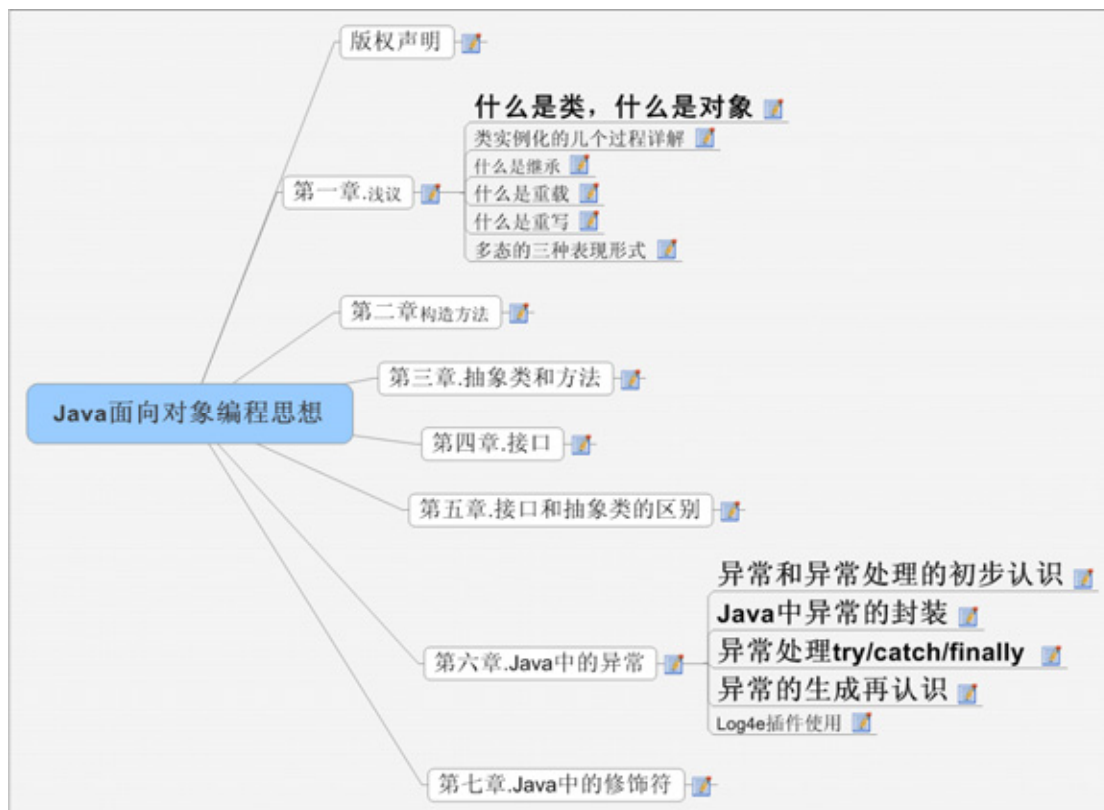


Java 面向对象编程思想



Java 面向对象编程思想.....	1
1 版权声明.....	2
2 第一章.浅议.....	2
2.1 什么是类，什么是对象.....	2
2.2 类实例化的几个过程详解.....	3
2.3 什么是继承.....	4
2.4 什么是重载.....	5
2.5 什么是重写.....	5
2.6 多态的三种表现形式.....	6
3 第二章.构造方法.....	6
4 第三章.抽象类和方法.....	7
5 第四章.接口.....	7
6 第五章.接口和抽象类的区别.....	8
7 第六章.Java 中的异常.....	13
7.1 异常和异常处理的初步认识.....	13
7.2 Java 中异常的封装.....	15
7.3 异常处理 try/catch/finally	16
7.4 异常的生成再认识.....	22
7.5 Log4e 插件使用	25
8 第七章.Java 中的修饰符.....	26

1 版权声明

此文档只能用于学习以及教学，请勿用作商业用途，因此而产生的法律问题，本人一概不负责。

本人声明，此文档资料为本人教学经验和网络资料收集合并之成果，如果在文档中引用了您的资料，而引起了侵犯您的权益的话，可以发送邮件知会，本人必定删除为是。

2 第一章.浅议

总所周知，Java 是一种面向对象的语言(所谓 OOP-Object Oriented Programming),但是很多人在学习 Java 的过程中对什么是类，什么是对象，什么是面向对象，为什么要有继承，为什么要有实现，为什么要有继承和实现，为什么要有抽象类和接口都不是十分的清楚，今天在这里，就不才浅议，未免贻笑大方，还望各位大大不吝赐教。

注：阅读此文档，不需要你是使用 Java 的高手，但是已经假设你有了一定的 Java 技术的基础。

理解面向对象，理解抽象，我想就应该是真正开始用面向对象的思想去分析问题，解决问题了吧。

2.1 什么是类，什么是对象

首先讲清楚类和对象的区别。

类是广泛的概念，表示一个具有相同属性和方法的多个对象的集合，是一个有共同性质的群体，而对象，所谓“万物皆对象”，指的是具体的一个实实在在的东西。

例如，“人”是一个类，它可以表示地球上所有的人；而“张三”、“李四”、“爱因斯坦”等则是

一个个的对象，或者说它们是“人”这个类的一个个实例。在 Java 中，我们可以定义类，然后创建类的对象。

例如：

```
// 声明一个类“Human”

class Human{

    private String name;

    public String getName(){

        return name;

    }

    public void setName(String value){

        this.name = value;

    }

}
```

通过一个类来创建一个对象：

```
Human human = new Human();
```

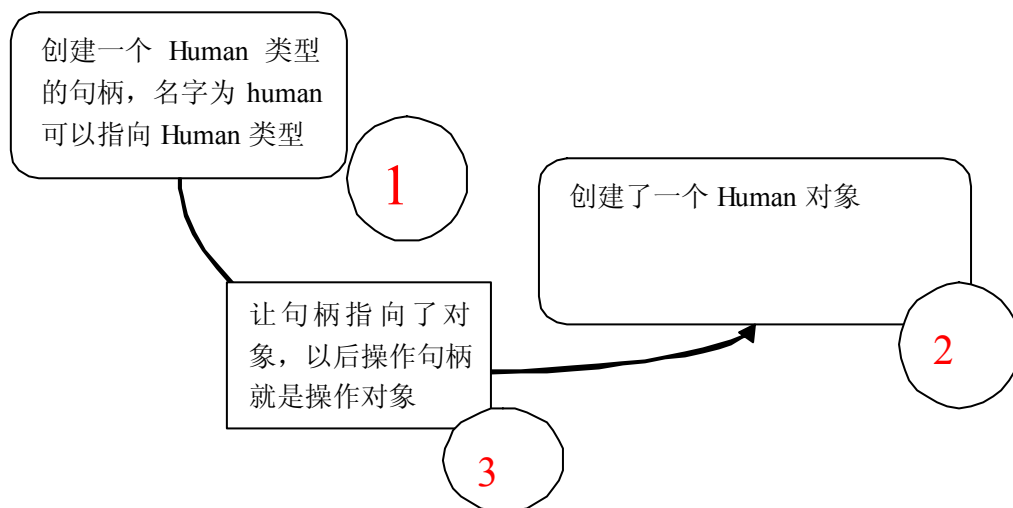
2.2 类实例化的几个过程详解

很多人对对象和对象的引用认识模糊,引用是程序操作对象的句柄，相当于 C 和 C++中的指针。

前面说了，对象是一个实实在在的东西，比如前面的代码：

```
Human human = new Human();
```

程序执行到这里之后，Java 虚拟机将会在内存中创建一个 Human 对象，并将这个对象的引用赋给 human 变量。这里有三步，首先是创建一个名字为 human 的 Human 类型的句柄，其次，声明一个 Human 对象，然后把创建的对对象的引用赋给 human 变量。



如果声明了一个对象的引用, 但没有将对象赋值给它, 或者给它的就是一个空的对象, 则这个引用指向了空的对象, 或者说引用了不存在的对象。这时如果想通过这个引用访问对象, 则会抛出空指针异常, 例如:

```
Human human;  
  
//.....  
  
human.setName("张三");
```

2.3 什么是继承

继承可以说是最大限度的发挥面向对象的复用性, 它是这样一个过程, 让一个类 B 去继承另一个类 A, 那么被继承的类 A 里的一些公开的可以被 B 看到的东西都可以被 B 继承过来, **就不必重新开发。**

如果“人”是一个基类, 则“男人”是“人”的一个子类。如果“张三”是一个“男人”, 也就是说“张三”是“男人”的一个对象, 那么显然“张三”也是“人”这个基类的一个对象。

子类具有父类的一般特性(包括属性和行为), 以及自身的特殊的特性。

Keep your code dry

这一切都目的都是为了达到一个目的, 那就是 Don't Repeat Yourself 即所谓的 DRY。如果某个属性或特性在父类中存在的话, 就不需要在子类里定义了。

而且, 如果发现两个类里有完全相同的属性或者方法的话, 就要考虑是不是应该有个父亲类了。

在父类里只定义一些通用的属性与方法。

在子类中通过 super 关键字来调用父类里的构造方法或者属性或者方法。

继承还直接带来了多态的好处。

在 Java 中使用 extends 关键字表示继承。

```
// 声明一个男人继承人类
```

```
public class Man extends Human{  
    }
```

// 声明一个人类的句柄指向男人

```
Human human = new Man();
```

程序设计的时候：先有子类-----

程序开发阶段：先做父类-----

多态带来了最重要的3个好处是：可复用性、可扩展性、可维护性。

2.4 什么是重载

同类，同名，不同参，与返回类无关，访问修饰无关，抛出异常无关。

重载： 在同一个类中，方法名相同，参数列表不同，所引起的两个方法的差异叫做重载，

- 在同一个类中
- 方法名相同
- 参数列表不同---
 - 参数的类型不同
 - 参数的个数不同
 - 参数的顺序不同
- 与访问修饰符无关
- 与返回值无关
- 与抛出异常无关

```
public void sayHello(int i) throws Exception{  
    }  
}
```

2.5 什么是重写

不同类，同名，同参，同返回，访问修饰不能更严格，抛出异常不能更广泛。

重写： 在存在父子继承关系的前提下，子类把从父类继承过来的方法，完全重写编写的过程叫做重写。

- 在父子类之间
- 方法名相同
- 参数列表相同

- 返回类型相同
- 访问修饰符，不能更加严格
- 抛出异常，不能更加广泛

2.6 多态的三种表现形式

1. 重写
2. 父类指针指向子类对象
3. 接口指向实现类

3 第二章构造方法

1. 构造方法是用来完成**对象的初始化的方法**, 要完成类的实例化, 产生一个对象, 就必须要有构造方法
2. **构造方法与类同名, 没有返回类型**(注意与没有返回值的区别)
3. 如果**不明显的指定类的构造方法**, Java 会为每个类隐式的生成一个不带任何参数的构造方法, **如果明显制定了任何一个合法的构造方法, Java 都不会为你默认生成。**
4. 构造方法是最先执行的方法(但是晚于静态代码块和非静态代码块)
5. 存在父子类继承的关系的前提下, 如果子类的构造方法没有与父类对应参数个数的构造方法, 而且如果不明显的指定运行父类的某一个构造方法的话, 会有编译错误。如果可以完成编译的话, **是先运行父类的构造方法, 然后执行子类的构造方法。**
6. **this(?)**关键字是运行本类的构造方法, **super(?)**关键字是用来运行父类的构造方法。
7. 构造方法不能继承, 更不可以被重写。
8. 构造方法可以被重载。
9. 抽象类/接口没有构造方法, 因为抽象类/接口不需要产生对象

```
public class Teacher{  
    public int Teacher(int i){  
        return 0;  
    }  
}
```

上面方法不是构造方法, 只不过是一个与类同名的普通方法。

4 第三章.抽象类和方法

- 如果一个方法中存在不可预知的方法实现，那么这个方法所在的类就应该声明为抽象类
- 如果在两个类当中有同名的方法，按照正常道理来说，应该提取到父类里，但是这个两个方法又有明显的方法实现上的不同，那么在父类里的方法，就应该定义成为抽象，抽象方法，只给出了方法的样子，而不给出方法的具体实现，具体实现由继承这个类的具体的子类去实现
- 抽象方法，必须在抽象类里，抽象类里不一定有抽象方法
- 没有抽象方法的抽象类的存在也是有意义的。这决定了这个类是不能被直接实例化的，他的作用基本上上是作为一个“框框”存在的。-----种猪
- 重写的前提是---父类里知道该方法如何去实现，子类不打算走原路，所以覆盖父类里的实现，
- 父类不给出实现，也不知道该如何实现，由具体的子类自己决定如何实现，但是给出一个定义，要求子类必须实现。

5 第四章.接口

- 接口是一种规范，是一种规则，它只给出了方法的样子，规定你要实现那些方法，而不给出方法的实现，让接口的实现类去实现这些方法，但是对于不同的实现类来说，对方法的实现可以完全不同。
- 接口的实现类如果实现了某一个接口，那么必须实现接口中定义的所有的待实现的方法。如果你不想要实现方法，那么实现类必须定义为抽象类，不想实现的方法必须定义为抽象方法。
- 接口中定义的属性和方法的默认访问级别都是`public`，所以方法肯定不被`private`所修饰，所有的方法前都默认包含了`abstract`，表明它是一个抽象方法，但是可以省略，而且默认是省略的
- 接口实际上是一个抽干了的抽象类，它里面所有的方法都是抽象的，都是不能给出任何实现部分的----- 僵尸类
- 因为接口是一种特殊的“抽象类”，而且因为抽象类可以继承别的类，所以接口也可以继承接口，也只能继承接口。但是接口不可以实现接口。
- 一个类可以实现多个接口使用关键字`implements`
- 一个接口可以继承多个接口使用关键字`extends`
- 一个类可以同时继承一个别的类，而且实现一个或多个接口，但是关键字`extends`要在前

6 第五章.接口和抽象类的区别

接口和抽象类的区别

- 接口是用来被实现的，而抽象类是用类被继承的
- 接口可以继承别的多个接口，组成一个新的接口
- 抽象类可以继承一个别的类，或实现一个或多个接口
- 接口是一个100%的抽象类，里面的方法没有任何一个有任何的实现
- 抽象类可以包含非抽象的方法，也就是说可以给出某一些方法的实现。
- 接口一般处于代码的最底层，作出一些规定，而接口之上一层抽象类层，对接口进行第一次的实现，把不可能一次完成的方法，交由自己的子类来实现。
- 抽象类可以有构造方法，而接口不可以有构造方法。

接口中定义的变量只能为 公有的，静态的，终态的，而且会默认增加。

面向对象思想，我觉得最关键的就是抽象。

一个软件设计的好坏，我很大程度上取决于它的整体架构，而这个整体架构其实就是你对整个宏观商业业务的抽象，当代表业务逻辑的高层抽象层结构合理时，你底层的具体实现需要考虑的就仅仅是一些算法和一些具体的业务实现了。当你需要再开发另一个相近的项目时，你以前的抽象层说不定还可以再次利用呢，面对对象的设计，复用的重点其实应该是抽象层的复用，而不是具体某一个代码块的复用。^^

说到了抽象，我就不能不提到 Java 接口和 Java 抽象类了。

既然面向对象设计的重点在于抽象，那 Java 接口和 Java 抽象类就有它存在的必然性了。

Java 接口和 Java 抽象类代表的就是抽象类型，就是我们需要提出的抽象层的具体表现。在 OOP(面向对象编程)中，如果要提高程序的复用性，增加程序的可维护性、可扩展性、可复用性，就必须是面向接口的编程，面向抽象的编程，正确地使用接口、抽象类这太有用的抽象类型做为结构层次上的顶层。

Java 接口和 Java 抽象类有太多相似的地方，又有太多特别的地方，究竟在什么地方，才是它们的最佳位置呢？把它们比较一下，你就可以发现了。

1、Java 接口和 Java 抽象类最大的一个区别，就在于 Java 抽象类可以提供某些方法的部分实现，而 Java 接口不可以，这大概就是 Java 抽象类唯一的优点吧，但这个优点非常有用。如果向一个抽象类里加入一个新的具体方法时，那么它所有的子类都一下子都得到了这个新方法，而

Java 接口做不到这一点，如果向一个 Java 接口里加入一个新方法，所有实现这个接口的类就无法成功通过编译了，因为你必须让每一个类都再实现这个方法才行，这显然是 Java 接口的缺点。

2、一个抽象类的实现只能由这个抽象类的子类给出，也就是说，这个实现处在抽象类所定义出的继承的等级结构中，而由于 Java 语言的单继承性，所以抽象类作为类型定义工具的效能大打折扣。在这一点上，Java 接口的优势就出来了，任何一个实现了一个 Java 接口所规定的方法的类都可以具有这个接口的类型，而一个类可以实现任意多个 Java 接口，从而这个类就有了多种类型。

3、从第 2 点不难看出，Java 接口是定义混合类型的理想工具，混合类表明一个类不仅仅具有某个主类型的行为，而且具有其他的次要行为。

4、结合 1、2 点中抽象类和 Java 接口的各自优势，具经典的设计模式就出来了：声明类型的工作仍然由 Java 接口承担，但是同时给出一个 Java 抽象类，且实现了这个接口，而其他同属于这个抽象类型的具体类可以选择实现这个 Java 接口，也可以选择继承这个抽象类，也就是说在层次结构中，Java 接口在最上面，然后紧跟着抽象类，哈，这下两个的最大优点都能发挥到极至了。这个模式就是“缺省适配模式”。

在 Java 语言 API 中用了这种模式，而且全都遵循一定的命名规范：Abstract + 接口名。

Java 接口和 Java 抽象类的存在就是为了用于具体类的实现和继承的，如果你准备写一个具体类去继承另一个具体类的话，那你的设计就有很大问题了。Java 抽象类就是为了继承而存在的，它的抽象方法就是为了强制子类必须去实现的。

其次，

下面重点谈一谈类、抽象类、接口和继承之间的关系

不少细心的初学者在论坛上问类似这样的问题：

1、接口不实现方法，但我却在程序中可以调用接口的方法，这是为什么？比如 java.sql 包中的 Connection、Statement、ResultSet 等都是接口，怎么可以调用 它们的方法呢？

2、抽象类不能实例化，但是 jdk 中却有很多抽象类的对象，这是为什么？比如 System.in 是一个 InputStream 类型对象，但 InputStream 是抽象类，怎么可以得到它的对象呢？

不管怎么样，大家应该明白一点：不管是抽象类中的抽象方法，还是接口中定义的方法，都是需要被调用的，否则这些方法定义出来就没有意义了。

可能有很多书上没有提到，或者提到了而读者没有注意到这一点：

一个子类如果继承了它的基类，则表示这个类也是其基类的一种类型，这个子类的一个对象是子

类类型，并且同时也是其基类的一个对象，它也具有基类的类型；一个类如果实现了一个接口，则表示这个类的一个对象也是这个接口的一个对象。

可能这样说不太好懂，又是子类、基类、类型、接口什么的，容易搞混。其实举个现实的例子你就会觉得其实很简单：

明白了这一点，就容易理解为什么我们可以得到抽象类的对象了：原来我们得到的抽象类的对象其实是它的已经实现了抽象方法的子类或子孙类的一个对象，但我们拿它当它的抽象类的基类来用。比如“人”这个类，每个人都会“悲伤”，男人悲伤的时候抽烟、喝酒，女人悲伤的时候哭泣、流泪。由于不同的子类在“悲伤”时所进行的动作不一样，因此这个动作(方法)在基类中不好实现，但基类中又需要有这个方法，因此，“人”这个类就可以定义一个抽象方法“悲伤”，由其子类“男人”和“女人”来实现“悲伤”这个方法。但是调用者只把男人和女人的对象当作其基类“人”的一个对象，调用它的“悲伤”方法。

读者可以去体验一下 jdk 的抽象类 `java.lang.Process`：

```
Runtime runtime = Runtime.getRuntime();  
  
Process process = runtime.exec("notepad.exe");  
  
Class cls = process.getClass();  
  
System.out.println(cls.getName());
```

这时会打印出 `process` 类的名字，如果在 Windows 下它会是一个类似于 `*Win32*` 的名字，它是 `Process` 的一个子类。因为 `process` 类用于管理打开的进程，而在不同的操作系统上都有不同的实现，因此它把方法定义为 `Process` 的抽象方法，而具体的操作只能由对应在不同操作系统下的子实现。

下面来谈接口，我们知道接口只定义了一些方法，而没有实现这些方法。而其实，接口是一个规范，它规定了实现这个接口所要做的事情，或者说规定了实现接口的类必须具备的能力(也就是方法)。

那么我们可以这样对比：

某种类型的驾驶执照，规定了拿到这个驾照的人必须能够“开小汽车”和“开公共汽车”。那么我们认为这个驾照是一个接口，它规定了实现它的类所必须有的能力。

我们可以定义一个类 `Driver`，继承自 `Human`，然后实现“驾照持有者”这个接口：

```
public interface DriverHolder{  
  
    public void driverCar();  
  
    public void driverBus();  
  
}  
  
public class Driver extends Human implements DriverHolder{
```

```
public void driverCar(){  
    // .....  
}  
  
public void driverBus(){  
    // .....  
}  
}
```

这样一来，一个“Driver”对象，它同时也是一个 `DriverHolder` 对象。即一个司机(Driver)同时是一个驾照执持有者对象。在程序中我们可以这样：

```
DriverHolder driverholder = new Driver();  
driverholder.driverCar();
```

这样我们就解释了为什么“接口没有实现方法，却可以得到接口类的对象”的问题。

但是这样一来，肯定有人会问：为什么要定义一个接口呢，为什么不直接把这个方法定义到 `Driver` 类中去，然后 `Driver driver = new Driver();` 一样可以调用它的 `driverCar();` 和 `driverBus()` 方法，这样做岂不是方便得多？

这是因为 `java` 是单继承的，它只能继承于一个类，这样它的类型就只限于其基类或者基类的基类。但是 `java` 可以实现多个接口，这样它就可以有很多个接口的类型。就象一个人，它继承自“脊椎动物”这个类，而“脊椎动物”又继承自“动物”这个类，因此“张三”是个人，他是一个“脊椎动物”，当然他也是一个“动物”。但他可以继承很多个接口，比如拿驾驶执照之后，他就是“驾照持有者”类型，他也可以拿英语六级证书，这样他就是一个六级证书持有者等等。

明白这一点之后，我们来看一看 `java` 的事件机制。

`java.awt.Button` 类有一个 `addActionListener(ActionListener l);`方法。这个方法传入的是一个接口类型：`ActionListener`，在实际中，我们需要实现 `ActionListener` 接口，并且把实现这个接口的类的对象引用作为参数传入。这样，`Button` 对象就得到了一个 `ActionListener` 对象，它知道这个 `ActionListener` 对象有一个 `actionPerformed` 方法，或者说它有处理 `Action` 事件的能力，当 `Action` 事件发生时，它就可以调用这个对象的 `actionPerformed` 方法。

比如一般我们会这样做：

```
public class TestButton extends Frame implements ActionListener{
```

```
private Button btn1 = new Button();

//.....

public TestButton(){
    btn.addActionListener(this);
this.add(btn);
}

public void actionPerformed(ActionEvent e){

}
}
```

现在我们假设 `ActionListener` 不是接口，而是一个类。那么我们只能继承 `ActionListener` 类，并且重写 `actionPerformed` 方法。但是 java 是单继承的，如果类继承了 `ActionListener` 类，那么它就不能继承其它的类(`Frame` 类)了，而不从 `Frame` 类继承的话，又怎么创建窗体，怎么把 `Button` 放到窗体中去呢？

其实接口不完全是为了解决 java 的单继承问题，它在某种程度上可以达到调用和实现细节的分离。

比如说，中国的民用电规范是一个接口：平均电压 220V、50Hz、Sin 交流电，水力发电厂、火力发电厂、核电厂，还有小型的柴油发电机如果按这个规范发电，则表示它们实现了这个民用电源的接口；冰箱、电视、洗衣机等家用电器使用这些电源，表示它们在调用这个接口。在这里，家用电器不管电从哪里来，只要它符合民用电源的规范就好，电源也不管它发的电用于什么工作，只管提供电源。

再回过头来看看 `Button` 的事件机制。要知道，`Button` 要保证所有的 `action` 事件发生时，程序员都可以在他的代码中处理它，图书馆管理系统、收银系统、进销存等等等等。而接口就可以做到这一点：找一个类实现 `ActionListener` 接口，并且让 `Button` 得到这个类的对象的引用（调用 `addActionListener` 方法），从而当 `Action` 事件发生时，`button` 创建一个包含了事件信息的对象(`ActionEvent`)，然后调用这个接口对象的方法，到底怎么处理这次事件，这就是实现接口的类的事情了。在这里，`Button` 丝毫不了解 `actionPerformed` 方法中到底干了什么事情，也不应该知道。`Button` 与具体的业务逻辑完全分离开了，它可以应用到所有的场合。

7 第六章.Java 中的异常

主要内容

异常和异常处理的初步认识

Java 中异常的封装

异常处理 **try/catch/finally**

异常的生成再认识

7.1 异常和异常处理的初步认识

异常处理是 Java 中唯一式的错误报告机制。ã

```
package net.tbq.oop.exception;
```

```
public class MathDemo {  
    public static int div(int a,int b){  
        return a/b;  
    }  
    // 1.语法没有问题  
    // 2.编译没有问题  
    public static void main(String[] args) {  
        System.out.println(MathDemo.div(1, 2));  
        System.out.println(MathDemo.div(11, 2));  
        System.out.println(MathDemo.div(12, 2));  
        System.out.println(MathDemo.div(13, 2));  
        System.out.println(MathDemo.div(14, 2));  
        System.out.println(MathDemo.div(15, 2));  
        System.out.println(MathDemo.div(15, 0));  
        System.out.println(MathDemo.div(16, 2));  
        System.out.println(MathDemo.div(71, 2));  
    }  
}
```

```
}  
}
```

// 3.运行时，出现了异常

// 错误栈

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at net.tbq.oop.exception.MathDemo.div(MathDemo.java:5)  
at net.tbq.oop.exception.MathDemo.main(MathDemo.java:16)
```

对付很长的异常堆栈的方法：

- _ 掌握好，栈，先进后出，越是先出的调用，越在下面，上面是最直接出异常的地方
- _ 不要去管跟你的代码无关的部分

异常和异常处理的初步认识

什么是异常？

一个程序是可行的，需要编译成功和运行无错两方面的配合。

异常（Exception）就是程序执行过程中意想不到的情况。

```
class Excep{  
    public int div(int a,int b){  
        return a/b;  
    }  
  
    class TestExcep{  
        public static void main(String[] aa){  
            Excep ec=new Excep();  
            ec.div(5,0);  
            System.out.println("OK");  
        }  
    }  
}
```

结论：

- ✧ 编译成功的程序未必可以正确运行。

如：除以零、打开一个不存在的文件、网络连接中断、数组下标越界、正在加载的类

文件丢失等。

- ✧ 不能正确运行，意味产生了**异常**。

什么是异常处理？

如果对异常不进行处理，JVM 就会显示出错信息，并中止程序执行。

异常处理是一种处理异常的机制。在程序中给出另一段（多段）错误处理的代码，当 JVM 发现异常时，转至该段代码执行。

7.2 Java 中异常的封装

异常的封装

常的封装，

Java 中，异常是以类的形式封装的。

- ✧ **程序中可处理的异常**：Exception 定义了程序中遇到的轻微的错误条件。
- ✧ **错误异常**：Error 定义了程序中不能恢复的严重错误条件。如内存溢出、类文件格式错误等。这一类错误程序无法处理。

编译时受检的异常

受检异常 (Checked Exception)

在编译时被能 Java 编译器检测到的异常。

- ✧ RuntimeException 之外的 Exception 都是受检异常。
- ✧ 受检异常必须在编译时进行处理。

非受检异常 (Unchecked Exception)

在编译时不能检测到的异常，包括：

- ✧ RuntimeException 的所有子类异常。
- ✧ Error 异常。

获得的异常的有关信息

Exception 的方法可用来获取当前异常的信息

```
public String getMessage()
```

返回描述当前异常性质的简略信息

```
public String toString()
```

返回描述当前异常类和异常性质的信息

```
public void printStackTrace()
```

在当前的标准输出上输出错误信息（错误异常类、错误性质、发生错误的类和方法）

7.3 异常处理 try/catch/finally

异常处理

有种异常处理方式：捕捉异常、转移异常。

捕捉异常方式的异常处理

对受检异常、运行时异常、动抛出的异常均适用。

try-catch 捕捉异常处理语句

```
try{  
    .....  
}  
catch(ExceptionType name){  
    .....  
}
```

流程分析说明：

- ◇ 生成异常，执行过程中遇到异常，自动产生一个对应异常

类的对象

- ✧ **抛出异常**，将生成的异常对象提交给 JVM
- ✧ **捕获异常**，JVM 寻找能处理这一异常的代码，并将当前异常对象作为参数传递过去，交由处理。

练习：给上例加入异常处理，发生异常时可显示异常的基本信息。

思考

1. 如果 `catch` 后给出空语句 `{ }`，情形会是怎样？
2. 如果 JVM 找不到对应的异常处理代码，会是怎样？
3. 异常没有发生，`catch` 段会执行到吗？
4. 异常发生了，`try` 体内产生异常语句的后续语句，有机会继续执行吗？

练习：编译、修改并执行以下程序，以创建磁盘文件。

```
import java.io.*;

class CreateFile{

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

```
File f1=new File("d://ok.txt");

f1.createNewFile();

System.out.println(f1+" have been created.");
```

完整的try-catch-finally 语句

```
try

    // 可能产生异常的 Java 代码

catch (ExceptionType1 name) { // 真的产生异常以
    后的处理..... }

catch (ExceptionType2 name) { ..... }

.....

finally { // 最终肯定要执行的代码 ..... }
```

- ✧ 具体的异常放在前，一般性的放在后
- ✧ 无论是否有异常产生，finally 后的语句块都被执行。
- ✧ 通常可用 finally 语句来做清理操作，比如关闭一个文件或把一个变量重新设为恰当的值。

嵌套

```
public static int div(int a,int b){  
  
    int result = 0;  
  
    try{  
  
        try {  
  
            Class.forName("");  
  
        } catch (ClassNotFoundException e) {  
  
            e.printStackTrace();  
  
        }  
  
        result = a/b;  
  
    }catch(ArithmeticException e){  
  
        //}catch(Throwable e){  
  
            e.printStackTrace();  
  
        }  
  
    return result;  
  
}
```

多重 catch

```
public static int div(int a,int b){  
  
    int result = 0;  
  
    try{  
  
        Class.forName("");  
  
        result = a/b;  
  
        } catch (ArithmeticException e) {  
  
            e.printStackTrace();  
  
        } catch (ClassNotFoundException e) {  
  
            e.printStackTrace();  
  
        }  
  
        return result;  
  
    }
```

转移异常方式的异常处理

在类的成员方法中，可通过 **throws** 短语抛出异常。

适用于需要强制进行异常捕捉、而并不想做具体的异常处理的情况。

如：

```
import java.io.*;
```

```
class CreateFile{

    public static void main(String[] aa) throws IOException

        File f1=new File("d://ok.txt");

        f1.createNewFile();

        System.out.println(f1+" have been created.");

    }

}
```

例中如果缺少 `throws IOException`，程序将不能通过编译。因为 `File` 类的 `createNewFile()` 方法要求进行强制异常捕捉。

说明：如果一个方法转移了异常，调用该方法的方法需要异常处理或转移异常。

练习：对 `Excep` 类中的方法 `div` 设置成转移异常，试修改程序编译运行。

Try---catch ----finally

肯定要出现的就是 `try` 自己，`catch` -`finally` 可以同时出现，也可以出现其中的一个

异常的生成再认识*

运行时自动生成

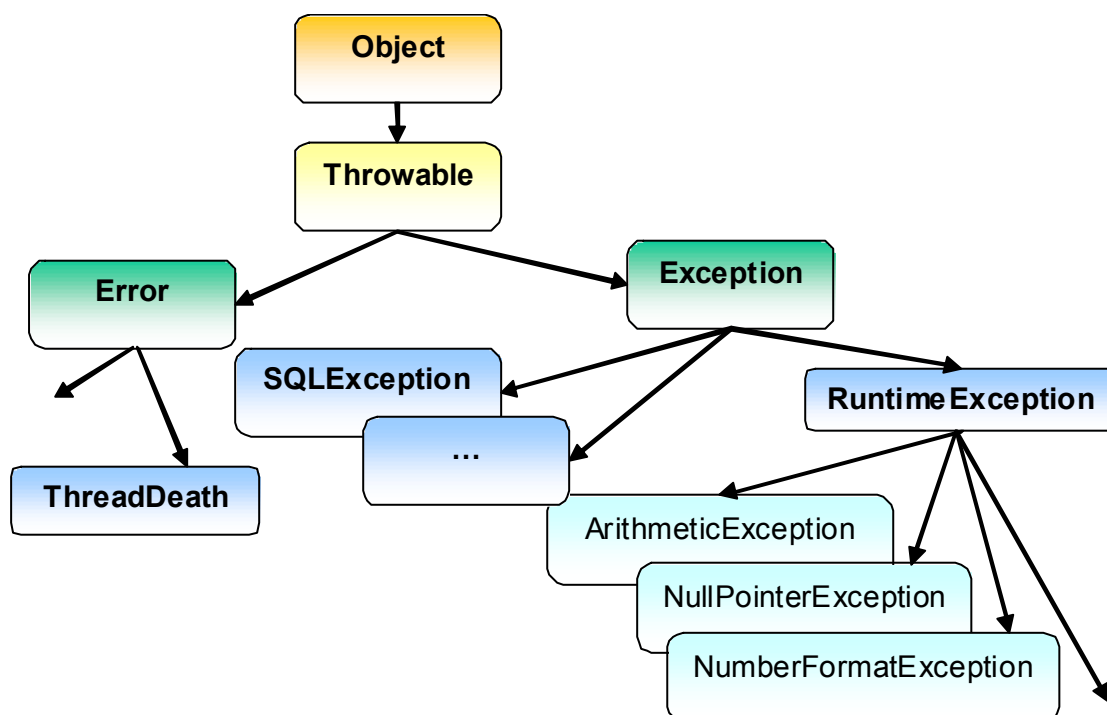
运行时虚拟机发现异常时，自动生成异常对象。

:

Throws	Throw
写在方法的声明时,表明这个方法是有异常的,告诉调用这个方法的那些方法,我有异常,如果用我的话,需要处理我的异常。	写在方法的里面,真正的抛出一个异常
抛出的是类型	抛出的对象

7.4 异常的生成再认识

- Throwable 具有两个子类, f 们是
 - Exception: 处理用户程序应当捕获的异常情况
 - Error: Error 类的异常为内部错误,因此在正常情况下不期望用户的程序捕获 f 们



异 类	说 明
Exception	异常层次结构的根类
RuntimeException	运行时异常，非受控
ArithmeticException	算术错误情形，如以零作除数
IllegalArgumentException	方法接收到非法参数
ArrayIndexOutOfBoundsException	数组大小小于或大于实际的数组大小
NullPointerException	尝试访问 null 对象成员
ClassNotFoundException	不能加载所需的类
NumberFormatException	数字转化格式异常，比如字符串到 float
IOException	I/O 异常的根类
FileNotFoundException	找不到文件
EOFException (End of file)	文件结束
InterruptedException	线程中断

异常抛出显式生成异常

throwThrowable 类对象

如：

```
thrownew ArithmeticException();
```

或：

```
ArithmeticException e=new ArithmeticExption();
```

```
throwe;
```

作用：故意抛出一个异常。（抛给调用者，一级一级抛出）

示例：分析程序运行结果。

```
class Throw_Test {  
  
    static void throw_something() {  
  
        System.out.println(1);  
  
        throw new ArithmeticException();  
  
  
  
    public static void main (String[] args) {  
  
        try {  
  
            throw_something();  

```



```
catch (ArithmeticException e) {
```

```
    System.out.println(3);
```

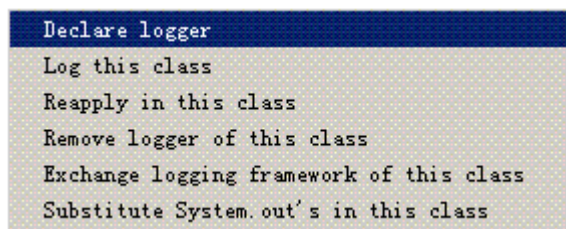
说明：

[1] 主动抛出的异常在虚拟机运行时发现，编译时不受检。

[2] Throw 语句后不可有别的语句，否则会出现编译错误：

“unreachable statement”

7.5 Log4e 插件使用



Declare -----声明---只声明一个对象 Logger

Log this class-----为这个类做日志---不光声明 Logger 还为每个方法增加一个开始时候的输出，再增加一个结束时候的输出，然后再增加一个级别判断

Reapply ----重新应用 Log

Remove-----删除

Exchange-----替换

Substitute system.out----替换 System.out

8 第七章.Java 中的修饰符

1.Java 中修饰符的类型：

Java 语言一共提供了 11 种修饰符，分别为 public,protected,private,final,abstract,static,native,transient,synchronized,volatile,strictfp.

2.Java 各修饰符应用对象

修饰符	(外部)类	属性	方法	构造方法	代码块	方法参数	内部类
public	可	可	可	可	不可	不可	可
protected	不可	可	可	可	不可	不可	可
(default)	可	可	可	可	可	可	可
private	不可	可	可	可	不可	不可	可
final	可	可	可	不可	不可	可	可
abstract	可	不可	可	不可	不可	不可	可
static	不可	可	可	不可	可	不可	可
native	不可	不可	可	不可	不可	不可	不可
transient	不可	可	不可	不可	不可	不可	不可
synchronized	不可	不可	可	不可	可	不可	不可
volatile	不可	可	不可	不可	不可	不可	不可
strictfp	可	不可	可	不可	不可	不可	可

注意：

— Java中外部类不可用被private和protected修饰，只可以是public 、默认、抽象、终态的修饰



对于内部类，可以修饰它的有公有，缺省，私有，保护，抽象，终态，静态

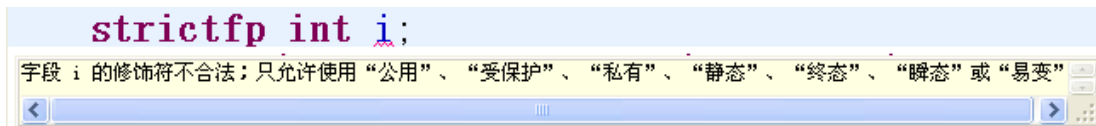


对类的声明来说，终态和抽象是冲突的两个概念，不能既是抽象的又是终态的。

对于方法的参数的修饰仅仅可以使用final关键字，表示此参数只可以用来为方法内的参数赋值，而本身不允许被修改。

对一个类的属性的修饰

只可以为以下



一个变量，如果出于类的层次定义，我们可以称它为属性，一个属性可以不赋予初值就使用，JVM 会根据属性的类型给予初值。

```
int i;

byte b;

char c;

short s;

long l;

float f;

double d;

boolean bool;

public void showArg(){

    System.out.println("byte 类型的属性的默认初值="+b);

    System.out.println("short 类型的属性的默认初值="+s);

    System.out.println("int 类型的属性的默认初值="+i);

    System.out.println("long 类型的属性的默认初值="+l);

    System.out.println("float 类型的属性的默认初值="+f);

    System.out.println("double 类型的属性的默认初值="+d);

    System.out.println("boolean 类型的属性的默认初值="+bool);

    System.out.println("char 类型的属性的默认初值="+c+"");

}
```

输出结果为：

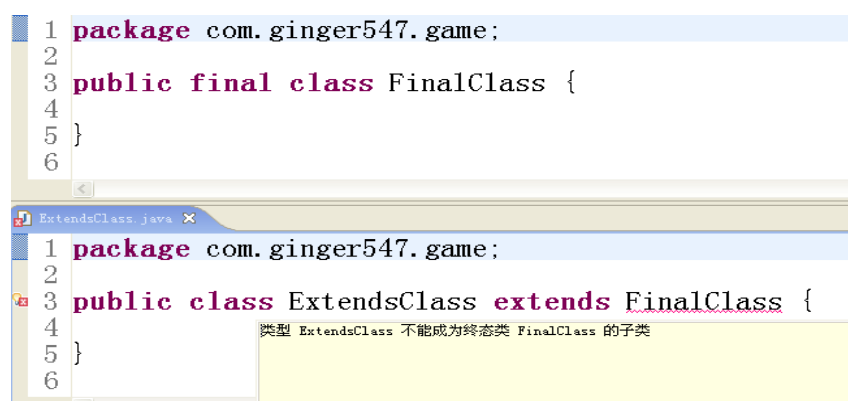
```
byte 类型的属性的默认初值=0
short 类型的属性的默认初值=0
int 类型的属性的默认初值=0
long 类型的属性的默认初值=0
float 类型的属性的默认初值=0.0
double 类型的属性的默认初值=0.0
boolean 类型的属性的默认初值=false
char 类型的属性的默认初值=|
```

而如果一个变量是在方法内部声明的，允许你声明的时候不给予初值，但是在第一次被使用之前一定要给予初值。否则的话会报错：

```
public void add( int a, int b) {
    int i;
    System.out.println(i);
    i=10;
}
```

局部变量 i 可能尚未初始化

final 可以修饰类，属性，方法，参数，表明该修饰符所修饰的对象具有不可变得特性：修饰类的时候说明，被修饰的类是一个终态的类，不可以被继承。



```
1 package com.ginger547.game;
2
3 public final class FinalClass {
4
5 }
6
```

```
1 package com.ginger547.game;
2
3 public class ExtendsClass extends FinalClass {
4
5 }
6
```

类型 ExtendsClass 不能成为终态类 FinalClass 的子类

修饰属性的时候说明这个属性只能被赋值一次，然后就不可被修改。

```
public final int i=10;
public void change() {
    i++;
}
```

不能对终态字段 FinalClass.i 赋值

但是注意：Java 中的常量是 final 决定的，而不是 final static 两个变量决定的，后者表示是一个静态常量，不需要实例化类就可以访问。

修饰方法的时候说明这个方法不可以在类继承的时候被重写。

```

    public final static int i=10;
    public final void change() {
        System.out.println(i);
    }
}

package com.ginger547.game;

public class ExtendsClass extends FinalClass {
    public void change() {
        // 不能覆盖 FinalClass 中的终态方法
    }
}

```

修饰参数的时候说明，这个参数只能用来为方法内的其他变量赋值，参数本身不能修改。

```

    public void changeIt(final int b) {
        b++;
        int a = b;
    }

    public void changeIt(final int b) {
        // b++;
        int a = b;
    }

```

不能对终态局部变量 b 赋值。它必须为空白，并且不使用复合赋值

不能对终态局部变量 b 赋值。它必须为空白，并且不使用复合赋值

如果一个 final 修饰符修饰一个引用性的变量，比如说一个类的一个实例，，表示该变量一旦被分配给一个对象以后，就不能再引用别的对象，更不能把其他对象的句柄赋值给它。但是可以改变所引用对象的内容。

```

1 package com.ginger547.game;
2
3 public class FinalClass {
4     private int i=10;
5     public void demo() {
6         System.out.println(i);
7     }
8     public class InnerClass{
9         public void demo() {
10             System.out.println(i);
11         }
12     }
13     public static void main(String[] args) {
14         final FinalClass fc = new FinalClass();
15         fc = new FinalClass();
16     }
17 }
18

```

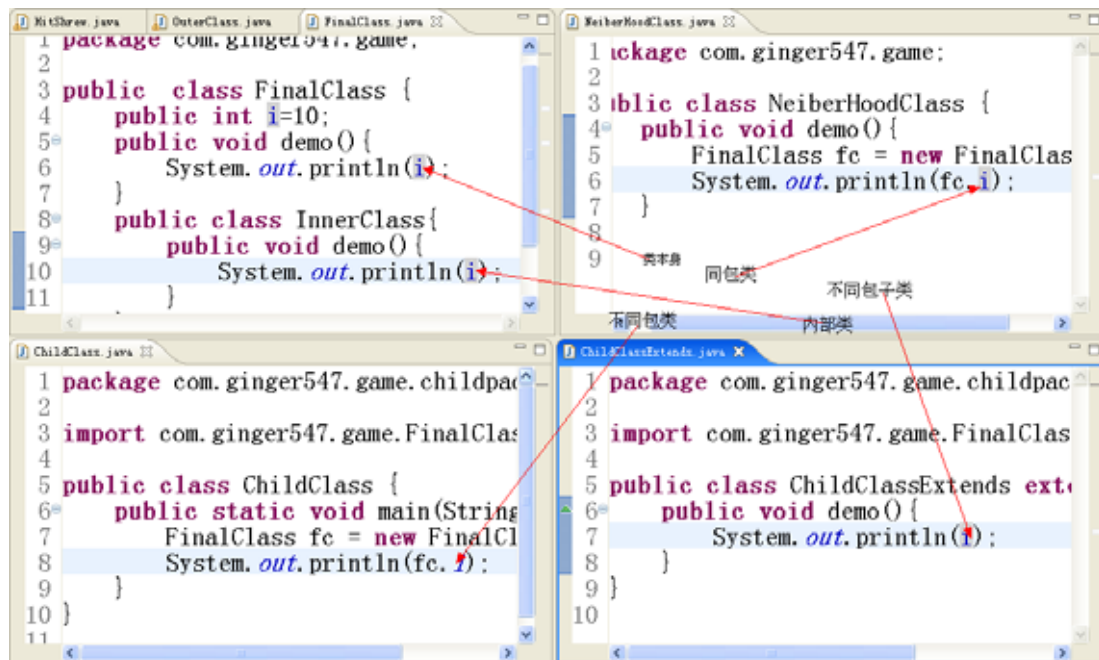
不能对终态局部变量 fc 赋值。它必须为空白，并且不使用复合赋值

final 是唯一可用于方法内局部变量的修饰符。

— private public (default) protected 修饰符的访问级别

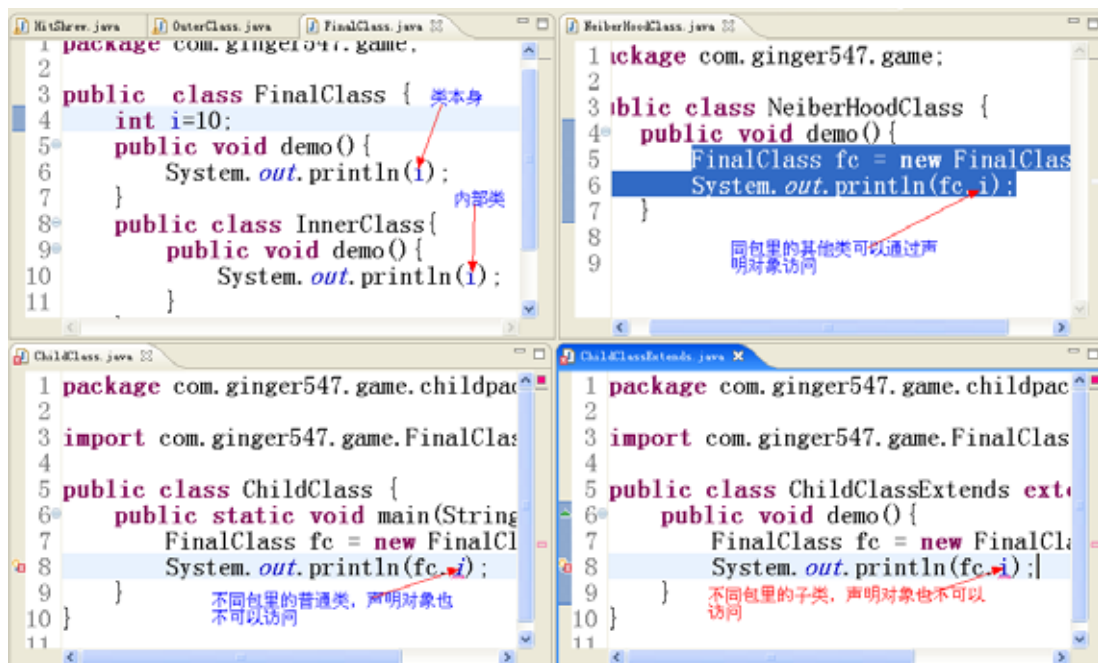
修饰符	类本身	同包类	不同包的子类	不同包类	内部类
public	可	可	可	可	可
protected	可	可	可	不可	可
(default)	可	可	不可	不可	可
private	可	不可	不可	不可	可

public:

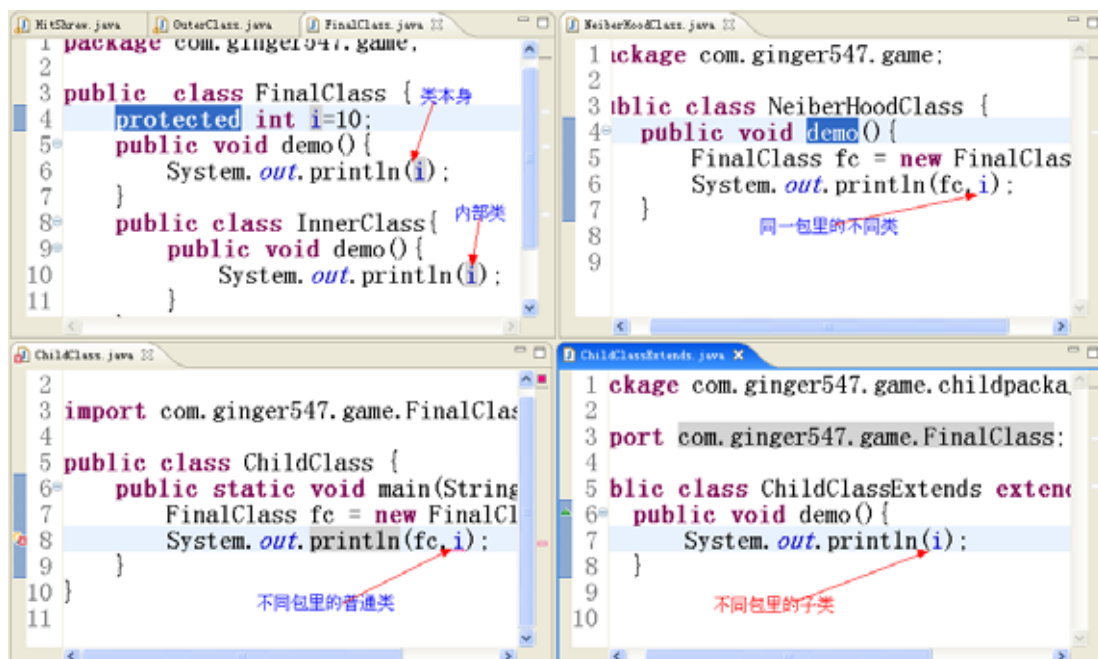


大家可以看到这内部类和不同包的子类里可以直接访问，不需要声明对象。

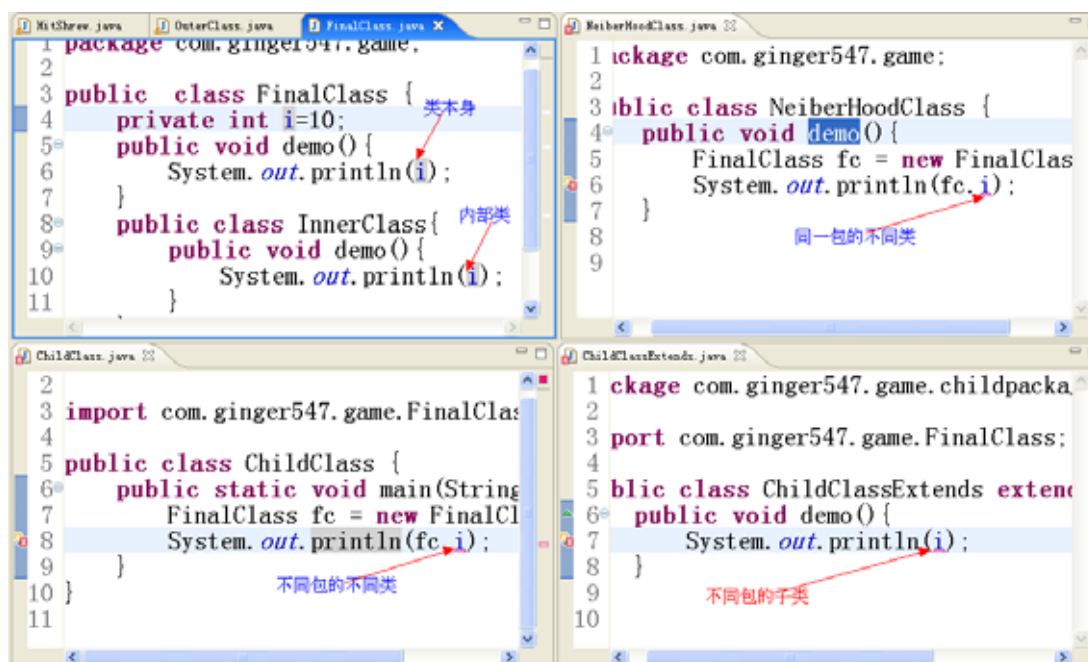
(default)



protected



private



可以这样理解：

public 是国家的
protected 是家族的
(default) 是家庭的
private 是个人的

或者是：

public 是妓女
protected 是情人
(default) 是坐公交车碰见的陌生女人
private 是老婆

— abstract 可以修饰类和方法：

abstract 修饰符只能用于修饰类和方法，不能修饰变量。因为该修饰符表明的是未实现的含义，而变量不存在未实现的概念。

修饰类的时候:说明这个类是一个抽象类，抽象类是不能被实例化的类，但是可以让一个抽象类指向一个非抽象的子类

```

1 package com.ginger547.game;
2
3 public abstract class AbstractParent {
4
5 }

```

```

1 package com.ginger547.game;
2
3 public class AbstractDemo {
4     public static void main(String[] args) {
5         AbstractParent abs = new AbstractParent();
6     }
7 }

```

不能实例化类型 AbstractParent

```

1 package com.ginger547.game;
2
3 public abstract class AbstractParent {
4
5 }
6

```

```

1 package com.ginger547.game;
2
3 public class AbstractDemo {
4     public static void main(String[] args) {
5         AbstractParent abs = new NoAbsChil();
6     }
7 }
8

```

```

1 package com.ginger547.game;
2
3 public class NoAbsChil extends AbstractParent {
4
5 }
6

```

一个抽象类里，可能没有抽象方法，仅仅是不想让别人实例自己，必须有自己的子类才可以。但是一个抽象方法，必须有一个宿主类，也就是必须在一个类里，而一旦一个类里有了抽象方法，那么这个类就必须为抽象类。

```

1 package com.ginger547.game;
2
3 public class AbstractParent {
4     public abstract void add();
5 }
6

```

只能由抽象类来定义类型 AbstractParent 中的抽象方法 add

抽象方法不得有方法体。

```

public abstract class AbstractParent {
    public abstract void add();
}

```

当子类继承一个抽象类时，该类必须被声明为抽象类，否则会发生编译错误。当类实现一个接口，但没有实现该接口中的所有方法时，则该类必须被声明为抽象类，否则会发生编译错

误。

- `static` 修饰符不能用于修饰顶层类(但是可以修饰内部类), 除了可以修饰方法和变量外, 还可以修饰一段代码块。该修饰符表明被修饰对象属于类范畴, 而不属于类实例, 即被修饰对象附属于类本身, 而非附属于各个类的实例。

一个被声明为静态的类属性, 它有两种引用方式, 一种是像其他非静态类属性一样, 通过类的实例来引用, 这尽管符合 Java 语言规定且不会引起编译错误, 但是会产生代码可读性差的问题; 另一种是根据静态的意义, 静态类的属性不得与任何一个类实例相关, 因此可直接通过类名来引用一个静态类的属性。

静态类的属性被所有的类实例所共享, 即任何一个实例改变了静态类属性的值, 所有类实例拥有的这个静态属性值都会同步得到改变。

```
1 package com.ginger547.ele;
2
3 public class Test1 {
4     public static void main(String[] args) {
5         System.out.println("测试开始");
6         StaticTest.i = 15;
7         Test2 test2 = new Test2();
8         test2.output();
9         Test3 test3 = new Test3();
10        test3.output();
11    }
12 }
```

问题 Javadoc 声明 搜索 控制台

<已终止> Test1 [Java 应用程序] D:\Java\jdk1.5.0_11\bin\javaw.exe (2008-7-23 上午11:12:10)

测试开始
15
15

注 1. 一个方法中定义的局部变量是不能被声明为静态的。因为静态修饰符 `static` 是一个与类而非类实例相关的概念, 而方法是一个局部概念, 它们附属于类实例, 其生命周期不同于类实例, 更不同于类, 而方法内部变量会随着方法的推出而被撤销, 因此一个方法体中声明的局部变量是不能被 `static` 修饰的。即使方法被声明为静态的, 其内部变量也不能被声明为静态的。一个被 `static` 修饰符修饰的静态方法, 除了可以访问中其内部定义的变量外, 则只能访问被 `static` 修饰符修饰的静态变量。如果需要访问非静态类的属性, 则必须先实例化一个类实例, 再通过该类实例引用非静态类的属性。但是, 一个非静态的方法可以访问一个静态变量。

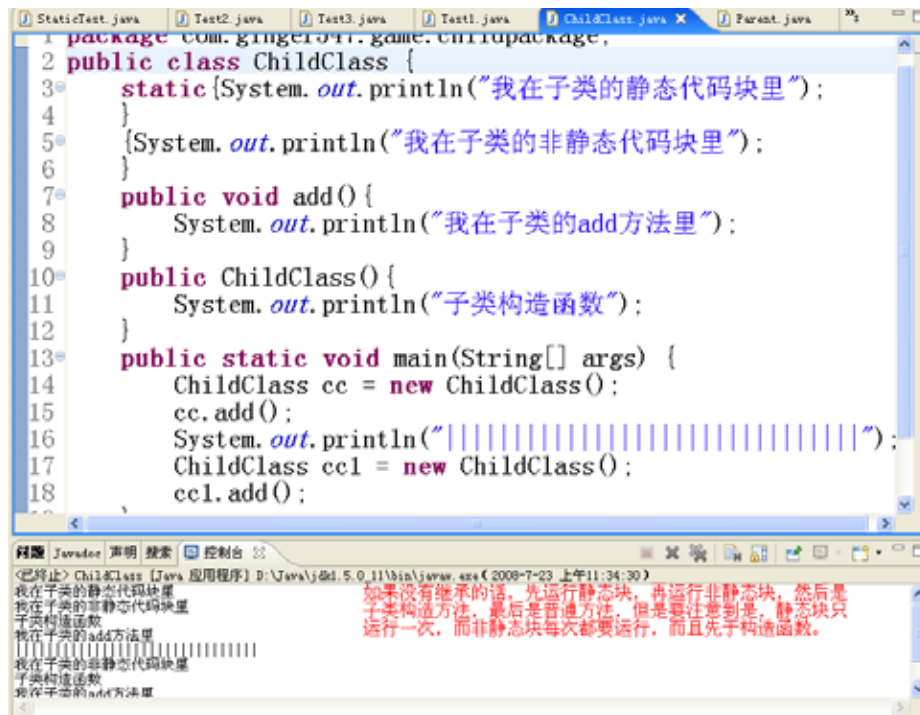
注 2. 对静态方法和非静态方法的调用方式不同, 对非静态方法的调用时在运行期决定的, 而对静态方法的调用则发生在编译期。一个非静态方法可以调用一个静态方法, 但是不允许一个静态方法直接调用非静态方法, 除非通过类实例调用。

注 3. 一个静态方法不能重写为一个非静态方法, 只能被重写为一个静态方法, 但是可以被

重载为一个非静态的方法，重写的含义是始终只有一个定义，只是原来的含义完全被后来的含义所取代，即形式不能变，而重载的含义是指同一个样的东西在不同的地方具有不同的含义。

注 4.静态代码块不是一个方法，实际上只是一个 `static` 修饰符，后跟一个方法主体(一对大括号内的一组语句)。静态代码块主要用于初始化，该代码块中的代码仅被执行一次，即在构造函数执行前执行，而且只执行一次，如果继承的父类里有静态代码块，先执行父类的，但是子类的静态代码块还是要先于父类的构造函数。如果一个类中存在多个静态代码块，那么其运行次序取决于在类中定义的次序。

注 5.非静态代码块不是一个方法，实际上只是一个方法主体(一对大括号内的一组语句)。每当创建类实例时，非静态代码块获得执行，其运行在父类构造器之后，所在类构造器之前。如果一个类中存在多个非静态代码块，那么其运行顺序取决于在类中定义的次序。



```
1 package com.ginger511.game.chirupackage;
2 public class ChildClass {
3     static {System.out.println("我在子类的静态代码块里");}
4 }
5 {System.out.println("我在子类的非静态代码块里");}
6 }
7 public void add() {
8     System.out.println("我在子类的add方法里");
9 }
10 public ChildClass() {
11     System.out.println("子类构造函数");
12 }
13 public static void main(String[] args) {
14     ChildClass cc = new ChildClass();
15     cc.add();
16     System.out.println("||||||||||||||||||||||||||||||||");
17     ChildClass ccl = new ChildClass();
18     ccl.add();
19 }
```

控制台输出：

```
<已终止> ChildClass [Java 应用程序] D:\Java\jdk-5.0.1\bin\java.exe (2008-7-23 上午11:34:30)
我在子类的静态代码块里
我在子类的非静态代码块里
我在子类的add方法里
||||||||||||||||||||||||||||||||
我在子类的add方法里
我在子类的非静态代码块里
我在子类的add方法里
```

如果没有继承的话，先运行静态块，再运行非静态块，然后是子类构造方法，最后是普通方法，但是要注意到是，静态块只运行一次，而非静态块每次都要运行，而且先于构造函数。

