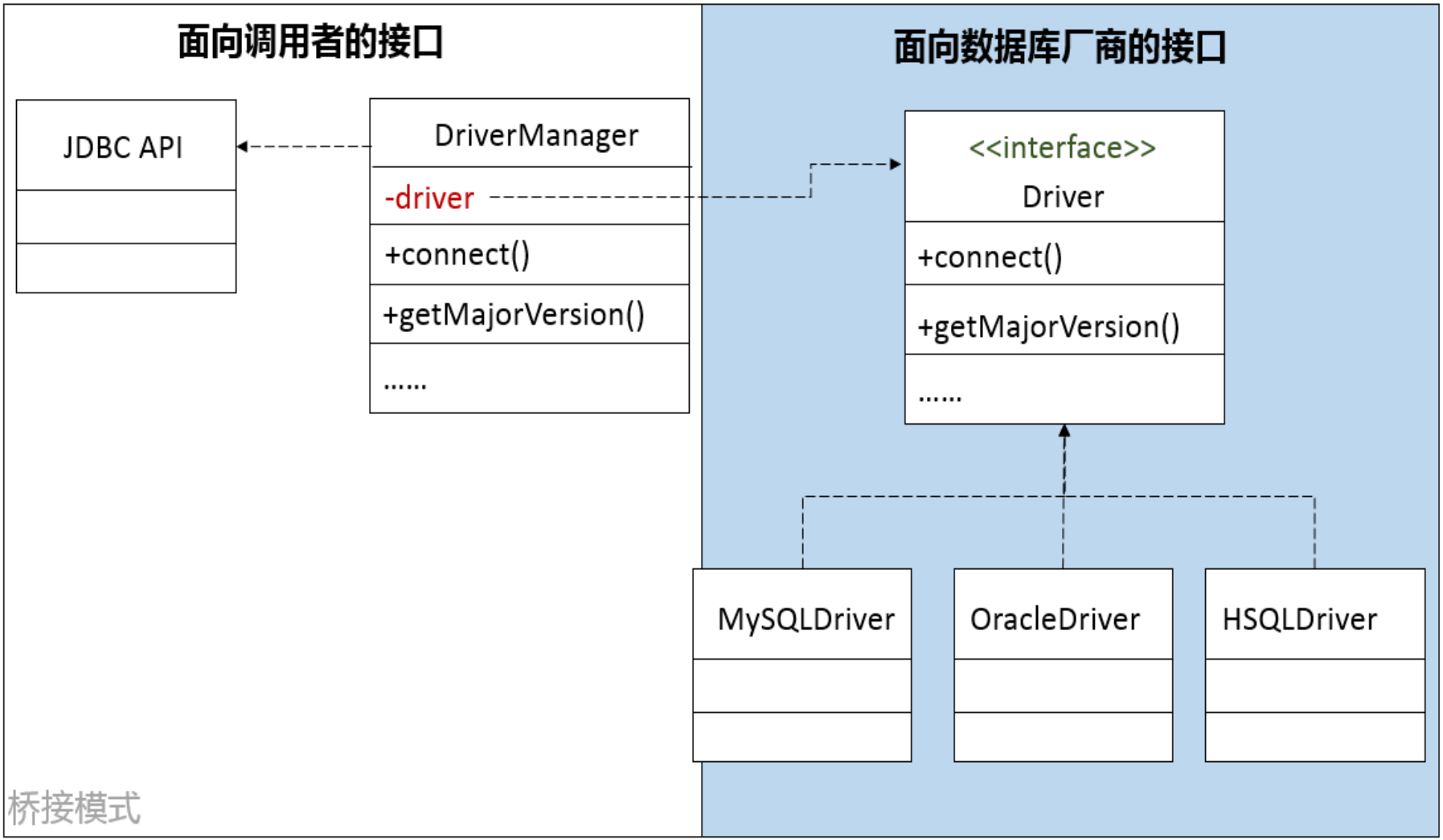
桥接模式（bridge）

- JDBC中的把面向厂商的接口(Driver)和面向使用者的API(DriverManager)做了拆分隔离。

```
Class.forName("com.mysql.jdbc.Driver");
Connection conn =
    DriverManager.getConnection(url, username, password);
```

开发者只需要关注JDBC API, 无需关注不同数据库Driver接口实现

桥接模式（bridge）


面向调用者的接口
面向数据库厂商的接口
JDBC API
DriverManager
-driver
+connect()
+getMajorVersion()
.....
<<interface>>
Driver
+connect()
+getMajorVersion()
.....
MySQLDriver
OracleDriver
HSQLDriver
桥接模式

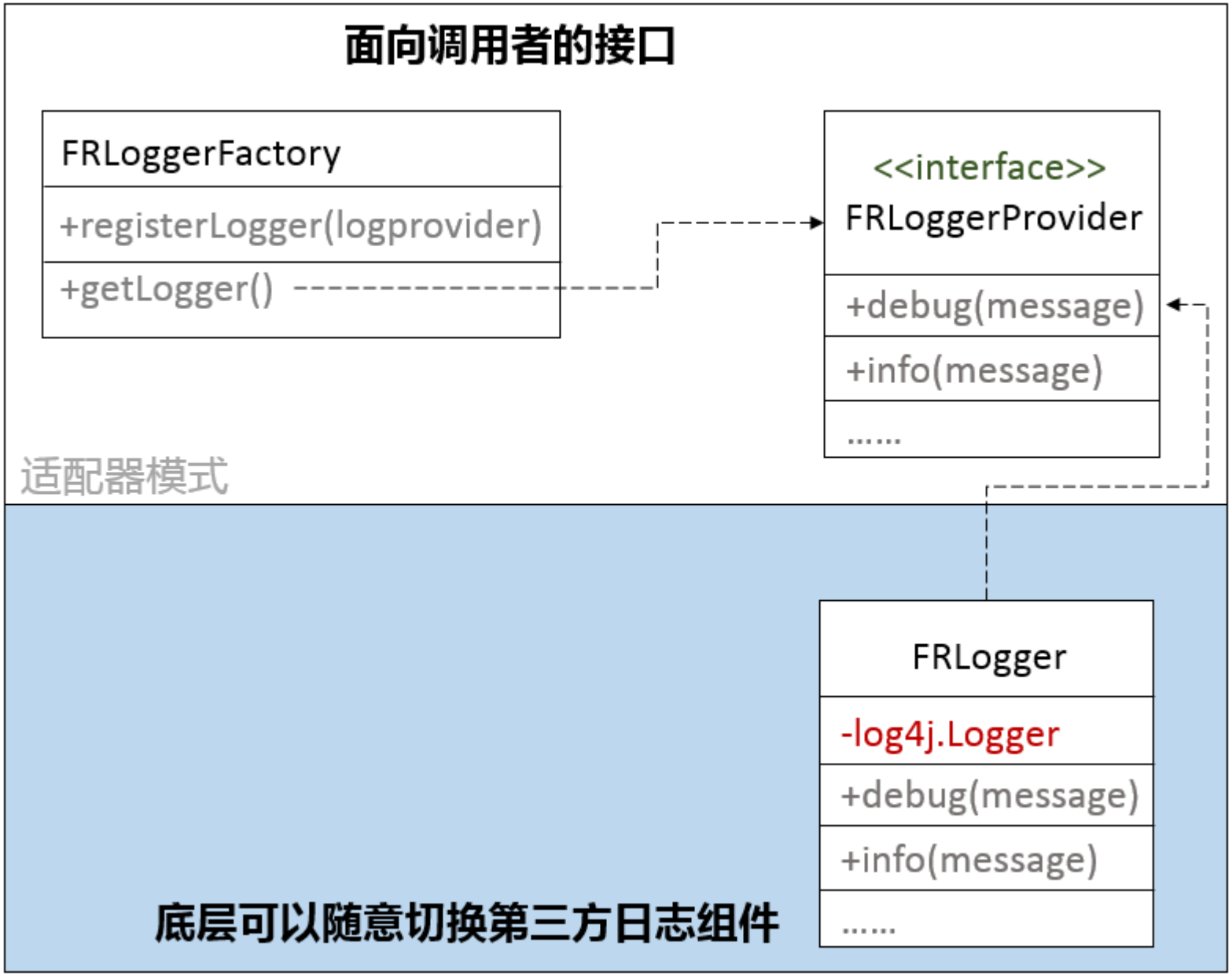
适配器模式

- 引入第三方库(hibernate, log4j), 不应该直接在代码中继承或者使用其实体类。
- 需要抽出上层统一接口, 然后增加实现类, 对外暴露接口。

```
1 // 代码与log4j强耦合, 不推荐
2 org.apache.log4j.Logger.getRootLogger().info("info");

3 // 底层可以随意更换log框架
4 FRLoggerFactory.getLogger().info("info");
```

适配器模式


面向调用者的接口
FRLoggerFactory
+registerLogger(logprovider)
+getLogger()
<<interface>>
FRLoggerProvider
+debug(message)
+info(message)
.....
适配器模式
FRLogger
-log4j.Logger
+debug(message)
+info(message)
.....
底层可以随意切换第三方日志组件

代码复用

- 代码复用是绝大多数程序员所期望的，也是OO的目标之一。
- 为了使代码能够最大程度上复用，应注意以下几点：
  - 第一原则：面向接口的编程
  - 第二原则：优先使用对象组合，而不是类继承。
  - 相关的设计模式有： Bridge、Composite、Decorator、Observer、Strategy等

面向接口的编程（1）

- 面向接口的编程：是面向对象设计（OOD）的第一个基本原则。
- 它的含义是：使用接口和同类型的组件通讯，即，对于所有完成相同功能的组件，应该抽象出一个接口，它们都实现该接口。

面向接口的编程（2）

- 具体到JAVA中，可以是接口（interface），或者是抽象类（abstract class），所有完成相同功能的组件都实现该接口，或者从该抽象类继承。
- 客户代码只应该和该接口通讯。

面向接口的编程（3）

- **场景1：** 当我们需要用其它组件完成任务时，只需要替换该接口的实现，代码的其它部分不需要改变。
- **场景2：** 当现有的组件不能满足要求时，我们可以创建新的组件，实现该接口，或者，直接对现有的组件进行扩展，由子类去完成扩展的功能。
- **减少了代码耦合，实现了代码复用**

优先使用对象组合，而不是类继承（1）

- 继承破坏了类的封装性，这种通过生成子类的复用，通常被称为白箱复用(white-box reuse)。
- 术语"白箱"是相对可视性而言：在继承方式中，父类的内部细节对子类可见。
- 继承之间的类的依赖关系远远强于组合之间的依赖关系

优先使用对象组合，而不是类继承（2）

- 对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组合对象来获得。
- 对象组合要求对象具有良好定义的接口。这种复用风格被称为**黑箱复用(black-box reuse)**，因为被组合的对象的内部细节是不可见的,对象只以"黑箱"的形式出现。

优先使用对象组合，而不是类继承（3）

- 对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。
- 由于组合要求对象具有良好定义的接口，而且，对象只能通过接口访问，**所以并不破坏封装性；**
- 只要类型一致，运行时刻还可以用一个对象来替代另一个对象；**更进一步，因为对象的实现是基于接口写的，所以实现上存在较少的依赖关系。**

最后

- 建议大家自学JAVA常用类库一章，这个部分很基础。
  - Object类
  - 数据类型类
  - String和StringBuffer类
  - Math类