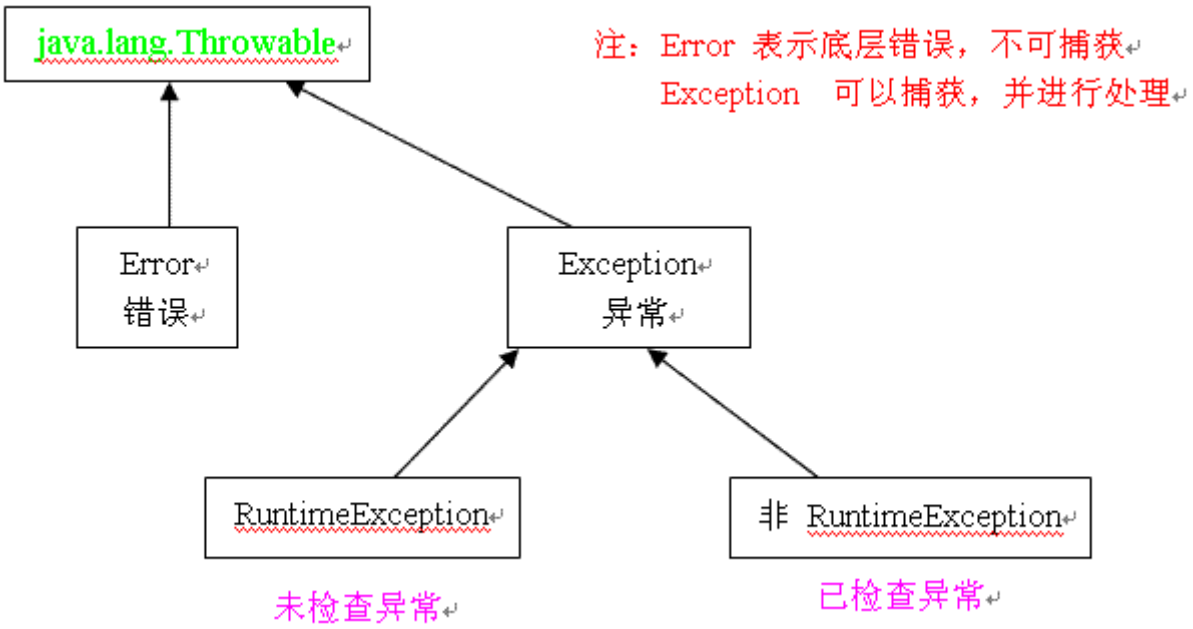
一、 异常 Exception

Java 的异常机制，可以提高程序的容错性。

特点： 编译时不报错， 在运行时发生例外退出

Java 中的异常机制链：



我们把 RuntimeException 叫做未检查异常，这类异常是可以避免的，如： 人为火灾

而非 RuntimeException (除 RuntimeException 之外的) 叫做已检查异常，这类异常不可避免，它必需被处理，如： 地震，等自然灾害

注： 对于未检查异常（RuntimeException），SUN 公司不鼓励 JAVA 程序员去做处理，但是要求程序员在编码时谨慎，应该尽量地避免未

检查异常的发生，如： 数组下标越界异常(ArrayIndexOutOfBoundsException)，空指针异常(NullPointerException)等.

而对于已检查异常， 虚拟机是要求你一定要做处理的。如：IOException, EOFException , SQLException 等

在 JAVA 中，一般以 JVM（虚拟机）为边界，界定了这两种异常：

凡是在 JVM 内部发生的异常，则一定是未检查异常，它是可避免的，而在 JVM 外部发生的异常，则必定是已检查异常，它不可避免，如 IO 异常。网络异常

1. 抛出异常的方法
   a) 是异常都可以抛出异常，当然也包括 构造方法
      使用： throw new Exception(); 来抛出一个异常。
      如果这个异常一直没有处理，(当然必需是未检查异常了)，则会一直抛到 main 方法，main 方法会中断退出.
2. 处理异常的方法
   a) 消极的处理方式：

i. 我们说过，已检查异常是一定要处理的。

如：

. . . .

Static void method() throws Exception {

. . . . . throws Exception 告诉编译器，我有可能发生异常，但它自己不处理，它交给调用它的方法去处理。

}

所以，throws XXX 异常只是一种申明，它不做任何的处理，我们叫这种方式为： 消极的处理方式。

b) 积极的处理方式

i. Try . . . catch

1. try {

//有可能出现异常的代码块;

} catch (Exception e) { //catch 后面跟得一定是个异常对像， 包括已检查和未检查异常。

// . . .

}

一旦 try 块中发生了异常，并能被 catch 后的异常所匹配，则语句会接着 catch 后面执行，正常返回。

一个 try {} 块， 后面可以跟多个 catch() 语句。

注意：多个 catch() 语句时，子类型在前面，父类型在后面，

因为：如果父类型在前面的话，父类型一旦匹配了，则子类型就不起作用了呀，这时 JVM 会出现编译错误，因为它不会永远不会执行

的代码存在的。

ii. Try . . . catch . . . finally

1. 最后，还可以加上 finally { . . . }
2. finally{ . . . } 中的代码块是一定会被执行的，不管有无异常发生。
3. 除非： try 或 catch 代码块中有 System.exit(0) ，提前退出
4. 所以， 根据 finally 块的这种特性，它一般用来释放资源。
5. 运行规则： 从 try 块出来后，如没有异常，进行 finally 块，如出现异常，会先进行匹配 catch 块，然后再进入 finally.

iii. Try . . . finally

1. 从 try 块出来到 finally 块，完后，会逐层抛出异常（如果有的话）

常见的 catch 语句处理方式如下：

```
Try {
    . . .    . . .
} catch (XXXException e ) {
    e.printStackTrace(); //此方法用来打印内存的堆栈信息
    System.out.println(e.getMessage()); //此方法返回在构造异常对象时所传递的那个字符串。
    }
```

3. 处理异常的原则

a) 什么时候 throws？ 什么时候 try . . . catch ?

i. 把一个异常留给真正该去处理的那个方法，通过 throws 逐层地传递到这

PDF-XChange Click to buy NOW! www.docu-track.com

个方法之外，（一般是由调用法来选择 是否处理）

b) 如何控制一个 try { } 块的范围？

i. 它应该以关联操作为准，把一组相关联的操作放在一个 try{} 块里面.

4. 如何自定义异常

```
Class MyException1 extends Exception {   // 一个已检查异常
    Public MyException1(String msg) {
        Super(msg);
    }
}
Class MyException2 extends RuntimeException { // 一个未检查异常
    Public MyException2(String msg) {
        Super(msg);
    }
}
```

例子： 写一个方法计算两整数之和，只要 a + b == 100 ，就抛出一个自定义的已检查异常。

Coding:

```
Public class TestException {
    Public static void main(String[] args) {
        Try {
            Add(20,80);
            . . .   . . .
        } catch (MyException e) {
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
    Public static void add(int a, int b) throws MyException {
        If((a + b) == 100 ) throw new MyException("custom exception");
    }
}
Class MyException extends Exception {
    Public MyException(String msg) {
        Super(msg);
    }
}
```

二、 JDK1。5 包装类简介：

a) 在 JAVA 中，共有 8 种简单类型， char, byte, short, int, long, float, double , boolean

b) 我们都知道 JAVA 是一门纯面向对像的语言，有很多的方法都适用于所有的对象类型，但不能用于简单类型，于是

JAVA 把 8 种简单类型都做了各自的封装类。如：

int → Integer, short → Short , byte → Byte, char → Character , long → Long, float → Float, double → Double, boolean → Boolean

在 JDK5。0 中，由简单类型到对象类型叫做： 封箱（Boxing）
由对象类到简单类型叫做： 解封（Unboxing）

如： Integer i1 = new Integer(4);

int i2 = Integer.parseInt("5"); int i3 = i1.intValue ; (把一个对象类型转成为基本类型)

Integer i4 = Integer.valueOf(i2) ; 把一个基本类型转成对象类型。

对于其它类型及方法详情，请查看 API。 包： java.lang 包

在 JDK5。0 中，支持自动封解箱操作。。

三、 内部类

定义在其它类内部的 类 叫做： 内部类， 所有，对于内类部，它可以访问外部类的私有成员。

根据内部类的定义方式不同，可以将它分为四大类：

1. 成员内部类 Member Inner Class : 做为一个类的成员， 与成员属性类似
2. 静态内部类 Static Inner Class ： 在成员类部类前加一个 static 修饰，是一种特殊的成员内部类
3. 局部内部类 Local Inner Class ： 定义在成员方法内部。 外面不能访问到。
4. 匿名内部类 Anonymous Inner Class ： 一种无名的局部内部类，也是一种最特殊的局部内部类。

注： 所谓的‘内部类’其实是编译时的概念， 在运行的时候还是两个类，（内部类也会产生对应的字节码文件）

现在分别来介绍这四种‘内部类’：

1. 成员内部类 Member Inner Class

a) 特点：在一个成员内部类里面，不允许有静态成员。 （想一想，为什么？）

i. 答： 因为成员内部类只是做为外部类的一个成员，而我们都知道，对于静态成员，JVM 在装载一个类的时候就会初始化，而此时并没有对象存在； 试想，即然外部类都没有对象，成员内部类也就不可以存在（非静态，依对象而存在），所以它不允许有静态成员。

例：

```
public class MemeberInnerDemo {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.disp();
    }
}
class Outer {
    private int index = 200;
    /**********************
    * @author yejf
    * 成员内部类: 可以访问外部类的私有成员
    * 使用时> 必需先创建外部类,然后才能创建内部类.
    */
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
    class Inner {
        private int index = 100;
        //static int si = 1000; //成员内部类不能有‘静态成员’
        public void disp() {
            System.out.println("Inner: " + index);
            System.out.println("Outer: " + Outer.this.index);
            Outer.this.disp();
        }
    }
    public void disp() {
        System.out.println("Index: "+index);
    }
}
```

对于成员内部类的构造，先要有外部类的对象，然后再用外部类对象去构造一个内部类对象： `Outer.Inner inn = out.new Inner();`

如要访问外部类中与内部类同名的成员：使用： Outer.this 加以限定： 如：Outer.this.index;

注：成员内部类的所有构造方法在运行时都会被自动加上一个外围类名的参数，这也就是为什么首先要有外围类对象名，才能构造成员内部类对象的原因。 所以，根据反射机制，获得一个成员内部类的默认构造成器是做不到的。。。

- 成员内部类的应用：
  - 我们都知道，单纯的接口并不能完全实现 C++中的‘多重继承’功能。所以：接口+内部类技求，才能做到这一点：

例如：人(Person)有一个 run()方法，机器(Machine)也有一个 run()方法，如果要写一个机器人(Robot)类，它要有人的功能，也要有机器的功能。实现如下：

```
abstract class Person {
    public abstract void run();
}
interface Machine {
    void run();
}
class Robot extends Person {
    public void run() { //覆盖父类的方法
        //实现代码: 人跑
    }
    //接下来要依赖内部类来实现 机器(Machine) 的功能
    private class Heart implements Machine { //写成私有，外面就不能访问
        public void run() {
            //实现代码: 机器启动
        }
```

```
        }
        //现在，提供一个对外公开的启动机器的方法：
        public void powerOn() {
            heart.run();
        }
        //提供一个内部类的对象：
        private Heart heart = new Heart();
    }
    //写一个测试类，
    class TestMemberInner {
        public static void main(String[] args) {
            Robot r = new Robot();  //生成一个机器人对象
            r.run();  // 首先是人 跑
            r.powerOn(); //再启动机器
        }
    }
```

2. 静态内部类 Static Inner Class

a) 特点：一种特殊的成员内部类，因为它是‘静态成员’，所以在构造静态内部类对象时，就不再需要外部类对象了，它可以直接用外部类名就可以。 如： Outer.Inner inn = new Outer.Inner();

b) 不能访问外部类的非静态成员。（想一想，为什么？）

i. 答： 原因就是： 静态成员不能访问非静态成员。

由于静态内部类的特点，所以可以把一个静态内部类看成是一个‘顶级类’

例子：

```
public class StaticInnerDemo {
    public static void main(String[] args) {
        //Outer2 outer2 = new Outer2(); //可以不用先有外部类对象了
        Outer2.Inner inner = new Outer2.Inner();
        inner.disp();
    }
}
class Outer2 {
    private int index = 100;//外部内非静态成员
    private static int si = 200; //外部内静态成员
    /******************
     * @author yejf
    * 静态内部类：只能访问外部类中静态成员
    */
    public static class Inner {
        private int index = 300;
        public void disp() {
            System.out.println("Outer: "+si);
            System.out.println("Inner: "+index);
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
                //System.out.println("Outer: "+Outer2.this.index); //
不能访问非静态成员
            }
        }
    }
```

3. 局部内部类 Local Inner Class

a) 特点：1. 能够让局部内部类访问的外部类的局部变量必需是 final 的，（也就是指此局部变量的值不会变）. (想一想：为什么？)

2. 它还可以屏蔽一些用户不关心的类。（可以访问外围类的所有成员）

3. 不能使用访问修饰符，而只能用： abstract 或 final 来修饰， 或什么都不加

现在，让我们来解答一个特点 1 的原因：

局部内部类只能在定义它的方法体内部使用，所以它产生的对象肯定是一个局部变量，而局部变量的生命周期仅在此方法体内， 但是对于 JVM 来说，

它肯定会把局部内部类的字节码文件加载进 JVM，所以 JVM 会做一件事情，就是为这些 final 就量增加一个拷贝，因为 final 决定了这些局部变量的值

不会改变。这样就算局部变量消息了也还可以使用。

例子：

```
public class LocalInnerDemo {
    public static void main(String[] args) {
        Outer3 outer3 = new Outer3();
        outer3.fn(70);
    }
}
abstract class AA {
    public abstract void m();
}
class Outer3 {
    private int index = 200;

    public void fn(final int a) {
        final int b = 30;
        /***************
        *  局部内部类: 可以访问外部类的私有成员,也可以访问局部变量,但
要求是此局部变量必需是 final的.
        */
        class Inner extends AA {
            private int index = 400;
            public void disp() {
                System.out.println("Outer3:
"+Outer3.this.index);
                System.out.println("Inner: "+index);
                System.out.println("a == "+a);
                System.out.println("b == "+b);
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
            }
            public void m() {
                System.out.println("Inner.m()");
            }
        }
        Inner inner = new Inner();
        inner.disp();
        inner.m();
    }
}
```

4. 匿名内部类 Anonymous Inner Class

a) 特点：

i. 它是一种特殊性的局部内部类

ii. 匿名内部类中不应该再添加一个方法，它是没意义的，因为连名字都没有，写了也没法调用

iii. 不能写构造方法，原因 （因为它根本没有类名）

注： 但绝对应该覆盖父类的方法或接口的方法。

匿名内部类应用非常灵活，基本上是用来继承一个抽象类或实现一个接口。下面我写两个例子，希望大家好好体会。

例子 1 ：

```
public class AnonymousInnerDemo {
    public static void main(String[] args) {
        Outer4 outer4 = new Outer4();
        IA ia = outer4.getIA(true);
        ia.print();
    }
}
interface IA {
    void print();
}
class Outer4 {
    private int index = 100;
    public IA getIA(boolean flag) {
        if(flag) {
            class AnonyIA implements IA {  //局部内部类
                public void print() {
                System.out.println("局部内部类: "+index);
                }
            }
            return new AnonyIA();
            } else {
                return new IA() { //匿名内部类
                    public void print() {
                    System.out.println("匿名内部类: "+index);
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
                    }
                };
            }
        }
    }
```

例子 2：我们根据用户的选择类型来决定返回一个什么样的‘比较器’， 比较器接口系统已提供，在 java.util.Comparator

```
    import java.util.Comparator;
    /**********************
    * @param type:  1 表示按书名字母； 2 表示按价钱； 3 表示按出版社字母； 。 。 。
    */
    class BookUtil {
        public static Comparator getSortRole(int type) {  //获取排序规则
            if(type == 1) {
                return new Comparator(){  //这里就是一个匿名内部类的典型应用，下同
                    public int compare(Object o1, Object o2) {
                        Book b1 = (Book)o1;
                        Book b2 = (Book)o2;
                        return b1.name.compare(b2.name);
                    }
                };
            } else if(type == 2) {
                    return new Comparator(){  //这里就是一个匿名内部类的典型应用
                    public int compare(Object o1, Object o2) {
                        Book b1 = (Book)o1;
                        Book b2 = (Book)o2;
                        if(b1.price > b2.price) return 1;   //按价格进行排序
                        else if(b1.price < p2.price) return -1;
                        else return 0;
                    }
                };
            } else if(type == 3) {
                    return new Comparator(){  //这里就是一个匿名内部类的典型应用
                    public int compare(Object o1, Object o2) {
                        Book b1 = (Book)o1;
                        Book b2 = (Book)o2;
                        return b1.publish.compare(b2.publish); //按
```

出版社进行排序

```
                    }
                };
            }
        }
    }

    class Book {
        private String name;
        private float price;
        private String publish;
        . . .     . . . //略 （基本的 getter/setter 方法）
    }
```

以上是针对 内部类的相关讲解，请仔细阅读并参照编写代码，以验证。

作者： 叶加飞 (steven ye)
mailto: leton.ye@gmail.com