# 1 内部类

## 1.1 什么是内部类

在一个类的内部，再定义一个类，那么这个类就是内部类。

特点：内部类中可以直接访问其外部类的成员，包括私有的成员。

内部类分为成员内部类和局部内部类：在类的成员位置定义的类，被称为成员内部类；在类的方法中(局部位置)定义的类，被称为局部内部类。例如：

|  |
| --- |
| **public class** Outer {  **private int outerNum** = 100;  **class** Inner1 {  *// 这是成员内部类*  **public void** show() {  System.***out***.println(**outerNum**); *// 可访问外部类私有变量*  }  }  **public void** method() {  *// 这是局部内部类*  **class** Inner2 {  **public void** show() {  System.***out***.println(**outerNum**); *// 也可访问外部类私有变量*  }  }  }  } |

内部类生产的class文件为 “外部类$内部类.class”，为了标明该内部类是属于具体哪个外部类的。

成员内部类和局部内部类都能访问外部类的私有变量，下面主要介绍一些细节和内部类是如何被外界访问的。

## 1.2 成员内部类

成员内部类可用访问修饰符修饰，比如public private和默认。实际开发中，为了保护内部类，将内部类用private修饰，不让外界直接访问，这样只能在该其外部类使用这个内部类。

成员内部类还能用static修饰，

（1）当不加static修饰时，成员内部类中不能有静态成员，并且外部类和外界访问该内部类的方法如下：

外部类名.内部类名 对象名 = 创建外部类对象.创建内部类对象。例如：

|  |
| --- |
| **public class** Outer {  **private int num** = 100;  **public class** Inner {  *// 成员内部类*  **public void** showNum() {  System.***out***.println(**num**);  }  }  **public void** show() {  *// 在外部类中可以直接写Inner*  Inner i = **new** Outer().**new** Inner();  i.showNum();  }  } |

Demo使用类中：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Outer outer = **new** Outer();  outer.show();  *// 外界必须写成Outer.Inner*  Outer.Inner i = **new** Outer().**new** Inner();  i.showNum();  }  } |

（2）当用static修饰时，成员内部类中只能访问外部类的静态成员；内部类中静态和实例成员都能有（实例方法也只能访问外部类的静态成员），并且这时外界访问更加方便，格式如下：

使用内部类静态方法，直接用：外部类名.内部类名.方法名()；

使用内部类实例方法：

1.创建内部类对象格式：

外部类名.内部类名 对象名 = new 外部类名.内部类名()；

2.使用实例方法：对象名.实例方法名()。

例子：

外部类Outer：

|  |
| --- |
| **public class** Outer {  **private int num1** = 100;  **private static int** *num2* = 200;  **public static class** Inner {  *// 内部类静态方法*  **public static void** showStaticNum() {  System.***out***.println(*num2*); *// 只能访问外部静态变量*  }  *// 内部类实例方法*  **public void** showNum() {  System.***out***.println(*num2*); *// 实例方法也只能访问外部变量*  }  }  **public void** showInner() {  *// 外部类中Outer.都能省略*  Inner.*showStaticNum*(); *// 直接调用内部类静态方法*  *// 创建对象调用内部类实例方法*  Inner inner = **new** Inner();  inner.showNum();  }  } |

Demo使用类：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  *// 测试外部类访问*  Outer outer = **new** Outer();  outer.showInner();  *// 外界访问不能省略Outer.*  Outer.Inner.*showStaticNum*();  Outer.Inner inner = **new** Outer.Inner();  inner.showNum();  }  } |

案例补充：

|  |
| --- |
| **public class** Outer {  **public int num** = 10;  **public class** Inner {  **public int** num = 20;  **public void** show() {  **int** num = 30;  System.***out***.println(num); *// 直接访问的是本局部变量的num，值是30*  System.***out***.println(**this**.num); *// 用this.num访问的是内部类的num，值是20*  System.***out***.println(Outer.**this**.**num**); *// 用Outer.this.num表示的是外部类的num，而不能用Outer.num*  }  }  } |

## 1.3 局部内部类

局部内部类不能写任何修饰符，包括public和private。同样，局部内部类也不能用static关键字修饰，并且局部内部类中也不能有静态成员。

在类的静态方法和实例方法中都可以写局部内部类，局部内部类只能在包含该局部内部类的方法中使用，在该方法中创建局部内部类对象并使用，外部的别的地方无法直接使用。

例子：

（1）Outer类

|  |
| --- |
| **public class** Outer {  **private int num1** = 100;  **private static int** *num2* = 200;  **public static void** staticMethod() {  *// 静态方法中的内部类*  **class** Inner1 {  *// 内部类的方法*  **public void** show() {  System.***out***.println(*num2*); *// 静态方法中的内部类只能外部类的访问静态变量*  }  }  *// 在该方法中使用内部类*  Inner1 inner1 = **new** Inner1();  inner1.show();  }  **public void** instanceMethod() {  *// 实例方法中的内部类*  **class** Inner2 {  **public void** show() {  System.***out***.println(**num1** + *num2*); *// 都可访问，这是正常的*  }  }  *// 同样在该方法中使用内部类*  Inner2 inner2 = **new** Inner2();  inner2.show();  }  } |

（2）Demo测试类

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  *// 测试*  Outer outer = **new** Outer();  outer.*staticMethod*(); *// 200*  outer.instanceMethod(); *// 300*  }  } |

特别需要注意的是，如果局部内部类中想要访问该方法的局部变量，那么必须将这个变量用final关键字修饰。原因：局部变量在外部类的这个方法中，该方法一旦执行完毕，变量就会被销毁释放，那么内部类就无法访问了，因此会出错。所以需要把内部类访问的局部变量用final关键字修饰，将这个变量提升为“类”的字段。如果不加，JDK8会自动加上隐含的final。建议还是自己加上。（这里所说的局部变量是外部类的包含这个局部内部类方法中的变量，而不是内部类自己的方法中的局部变量）

例子：

（1）Outer类

|  |
| --- |
| **public class** Outer {  **public void** method() {  **final int** num = 100; *// 用final修饰将要被局部内部类访问的变量*  *// 定义局部内部类*  **class** Inner {  **public void** show() {  System.***out***.println(num);  }  }  *// 使用局部内部类*  Inner inner = **new** Inner();  inner.show();  }  } |

（2）Demo测试类

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  *// 测试*  Outer outer = **new** Outer();  outer.method();  }  } |

## 1.4 匿名内部类

匿名内部类就是没有名字的内部类，他的作用是简化内部类的书写。写匿名内部类的前提是有一个类或者接口。匿名内部类的格式：

new类名或者接口名(){

可重写方法；

};

上述就是一个匿名内部类，匿名内部类的本质就是一个继承了该类或者实现了该接口的子类匿名对象。比如：

|  |
| --- |
| **new** Date(){  *// 这里没有重写方法* }; |

就是一个匿名内部类，并且返回的是继承了Date类的一个匿名子类对象。然后可以直接使用这个对象，比如：

|  |
| --- |
| **new** Date(){}.getTime(); |

可以输出当前时间。看似和：new Date().getTime();的效果是一样的，实际上不同。使用匿名内部类可以重写方法，比如如果这样写，得到的就不是当前时间的毫秒值了，而是1L。

|  |
| --- |
| **new** Date(){  **public long** getTime() {  **return** 1L;  } }.getTime(); |

并且，使用匿名内部类，可以在需要接口或者父类参数的地方，创建一个匿名内部类，以参数的方式传递进去。在Android中就有很多这样的用法。因为匿名内部类使用一次。使用完后立即释放，不占用太多资源。如果需要多次使用这个对象，那么建议还是创建一个类继承类或者实现接口。当然，也可以先将这个匿名内部类赋给一个对象，然后多次调用，比如：

（1）Fly接口

|  |
| --- |
| **public interface** Fly {  **void** fly(); } |

（2）Demo类

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Fly fly = **new** Fly() {  **public void** fly() {  System.***out***.println(**"I can fly!"**);  }  };  *// 可多次调用* fly.fly();  fly.fly();  } } |

这种方式只适合在一个文件中使用。

# 2 Eclipse的使用

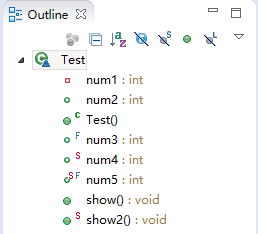
## 2.1 基本使用

Eclipse是Java集成开发环境，免费，开源，绿色，可扩展。

Eclipse中有多个视图，比如有Java视图、Debug视图。每个视图有多种视窗，主要有

（1）Package Explorer：包资源管理器，显示项目结构以及所有源文件。

（2）OutLine：显示类的结构，方便查看、定位等。如图：



主体颜色(红/绿)代表修饰符，右上角的C表示构造方法，F表示以final修饰，S代表static修饰。

（3）Console：显示控制台输出信息。

（4）Type Hierarchy：显示Java继承层次结构，选中java文件后按F4即可查看。

（5）Eclipse代码区：红色标记表示错误，橙色表示警告。

使用：开启时选择工作空间，再新建：Java Project🡪Package🡪Class。

Eclipse会在每次保存Java文件时自动将\*.java文件编译成\*.class文件。

## 2.2 常用设置

（1）显示行号：代码左边的区域，右键Show Line Numbers即可

（2）导入项目文件夹：右键🡪import🡪General🡪Existing Project into Workspace🡪选择文件夹/zip文件等🡪完成。

（3）设置文档为UTF-8编码：Project->Proporties->Resource出现的Text file encoding选择Other的UTF-8。

（4）设置字体大小：Window -> Preferences -> General -> Appearance -> Colors and Fonts中的Basic，选中Text Font，点击Edit按钮编辑字体设置即可。

（5）设置代码提示：Window -> Preferences -> Java -> Editor -> Content Assist中，在“Auto activation triggers for java”里面填上所有的字母。

## 2.3 常用快捷键

（1）代码提示：Alt + /；

（2）格式化代码：Ctrl + Shift + F

（3）导包：Ctrl + Shift + O，如果有多个包存在同名的类，则要选择。

（4）注释：单行注释：Ctrl + /。如果想取消，再来一次。

多行注释：Ctrl + Shift + /。取消：Ctrl + Shift + \。

（5）查看源代码：选中代码，按F3或者Ctrl + 鼠标点击。【JDK目录下有src.zip文件，里面包含了源代码，不要删除，否则IDE会找不到源码】

（6）如何使用Source中的快捷键：按下Alt + Shift + S，再看下滑线字母进行选择。

## 2.4 Eclipse中导出jar包和使用jar包

（1）将源码导出为jar包。右键项目🡪Export…🡪Java🡪JAR file🡪Next🡪选择导出jar文件的名称和路径🡪完成。为了让人使用自己的jar包，还需要给出说明书。生成说明书的方法：右键项目🡪Export…🡪Javadoc🡪选择输出路径🡪完成。

（2）使用别人的jar包：在项目下新家一个lib文件夹专门用于存放第三方jar包，把需要的jar包复制进文件夹中。然后，右键这个jar包🡪Build Path🡪Add to Build Path，这样就能使用jar包中的类了。如果不依赖于IDE，实际上只要将jar包配置到classpath路径下，就能使用jar包了。

## 2.5 调试程序

调试程序就是设置断点，程序运行到断点的地方会停止，然后可以查看这时的变量状态以便找出程序的出错位置和出错原因。

调试的方法：

双击一行代码的左侧设置断点，然后Debug as调试程序，会弹出是否进入Debug视图的选择框，选择“Yes”进入Debug视图。运行到断点时可以使用Step Over(F6)逐行调试。Variables视窗可以观察变量值的变化。如果想监视表达式和变量，则打开Expressions选项卡：Window 🡪 Show View 🡪 Expressions。再次双击断点会取消断点。

通过观看多个函数的逐步调试调用过程，可以知道函数的入栈和出栈。

# 3 异常

## 3.1 异常介绍

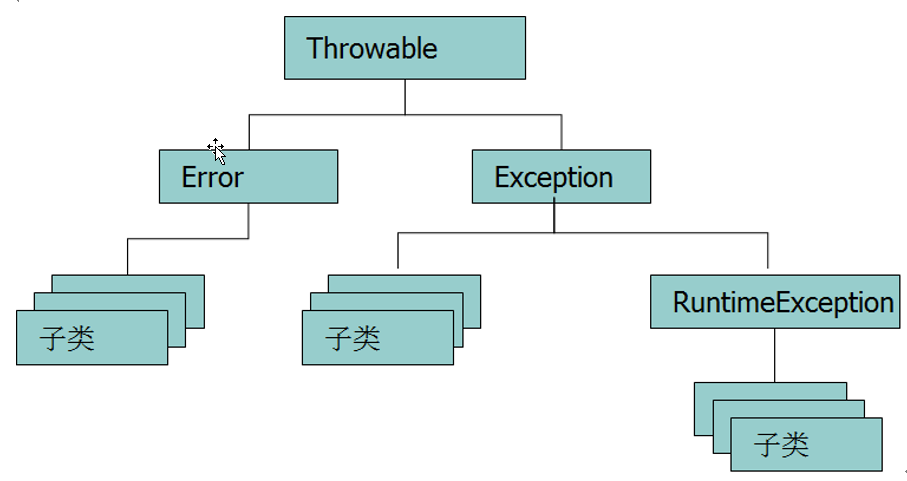
异常就是程序出现了不正常的情况。比如执行Integer.parseInt("abc")时，由于程序无法将abc转换成数字，因此会出现异常。

Java中，异常类叫Throwable，他有两个子类，分别是Error和Exception。

Error错误可不处理，比如内存溢出的Error（OutOfMemoryError）；而Exception中又分为RuntimeException（运行时异常）和非RuntimeException（非运行时异常）的异常。运行时异常一般是代码不严谨导致的，可手动处理；而非运行时异常是编译期异常，必须进行处理，否则代码不能编译通过。

若程序出现异常，JVM会做出默认处理，会输出异常的信息并立即结束程序。

异常体系图：



### 3.1.1 Exception类

Exception类中主要的方法：

public String getMessage()：返回异常消息的字符串

public String toString()：返回异常的简单信息描述，包括：此对象的完整类名和调用getMessage()返回的内容。

public void printStackTrace()：获取异常类名、异常信息和出现异常的位置，并输出这些信息。看起来在catch中使用printStackTrace()方法和程序出现异常的结果没有什么不同，但是注意：有了try...catch处理后，程序还会继续执行，而如果不进行异常处理，那么程序遇到异常就直接输出异常信息并结束。

## 3.2 处理异常

有时我们需要自行处理异常，如果让JVM默认处理，那么一旦程序出现异常就会立即停止运行。处理异常有两种方式，一种是自行处理，另一种是抛出异常。

### 3.2.1 自行处理

使用try...catch...finally捕获处理异常。格式：

try{

可能出现异常的代码;

...

} catch(异常类型名 变量名){

如果程序出现异常，就会执行这里的代码块

这里会捕获到异常，可进行处理...

} finally{

不管程序是否有异常，都会执行这里的语句，一般用于释放资源

}

通常，try里面包含的代码块尽量的少，以便准确定位异常。

其中，finally及其代码块可不写，如果这段代码有多个异常要处理，可以写多个catch来分别处理不同的异常，形式类似于：try{…}catch(){…}catch(){…}。需要注意：如果这些异常有父子类关系，则必须先捕获处理子类异常，再处理父类异常。因为会先匹配前面的异常，出现异常，try中就不执行了，而是执行catch中代码，如果有finally，再执行其中的代码，最后，会继续执行程序中其他代码。

上述格式还能简化成：

try{

catch(异常类型1 变量1 | 异常类型2 变量2 | ...){

...

}

}

这是JDK7支持的新方案。但是这样就是为所有的异常执行相同的处理方法，而且各个异常间不能有父子类关系，也就是要“平级”的。

### 3.2.2 抛出异常

抛出异常，适用于现在不适合处理或是没有权限处理的异常。用关键字throws向上抛出异常，让方法的调用者来处理异常。抛出异常要写在方法形参的一对小括号后边，格式：

throws 异常类名。

例如：public void readFile(String file) throws IOException{...}。

当然，如果抛出非运行时异常（编译期异常），则调用者必须处理，如果抛出运行时异常，调用者可以不处理。

### 3.2.3 自定义异常

实际开发中，可以自定义异常类。自定义异常类必须继承Exception(编译期异常)或者RuntimeException(运行期异常)类。

例子：

（1）自定义异常类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyException **extends** Exception {  **public** MyException() {}  **public** MyException(String message) {  *// 自定义异常只需要继承并写构造方法  // 该构造调用父类构造目的是显示错误信息* **super**(message);  } } |

（2）教师类中用到了自定义异常并抛出此异常对象。注意：throw关键字是主动抛出异常，抛出的是异常的对象，使用throw则一定会有异常。而throws是向上抛出可能出现的异常，由调用者处理可能出现的异常。

由于MyException继承了Exception，而不是RuntimeException，所以所在方法必须处理异常，可以抛出。

|  |
| --- |
| **package** com.zhang.test;  **public class** Teacher {  **public void** checkScore(**int** score) **throws** MyException {  **if**(score < 0 || score > 100) {  *// 这里抛出异常* **throw new** MyException();  }  } } |

（3）主方法可处理异常

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  Teacher teacher = **new** Teacher();  **try**{  teacher.checkScore(1000);  } **catch** (MyException e) {  e.printStackTrace();  }  } } |

### 3.2.4 细节

（1）子类重写父类方法时，必须抛出相同的异常或父类异常的子类异常，而不能抛出父类没有的异常。如果被重写的父类方法没有异常抛出,那么子类的方法绝对不可以抛出异常。这时如果子类方法内需要处理异常,只能用try...catch方式处理，不能throws抛出。

（2）使用多态机制处理异常：比如当一段代码有多个异常类需要处理时，可以直接处理Exception类异常即可，或者直接抛出Exception异常，因为这是所有需要处理的异常的父类。