

Lesson4

面向对象的三大特称之一：封装



2021/3/26

Xueping Shen



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

主要内容

- 封装含义
- 信息隐藏的必要性
- 访问控制修饰符
 - 私有成员（变量和方法）的理解和使用
 - 共有成员的理解和使用
 - 保护成员的理解和使用
 - 使用不加任何权限修饰符的成员
- 封装的好处
- 深刻理解封装（从系统角度）
- 案例分析
 - 封装、单例模式

封装的含义：现实世界案例（机箱）：

- 一台电脑，它是由CPU、主板、显卡、内存、硬盘、电源等部件组成，我们将这些部件组装在一起就可以使用电脑了。
- 如果这些部件散落在外面，很容易造成不安全因素，于是，使用机箱壳子，把这些部件都装在里面。
- 同时，会在机箱壳上留下一些插口等，若不留插口，大家想想会是什么情况？
- 总结：机箱其实就是隐藏了设备的细节，对外提供了插口以及开关等访问内部细节的方式。



为什么要使用封装



- 下面代码有什么问题？

```
Dog d = new Dog();  
d.age = 1000;
```



不合理的赋值

- 如何解决上面设计的缺陷？

使用**封装**



封装 (encapsulation)

- 封装是面向对象方法的核心思想之一，他有两个含义
 - 一层含义是把对象的属性和行为看成为一个密不可分**的整体**，将这两者封装在一个不可分割的独立单位（即对象）中。
 - 另一层含义指**信息隐藏**，把不需要让外界知道的信息隐藏起来，有些对象的属性及行为允许外界用户知道或使用，但不允许更改，而另一些属性和行为则不允许外界知晓或只允许使用对象的功能，而尽可能隐藏对象的功能实现细节。



封装的示例 (访问权限加以控制)



花园：所有人可以访问

田地：本人及儿子可以访问

房屋：本人及同一家族可以访问

小金库：归自己私有

为了安全，我们对花园、天地、房屋及小金库加入了访问权限



封装的示例

法拉力公司



迪斯尼汽车公司



市场经理
雪莉女士

要求



采购经理
罗杰斯先生

接口



封装的示例

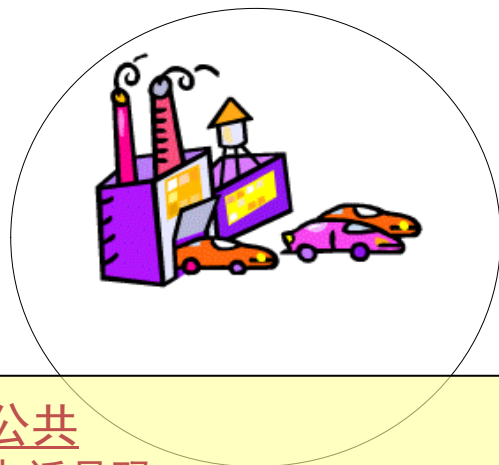
法拉利公司



公共
电话号码
电子邮箱
产品种类

私有
备件的制造方法
备件库存
备件的制造成本

迪斯尼汽车公司



公共
电话号码
电子邮箱
汽车种类

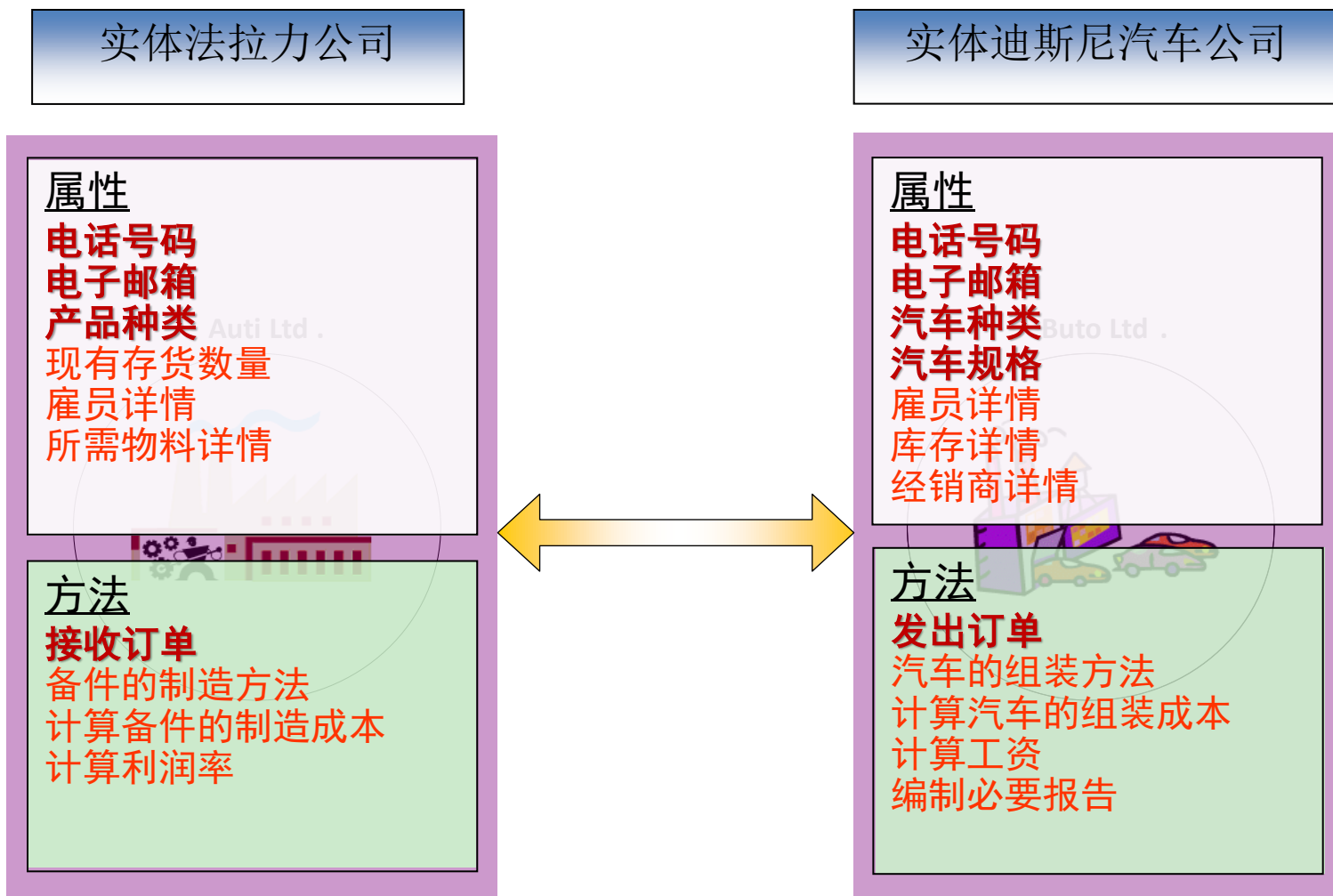
私有
汽车的组装方法
汽车库存
汽车的组装成本



有选择地提供数据



封装的示例



主要内容

- 封装含义
- **信息隐藏的必要性**
- 访问控制修饰符
 - 私有成员（变量和方法）的理解和使用
 - 共有成员的理解和使用
 - 保护成员的理解和使用
 - 使用不加任何权限修饰符的成员
- 封装的好处
- 深刻理解封装（从系统角度）
- 案例分析
 - 封装、单例模式
 - 双向关联

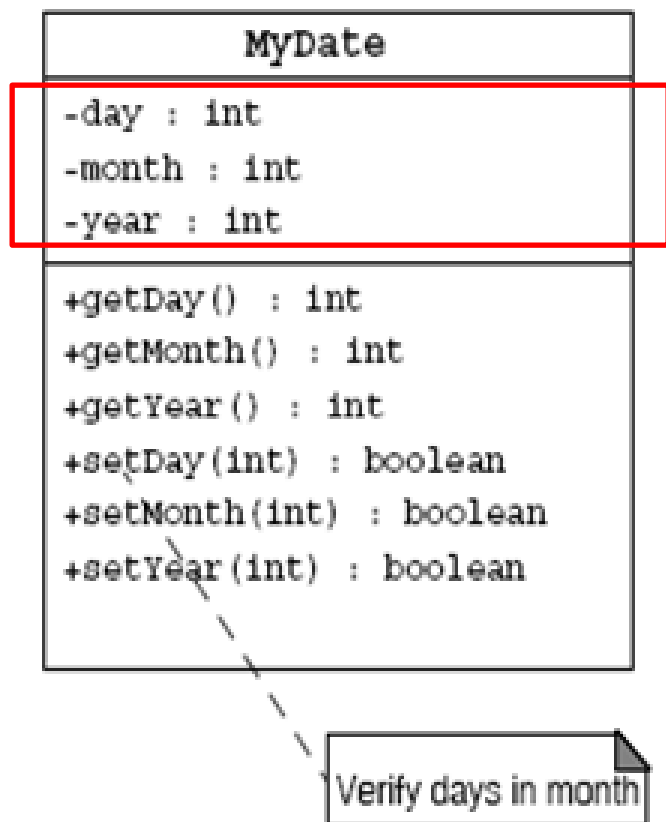
信息隐藏的必要性 (Information Hiding)

MyDate
+day : int +month : int +year : int

```
d.day = 32;  
// invalid day  
  
d.month = 2; d.day = 30;  
// plausible but wrong  
  
d.day = d.day + 1;  
// no check for wrap around
```



信息隐藏的必要性 (Information Hiding) — 解决方案



```
MyDate d = new MyDate();

d.setDay(32);
// invalid day, returns false

d.setMonth(2);
d.setDay(30);
// plausible but wrong,
// setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around
// needs to occur
```



信息隐藏的必要性(Information Hiding)——更好的方案

MyDate
-date : long
+getDay() : int +getMonth() : int +getYear() : int +setDay(int) : boolean +setMonth(int) : boolean +setYear(int) : boolean -isDayValid(int) : boolean



案例分析（1）Employee类

```
class Employee {  
public String name;  
public String id;  
public String gender;  
  
    public void work() {  
        System.out.println(id + ":" + name + ":" + gender + " 努力工作中!!!");  
    }  
}
```



测试类（假设两个类在同一个包中）

```
public class EmployeeDemo {  
    public static void main(String[] args) {  
        // 创建对象  
        Employee jack = new Employee();  
  
        // 进制通过类名.成员的形式调用成员。初始化实例变量  
        jack.name = "jack";  
        jack.id = "123456";  
        jack.gender = "男";  
  
        // 调用成员方法  
        jack.work();  
        System.out.println();  
  
        // 传入非法的参数  
        jack.gender = "不是男人";  
        jack.work();  
    }  
}
```

Jack的性别
被恶搞了



代码分析：

- 缺陷：如果不使用封装，很容易赋值错误，并且任何人都可以更改，造成信息的不安全。
- 问题解决：使用封装



Employee类（更好的解决方案）

```
class Employee {  
    //使用了private修饰了成员变量  
    private String name;  
    private String id;  
    private String gender;  
  
    public void work() {  
        System.out.println(id + ":" + name + ":" + gender + " 努力工作中!!!");  
    }  
}
```



为什么编译出错？

```
public class EmployeeDemo {  
    public static void main(String[] args) {  
        // 创建对象  
        Employee jack = new Employee();  
  
        //编译报错  
        jack.name = "jack";  
        jack.id = "123456";  
        jack.gender = "男";  
  
        // 编译报错  
        jack.gender = "不是男人";  
        jack.work();  
    }  
}
```



更好的解决方案

- 将所有的成员变量封装加上`private`，对外提供公开的用于设置对象属性的`public`方法
 - 设置`set`
 - 获取`get`
- `Employee`类的`gender`的修饰符修改为`private`后，无法在类外调用，那么如何给`gender`设置合适的值？
 - 在`setGender()`方法中加入逻辑判断，过滤掉非法数据。



```
class Employee {  
    private String name;  
    private String id;  
    private String gender;  
    // 提供公有的get set方法  
    public String getName() {  
        return name;  
    }  
}
```

哈哈，这样，别人就不能恶搞对象的性别了

```
    public void setGender(String gen) {  
        if ("男".equals(gen) || "女".equals(gen)) {  
            gender = gen;  
        } else {  
            System.out.println("请输入\"男\"或者\"女\"");  
        }  
    }  
  
    public void work() {  
        System.out.println(id + ":" + name + ":" + gender + "努力工作中!!!");  
    }  
}
```

```
return gender;  
}
```



案例分析（2）：该看的看，不该看的别看

属性私有
(用private关键字修饰)

```
public class Student2 {  
    private String name; // 姓名  
    private int rp;      // 人品（取值在1-10之间，越高越好）
```

```
    public int getRp() {  
        return rp;  
    }
```

提供公有的方法访问私有属性，
可以在方法中实现对属性的控制。

```
    public void setRp(int rp) {  
        if (rp < 1 || rp > 10) {  
            System.out.println("错误！人品值应该在1-10之间！");  
            this.rp = 1; // 人品不符合要求，赋予默认值1  
        } else {  
            this.rp = rp;  
        }  
    }
```

.....

```
}
```



案例分析（2）：该看的看，不该看的别看

哈哈，这样，别人就不能恶搞对象的人品了

```
public class Student2Test {  
    public static void main(String[] args) {  
        //实例化学员对象,对其属性进行初始化  
        Student2 xiaoxin = new Student2();  
        xiaoxin.setName("小明");  
        xiaoxin.setRp(-1);  
        //小新自我介绍  
        xiaoxin.introduction();  
    }  
}
```

通过公有的
setter方法给两
个属性赋值

➤ 我们可以通过公有的getter（取值）、setter（赋值）方法访问这两个属性，而且在人品的赋值方法中加入了属性访问的限制，成功的实现了对小明人品的拯救。



案例分析（3）：该看的看，不该看的别看

```
public class Teacher3 {  
    1 private String name; // 教员姓名  
    private int age;      // 年龄  
  
    2 public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        if (age < 22) {  
            3 System.out.println("错误！最小年龄应为22岁！");  
            this.age = 22; // 如果不符合年龄要求，则赋予默认值  
        } else {  
            this.age = age;  
        }  
    }  
    // 此处省略对name属性的setter、getter方法  
}
```



案例分析（3）：该看的看，不该看的别看

● 测试类通过调用setter方法，为对象的各个属性赋值

```
public class Teacher3Test {  
    public static void main(String[ ] args) {  
        Teacher3 teacher = new Teacher3();  
        teacher.setName ("Mary");  
        teacher.setAge(10);  
        System.out.println(teacher.introduction());  
    }  
}
```

错误！最小年龄应为22岁！
大家好！我是Mary，我今年22岁

使用封装，增加了数据访问限制，增强了程序的可维护性

小结:

- 通过封装：
 - 1: 隐藏了类的具体实现
 - 2: 操作简单
 - 3: 提高对象数据的安全性
 - 4: 减少了冗余代码，数据校验等写在方法里，可以复用
- 所以在解决实际问题的時候，我們需要對成員變量和成員方法進行歸類，設置訪問級別，提供不同程度的對外訪問權限。

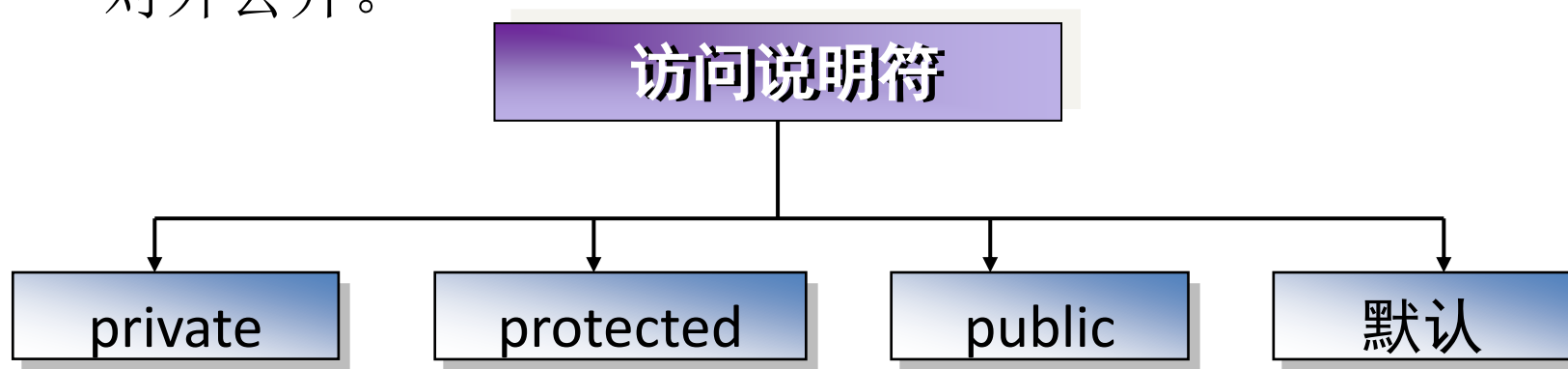


主要内容

- 封装含义
- 信息隐藏的必要性
- 访问控制修饰符
 - 私有成员（变量和方法）的理解和使用
 - 共有成员的理解和使用
 - 保护成员的理解和使用
 - 使用不加任何权限修饰符的成员
- 封装的好处
- 深刻理解封装（从系统角度）
- 案例分析
 - 封装、单例模式
 - 双向关联

访问控制修饰符（JAVA）

- 访问控制分四种级别：
 - 公开级别：用**public**修饰，对外公开。
 - 受保护级别：用**protected**修饰，向子类以及同一个包中的类公开。
 - 默认级别：没有访问控制修饰符，向同一个包中的类公开。
 - 私有级别：用**private**修饰，只有类本身可以访问，不对外公开。



四种访问级别的被访问范围

- 可以对Java类中定义的**属性**和**方法**进行访问控制----规定不同的保护等级：**public**、**protected**、**default**、**private**

修饰符	同一个类	同一个包	子类	整体
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes



Java中的protected修饰符



2021/3/26

Xueping Shen



北京航空航天大学
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY 软件学院

Java中的protected修饰符

1. 父类的protected成员是包内可见的，并且对子类可见；
2. 若子类与父类不在同一包中，那么在子类中，子类实例可以访问其从父类继承而来的protected方法，而不能访问父类实例的protected方法。
3. 对于protected修饰的静态变量，无论是否同一个包，在子类中均可直接访问



思考题

1. 同一个包中， 子类对象能访问父类的 `protected` 方法吗？
 - ◆ 在同一个包中， 普通类或者子类都可以访问基类的 `protected` 方法。
2. 不同包下， 在子类中创建该子类对象能访问父类的 `protected` 方法吗？
3. 不同包下， 在子类中创建父类对象能访问父类的 `protected` 方法吗？
4. 不同包下， 在子类中创建另一个子类的对象能访问公共父类的 `protected` 方法吗？
5. 父类 `protected` 方法加上 `static` 修饰符又会如何呢？



父类为非静态 protected 修饰类

```
1 package com.protectedaccess.parentpackage;  
2  
3 public class Parent {  
4  
5     protected String protect = "protect field";  
6  
7     protected void getMessage(){  
8         System.out.println("i am parent");  
9     }  
10 }
```



不同包下，在子类中通过父类引用不可以访问其 `protected` 方法

```
1 package com.protectedaccess.parentpackage.sonpackage1;  
2  
3 import com.protectedaccess.parentpackage.Parent;  
4  
5 public class Son1 extends Parent{  
6     public static void main(String[] args) {  
7         Parent parent1 = new Parent();  
8         // parent1.getMessage(); 错误  
9  
10        Parent parent2 = new Son1();  
11        // parent2.getMessage(); 错误  
12    }  
13 }
```



不同包下，在子类中通过该子类引用可以访问其 **protected** 方法

```
1 package com.protectedaccess.parentpackage.sonpackage1;
2
3 import com.protectedaccess.parentpackage.Parent;
4
5 public class Son1 extends Parent{
6     public static void main(String[] args) {
7         Son1 son1 = new Son1();
8         son1.getMessage(); // 输出: i am parent,
9     }
10    private void message(){
11        getMessage(); // 如果子类重写了该方法， 则输出重写方法中的内容
12        super.getMessage(); // 输出父类该方法中的内容
13    }
14 }
```

子类中实际上把父类的方法继承下来了， 可以通过该子类对象访问， 也可以在子类方法中直接访问。 还可以通过 **super** 关键字调用父类中的该方法。



- 不同包下，在子类中不能通过另一个子类引用访问共同基类的 **protected** 方法



```
1 package com.protectedaccess.parentpackage.sonpackage2;  
2  
3 import com.protectedaccess.parentpackage.Parent;  
4  
5 public class Son2 extends Parent {  
6  
7 }
```



```
1 package com.protectedaccess.parentpackage.sonpackage1;
2
3 import com.protectedaccess.parentpackage.Parent;
4 import com.protectedaccess.parentpackage.sonpackage2.Son2;
5
6 public class Son1 extends Parent{
7     public static void main(String[] args) {
8         Son2 son2 = new Son2();
9         // son2.getMessage(); 错误
10    }
11 }
```



父类为静态 protected 修饰类

- 对于protected修饰的静态变量，无论是否同一个包，在子类中均可直接访问
- 在不同包的非子类中则不可访问



```
1 package com.protectedaccess.parentpackage;
2
3 public class Parent {
4
5     protected String protect = "protect field";
6
7     protected static void getMessage(){
8         System.out.println("i am parent");
9     }
10 }
```



```
1 package com.protectedaccess.parentpackage.sonpackage1;
2
3 import com.protectedaccess.parentpackage.Parent;
4
5 public class Son3 extends Parent{
6     public static void main(String[] args) {
7         Parent.getMessage(); // 输出: i am parent
8     }
9 }
```



在不同包下，非子类不可访问

```
1 package com.protectedaccess.parentpackage.sonpackage1;
2
3 import com.protectedaccess.parentpackage.Parent;
4
5 public class Son4{
6     public static void main(String[] args) {
7         // Parent.getMessage(); 错误
8     }
9 }
```



Java类的访问级别

- Java中，类可以是**public**和**默认访问级别**
 - public级别的类可以被所有其他类访问。
 - 默认级别的类只能被同一个包中的类访问。



示例代码(默认级别的类只能被同一个包中的类访问)

```
package mpack1;  
class Base1{}  
public class Base2{}
```

```
package mpack2;  
class Sub1 extends Base1{} //非法  
class Sub2 extends Base2{}  
  
public class Guest{  
    public void test(){  
Base1 b1=new Base1(); //非法  
        Base2 b2=new Base2();  
    }  
}
```



类的封装（小结）

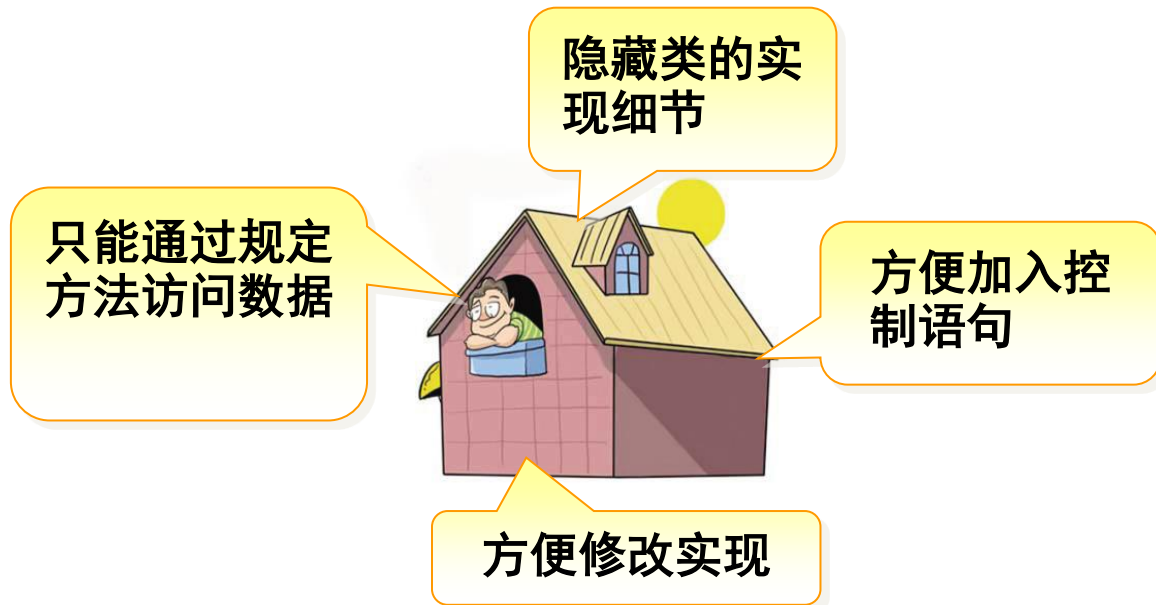
- 含义：
 - java中，对象就是一个封装体。
 - 把对象的属性和服务结合成一个独立的单位，并尽可能隐蔽对象的内部细节（尤其是私有数据）
 - 目的：使对象以外的部分不能随意存取对象的内部数据（如属性），从而，使软件错误能够局部化，大大减少查错和排错的难度。

“隐藏属性、方法或实现细节的过程称为封装。”



使用封装的好处

- 1、良好的封装能够减少耦合。
- 2、类内部的结构可以自由修改。
- 3、可以对成员进行更精确的控制。
- 4、隐藏信息，实现细节。



主要内容

- 封装含义
- 信息隐藏的必要性
- 访问控制修饰符
 - 私有成员（变量和方法）的理解和使用
 - 共有成员的理解和使用
 - 保护成员的理解和使用
 - 使用不加任何权限修饰符的成员
- 封装的好处
- 案例分析
 - 封装、单例模式
- 深刻理解封装（从系统角度）

案例分析（1）：

- 问题：构造方法可以使用private修饰吗？
- 问题：希望程序使用者不能随意自己创建对象，只能获取一个单独的对象，即一个类在内存中只有一个实例（对象）存在，如何实现？
 - 例如：比如多程序访问**同一个配置文件**，希望多程序操作的都是同一个配置文件中的数据，那么就需要保证该配置文件对象的唯一性。
 - 又比如系统日志，多线程访问一个程序，需要记录日志，需要把**日志单例化**。你不希望看到N多个日志文件实例吧？



- 单例模式：
 - 一个类在内存中只有一个实例（对象）存在，该类一般没有属性。
 - 无法继承，所以无法扩展，无法更改它的实现。
- 重点：
 - 1、单实例模式及其适用场合
 - 2、加深对private的理解
 - 3、加深对static的理解

“饿汉式”单实例模式

```
public class Singleton
{
    //静态的。保留自身的引用。类加载时就初始化
    private static Singleton test = new Singleton();
    //必须是私有的构造函数
    private Singleton(){}
    //公共的静态的方法。
    public static Singleton getInstance() {
        return test;
    }
}
```



“饿汉式”单实例模式

- 是否 Lazy 初始化：否
- 是否多线程安全：是
- 描述：这种方式比较常用，但容易产生垃圾对象。
- 优点：没有加锁，执行效率会提高。
- 缺点：类加载时就初始化，浪费内存。
- 特点：它基于 classloader 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化



“懒汉式”单实例模式

```
public class Singleton  
{
```

懒汉式：单例的延迟加载方式。
延迟：需要的时候，再创建

//静态的。保留自身的引用。

```
private static Singleton test = null;
```

//必须是私有的构造函数

```
private Singleton(){}  
//公共的静态的方法。
```

```
public static Singleton getInstance() {  
    if(test == null)
```

```
    {
```

```
    {
```

```
        test = new Singleton();
```

```
    }
```

```
    return test;
```

```
    }
```

```
}
```



“懒汉式”单实例模式

- 是否 Lazy 初始化：是
- 是否多线程安全：否
- 描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 `synchronized`，所以严格意义上它并不算单例模式。
- 特点这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。



但是单例的延迟加载方式会出现多线程中的安全问题，需要对上面的代码进行修改（了解）

```
class Single2
{
    private static Single s = null;
    private Single(){}
    public static Single getInstance()
    {
        if(s==null)
        {
            synchronized(Single.class)
            {
                if(s==null)

                    s = new Single();
            }
        }
        return s;
    }
}
```



单实例模式及其适用场合

- 只需要使用一个单独的资源，并且需要共享这个单独资源的状态信息时，就能用到单例模式。
- 需要频繁的创建和销毁的对象；
- 创建对象时耗时过多或耗费资源过多，但又经常用到的对象；
- 工具类对象；
- 频繁访问数据库或文件的对象。



单实例模式优点

- 由于单例模式在内存中只有一个实例，减少了内存开销。
- 系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统性能。
- 单例模式可以在系统设置全局的访问点，优化和共享资源访问。
- 例如一个播放器程序，当用户打开一个播放音乐界面，再想打开另一个音乐播放时，之前的界面就关闭了。这就是一个单例模式的具体应用



单例模式：小结

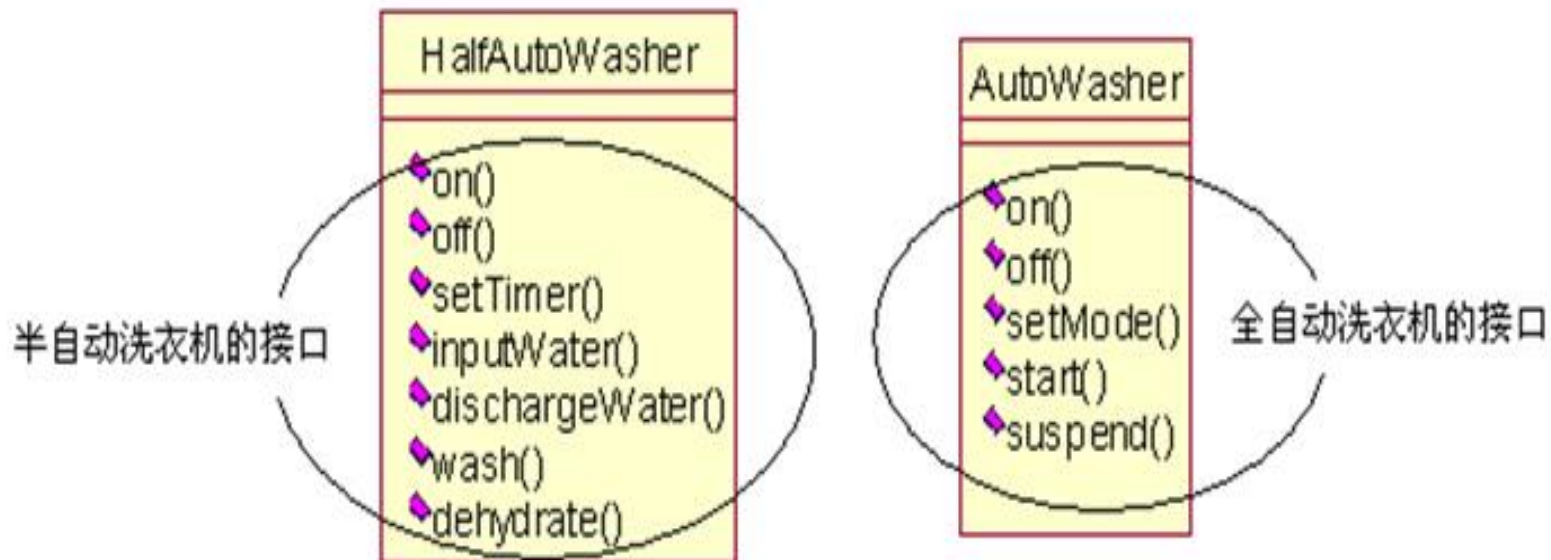
- 单例类是指仅有一个实例的类。
- 在系统中具有唯一性的组件可作为单例类，**这种类的实例通常会占用较多的内存，或者实例的初始化过程比较冗长，因此随意创建这些类的实例会影响系统的性能。**



主要内容

- 封装含义
- 信息隐藏的必要性
- 访问控制修饰符
 - 私有成员（变量和方法）的理解和使用
 - 共有成员的理解和使用
 - 保护成员的理解和使用
 - 使用不加任何权限修饰符的成员
- 封装的好处
- 案例分析
 - 封装、单例模式
 - 双向关联
- 深刻理解封装（从系统角度）

深刻理解封装（从系统的角度）



半自动洗衣机 HalfAutoWasher

全自动洗衣机 AutoWasher

用半自动洗衣机洗衣服（界面欠清晰，不容易操作）

```
HalfAutoWasher washer=new  
HafAutoWasher();
```

```
//开始洗衣服  
washer.on(); //开机
```

```
//洗涤  
washer.inputWater(); //放水  
washer.setTimer(5); //定时5分钟  
washer.wash(); //洗涤  
washer.dischargeWater(); //排水
```

```
//第一次清洗  
washer.inputWater(); //放水  
washer.setTimer(5); //定时5分钟  
washer.wash(); //洗涤  
washer.dischargeWater(); //排水
```

```
//第二次清洗
```

```
washer.inputWater(); //放水  
washer.setTimer(5); //定时5分钟  
washer.wash(); //洗涤  
washer.dischargeWater(); //排水
```

```
//为衣服脱水
```

```
washer.setTimer(8); //定时8分钟  
washer.dehydrate(); //脱水
```

```
washer.off(); //关机
```



用全自动洗衣机洗衣服（界面清晰，容易操作，调用更简单）

```
AutoWasher washer=new AutoWasher();
```

```
//开始洗衣服
```

```
washer.on(); //开机
```

```
washer.setMode("标准模式"); //设置洗衣机模式
```

```
//开始洗衣服，洗衣结束后，30分钟内洗衣机会自动关机
```

```
washer.start();
```



深刻理解封装（从系统的角度）

- 把尽可能多的东西藏起来，对外提供简洁的接口
- 系统的封装程度越高，那么它的相对独立性就越高，而且使用起来更方便。



深刻理解封装（从系统的角度）

- 封装是指隐藏对象的属性和实现细节，仅仅对外公开接口。
- 封装的优点：
 - （1）便于使用者正确的方便地理解和使用系统，防止使用者错误修改系统的属性。
 - （2）有助于建立各个系统之间的松耦合关系，提高系统的独立性。当某一个系统的实现发生变化，只要它的接口不变，就不会影响到其他的系统。
 - （3）提高软件的可重用性，每个系统都是一个相对独立的整体，可以在多种环境中得到重用。
 - （4）降低了构建大型系统的风险，即使整个系统不成功，个别的独立子系统有可能依然是有价值的。



思考题1

```
class Parent{  
    private int f1 = 1;  
    int f2 = 2;  
    protected int f3 = 3;  
    public int f4 = 4;  
    private void fm1() {System.out.println("in fm1() f1=" + f1);}   
    void fm2() {System.out.println("in fm2() f2=" + f2);}   
    protected void fm3() {System.out.println("in fm3() f3=" + f3);}   
    public void fm4() {System.out.println("in fm4() f4=" + f4);}   
}
```



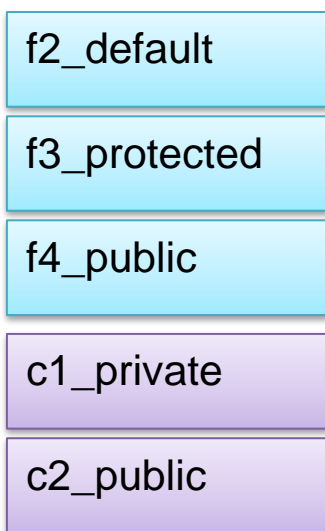
思考题1

```
class Child extends Parent{                                //设父类和子类在同一个包内
    private int c1 = 21;
    public int c2 = 22;
    private void cm1(){System.out.println("in cm1() c1=" + c1);}
    public void cm2(){System.out.println("in cm2() c2=" + c2);}
    public static void main(String args[]){
        int i;
        Parent p = new Parent();
        i = p.f2;      // i = p.f3;          i = p.f4;
        p.fm2();      // p.fm3();          p.fm4();
        Child c = new Child();
        i = c.f2;      // i = c.f3;          i = c.f4;
        i = c.c1;      // i = c.c2;
        c.cm1();      // c.cm2();  c.fm2();  c.fm3();  c.fm4()
    }
}
```



思考题1

- 父类Parent和子类Child在同一包中定义时:



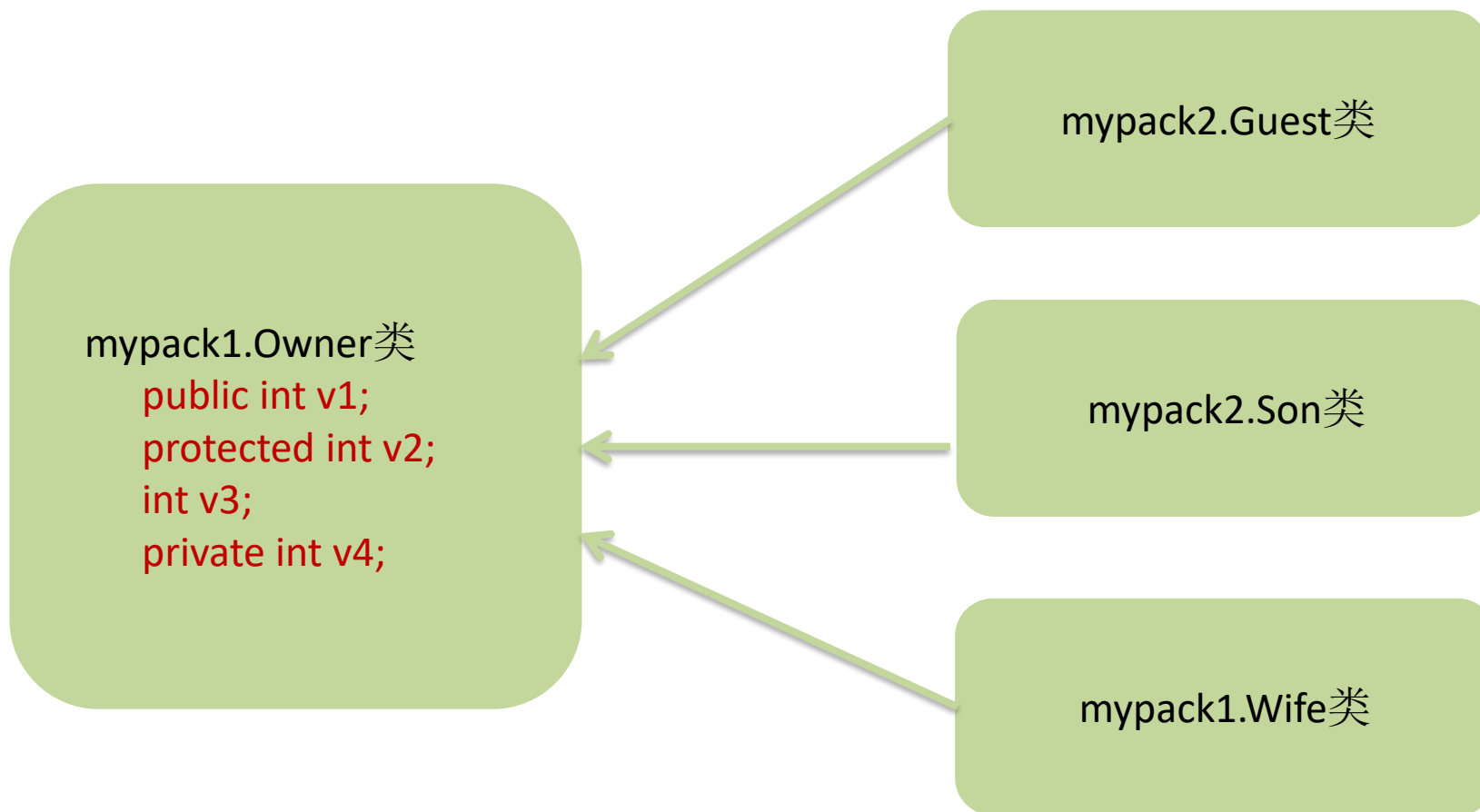
子类对象可以访问的成员数据



子类的对象可以调用的成员方法



思考题2



思考题2(不同包)

```
package mypack1;
```

```
public class Owner{  
    public int v1;  
    protected int v2;  
    int v3;  
    private int v4;  
  
    protected void test(){}  
}
```

```
package mypack2;
```

```
import mypack1.Base;
```

```
public class Guest{  
    public void method(){  
        Base a=new Base();  
        a.v1=1; //合法
```

```
        a.v2=1; //编译出错
```

```
        a.v3=1; //编译出错
```

```
        a.v4=1; //编译出错
```

```
        a.test(); //编译出错
```

```
    }  
}
```

```
}
```



思考题2（同一个包）

```
package mypack1;
```

```
public class Owner{  
    public int v1;  
    protected int v2;  
    int v3;  
    private int v4;  
  
    protected void test(){}  
}
```

```
package mypack1;
```

```
import mypack1.Owner;
```

```
public class Son extends Owner{  
    public void method(){  
        v1=1; //合法  
        v2=1; //合法  
        v3=1; //合法  
        test(); //合法  
        Base a=new Base();  
        a.v1=1; //合法  
        a.v2=1; //合法  
        a.v3=1; //合法  
        a.v4=1; //编译出错  
    }  
}
```



思考题2（不同包）

```
package mypack1;
```

```
public class Owner{  
    public int v1;  
    protected int v2;  
    int v3;  
    private int v4;  
  
    protected void test(){}  
}
```

```
package mypack2;
```

```
import mypack1.Owner;
```

```
public class Son extends Owner{  
    public void method(){  
        v1=1; //合法  
        v2=1; //合法  
        v3=1; //合法  
        test(); //合法  
        Base a=new Base();  
        a.v1=1; //合法  
        a.v2=1; //编译出错  
        a.v3=1; //编译出错  
        a.v4=1; //编译出错  
    }  
}
```

