# 【OOP：面向对象编程\_第五天】

## 主要内容

1. 异常
2. try-catch-finally
3. 异常分类
4. throws和throw
5. 自定义异常
6. 单一职责原则SRP
7. 开闭原则OCP
8. 里氏替代原则LSP
9. 依赖倒置原则DIP
10. 接口分离原则ISP
11. 迪比特法则LOD
12. 组合聚会复用元素CARP
13. 类和类的六种关系

## 学习目标

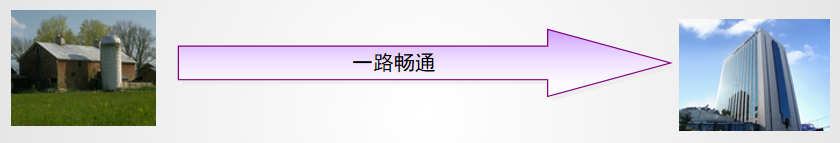
|  |  |
| --- | --- |
| 知识点 | 要求 |
| 引入异常 | 理解 |
| 异常处理：try, catch, finally | 掌握 |
| 异常分类 | 掌握 |
| 异常处理：throws, throw | 掌握 |
| 自定义异常 | 了解 |
| 单一职责原则SRP | 了解 |
| 开闭原则OCP | 了解 |
| 里氏替代原则LSP | 了解 |
| 依赖倒置原则DIP | 了解 |
| 接口分离原则ISP | 了解 |
| 迪比特法则LOD | 了解 |
| 组合聚会复用元素CARP | 了解 |
| 类和类的六种关系 | 了解 |

## 第一节 异常的捕获

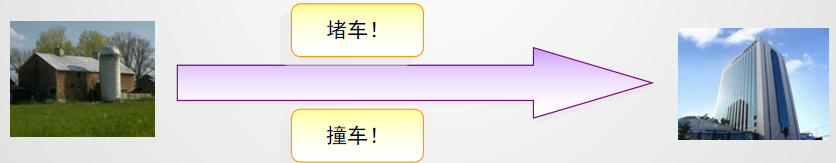
### 1.1 引入异常

**生活中的异常：**

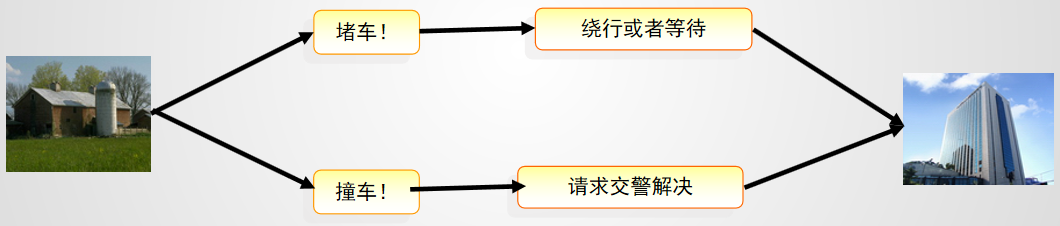
正常情况下，小王每日开车去上班，耗时大约30分钟



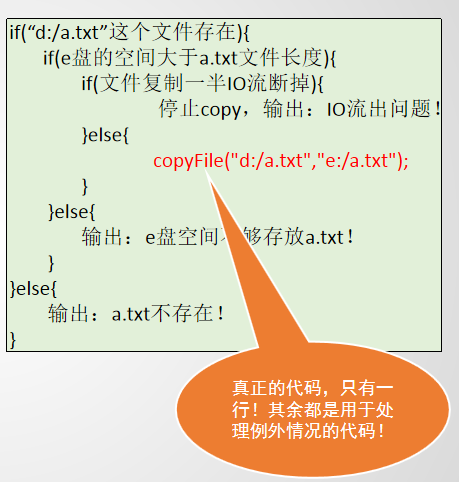
但是，异常情况迟早要发生！



面对异常该怎么办呢？生活中，我们会根据不同的异常进行相应的处理，而不会就此中断我们的生活



**程序中的异常**



示例1：给出除数和被除数，求商

* 如果除数为0，出异常
* 如果除数或者被除数不是数字，出异常

示例2：将d:/a.txt复制到e:/a.txt

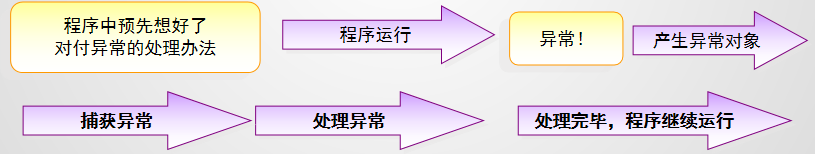
* 如果d:/a.txt不存在
* 如果e:/存在a.txt
* 如果e盘空间不足
* 如果复制过程中出错

面对异常该怎么办呢？

方式1：由开发者通过if-else来解决异常问题

* 代码臃肿：业务代码和异常处理代码放一起
* 程序员要花很大精力"堵漏洞“
* 程序员很难堵住所有“漏洞”，对程序员本身要求较高

方式2：**开发者不需要通过if-else来解决异常问题，而是Java提供异常处理机制。它将异常处理代码和和业务代码分离，使程序更优雅，更好的容错性，高键壮性**



异常（ Exception 也称例外）就是在程序的运行过程中所发生的不正常的事件，它会中断正在运行的程序

* 所需文件找不到
* 网络连接不通或中断
* 算术运算错 (被零除…)
* 数组下标越界
* 装载一个不存在的类或者对null对象操作
* 类型转换异常
* ……

当Java程序出现以上的异常时，就会在所处的方法中产生一个异常对象。这个异常对象包括异常的类型，异常出现时程序的运行状态以及对该异常的详细描述。

Java的异常处理是通过5个关键字来实现的：try、catch、 finally、throw、throws



#### 【示例1】引入异常

|  |
| --- |
| **package** exceptionDemo;  **public class** Test1 {  **public static void** main(String[] args) **throws** InterruptedException {  **int** a=0;  **if**(a!=0){  System.***out***.println(1/a);  }  *// int[] arr={1};  //System.out.println(arr[10]);  /\*Person p=null ;  System.out.println(p.equals(p));\*/* Thread.*sleep*(1000);    System.***out***.println(**"后续代码"**);  } } **class** Person{} |

张三谈了一个女友 ,要和女友出去吃饭,没有好看的衣服, 室友李四有一件非常好看的衬衫,张三想借过来穿,但是衣服有一些脏;

异常:脏

捕获处理

李四把衬衫洗干净,交给张三,李四把异常处理完毕,李四捕获异常

抛出处理

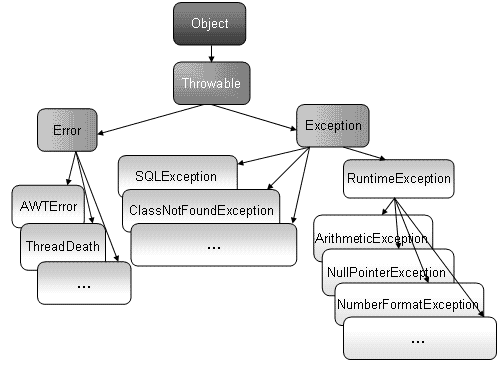
李四不洗,让张三自己去洗,李四把问题留个张三,李四抛出异常

优点：Java已经提供了异常处理机制，发生异常后，会给出异常类型、异常提示信息、异常的位置

缺点：出现异常后，后续语句不执行了；提示信息太专业，可读性差

**解决：try-catch-finally 处理异常；throws 抛出异常不处理，调用者处理**

### 1.2 异常体系



**Error**

Error类层次描述了Java运行时系统内部错误和资源耗尽错误，一般指与JVM或动态加载等相关的问题，如虚拟机错误，动态链接失败，系统崩溃等。

这类错误是我们无法控制的，同时也是非常罕见的错误。所以在编程中，不去处理这类错误。注：我们不需要管理Error！

**Exception**

所有异常类的父类，其子类对应了各种各样可能出现的异常事件。

**Exception分类**

* 运行时异常Runtime Exception（unchecked Exception）

可不必对其处理，系统自动检测处理

一类特殊的异常，如被 0 除、数组下标超范围等，其产生比较频繁，处理麻烦，如果显式的声明或捕获将会对程序可读性和运行效率影响很大

* 检查异常 Checked  Exception

必须捕获进行处理，否则会出现编译错误。

注意：只有Java提供了Checked异常，体现了Java的严谨性，提高了Java的健壮性。同时也是一个备受争议的问题。

#### 【示例2】区分运行时异常和检查异常

|  |
| --- |
| **public class** TestException5 {  **public static void** main(String[] args) {  //运行时异常，异常可以不进行捕获，运行时才会发生异常  String str= **"bjsxt"**;  System.**out**.println(str.toUpperCase());  NullPointerException ne;  ArithmeticException ae;  InputMismatchException ime;  ClassCastException cce;  **int** result = 10 / 0 ;  //检查异常 checked Exception  **try** {  Class.forName(**"com.bjsxt.adfadfa.afasdfad"**).newInstance();  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  } **catch** (IllegalAccessException e) {  e.printStackTrace();  } **catch** (InstantiationException e) {  e.printStackTrace();  }  **try** {  InputStream is = **new** FileInputStream(**"d:/bjsxt.txt"**);  } **catch** (FileNotFoundException e) {  e.printStackTrace();  }  } } |

### 1.3 异常处理：try-catch-finally

#### 【示例3】使用try-catch处理异常

|  |
| --- |
| **// try catch 简单使用**  **public static void** main(String[] args) {  *testA*();  } **public static void** testA(){  System.***out***.println(**"程序启动"**);   **try**{  System.***out***.println(1/0);  }**catch**(ArithmeticException e){  System.***out***.println(e.getMessage());  e.printStackTrace();  }    System.***out***.println(**"程序结束"**); } |

单个catch捕获多个不同异常

|  |
| --- |
| **package** com.bjsxt.tryDemo;  **public class** Test1 {  **public static void** main(String[] args) {   System.***out***.println(**"第一行"**);  System.***out***.println(**"第二行"**);  *// 使用try语句对可能出现异常的代码进行监控  // 通过catch语句 准备异常一旦发生,对应的处理机制  // java中万物皆对象* **try** {  **int**[] arr={1};  System.***out***.println(arr[10]);  String s=**null**;  System.***out***.println(s.length());  System.***out***.println(1 / 0);    }**catch**(NullPointerException|ArithmeticException|ArrayIndexOutOfBoundsException e){  *// 异常发生之后的处理办法  // 获得异常信息* String message = e.getMessage();  System.***out***.println(message);  e.printStackTrace();  }     System.***out***.println(**"后续代码"**);   } } |

多重catch捕获异常

|  |
| --- |
| **package** com.bjsxt.tryDemo;  **public class** Test3 {  **static** String *s*;  **public static void** main(String[] args) {  **try** {  *//System.out.println(s.length());  //System.out.println(1 / 0);  //String s0 = (String) new Object();* **int**[] arr={1};  System.***out***.println(arr[10]);  }**catch** (ClassCastException e){  */\*e.printStackTrace();\*/* System.***out***.println(**"类型转换异常的处理代码"**);  }**catch**(ArithmeticException e){  System.***out***.println(**"数学运算异常的处理代码"**);  }**catch**(NullPointerException e){  System.***out***.println(**"空指针异常处理代码"**);  }**catch** (RuntimeException e){  System.***out***.println(**"其他运行时异常处理代码"**);  }**catch** (Exception e){  System.***out***.println(**"其他检查型异常处理代码"**);  }  System.***out***.println(**"程序执行结束"**);   } } |

try catch嵌套

|  |
| --- |
| **package** com.bjsxt.tryDemo;  **public class** Test4 {  **static** String *s*;  **public static void** main(String[] args) {  **try** {  System.***out***.println(**"异常产生之前的代码"**);    **try** {  System.***out***.println(*s*.length());  }**catch** (NullPointerException e){  System.***out***.println(**"空指针异常了"**);  }  System.***out***.println(**"异常产生之后的代码"**);   }**catch** (Exception e){  e.printStackTrace();  }  System.***out***.println(**"程序执行结束"**);   } } |

finally语句块的使用

|  |
| --- |
| **package** com.bjsxt.exceptionDemo2;  **import** java.util.InputMismatchException; **import** java.util.Scanner;  */\*\*  \* finally语句块 无论异常是否出现,都会执行的 语句块  \* 一般用于释放资源,关闭资源  \*   \*/* **public class** Test5 {  **public static void** main(String[] args) {  Scanner sc =**new** Scanner(System.***in***);  System.***out***.println(**"请输入整数"**);  **try**{  **int** i=sc.nextInt();  }**catch** (InputMismatchException e){  System.***out***.println(**"输入格式有误"**);  }**finally**{  System.***out***.println(**"finally语句块执行"**);  sc.close();  }  } }  **package** com.bjsxt.exceptionDemo2;  */\*\*  \* 在方法体中,finally语句块一定会在return之前执行  \*/* **public class** Test6 {  **public static void** main(String[] args) {  **int** chu = *getChu*(10, 2);  System.***out***.println(chu);   }  **public static int** getChu(**int** a,**int** b){  **int** result=0;  **try**{  System.***out***.println(**"try"**);   result=a/b;  }**catch** (ArithmeticException e){  e.printStackTrace();  result= 0;  }**finally** {  System.***out***.println(**"finally执行完毕了"**);  *//return 100;* }  **return** result;  } } |

**try-catch执行情况**

* 情况1：try块中代码没有出现异常

不执行catch块代码，执行catch块后边的代码

* 情况2：try块中代码出现异常，catch中异常类型匹配（相同或者父类）

执行catch块代码，执行catch块后边的代码

* 情况3：try块中代码出现异常， catch中异常类型不匹配

不执行catch块代码，不执行catch块后边的代码，程序会中断运行

注意：

* 出现异常后，Java会生成相应的异常对象，Java系统,寻找匹配的catch块，找到后将异常对象付给catch块异常参数。
* 出现异常后，try块中尚未执行的语句不会执行；
* 出现异常后并处理后，catch块后面的语句还会执行

**catch块中如何处理异常**

* 输出用户自定义异常信息

System.err.println("除数不能为零。");

System.err.println("被除数和除数必须是整数。");

* 调用异常对象的方法输出异常信息

toString ( )方法，显示异常的类名和产生异常的原因

void printStackTrace() 输出异常的堆栈信息

String getMessage()返回异常信息描述字符串，printStackTrace()信息一部分

* 继续向上抛出异常

throw e

|  |  |
| --- | --- |
| **异常类型** | **说    明** |
| Exception | 异常层次结构的根类 |
| ArithmeticException | 算术错误情形，如以零作除数 |
| ArrayIndexOutOfBoundsException | 数组下标越界 |
| NullPointerException | 尝试访问 null 对象成员 |
| ClassNotFoundException | 不能加载所需的类 |
| InputMismatchException | 欲得到数据类型与实际输入类型不匹配 |
| IllegalArgumentException | 方法接收到非法参数 |
| ClassCastException | 对象强制类型转换出错 |
| NumberFormatException | 数字格式转换异常，如把"ab"转换成数字 |

问题1：在什么情况下，catch后面的语句是不执行的

情况1：throw e;

情况2：发生的异常和catch中异常类型不匹配

情况3：return

问题2：不管什么情况下，希望某些语句都执行，怎么办

finally语句

#### 【示例4】使用try-catch-finally处理异常

|  |
| --- |
| **package** com.bjsxt.tryDemo;  **import** java.util.InputMismatchException; **import** java.util.Scanner;  **public class** Test5 {  **public static void** main(String[] args) {  Scanner sc =**new** Scanner(System.***in***);  **try** {    System.***out***.println(**"请录入第一个数字"**);  **double** num1 = sc.nextDouble();  System.***out***.println(**"请录入符号"**);  String f = sc.next();  System.***out***.println(**"请录入第二个数字"**);  **double** num2 = sc.nextDouble();  **switch** (f) {  **case "+"**:  System.***out***.println(num1 + num2);  **break**;  **case "/"**:  System.***out***.println(num1 / num2);  **break**;  **case "\*"**:  System.***out***.println(num1 \* num2);  **break**;  **case "-"**:  System.***out***.println(num1 - num2);  **break**;  **default**:  System.***out***.println(**"操作符只能是+-\*/"**);  }    }**catch** (NullPointerException e){  System.***out***.println(**"出异常了"**);   } **finally** {  System.***out***.println(**"finally block"**);  *// 无论程序是否出现异常都会执行的代码 一般用来做资源释放工作* sc.close();  }    System.***out***.println(**"程序执行结束了"**);  }  } |

问题3：return和finally语句的执行顺序

执行return之前的语句----执行finally语句-----执行return

问题4：finally在实际开发中的使用场合

IO流的管理，数据库连接的关闭 socket的关闭

问题5：唯一的例外

System.exit(0); 终止当前正在运行的 Java 虚拟机。

|  |
| --- |
| **package** com.bjsxt.tryDemo;  **public class** Test6 {  *//定义我一个方法 返回a/b的结果* **public static void** main(String[] args) {  **int** c=*chu*(10,0);  System.***out***.println(c);  }   **public static int** chu(**int** a,**int** b){  *//当finally遇见return的时候,finally一定会在return之前先执行  // 要么就别在finally中写return 要么就只在finally里面写return* **int** res=0;  **try**{  res=a/b;  }**catch**(ArithmeticException e){  e.printStackTrace();  }**finally** {  **return** res;  }   }   } |

**问题：一段代码可能会引发多种类型的异常，是否可以分开处理**

* 当引发异常时，会按顺序来查看每个 catch 语句，并执行第一个与异常类型匹配的catch语句
* 执行其中一条 catch 语句后，其后 catch 语句将被忽略
* 在安排catch语句的顺序时，首先应该捕获最特殊的异常， 然后再逐渐一般化，即先子类后父类
* JDK7异常新处理方式：catch(InputMismatchException | ArithmeticException e){}

### 本节作业

1. 使用try-catch-finally处理异常
2. 写出6个常见的异常类型并说明其作用
3. 说明finally语句的相关技能点

## 第二节 异常的抛出

### 2.1 异常处理：throws、throw

**人为抛出异常throw**

* Java异常类对象除在程序执行过程中出现异常时由系统自动生成并抛出，也可根据需要手工创建并抛出。
* 在捕获一个异常前，必须有一段代码先生成异常对象并把它抛出。这个过程我们可以手工做，也可以由JRE来实现，但是他们调用的都是throw子句。
* **注意抛出运行时异常和Checked异常的区别**
  + 抛出Checked异常，该throw语句要么处于try块中，要么方法签名中是由throws抛出
  + 抛出运行时异常，没有以上要求

**声明方法的异常列表throws**

* 当Checked Exception产生时，不一定立刻处理它，可以再把异常Throws出去
* 如果一个方法抛出多个已检查异常，就必须在方法的首部列出所有的异常，之间以逗号隔开
* **子类声明的异常范围不能超过父类声明范围：**父类没有声明异常，子类也不能；不可抛出原有方法抛出异常类的父类或上层类

#### 【示例5】throws的使用：声明方法的异常列表

|  |
| --- |
| **import** java.io.FileInputStream; **import** java.io.FileNotFoundException;  **public class** Test2 {  **public static void** main(String[] args) **throws** InterruptedException, FileNotFoundException {  *showHahaha*();  }  */\*  \* 方法头  \* 访问修饰符  \* 返回值类型  \* 方法名  \* 参数列表  \* 异常列表:代表当前方法在使用时可能会出现的异常,如果要使用该方法,则就要对异常作出处理  \* 方法体{}  \*  \* \*/  // 定义一个方法,每隔一秒向控制台输出一个"哈" 5个* **public static void** showHahaha()**throws** InterruptedException, FileNotFoundException,ArithmeticException {  **for** (**int** i = 0; i < 5; i++) {  System.***out***.println(**"哈"**);  Thread.*sleep*(1000);  }   System.***out***.println(1/0);  FileInputStream fis = **new** FileInputStream(**"e:/aaa.txt"**);  } } |

#### 【示例6】throw的使用 :人为抛出异常

|  |
| --- |
| **public class** Test {  **public static void** main(String[] args) {  Scanner sc=**new** Scanner(System.***in***);  System.***out***.println(**"请录入一个学生的年龄"**);  **int** age = sc.nextInt();  **if**(age >0 && age <100){  System.***out***.println(**"OK "**);  }**else**{  */\*人为抛出异常的关键字 throw  \* 在throw后面必须放一个异常对象  \* \*/* **throw new** ArithmeticException(**"/ 测试人为抛出异常"**);  }  sc.close();  } } |

### 2.2 自定义异常

在程序中，可能会遇到任何标准异常类都没有充分的描述清楚的问题，这种情况下可以创建自己的异常类。

从Exception类或者它的子类派生一个子类即可。习惯上，定义的类应该包含2个构造器：一个是默认构造器，另一个是带有详细信息的构造器

#### 【示例7】自定义异常

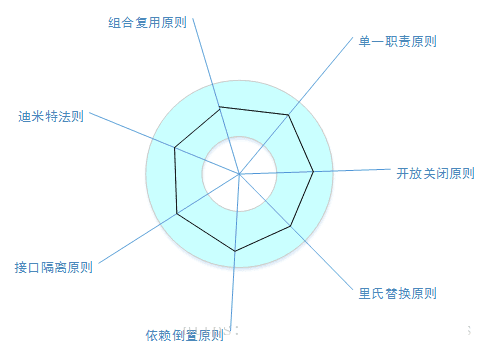
|  |
| --- |
| **package** com.bjsxt.throwDemo;  **import** java.util.Scanner;  **public class** Test {  **public static void** main(String[] args) {  Scanner sc=**new** Scanner(System.***in***);  System.***out***.println(**"请录入一个学生的年龄"**);  **int** age = sc.nextInt();  **if**(age >0 && age <100){  System.***out***.println(**"OK "**);  }**else**{  */\*人为抛出异常的关键字 throw  \* 在throw后面必须放一个异常对象  \* \*/* **throw new** AgeOutofRangeException(**"年龄范围有误:"**+age);  }  sc.close();  } } */\* \* 检查型异常 Exception \* 运行时异常 RuntimeException \* \* \*/* **class** AgeOutofRangeException **extends** RuntimeException {  **public** AgeOutofRangeException(){   }  **public** AgeOutofRangeException(String message){  **super**(message);   } } |

### 本节作业

1. 检查异常和运行时异常的区别
2. 进行求平均分运算，要求人数不能是负数，使用throw抛出异常并进行处理
3. 自定义异常CountMinusException，表示班级人数不能为负数的异常

## 第三节 面向对象设计原则

面向对象设计原则是面向对象设计的基石，面向对象设计质量的依据和保障。一共有7个设计原则。设计模式就是面向对象设计原则的经典应用。



### 3.1 单一职责原则

1. **概念**

单一职责原则 SRP --- Single Responsibility Principle

There should never be more than one reason for a class to change。

应该有且仅有一个原因引起类的变更

系统中的每个类都应只有一个职责，而所有类所关注的就是自身职责的完成

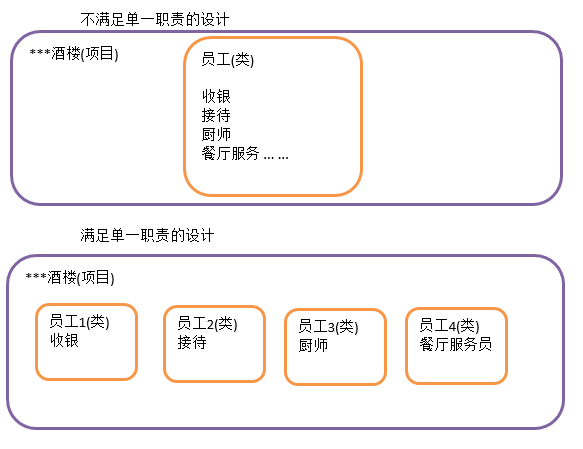
* 职责是指为“变化的原因”
* 如果能想到多个原因去改变一个类，这个类就具有多个职责
* 并不是单一功能原则，并不是每个类只能有一个方法，而是单一“变化的原因”原则
* 如果一个类有多个职责，这些职责就耦合在了一起，当一个职责发生变化时，**可能会影响其它的职责**
* 多个职责耦合在一起，会影响复用性（可能只需要复用该类某一个职责，但该职责跟其它职责耦合在一起，很难分离出来）

1. **好处**

* 单一职责原则的意思就是经常说的“高内聚、低耦合”

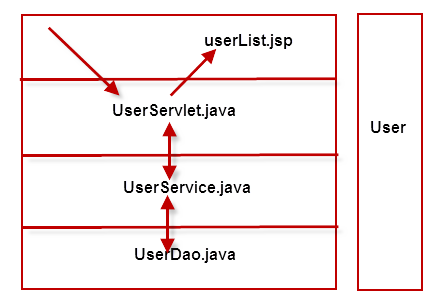
1. **举例**

饭店不同员工职责不同，我们不需要老板一肩扛起所有工作，可以安排不同岗位的员工。即使某个员工离职了，也比较好招聘到新员工。



**JavaEE中的分层框架模式实际上体现了单一职责原则**

对于用户操作，可以分解存储数据的实体类User，完成数据库操作的DAO类UserDao、完成业务操作的业务类UserService、显示用户信息的页面userList.jsp，甚至增加负责调度的UserServlet。



1. **注意**

* 单一职能原则是所有原则中最简单的、最难应用的一个；要注意过犹不及
* “变化的原因”，只有实际发生时才有意义。可能预测到会有多个原因引起这个类的变化，但这仅仅是预测，并没有真的发生，这个类仍可看做具有单一职责，不需要分离

### 3.2 开闭原则OCP

1. **概念**

开闭原则OCP--Open Closed Principle

Software entities should be open for extension,but closed for modification。

软件实体应当对扩展开放，对修改关闭。更通俗翻译：软件系统中的各种组件，如模块（Modules）、类（Classes）以及功能（Functions）等，应该在不修改现有代码的基础上，引入新功能。

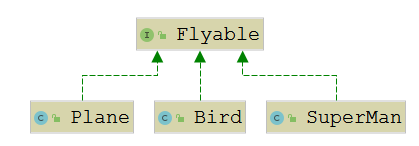
* 实现开闭原则的关键是抽象
* 定义一个抽象层，只规定功能而不提供实现，实现通过定义具体类来完成
* 需求变化时不是通过修改抽象层来完成，而是通过定义抽象层新实现完成
* 通过抽象类及接口，规定了具体类的特征作为抽象层，相对稳定，不需修改，从而满足“对修改关闭”；从抽象类导出的具体类可以改变系统的行为，从而满足“对扩展开放”

1. **好处**

* 通过扩展已有软件系统，可提供新的行为，以满足对软件的新需求，提高了软件系统的适应性和灵活性
* 已有的软件模块，特别是最重要的抽象层模块不能再修改，提高了软件系统的一定的稳定性和延续性
* 这样的设计同时也满足了可复用性与可维护性

1. **举例**

定义飞行接口Flyable，可以有多个实现类。showFly()使用Flyable作为参数，可以传递所有实现类的对象。有了新选手，增加实现类即可。不需要修改showFly()所在的类。

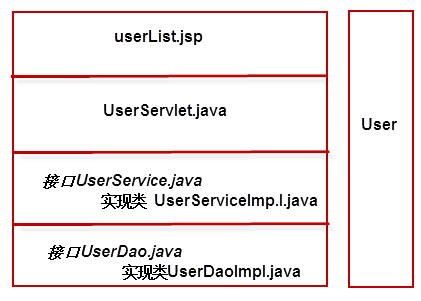


#### 【示例8】开闭原则:

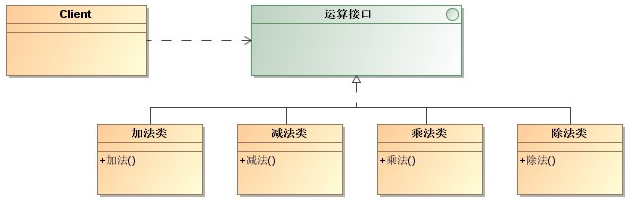
简单工厂模式 >>> 抽象工厂模式

|  |
| --- |
| **package** com.bjsxt.factoryDemo;  **public class** Test {  **public static void** main(String[] args) {  *// 获得工厂对象* Factory factory=**new** ToyotaFactory();  Car car = factory.getCar();  car.run();  } }  **interface** Factory{  Car getCar(); }  **class** AudiFactory **implements** Factory{  @Override  **public** Car getCar() {  **return new** Audi();  } }  **class** BMWFactory **implements** Factory{  @Override  **public** Car getCar() {  **return new** BMW();  } } **class** BYDFactory **implements** Factory{  @Override  **public** Car getCar() {  **return new** BYD();  } }  **class** ToyotaFactory **implements** Factory{  @Override  **public** Car getCar() {  **return new** BYD();  } }  **interface** Car{  **void** run(); }  **class** BMW **implements** Car{  @Override  **public void** run() {  System.***out***.println(**"宝马跑"**);  } } **class** Audi **implements** Car{  @Override  **public void** run() {  System.***out***.println(**"奥迪跑"**);  } } **class** BYD **implements** Car{  @Override  **public void** run() {  System.***out***.println(**"乱跑"**);  } }  **class** Toyota **implements** Car{  @Override  **public void** run() {  System.***out***.println(**"丰田跑"**);  } } |

JavaEE多层模式下,可以定义业务、DAO接口，是和其他层对接的规范。可以增加不同的实现类，而不是去修改原来的实现类。



开发一个计算器类。可以定义一个运算接口，提供基本的加减乘除运算的实现类，如果需要增加取模、开平方、幂等运算，增加新的实现类即可。



1. **总结**

* **实现开闭原则的关键是抽象**
* 抽象层相对稳定不需修改，需求变化后通过重新定义抽象层的新实现来完成
* 即使无法百分之百的做到开闭原则，但朝这个方向努力，可以显著改善一个系统的结构
* 对系统每个部分都肆意地进行抽象也不是一个好主意，应该仅仅对程序中需求频繁变化部分进行抽象。拒绝不成熟的抽象和抽象本身一样重要
* **开闭原则具有理想主义的色彩，它是面向对象设计的终极目标。其他设计原则都可以看作是开闭原则的实现手段或方法**

### 3.3 里氏替代原则

1. **概念**

里氏替代原则LSP--Liskov Substitution Principle

functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.  
所有引用基类的地方必须能透明地使用其子类的对象。

通俗点讲只要父类能出现的地方子类就可以出现，而且调用子类还不产生任何的错误或异常，调用者可能根本就不需要知道是父类还是子类。但是反过来就不成了，有子类出现的地方，父类未必就能适应。

里氏替换法则包含了四层意思：

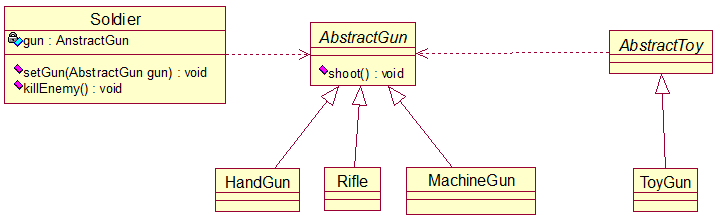
1. 子类必须完全实现父类的方法--重写
2. 子类可以有自己的个性----子类中可以定义特有的方法
3. 覆盖和实现父类的方法时，输出结果（返回值）可以被缩小，但不能被放大
4. **好处**

* 里氏代换原则对如何良好继承提出了衡量依据

1. **举例**

1、企鹅/鸵鸟是鸟吗？

2、玩具手枪是手枪吗？



建议：如果子类不能完整实现父类的方法，或者是父类的某些方法在子类中已经发生“畸变”，那么建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

1. **总结**

* 采用开闭原则必然用到抽象和多态，而这离不开继承。**而里氏代换原则对如何良好继承提出了衡量依据。里氏代换原则是使代码符合开闭原则的一个重要保证。**

### 3.4 依赖倒置原则DIP（接口）

1. **概念**

依赖倒置原则DIP --Dependence Inversion Principle

High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions。翻译过来，包含三层含义：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

抽象：即抽象类或接口，两者是不能够实例化的。

细节：即具体的实现类，实现接口或者继承抽象类的类，可通过关键字new直接被实例化。

**依赖正置就是类间的依赖是实实在在的实现类间的依赖，也就是面向实现编程**

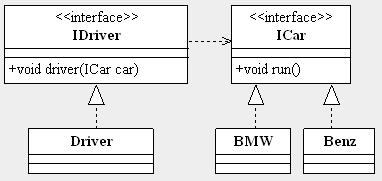
**依赖倒置原则的本质其实就是通过抽象（抽象类或接口）使各个类或模块的实现彼此独立，不相互影响，实现模块间的松耦合 面向要求进行依赖**

1. **好处**

* 采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定性，减少并行开发引起的风险，提高代码的可读性和可维护性。

1. **举例**

* 早期电脑所有硬件整合在一起，一个模块坏全部坏，现在的电脑依赖于插槽（规范），更换cpu、内存、卡等方便。
* 司机驾校培训出来不是只会开宝马、或者只会开奔驰，而是可以开所有汽车。



1. **总结**

* **依赖倒置原则核心就是要面向接口编程，理解了面向接口编程，也就理解了依赖倒置**
* **如果没有实现依赖倒置原则，那么也就意味着开闭原则也无法实现。**
* 结合实际情况使用此原则，要考虑生产和成本，不能生搬硬套

### 3.5 接口分离原则ISP

1. **概念**

接口分离原则ISP-- Interface Segregation Principle

有两种定义

第一种：Clients should not be forced to depend upon interfaces that they don't use.客户端不应该强行依赖它不需要的接口

第二种：The dependency of one class to another one should depend on the smallest possible interface.类间的依赖关系应该建立在最小的接口上

接口隔离原则的含义：

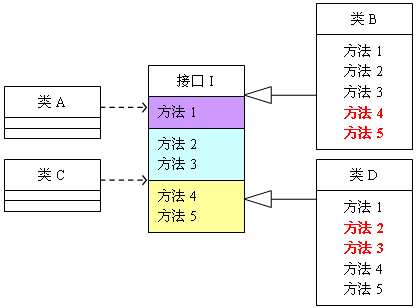
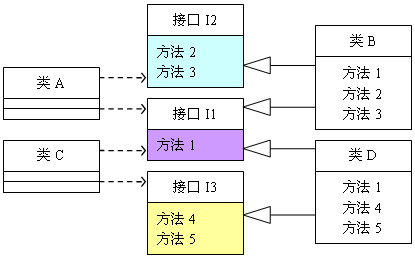
**建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。**也就是说，要为各个类建立专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

1. **好处**

* 防止庞大、臃肿的接口，避免“接口污染”，提高灵活性和可维护性

1. **举例**

* 在车站售票窗口排队的人有买票的，有查开车信息，有退票的，不必排在同一窗口中。多开几窗口，每个窗不同功能，让不同需求的人排在不同窗口，可以节约时间和人力。

1. **总结**

* 注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；
* 接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来很不方便。
* 一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强迫客户依赖于那些它们不用的方法。

### 3.6 迪米特法则LOD

1. **概念**

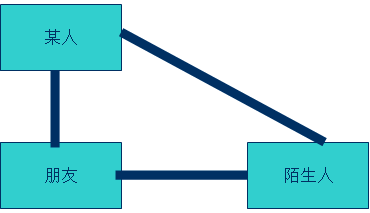
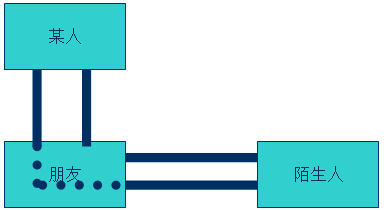
迪米特法则LOD-- Law of Demeter

talk only to your immediate friends

只与你直接的朋友通信（不跟陌生人说话，朋友越少越好）

如果两个类不必彼此直接通信,那么这两个类就不应当发生直接的相互作用.如果其中一个类需要调用另一个类的方法的话,可以通过第三者转发这个调用.

迪米特法则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

不符合LOD 符合LOD

1. **好处**

* 尽量降低类与类之间的耦合

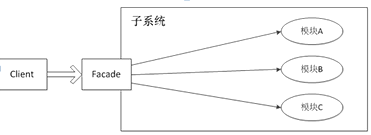
1. **举例**

买房找一个中介即可，不用是认识很多卖房人。去医院看病不用自己知道所有科室的位置，找一个导医即可。



1. **总结**

* 过分使用迪米特原则会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。
* 外观模式和中介者模式都是迪米特法则的应用



### 3.7 合成/聚合复用原则CARP

1. **概念**

合成/聚合复用原则CARP-- Composite Aggregate Reuse Principle

Favor object composition over class inheritance.

优先使用对象组合，而不是类继承.（要尽量使用合成和聚合，尽量不要使用继承）

合成聚合复用原则是指在一个新对象中通过关联关系（组合和聚合关系）使用原来已经存在的一些对象，是之成为新对象的一部分，新的对象通过向这些原来已经具有的对象委派相应的动作或者命令达到复用已有功能的目的。

为何“要尽量使用合成和聚合，尽量不要使用继承”呢？这是因为：

1. 继承复用破坏包装，它把超类的实现细节直接暴露给子类，这违背了信息隐藏的原则；
2. 如果超类发生了改变，那么子类也要发生相应的改变，这就直接导致了类与类之间的高耦合，不利于类的扩展、复用、维护等，也带来了系统僵硬和脆弱的设计。
3. 从超类继承而来的实现是静态的，不可能再运行时间内发生改变，因此没有足够的灵活性。

而是用合成和聚合的时候新对象和已有对象的交互往往是通过接口或者抽象类进行的，就可以很好的避免上面的不足，而且这也可以让每一个新的类专注于实现自己的任务，符合单一职责原则。

1. **好处**

* 非常有利于构建可维护、可复用、可扩展和灵活性好的软件系统。

1. **举例**

#### 【示例9】组合聚合复用原则

素材

|  |
| --- |
| **public class** FatherClass {  **public void** method1(){  System.***out***.println(**"FatherClass method1"**);  }   **public void** method2(){  System.***out***.println(**"FatherClass method2"**);  } }  **public class** SubClass1 **extends** FatherClass{  **public void** method2(){  System.***out***.println(**"SubClass1 method2"**);  } }  **public class** SubClass2 **extends** FatherClass{  **public void** method2(){  System.***out***.println(**"SubClass2 method2"**);  } } |

使用继承：不灵活；编译的时候就知道运行的结果；可以重用父类的代码，但是不能重用父类的子类的代码

|  |
| --- |
| **public class** MyClass **extends** FatherClass{  @Override  **public void** method1() {  System.***out***.println(**"MyClass method1"**);  }   **public static void** main(String[] args) {  MyClass myClass = **new** MyClass();  myClass.method1();  myClass.method2();  } } |

使用组合：灵活。根据注入的具体的子类类型，而出现不同的运行结果。

|  |
| --- |
| **public class** MyClass2 {  **private** FatherClass **clazz**;//使用组合，关联  **public** MyClass2() {  **super**();  }  **public** MyClass2(FatherClass clazz) {  **super**();  **this**.**clazz** = clazz;  }  **public** FatherClass getClazz() {  **return clazz**;  }  **public void** setClazz(FatherClass clazz) {  **this**.**clazz** = clazz;  }  **public void** method1() {  **clazz**.method1();  }  **public void** method3() {  **clazz**.method2();  }  **public static void** main(String[] args) {  MyClass2 clazz = **new** MyClass2();  //FatherClass fc = new FatherClass();  FatherClass fc = **new** SubClass2();//多态  clazz.setClazz(fc);  clazz.method1();  clazz.method3();  }  } |

1. **总结**

* 组合与继承都是重要的复用方法。组合称为黑箱复用，继承称为白箱复用。
* 在OO开发的早期，继承被过度地使用；随着时间发展，人们发现优先使用组合可以获得复用性与简单性更佳的设计
* 并非不要使用继承，并非继承一无是处，而是不要滥用继承。合成/聚合也有自己的缺点

|  |  |
| --- | --- |
| 单一职责原则 | ★★★★ |
| **开闭原则** | **★★★★★** |
| 里氏代换原则 | ★★★ |
| **依赖倒置原则** | **★★★★★** |
| 接口分离原则 | ★★★ |
| 迪米特原则 | ★★ |
| 组合/聚合复用原则 | ★★★★ |

### 本节作业

1. 面向对象设计七大原则是哪些
2. 如何理解开闭原则
3. 为什么组合比继承更灵活

## 第四节 类和类之间的6种关系

### 4.1 认识UML

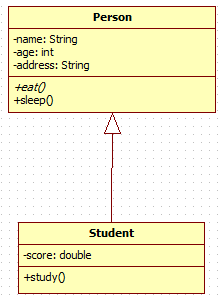
1. Unified Modeling Language (UML) 统一建模语言或标准建模语言
2. 它是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持
3. 它不仅统一了Booch、Rumbaugh和Jacobson的表示方法，而且对其作了进一步的发展，并最终统一为大众所接受的标准建模语言
4. UML定义了10种模型图,对应软件设计开发的不同阶段
   * + 用例图
     + 静态图：类图，包图，对象图。
     + 行为图：状态图和活动图
     + 交互图：顺序图和协作图
     + 实现图：组件图、部署图

### 4.2 认识PowerDesigner

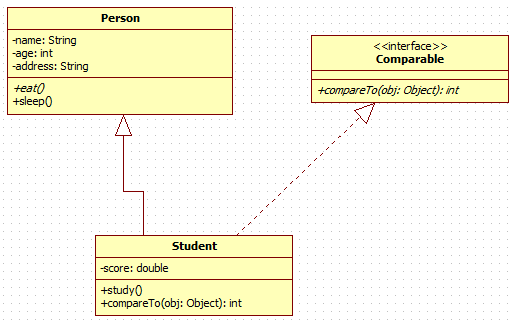
1. 概要
   * + Rational Rose和SyBase PowerDesigner是目前流行两大计算机建模工具。
     + Rose出道是时走的是UML面向对象建模，而后再向数据库建模发展，而PowerDesigner则反其道而行之，它先是一个纯粹的数据库建模工具，后来才向面向对象建模，业务逻辑建模及需求分析建模进军。
     + Rose和PowerDesigner都既可以进行数据库建模，也可以进行面向对象建模，只是存在支持上的偏重而已。
     + Rational被IBM收购。 SDP被Powersoft公司收购，同年Sybase这只大黄雀又吃下了Powersoft这只螳螂。
2. 主要功能
   * **数据库建模**
     + 利用实体-关系图创建“概念数据模型”－CDM。
     + 根据CDM 产生基于特定数据库的“物理数据模型”-PDM。
     + 根据PDM产生为SQL 语句并可以文件形式存储。
     + 由已存在的数据库或者SQL语句反向生成PDM，CDM
   * **面向对象设计（UML建模）**
     + **用例图 类图**  对象图 **时序图** 状态图
     + 协作图 状态图 组件图 部署图
   * **业务流程图(BPM)**

### 4.3类和类的六种关系

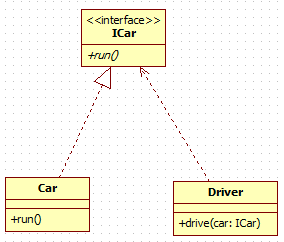
1. **继承关系（泛化关系 Generalization）**
   1. 语义：
      1. 类和子类的关系，接口和子接口的关系；
      2. 一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能
   2. 语法：extends
   3. 符号：
      1. 一条带空心三角箭头的实线，从子类指向父类，或者子接口指向父接口。



1. **实现关系(Realization)**
   1. 语义：
      1. 类和接口之间的关系；
      2. 一个类可以实现多个接口，实现所有接口的功能；体现了规范和实现分离的原则
   2. 语法： implements
   3. 符号
      1. 实现用一条带空心三角箭头的虚线表示，从类指向实现的接口

****

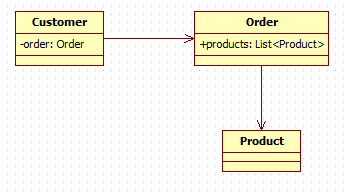
1. **依赖关系(Dependency)**
   1. 语义：一个类A使用到了另一个类B，但是这种使用关系是具有偶然性的、临时性的、非常弱的，但是类B的变化会影响到类A
   2. 语法：类B作为类A的方法的参数（或者局部变量）存在
   3. 符号：
      1. 由类A指向类B的带箭头虚线表示



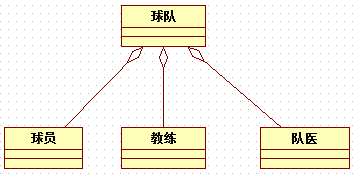
1. **关联关系(Association)**
   1. 语义：
      1. 比依赖关系强，必然的，长期的，强烈的；
      2. 分为单向关联（只是班级中增加了学生）、双向关联（在学生中也添加班级属性）
      3. 分为一对一（学生和学生证）、一对多（班级和学生）、多对多关联（学生和课程）
      4. 有两个类的关联（客户和订单、订单和商品）、还有一个类和自身关联（领导也是员工）

|  |
| --- |
| Class Customer {  Order[] orders;  }  Class Order{  Product[] products;  }  Class Product{  } |

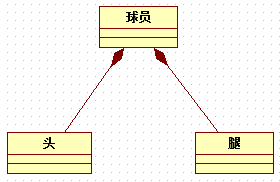
* 1. 语法：类B作为成员变量形成存在于类A中
  2. 符号：
     1. 由类A指向类B的带箭头虚线表示；双向关联可取消两个箭头
     2. 在关联的两端可以标注关联双方的角色和多重性标记



1. **聚合关系(Aggregation)**
   1. 语义：
      1. 关联关系的一种特例
      2. 整体和部分的关系
      3. 整体部分可分离，整体的生命周期和部分的生命周期不同，has-a的关系
      4. 计算机与CPU、公司与员工的关系、班级和学生的关系
   2. 语法：同关联关系
   3. 符号：空心菱形加实线箭头



1. **组合关系(Composition)**
   1. 语义：
      1. 关联关系的一种特例
      2. 整体和部分关系、整体部分不可分离、比聚合更强 ，contains-a的关系
      3. 整体的生命周期和部分的生命周期相同
      4. 人和四肢的关系
   2. 语法：同关联关系
   3. 符号：实心菱形加实线箭头



1. **总结**
   1. 继承和实现一般没有争议
   2. 后四种关系的强弱：组合>聚合>关联>依赖。
   3. 关联和依赖的区别：
      1. 关联关系强、长期
      2. 关联关系是通过属性来实现；依赖关联是通过方法形参或者局部变量实现
   4. 关联、组合/聚合的异同
      1. 相同：都是关联，都是做类的属性
      2. 不同点：组合 /聚合表示的是整体和部分的关系，关联可以表示所有关系
   5. 组合和聚合的异同
      1. 相同：都是关联的特例，都是表示的整体和部分的关系
      2. 不同点：整体部分的生命周期是否相同?组合更强

### 本节作业

1. 类和类的六种关系的符号表示
2. 依赖和关联关系的区别联系
3. 聚合和组合关系的区别联系