

## 1. 纲要

final 关键字

抽象类

接口

抽象类与接口的区别

Object 类

package 和 import

访问权限控制

## 2. 内容

### 1.1、final 关键字

**final** 表示不可改变的含义

- 采用 final 修饰的类不能被继承
- 采用 final 修饰的方法不能被覆盖
- 采用 final 修饰的变量不能被修改
- final 修饰的变量必须显示初始化
- 如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个引用不能再次赋值，但被指向的对象是可以修改的
- 构造方法不能被 final 修饰
- 会影响 JAVA 类的初始化:final 定义的静态常量调用时不会执行 java 的类初始化方法，也就是说不会执行 static 代码块等相关语句，这是由 java 虚拟机规定的。我们不需要了解的很深，有个概念就可以了。

#### 1.1.1、采用 final 修饰的类不能被继承

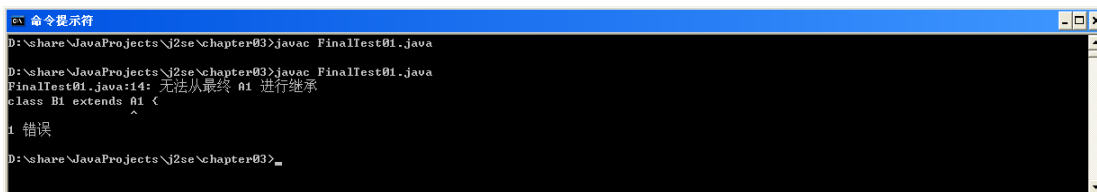
```
public class FinalTest01 {
```

```
public static void main(String[] args) {  
  
}  
}
```

```
final class A1 {  
    public void test1() {  
  
    }  
}
```

//不能继承 A1，因为 A1 采用 final 修饰了

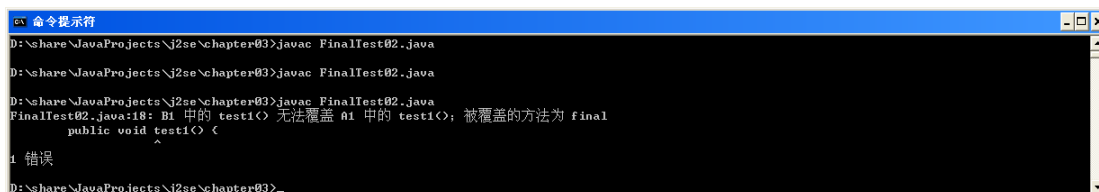
```
class B1 extends A1 {  
  
    public void test2() {  
  
    }  
}
```



### 1.1.2、采用 final 修饰的方法不能被覆盖

```
public class FinalTest02 {  
  
    public static void main(String[] args) {
```

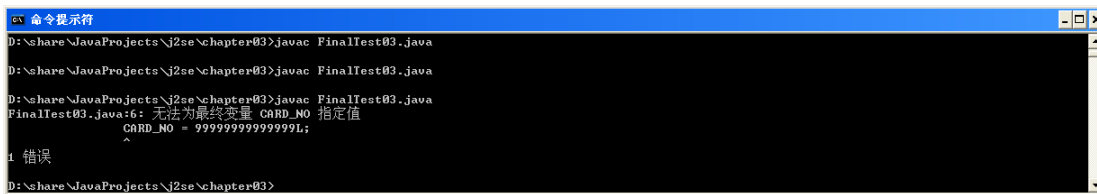
```
}  
}  
  
class A1 {  
  
    public final void test1() {  
  
    }  
}  
  
class B1 extends A1 {  
  
    //覆盖父类的方法，改变其行为  
    //因为父类的方法是 final 修饰的，所以不能覆盖  
    public void test1() {  
  
    }  
  
    public void test2() {  
  
    }  
}  
}
```



```
命令提示符  
D:\share\JavaProjects\j2se\chapter03>javac FinalTest02.java  
D:\share\JavaProjects\j2se\chapter03>javac FinalTest02.java  
D:\share\JavaProjects\j2se\chapter03>javac FinalTest02.java  
FinalTest02.java:18: B1 中的 test1() 无法覆盖 A1 中的 test1(); 被覆盖的方法为 final  
    public void test1() {  
            ^  
1 错误  
D:\share\JavaProjects\j2se\chapter03>
```

### 1.1.3、采用 final 修饰的变量(基本类型)不能被修改

```
public class FinalTest03 {  
  
    private static final long CARD_NO = 878778878787878L;  
  
    public static void main(String[] args) {  
        //不能进行修改，因为 CARD_NO 采用 final 修改了  
        CARD_NO = 999999999999999L;  
    }  
}
```



### 1.1.4、final 修饰的变量必须显示初始化

```
public class FinalTest04 {  
  
    //如果是 final 修饰的变量必须初始化  
    private static final long CARD_NO = 0L;  
  
    public static void main(String[] args) {  
        int i;  
        //局部变量必须初始化  
        //如果不使用可以不初始化  
        System.out.println(i);  
    }  
}
```

### 1.1.5、如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个引用不能再次赋值，但被指向的对象是可以修改的

```
public class FinalTest05 {

    public static void main(String[] args) {

        Person p1 = new Person();

        //可以赋值
        p1.name = "张三";
        System.out.println(p1.name);

        final Person p2 = new Person();
        p2.name = "李四";
        System.out.println(p2.name);

        //不能编译通过
        //p2 采用 final 修饰，主要限制了 p2 指向堆区中的地址不能修改(也就是
        //p2 只能指向一个对象)
        //p2 指向的对象的属性是可以修改的
        p2 = new Person();

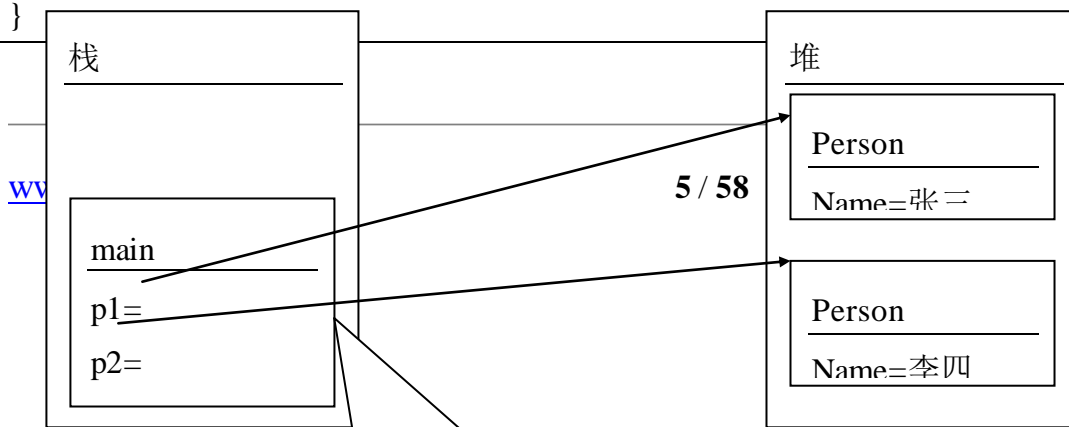
    }

}
```

```
class Person {

    String name;

}
```



## 1.2、抽象类

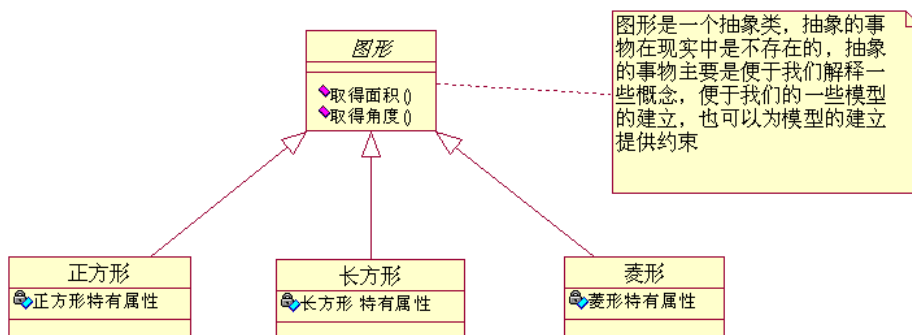
看我们以前示例中的 Person、Student 和 Employee，从我们使用的角度来看主要对 Student 和 Employee 进行实例化，Person 中主要包含了一些公共的属性和方法，而 Person 我们通常不会实例化，所以我们可以把它定义成抽象的：

- 在 java 中采用 **abstract** 关键字定义的类就是抽象类，采用 **abstract** 关键字定义的方法就是抽象方法
- 抽象的方法只需在抽象类中，提供声明，不需要实现
- 如果一个类中含有抽象方法，那么这个类必须定义成抽象类
- 如果这个类是抽象的，那么这个类被子类继承，抽象方法必须被重写。如果在子类中

不复写该抽象方法，那么必须将此类再次声明为抽象类

- 抽象的类是不能实例化的，就像现实世界中人其实是抽象的，张三、李四才是具体的
- 抽象类不能被 **final** 修饰
- 抽象方法不能被 **final** 修饰，因为抽象方法就是被子类实现的

抽象类中可以包含方法实现，可以将一些公共的代码放到抽象类中，另外在抽象类中可以定义一些抽象的方法，这样就会存在一个约束，而子类必须实现我们定义的方法，如：**teacher** 必须实现 **printInfo** 方法，**Student** 也必须实现 **printInfo** 方法，方法名称不能修改，必须为 **printInfo**，这样就能实现多态的机制，有了多态的机制，我们在运行期就可以动态的调用子类的方法。所以在运行期可以灵活的互换实现。



### 1.2.1、采用 **abstract** 声明抽象类

```

public class AbstractTest01 {

    public static void main(String[] args) {

        //不能实例化抽象类
        //抽象类是不存在，抽象类必须有子类继承
        Person p = new Person();

        //以下使用是正确的，因为我们 new 的是具体类
        Person p1 = new Employee();
    }
}
    
```

```
p1.setName("张三");
System.out.println(p1.getName());

}
}
```

//采用 **abstract** 定义抽象类

//在抽象类中可以定义一些子类公共的方法或属性

//这样子类就可以直接继承下来使用了，而不需要每个

//子类重复定义

```
abstract class Person {
```

```
    private String name;
```

```
    public void setName(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
```

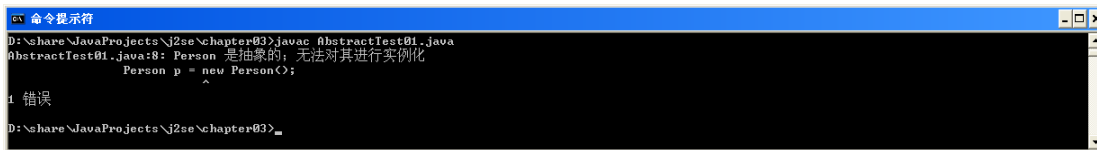
//此方法各个子类都可以使用

```
    public void commonMethod1() {
        System.out.println("-----commonMethod1 -----");
    }
}
```

```
class Employee extends Person {
```



```
}  
  
class Student extends Person {  
  
}
```



### 1.2.2、抽象的方法只需在抽象类中，提供声明，不需要实现，起到了一个强制的约束作用，要求子类必须实现

```
public class AbstractTest02 {  
  
    public static void main(String[] args) {  
        //Person p = new Employee();  
        //Person p = new Student();  
        //Person p = new Person();  
        p.setName("张三");  
        p.printInfo();  
    }  
}  
  
abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;
```

```
}

public String getName() {
    return name;
}

//此方法各个子类都可以使用
public void commonMethod1() {
    System.out.println("-----commonMethod1-----");
}

//public void printInfo() {
//    System.out.println("-----Person.printInfo()-----");
//}

//采用 abstract 定义抽象方法
//如果有一个方法为抽象的，那么此类必须为抽象的
//如果一个类是抽象的，并不要求具有抽象的方法
public abstract void printInfo();
}

class Employee extends Person {

    //必须实现抽象的方法
    public void printInfo() {
        System.out.println("Employee.printInfo()");
    }
}

class Student extends Person {
```

```
//必须实现抽象的方法
```

```
public void printInfo() {  
    System.out.println("Student.printInfo()");  
}  
}
```

1.2.3、如果这个类是抽象的，那么这个类被子类继承，抽象方法必须被覆盖。如果在子类中不覆盖该抽象方法，那么必须将此方法再次声明为抽象方法

```
public class AbstractTest03 {  
  
    public static void main(String[] args) {  
        //此时不能再 new Employee 了  
        Person p = new Employee();  
    }  
}
```

```
abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {
```

```
        return name;
    }

    //此方法各个子类都可以使用
    public void commonMethod1() {
        System.out.println("-----commonMethod1-----");
    }

    //采用 abstract 定义抽象方法
    public abstract void printInfo();
}

abstract class Employee extends Person {

    //再次声明该方法为抽象的
    public abstract void printInfo();
}

class Student extends Person {

    //实现抽象的方法
    public void printInfo() {
        System.out.println("Student.printInfo()");
    }
}
```

#### 1.2.4、抽象类不能被 final 修饰

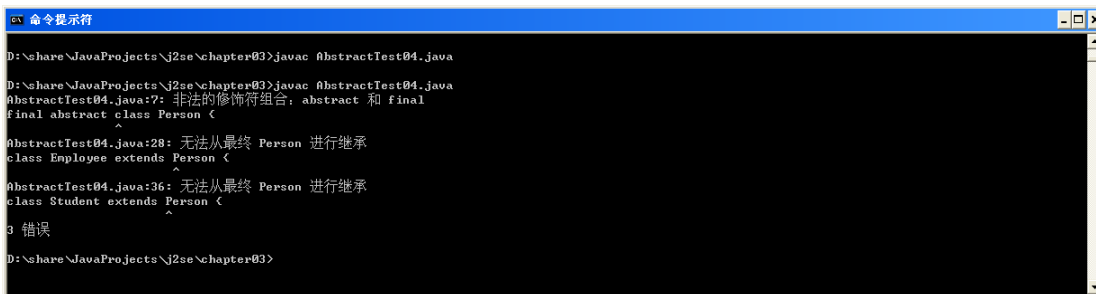
```
public class AbstractTest04 {
```

```
public static void main(String[] args) {  
    }  
}  
  
//不能采用 final 修改抽象类  
//两个关键字是矛盾的  
final abstract class Person {  
  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    //此方法各个子类都可以使用  
    public void commonMethod1() {  
        System.out.println("-----commonMethod1-----");  
    }  
  
    //采用 abstract 定义抽象方法  
    public abstract void printInfo();  
}  
  
class Employee extends Person {
```

```
//实现抽象的方法
public void printInfo() {
    System.out.println("Student.printInfo()");
}
}

class Student extends Person {

    //实现抽象的方法
    public void printInfo() {
        System.out.println("Student.printInfo()");
    }
}
```



### 1.2.5、抽象方法不能被 final 修饰

```
public class AbstractTest05 {

    public static void main(String[] args) {
    }
}

abstract class Person {
```

```
private String name;

public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

//此方法各个子类都可以使用
public void commonMethod1() {
    System.out.println("-----commonMethod1-----");
}

//不能采用 final 修饰抽象的方法
//这两个关键字存在矛盾
public final abstract void printInfo();
}

class Employee extends Person {

    //实现抽象的方法
    public void printInfo() {
        System.out.println("Student.printInfo()");
    }
}

class Student extends Person {
```

//实现抽象的方法

```
public void printInfo() {  
    System.out.println("Student.printInfo()");  
}  
}
```



```
命令提示符  
D:\share\JavaProjects\j2se\chapter03>javac AbstractTest05.java  
AbstractTest05.java:25: 非法的修饰符组合, abstract 和 final  
    public final abstract void printInfo();  
                        ^  
AbstractTest05.java:31: Employee 中的 printInfo() 无法覆盖 Person 中的 printInfo(); 被覆盖的方法为 final  
    public void printInfo() {  
                        ^  
AbstractTest05.java:39: Student 中的 printInfo() 无法覆盖 Person 中的 printInfo(); 被覆盖的方法为 final  
    public void printInfo() {  
                        ^  
3 错误  
D:\share\JavaProjects\j2se\chapter03>
```

## 1.2.6、抽象类中可以没有抽象方法（参见 1.16.1）

## 1.3、接口(行为)

接口我们可以**看作**是**抽象类的一种特殊情况**，在接口中只能定义抽象的方法和常量

- 1) 在 java 中接口采用 interface 声明
- 2) 接口中的方法**默认**都是 **public abstract** 的，不能更改
- 3) 接口中的变量**默认**都是 **public static final** 类型的，不能更改，所以必须显示的初始化
- 4) 接口不能被实例化，接口中没有构造函数的概念
- 5) 接口之间可以继承，但接口之间不能实现
- 6) 接口中的方法只能通过类来实现，通过 **implements** 关键字
- 7) 如果一个类实现了接口，那么接口中所有的方法必须实现
- 8) 一类可以实现多个接口

### 1.3.1、接口中的方法默认都是 public abstract 的，不能更改

```
public class InterfaceTest01 {
```



```
public static void main(String[] args) {  
    }  
}  
  
//采用 interface 定义接口  
//定义功能，没有实现  
//实现委托给类实现  
interface StudentManager {  
  
    //正确，默认 public abstract 等同 public abstract void addStudent(int id, String  
name);  
    public void addStudent(int id, String name);  
  
    //正确  
    //public abstract void addStudent(int id, String name);  
  
    //正确，可以加入 public 修饰符，此种写法较多  
    public void delStudent(int id);  
  
    //正确，可以加入 abstract，这种写法比较少  
    public abstract void modifyStudent(int id, String name);  
  
    //编译错误，因为接口就是让其他人实现  
    //采用 private 就和接口原本的定义产生矛盾了  
    private String findStudentById(int id);  
}
```

### 1.3.2、接口中的变量是 `public static final` 类型的，不能更改，所以必须显示的初始化

```
public class InterfaceTest02 {

    public static void main(String[] args) {

        //不能修改，因为 final 的
        //StudentManager.YES = "abc";

        System.out.println(StudentManager.YES);
    }
}

interface StudentManager {

    //正确，默认加入 public static final
    String YES = "yes";

    //正确，开发中一般就按照下面的方式进行声明
    public static final String NO = "no";

    //错误，必须赋值，因为是 final 的
    //int ON;

    //错误，不能采用 private 声明
    private static final int OFF = -1;
}
```

### 1.3.3、接口不能被实例化，接口中没有构造函数的概念

```
public class InterfaceTest03 {  
  
    public static void main(String[] args) {  
  
        //接口是抽象类的一种特例，只能定义方法和变量，没有实现  
        //所以不能实例化  
        StudentManager studentManager = new StudentManager();  
    }  
}  
  
interface StudentManager {  
  
    public void addStudent(int id, String name);  
}
```

### 1.3.4、接口之间可以继承，但接口之间不能实现

```
public class InterfaceTest04 {  
  
    public static void main(String[] args) {  
    }  
}  
  
interface inter1 {  
    public void method1();  
  
    public void method2();  
}
```

```
interface inter2 {  
    public void method3();  
}  
  
//接口可以继承  
interface inter3 extends inter1 {  
  
    public void method4();  
}  
  
//接口不能实现接口  
//接口只能被类实现  
interface inter4 implements inter2 {  
    public void method3();  
}
```

### 1.3.5、如果一个类实现了接口，那么接口中所有的方法必须实现

```
public class InterfaceTest05 {  
  
    public static void main(String[] args) {  
        //Iter1Impl 实现了 Inter1 接口  
        //所以它是 Inter1 类型的产品  
        //所以可以赋值  
        Inter1 iter1 = new Iter1Impl();  
        iter1.method1();  
  
        //Iter1Impl123 实现了 Inter1 接口  
        //所以它是 Inter1 类型的产品
```

```
//所以可以赋值
```

```
iter1 = new Iter1Impl123();
```

```
iter1.method1();
```

```
//可以直接采用 Iter1Impl 来声明类型
```

```
//这种方式存在问题
```

```
//不利于互换，因为面向具体编程了
```

```
Iter1Impl iter1Impl = new Iter1Impl();
```

```
iter1Impl.method1();
```

```
//不能直接赋值给 iter1Impl
```

```
//因为 Iter1Impl123 不是 Iter1Impl 类型
```

```
//iter1Impl = new Iter1Impl123();
```

```
//iter1Impl.method1();
```

```
}
```

```
}
```

```
//接口中的方法必须全部实现
```

```
class Iter1Impl implements Inter1 {
```

```
    public void method1() {
```

```
        System.out.println("method1");
```

```
    }
```

```
    public void method2() {
```

```
        System.out.println("method2");
```

```
    }
```

```
    public void method3() {
```

```
        System.out.println("method3");
    }
}

class Iter1Impl123 implements Inter1 {

    public void method1() {
        System.out.println("method1_123");
    }

    public void method2() {
        System.out.println("method2_123");
    }

    public void method3() {
        System.out.println("method3_123");
    }
}

abstract class Iter1Impl456 implements Inter1 {

    public void method1() {
        System.out.println("method1_123");
    }

    public void method2() {
        System.out.println("method2_123");
    }

    //再次声明成抽象方法
```

```
public abstract void method3();  
}
```

//定义接口

```
interface Inter1 {  
  
    public void method1();  
  
    public void method2();  
  
    public void method3();  
}
```

### 1.3.6、一类可以实现多个接口

```
public class InterfaceTest06 {  
  
    public static void main(String[] args) {  
  
        //可以采用 Inter1 定义  
        Inter1 inter1 = new InterImpl();  
        inter1.method1();  
  
        //可以采用 Inter1 定义  
        Inter2 inter2 = new InterImpl();  
        inter2.method2();  
    }  
}
```

```
//可以采用 Inter1 定义
Inter3 inter3 = new InterImpl();
inter3.method3();
}
}

//实现多个接口，采用逗号隔开
//这样这个类就拥有了多种类型
//等同于现实中的多继承
//所以采用 java 中的接口可以实现多继承
//把接口粒度划分细了，主要使功能定义的含义更明确
//可以采用一个大的接口定义所有功能，替代多个小的接口，
//但这样定义功能不明确，粒度太粗了
class InterImpl implements Inter1, Inter2, Inter3 {

    public void method1() {
        System.out.println("----method1-----");
    }

    public void method2() {
        System.out.println("----method2-----");
    }

    public void method3() {
        System.out.println("----method3-----");
    }
}

interface Inter1 {
```



```
    public void method1();
}

interface Inter2 {

    public void method2();
}

interface Inter3 {

    public void method3();
}

/*
interface Inter {

    public void method1();

    public void method2();

    public void method3();
}
*/
```

## 1.4、接口的进一步应用

在 java 中接口其实描述了类需要做的事情，类要遵循接口的定义来做事，使用接口到底有什么本质的好处？可以归纳为两点：

- 采用接口明确的声明了它所能提供的服务
- 解决了 Java 单继承的问题
- 实现了可接插性（重要）

示例：完成学生信息的增删改操作，系统要求适用于多个数据库，如：适用于 Oracle 和 MySQL；

- 第一种方案，不使用接口，每个数据库实现一个类：

StudentOracleImpl
<ul style="list-style-type: none"><li>add(int id, String name)</li><li>del(int id)</li><li>modify(int id, String name)</li></ul>

StudentMysqlImpl
<ul style="list-style-type: none"><li>addStudent(int id, String name)</li><li>deleteStudent(int id)</li><li>updateStudent(int id, String name)</li></ul>

//Oracle 的实现

```
public class StudentOracleImpl {  
  
    public void add(int id, String name) {  
        System.out.println("StudentOracleImpl.add()");  
    }  
  
    public void del(int id) {  
        System.out.println("StudentOracleImpl.del()");  
    }  
  
    public void modify(int id, String name) {  
        System.out.println("StudentOracleImpl.modify()");  
    }  
}
```

需求发生了变化了，客户需要将数据移植 Mysql 上

////Mysql 的实现

```
public class StudentMysqlImpl {  
  
    public void addStudent(int id, String name) {  
        System.out.println("StudentMysqlImpl.addStudent()");  
    }  
  
    public void deleteStudent(int id) {
```

```
        System.out.println("StudentMysqlImpl.deleteStudent()");
    }

    public void udpateStudent(int id, String name) {
        System.out.println("StudentMysqlImpl.udpateStudent()");
    }
}
```

调用以上两个类完成向 Oracle 数据库和 Mysql 数据存储数据

```
public class StudentManager {

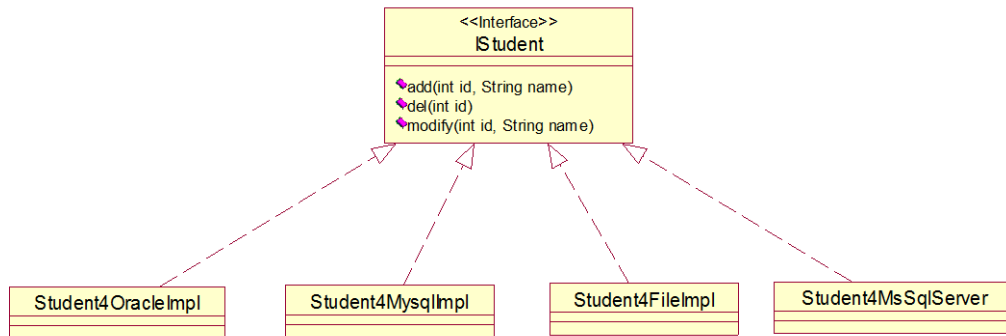
    public static void main(String[] args) {
        //对 Oracle 数据库的支持
        /*
        StudentOracleImpl studentOracleImpl = new StudentOracleImpl();
        studentOracleImpl.add(1, "张三");
        studentOracleImpl.del(1);
        studentOracleImpl.modify(1, "张三");
        */

        //需要支持 Mysql 数据库
        StudentMysqlImpl studentMysqlImpl = new StudentMysqlImpl();
        studentMysqlImpl.addStudent(1, "张三");
        studentMysqlImpl.deleteStudent(1);
        studentMysqlImpl.udpateStudent(1, "张三");
    }
}
```

以上代码不能灵活的适应需求，当需求发生改变需要改动的代码量太大，这样可能会导致代码的冗余，另外可能会导致项目的失败，为什么会导致这个问题，在开发中没有考虑到程序的扩展性，就是一味的实现，这样做程序是不行的，所以大的项目比较追求程序扩展性，有了扩展性才可以更好的适应需求。

- 第二种方案，使用接口

UML，统一建模语言



```
public class Student4OracleImpl implements ISStudent {

    public void add(int id, String name) {
        System.out.println("Student4OracleImpl.add()");
    }

    public void del(int id) {
        System.out.println("Student4OracleImpl.del()");
    }

    public void modify(int id, String name) {
        System.out.println("Student4OracleImpl.modify()");
    }

}
```

```
public class Student4MysqlImpl implements ISStudent {

    public void add(int id, String name) {
        System.out.println("Student4MysqlImpl.add()");
    }

}
```

```
public void del(int id) {  
    System.out.println("Student4MysqlImpl.del()");  
}  
  
public void modify(int id, String name) {  
    System.out.println("Student4MysqlImpl.modify()");  
}  
}
```

```
public class StudentService {  
  
    public static void main(String[] args) {  
        /*  
        IStudent istudent = new Student4OracleImpl();  
        IStudent istudent = new Student4MysqlImpl();  
        istudent.add(1, "张三");  
        istudent.del(1);  
        istudent.modify(1, "张三");  
        */  
        //IStudent istudent = new Student4OracleImpl();  
        //IStudent istudent = new Student4MysqlImpl();  
        //doCrud(istudent);  
        //doCrud(new Student4OracleImpl());  
        //doCrud(new Student4MysqlImpl());  
  
        //doCrud(new Student4OracleImpl());  
        doCrud(new Student4MysqlImpl());  
  
    }  
}
```

//此种写法没有依赖具体的实现

//而只依赖的抽象，就像你的手机电池一样：你的手机只依赖电池（电池是一个抽象的事物），

//而不依赖某个厂家的电池(某个厂家的电池就是具体的事物了)

//因为你依赖了抽象的事物，每个抽象的事物都有不同的实现

//这样你就可以利用多态的机制完成动态绑定，进行互换，是程序具有较高的灵活

//我们尽量遵循面向接口（抽象）编程,而不要面向实现编程

```
public static void doCrud(IStudent istudent) {  
    istudent.add(1, "张三");  
    istudent.del(1);  
    istudent.modify(1, "张三");  
}
```

//以下写法不具有扩展性

//因为它依赖了具体的实现

//建议不要采用此种方法，此种方案是面向实现编程，就依赖于具体的东西了

/\*

```
public static void doCrud(Student4OracleImpl istudent) {  
    istudent.add(1, "张三");  
    istudent.del(1);  
    istudent.modify(1, "张三");  
}  
*/
```

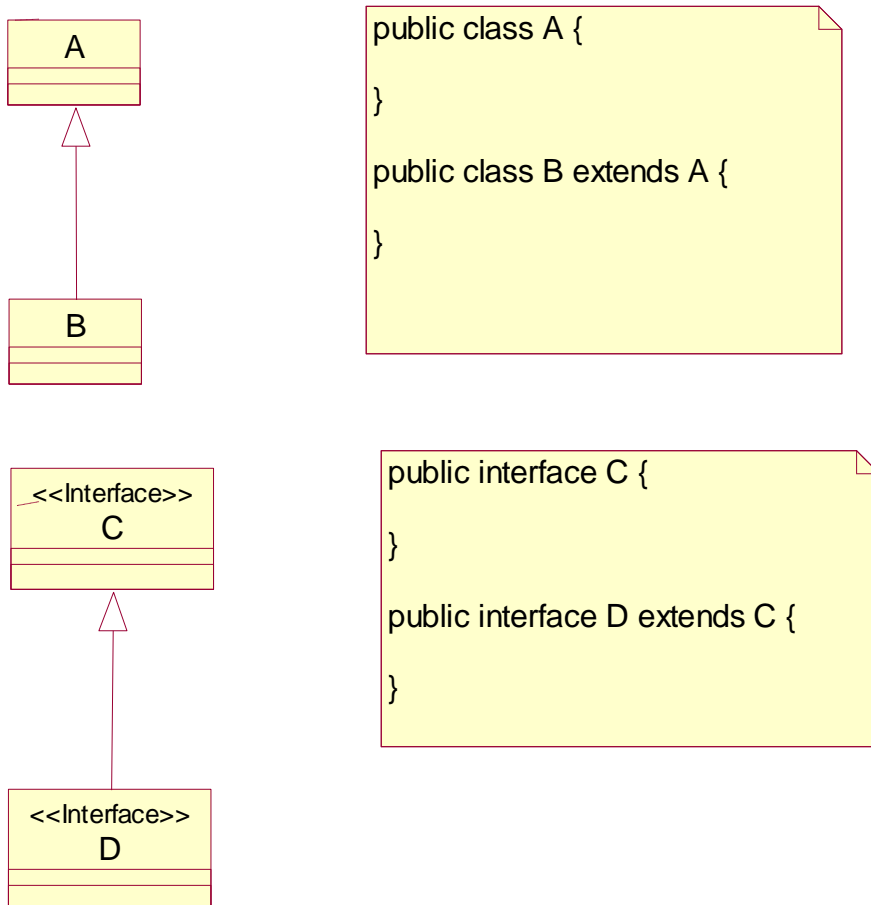
}

## 1.5、接口和抽象类的区别？

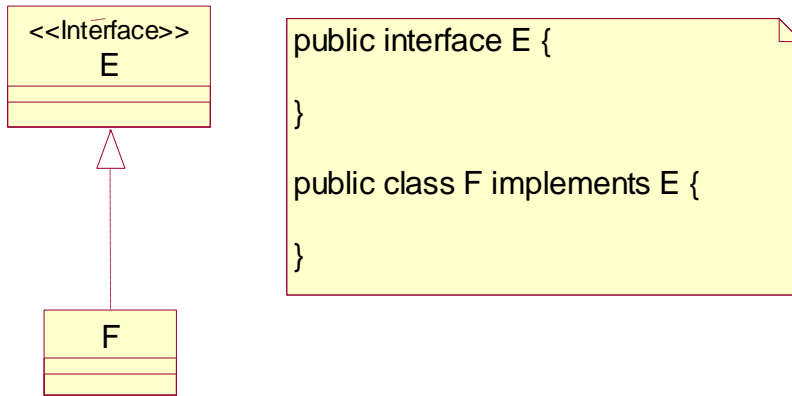
- a) 接口描述了方法的特征，不给出实现，一方面解决 java 的单继承问题，实现了强大的可接插性
- b) 抽象类提供了部分实现，抽象类是不能实例化的，抽象类的存在主要是可以把公共的代码移植到抽象类中
- c) 面向接口编程，而不要面向具体编程（面向抽象编程，而不要面向具体编程）
- d) 优先选择接口（因为继承抽象类后，此类将无法再继承，所以会丧失此类的灵活性）

## 1.6、类之间的关系

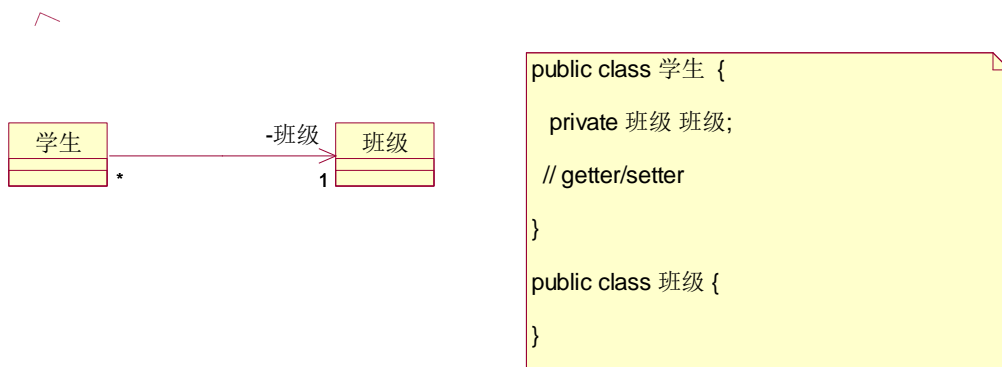
1. 泛化关系，类和类之间的继承关系及接口与接口之间的继承关系



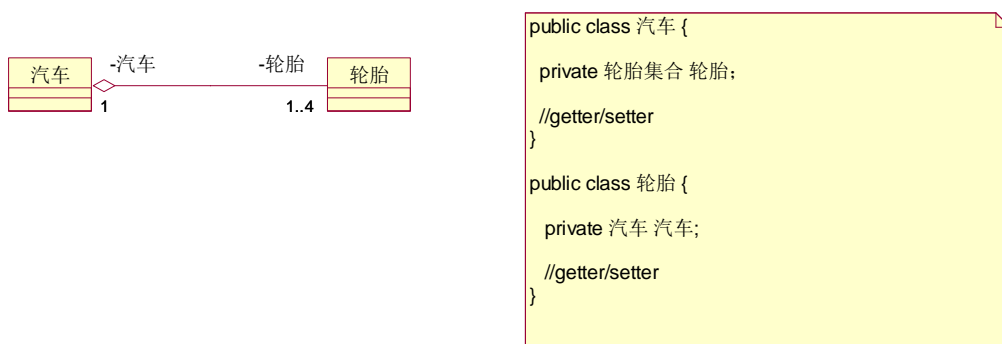
2. 实现关系，类对接口的实现



3. 关联关系，类与类之间的连接，一个类可以知道另一个类的属性和方法，在 java 语言中使用成员变量体现



4. 聚合关系，是关联关系的一种，是较强的关联关系，是整体和部分的关系，如：汽车和轮胎，它与关联关系不同，关联关系的类处在同一个层次上，而聚合关系的类处在不平等的层次上，一个代表整体，一个代表部分，在 java 语言中使用实例变量体现



5. 合成关系，是关系的一种，比聚合关系强的关联关系，如：人和四肢，整体对象决定部分对象的生命周期，部分对象每一时刻只与一个对象发生合成关系，在 java 语言中使用实例变量体现





```

public class 人 {
    private 四肢集合 四肢;
    //getter/setter
}

public class 四肢 {
    private 人 人;
    //getter/setter
}
    
```

6. 依赖关系，依赖关系是比关联关系弱的关系，在java语言中体现为返回值，参数，局部变量和静态方法调用



```

public class Test {
    public static void main(String[] args) {
        Person person = new Person();
    }
}

class Person {
}
    
```

## 1.7、is-a、is-like-a、has-a

### ● Is-a

```

public class A {
    public void method1() {}
}

public class B extends A {
    public void method1() {}
}
    
```

### ● is-like-a

```
public interface I {  
    public void method1();  
}  
  
public class A implements I {  
    public void method1() {  
        //实现  
    }  
}
```

● has-a

```
public class A {  
    private B b;  
  
}  
  
public class B {  
  
}
```

## 1.8、Object 类

- a) Object 类是所有 Java 类的根基类
  - b) 如果在类的声明中未使用 extends 关键字指明其基类，则默认基类为 Object 类
- 如：

```
public class User {  
    .....  
}  
相当于  
public class User extends Object {
```

```
.....  
}
```

### 1.8.1、toString()

返回该对象的字符串表示。通常 `toString` 方法会返回一个“以文本方式表示”此对象的字符串，`Object` 类的 `toString` 方法返回一个字符串，该字符串由类名加标记@和此对象哈希码的无符号十六进制表示组成，`Object` 类 `toString` 源代码如下：

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

在进行 `String` 与其它类型数据的连接操作时，如：

`System.out.println(student);`，它自动调用该对象的 `toString()` 方法

【代码示例】

```
public class ToStringTest01 {  
  
    public static void main(String[] args) {  
        int i = 100;  
        System.out.println(100);  
  
        Person person = new Person();  
        person.id = 200;  
        person.name = "张三";  
  
        //会输出 Person@757aef  
        //因为它调用了 Object 中的 toString 方法  
        //输出的格式不友好，无法看懂  
        System.out.println(person);  
  
    }  
}
```

```
//class Person extends Object { //和以下写法等同
class Person{

    int id;

    String name;

}
```

【代码示例】，覆盖 Person 中的 toString 方法

```
public class ToStringTest02 {

    public static void main(String[] args) {
        Person person = new Person();
        person.id = 200;
        person.name = "张三";

        //System.out.println(person.toString());

        //输出结果为: {id=200,name=张三}
        //因为 println 方法没有带 Person 参数的
        //而 Person 是 Object，所以他会调用 println(Object x)方法
        //这样就是产生 object 对其子类 Person 的指向，而在 Person 中
        //覆盖了父类 Object 的 toString 方法，所以运行时会动态绑定
        //Person 中的 toString 方法，所以将会按照我们的需求进行输出
        System.out.println(person);

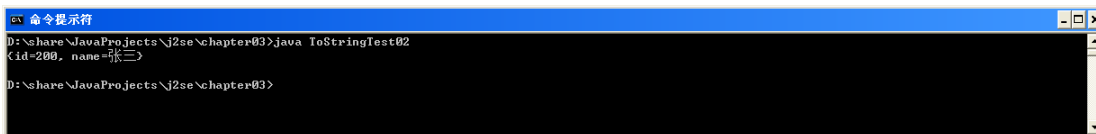
    }
}
```

```
//class Person extends Object { //和以下写法等同
class Person{

    int id;

    String name;

    public String toString() {
        return "{id=" + id + ", name=" + name + "}";
    }
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter03>java ToStringTest02
{id=200, name=张三}
D:\share\JavaProjects\j2se\chapter03>
```

## 1.8.2、finalize

垃圾回收器（Garbage Collection），也叫 **GC**，垃圾回收器主要有以下特点：

- 当对象不再被程序使用时，垃圾回收器将会将其回收
- 垃圾回收是在后台运行的，我们无法命令垃圾回收器**马上**回收资源，但是我们可以告诉他，尽快回收资源（**System.gc** 和 **Runtime.getRuntime().gc()**）
- 垃圾回收器在回收某个对象的时候，首先会调用该对象的 **finalize** 方法
- **GC 主要针对堆内存**
- **单例模式的缺点**

当垃圾收集器将要收集某个垃圾对象时将会调用 **finalize**，建议不要使用此方法，因为此方法的运行时间不确定，如果执行此方法出现错误，程序不会报告，仍然继续运行

```
public class FinalizeTest01 {
```

```
public static void main(String[] args) {  
    Person person = new Person();  
    person.id = 1000;  
    person.name = "张三";  
  
    //将 person 设置为 null 表示，person 不再执行堆中的对象  
    //那么此时堆中的对象就是垃圾对象  
    //垃圾收集（GC）就会收集此对象  
    //GC 不会马上收集，收集时间不确定  
    //但是我们可以告诉 GC，马上来收集垃圾，但也不确定，会马上来  
    //也许不会来  
    person = null;  
  
    //通知垃圾收集器，来收集垃圾  
    System.gc();  
    /*  
    try {  
        Thread.sleep(5000);  
    } catch (Exception e) {  
    }  
    */  
}  
  
}  
  
class Person {  
  
    int id;  
  
    String name;
```

```
//此方法垃圾收集器会调用
public void finalize() throws Throwable {
    System.out.println("Person.finalize()");
}
}
```

注意以下写法

```
public class FinalizeTest02 {

    public static void main(String[] args) {
        method1();
    }

    private static void method1() {
        Person person = new Person();
        person.id = 1000;
        person.name = "张三";

        //这种写法没有多大的意义，
        //执行完成方法，所有的局部变量的生命周期全部结束
        //所以堆区中的对象就变成垃圾了（因为没有引用指向对象了）
        //person = null;
    }
}

class Person{

    int id;

    String name;
```

```
public void finalize() throws Throwable {  
    System.out.println("Person.finalize()");  
}  
}
```

### 1.8.3、==与 equals 方法

#### ■ 等号 “==”

等号可以比较基本类型和引用类型，等号比较的是值，特别是比较引用类型，比较的是引用的内存地址

```
public class EqualsTest01 {  
  
    public static void main(String[] args) {  
        int a = 100;  
        int b = 100;  
  
        //可以成功比较  
        //采用等号比较基本它比较的就是具体的值  
        System.out.println((a == b)?"a==b":"a!=b");  
  
        Person p1 = new Person();  
        p1.id = 1001;  
        p1.name = "张三";  
  
        Person p2 = new Person();  
        p2.id = 1001;  
        p2.name="张三";  
  
        //输出为 p1!=p2  
        //采用等号比较引用类型比较的是引用类型的地址（地址也是值）
```



//这个是不符合我们的比较需求的

//我们比较的应该是对象的具体属性，如：id 相等，或 id 和 name 相等

```
System.out.println((p1 == p2)?"p1==p2":"p1!=p2");
```

```
Person p3 = p1;
```

//输出为 p1==p3

//因为 p1 和 p3 指向的是一个对象，所以地址一样

//所以采用等号比较引用类型比较的是地址

```
System.out.println((p1 == p3)?"p1==p3":"p1!=p3");
```

```
String s1 = "abc";
```

```
String s2 = "abc";
```

//输出 s1==s2

```
System.out.println((s1==s2)?"s1==s2":"s1!=s2");
```

```
}
```

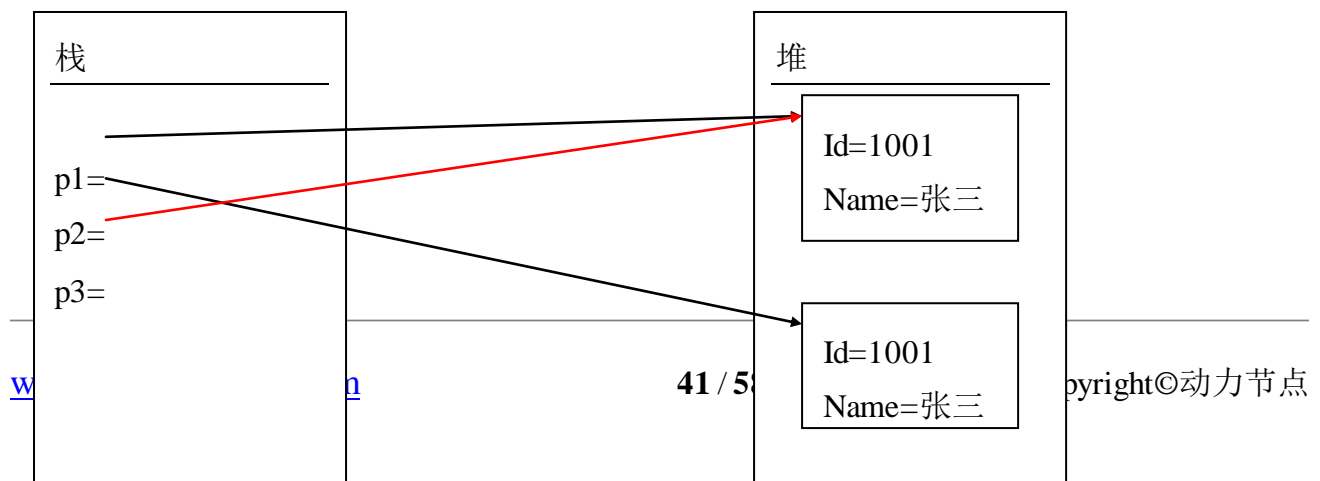
```
}
```

```
class Person{
```

```
    int id;
```

```
    String name;
```

```
}
```



## ■ 采用 equals 比较两个对象是否相等

```
public class EqualsTest02 {  
  
    public static void main(String[] args) {  
        String s1 = "abc";  
        String s2 = "abc";  
        //输出 s1==s2  
        System.out.println((s1==s2)?"s1==s2":"s1!=s2");  
  
        String s3 = new String("abc");  
        String s4 = new String("abc");  
        System.out.println((s3==s4)?"s3==s4":"s3!=s4");  
  
        //输出 s3 等于 s4，所以确定 string 的 equals 比较的是具体的内容  
        System.out.println(s3.equals(s4)?"s3 等于 s4":"s3 不等于 s4");  
  
        Person p1 = new Person();  
        p1.id = 1001;  
        p1.name = "张三";  
  
        Person p2 = new Person();
```

```
p2.id = 1001;  
p2.name="张三";
```

//输出: p1 不等于 p2

//因为它默认调用的是 Object 的 equals 方法

//而 Object 的 equals 方法默认比较的就是地址,Object 的 equals 方法代码如下:

```
//    public boolean equals(Object obj) {  
//    return (this == obj);  
//    }
```

//如果不准备调用父类的 equals 方法,那么必须覆盖父类的 equals 方法行为

```
System.out.println(p1.equals(p2)? "p1 等于 p2": "p1 不等于 p2");
```

```
}
```

```
}
```

```
class Person{
```

```
    int id;
```

```
    String name;
```

```
}
```

在进一步完善

```
public class EqualsTest03 {
```

```
    public static void main(String[] args) {
```

```
        Person p1 = new Person();
```

```
        p1.id = 1001;
```

```
        p1.name = "张三";
```

```
    Person p2 = new Person();
    p2.id = 1001;
    p2.name="张三";

    System.out.println(p1.equals(p2)? "p1 等于 p2": "p1 不等于 p2");

}
}

class Person{

    int id;

    String name;

    //覆盖父类的方法
    //加入我们自己的比较规则
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        //确定比较类型为 person
        //同一类型，才具有可比性
        if (obj instanceof Person) {
            //强制转换，必须实现知道该类型是什么
            Person p = (Person)obj;
            //如果 id 相等就认为相等
            if (this.id == p.id) {
                return true;
            }
        }
    }
}
```

```
        }  
    }  
    return false;  
}  
  
}
```

以上输出完全正确，因为执行了我们自定义的 `equals` 方法，按照我们的规则进行比较的，注意 `instanceof` 的使用，注意强制转换的概念。将父类转换成子类叫做“**向下转型**（造型）”，向下造型是不安全的。“**向上转型**（造型）”是安全，子类转换成父类，如：将 `Student` 转成 `Person`，如 `Dog` 转成动物

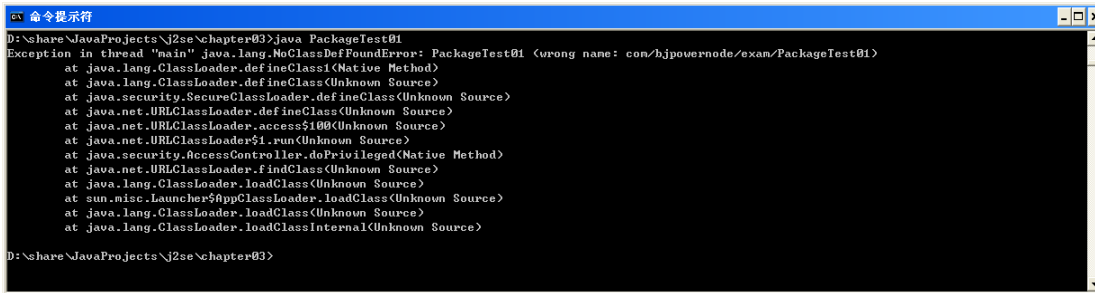
## 1.9、包和 import

### 1.9.1、包

包其实就是目录，特别是项目比较大，java 文件特别多的情况下，我们应该分目录管理，在 java 中称为分包管理，包名称**通常采用小写**

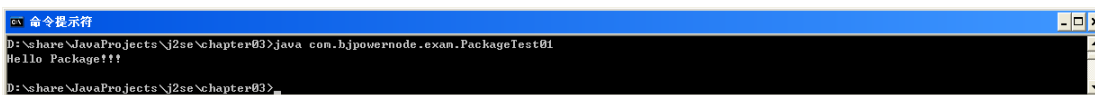
```
/*  
    1、包最好采用小写字母  
    2、包的命名应该有规则，不能重复，一般采用公司网站逆序，  
    如：com.bjpowernode.项目名称.模块名称  
    com.bjpowernode.exam  
*/  
  
//package 必须放到 所有语句的第一行，注释除外  
  
package com.bjpowernode.exam;  
  
public class PackageTest01 {  
  
    public static void main(String[] args) {
```

```
System.out.println("Hello Package!!!");  
  
}  
  
}
```



```
C:\ 命令提示符  
D:\share\JavaProjects\j2se\chapter03>java PackageTest01  
Exception in thread "main" java.lang.NoClassDefFoundError: PackageTest01 (wrong name: com/bjpowernode/exam/PackageTest01)  
    at java.lang.ClassLoader.defineClass1(Native Method)  
    at java.lang.ClassLoader.defineClass(Unknown Source)  
    at java.security.SecureClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.defineClass(Unknown Source)  
    at java.net.URLClassLoader.access$100(Unknown Source)  
    at java.net.URLClassLoader$1.run(Unknown Source)  
    at java.security.AccessController.doPrivileged(Native Method)  
    at java.net.URLClassLoader.findClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  
    at java.lang.ClassLoader.loadClass(Unknown Source)  
    at java.lang.ClassLoader.loadClassInternal(Unknown Source)  
D:\share\JavaProjects\j2se\chapter03>
```

运行出现类不能找到错误，提示给的很明显，如 `com.bjpowernode.exam.PackageTest01` 类不能找到，因为我们加入了包，所以我们的 `class` 文件必须放到和包一样的目录里才可以，这就是采用包来管理类，也就是采用目录来管理类，建立目录 `com/bjpowernode/exam`，采用 `java PackageTest01` 执行，同样出现上面的错误，如果采用了包在执行该类时必须加入完整的包名称，正确的执行方式为 `java com.bjpowernode.exam.PackageTest01`



```
C:\ 命令提示符  
D:\share\JavaProjects\j2se\chapter03>java com.bjpowernode.exam.PackageTest01  
Hello Package!!!  
D:\share\JavaProjects\j2se\chapter03>
```

正确执行通过。

另外还有一点需要注意：必须在最外层包采用 `java` 来执行，也就是 `classpath` 必须设置在 `chapter03` 目录上，以 `chapter03` 目录为起点开始找我们的 `class` 文件

## 1.9.2、import

如何使用包下的 `class` 文件

```
package com.bjpowernode.exam;  
  
//采用 import 引入需要使用的类  
//import com.bjpowernode.exam.model.User;  
  
//import com.bjpowernode.exam.model.Student;  
//import com.bjpowernode.exam.model.Employee;
```

```
//可以采用 * 通配符引入包下的所有类
//此种方式不明确，但简单
import com.bjpowernode.exam.model.*;

//package 必须放到 所有语句的第一行，注释除外
//package com.bjpowernode.exam;

public class PackageTest02 {

    public static void main(String[] args) {
        User user = new User();
        user.setUserId(10000);
        user.setUserName("张三");

        System.out.println("user.id=" + user.getUserId());
        System.out.println("user.name=" + user.getUserName());
    }
}
```

如果都在同一个包下就不需要 `import` 引入了，以上的示例都没有包，可以理解为都在同一个包下，在实际开发过程中不应该这样做，必须建立包

### 1.9.3、JDK 常用开发包

- `java.lang`，此包 Java 语言标准包，使用此包中的内容无需 `import` 引入
- `java.sql`，提供了 JDBC 接口类
- `java.util`，提供了常用工具类
- `java.io`，提供了各种输入输出流

## 1.10、访问控制权限

java 访问级别修饰符主要包括：`private` `protected` 和 `public`，可以限定其他类对该类、属性和方

法的使用权限，

修饰符	类的内部	同一个包里	子类	任何地方
private	Y	N	N	N
default	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

注意以上对类的修饰只有：public 和 default，内部类除外

### 1.10.1、private

【示例代码】

```
public class PrivateTest01 {  
  
    public static void main(String[] args) {  
        A a = new A();  
        //不能访问，private 声明的变量或方法，只能在同一个类中使用  
        System.out.println(a.id);  
    }  
}  
  
class A {  
  
    private int id;  
  
}
```

### 1.10.2、protected

【代码实例】，在同一个包下，建立类 ProtectedTest01、A，并建立 B 继承 A

```
public class ProtectedTest01 {
```



```
public static void main(String[] args) {  
    A a = new A();  
    a.method1();  
    A b = new B();  
    b.method1();  
  
    B b1 = new B();  
    b1.method3();  
}  
}  
  
class A {  
  
    //采用 protected 声明的变量或方法只有子类或同一个包下的类可以调用  
    protected int id = 100;  
  
    public void method1() {  
        System.out.println(id);  
    }  
  
    protected void method2() {  
        System.out.println("A.method2()");  
    }  
}  
  
class B extends A {  
  
    public void method1() {  
        //可以正确调用
```

```
//因为和 A 在同一个包下
System.out.println(id);
}

public void method3() {
    //可以正确调用
    method2();
}
}
```

【代码示例】，在 test1 下建立类 C1，在 test2 下建立 ProtectedTest02 和 C2 类

```
package test2;

import test1.C1;

public class ProtectedTest02 {

    public static void main(String[] args) {
        new C2().method2();
    }
}

class C2 extends C1 {

    public void method2() {
        //可以调用，主要原因 C2 是 C1 的子类
        //所以可以访问 protected 声明的变量
        System.out.println(id);
        method1();
    }
}
```

```
class C3 {  
  
    public void method3() {  
        //不能在其他类中访问 protected 声明的方法或变量  
        //new C1().method1();  
    }  
}
```

### 1.10.3、default

如果 class 不采用 public 修饰，那么此时的 class，只能被该包下的类访问，其他包下无法访问

```
import test3.D;  
  
public class DefaultTest01 {  
  
    public static void main(String[] args) {  
        D d = new D();  
        d.method1();  
  
        //不能访问，只有在一个类中或在同一个包下可以访问  
        //在子类中也不能访问  
        d.method2();  
    }  
}
```

## 1.11、 内部类

在一个类的内部定义的类，称为内部类

内部类主要分类：

- 实例内部类
- 局部内部类
- 静态内部类

### 1.11.1、 实例内部类

- 创建实例内部类，外部类的实例必须已经创建
- 实例内部类会持有外部类的引用
- 实例内部不能定义 `static` 成员，只能定义实例成员

```
public class InnerClassTest01 {  
  
    private  int a;  
  
    private int b;  
  
    InnerClassTest01(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    //内部类可以使用 private 和 protected 修饰  
    private class Inner1 {  
        int i1 = 0;  
        int i2 = 1;  
  
        int i3 = a;  
        int i4 = b;  
    }  
}
```

```
//实例内部类不能采用 static 声明
//static int i5 = 20;
}

public static void main(String[] args) {
    InnerClassTest01.Inner1 inner1 = new InnerClassTest01(100, 200).new
    Inner1();
    System.out.println(inner1.i1);
    System.out.println(inner1.i2);
    System.out.println(inner1.i3);
    System.out.println(inner1.i4);
}
}
```

### 1.11.2、静态内部类

- 静态内部类不会持有外部的类的引用，创建时可以不用创建外部类
- 静态内部类可以访问外部的静态变量，如果访问外部类的成员变量必须通过外部类的实例访问

【示例代码】

```
public class InnerClassTest02 {

    static int a = 200;

    int b = 300;

    static class Inner2 {
        //在静态内部类中可以定义实例变量
        int i1 = 10;
        int i2 = 20;
    }
}
```

```
//可以定义静态变量
static int i3 = 100;

//可以直接使用外部类的静态变量
static int i4 = a;

//不能直接引用外部类的实例变量
//int i5 = b;

//采用外部类的引用可以取得成员变量的值
int i5 = new InnerClassTest02().b;

}

public static void main(String[] args) {
    InnerClassTest02.Inner2 inner = new InnerClassTest02.Inner2();
    System.out.println(inner.i1);
}
}
```

### 1.11.3、局部内部类

局部内部类是在方法中定义的，它只能在当前方法中使用。和局部变量的作用一样  
局部内部类和实例内部类一致，不能包含静态成员

```
public class InnerClassTest03 {

    private int a = 100;
```

//局部变量，在内部类中使用必须采用 **final** 修饰

```
public void method1(final int temp) {  
    class Inner3 {  
  
        int i1 = 10;  
  
        //可以访问外部类的成员变量  
        int i2 = a;  
  
        int i3 = temp;  
    }  
  
    //使用内部类  
    Inner3 inner3 = new Inner3();  
  
    System.out.println(inner3.i1);  
    System.out.println(inner3.i3);  
}  
  
public static void main(String[] args) {  
    InnerClassTest03 innerClassTest03 = new InnerClassTest03();  
    innerClassTest03.method1(300);  
}  
}
```

#### 1.11.4、匿名内部类

是一种特殊的内部类，该类没有名字

- 没有使用匿名类

```
public class InnerClassTest04 {
```

```
public static void main(String[] args) {  
    MyInterface myInterface = new MyInterfaceImpl();  
    myInterface.add();  
}  
}  
  
interface MyInterface {  
  
    public void add();  
}  
  
class MyInterfaceImpl implements MyInterface {  
  
    public void add() {  
        System.out.println("-----add-----");  
    }  
}
```

- 使用匿名类

```
public class InnerClassTest05 {  
  
    public static void main(String[] args) {  
        /*  
        MyInterface myInterface = new MyInterface() {  
            public void add() {  
                System.out.println("-----add-----");  
            }  
        };  
        myInterface.add();  
    }  
}
```



```
*/

/*
MyInterface myInterface = new MyInterfaceImpl();
InnerClassTest05 innerClassTest05 = new InnerClassTest05();
innerClassTest05.method1(myInterface);
*/

InnerClassTest05 innerClassTest05 = new InnerClassTest05();
innerClassTest05.method1(new MyInterface() {
    public void add() {
        System.out.println("-----add-----");
    }
});

}

private void method1(MyInterface myInterface) {
    myInterface.add();
}
}

interface MyInterface {

    public void add();
}

/*
class MyInterfaceImpl implements MyInterface {
```

```
public void add() {  
    System.out.println("-----add-----");  
}  
}  
*/
```