



异常和断言

李玮玮

讲授思路

- 异常概述
- 异常处理机制
- 自定义异常
- 断言

异常概述

- 异常的层次结构
- 异常的分类

引入

- 发现错误的理想时机是在编译阶段，也就是在你试图运行程序之前。然而编译期间并不能找出所有的错误，余下的问题必须在运行期间解决。--Think in Java
- 这就需要有一个机制在运行期间如果出问题了，能够知道如何正确处理该问题。



什么是异常

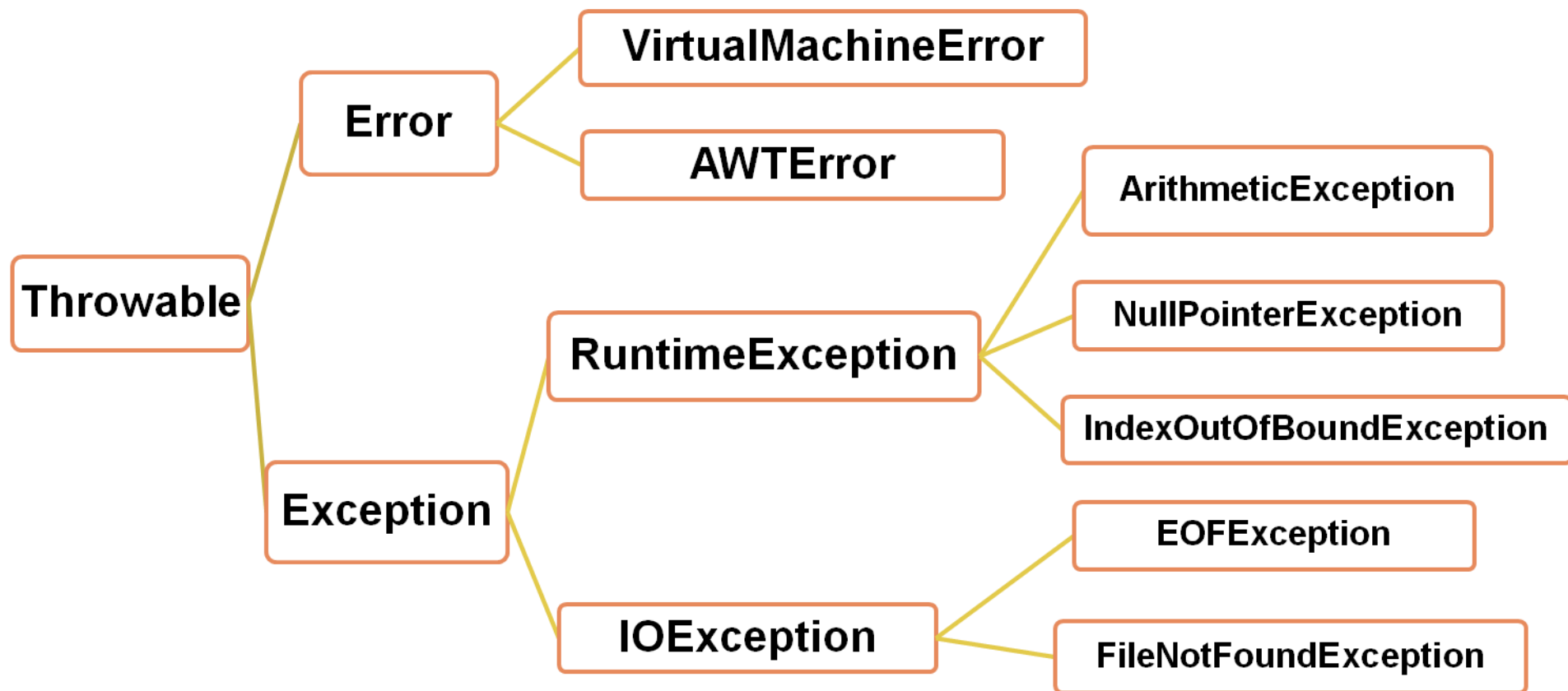
- 异常就是在程序运行的过程中所发生的不正常的事件，它会中断指令的正常执行
- 异常机制提供了程序退出的安全通道。当出现错误后，程序执行的流程发生改变，程序的控制权转移到异常处理器
- 导致异常的原因
 - 用户输入错误
 - 设备错误
 - 代码错误
 - 磁盘空间不足
 -

异常的提出

- 异常发生的后果
 - 丢失用户数据
 - 程序崩溃
- 用户期望的友好操作
 - 向用户通告错误
 - 保存所有的操作结果
 - 允许用户以适当的形式退出程序
- 解决方法——异常处理

异常的层次结构

- Java提供异常类来表示程序运行中发生的异常



异常的分类

- 异常分为2大类

- Error : 描述了Java运行系统中的内部错误以及资源耗尽错误
 - 唯一的解决方法：尽力使程序安全地终止
- Exception : 程序中需要关注的
 - 运行时错误 (RuntimeException) : 在 Java 虚拟机正常运行期间抛出的异常，由程序错误导致。Java编译器允许程序中不对这类异常做出处理。
 - 错误的类型转换
 - 数组下标越界
 - 访问空指针

未检查
异常
(unchecked)

其他错误 (如：IO异常、SQL异常) : 一般是外部错误，Java编译器要求在程序中必须处理这类异常

讲授思路

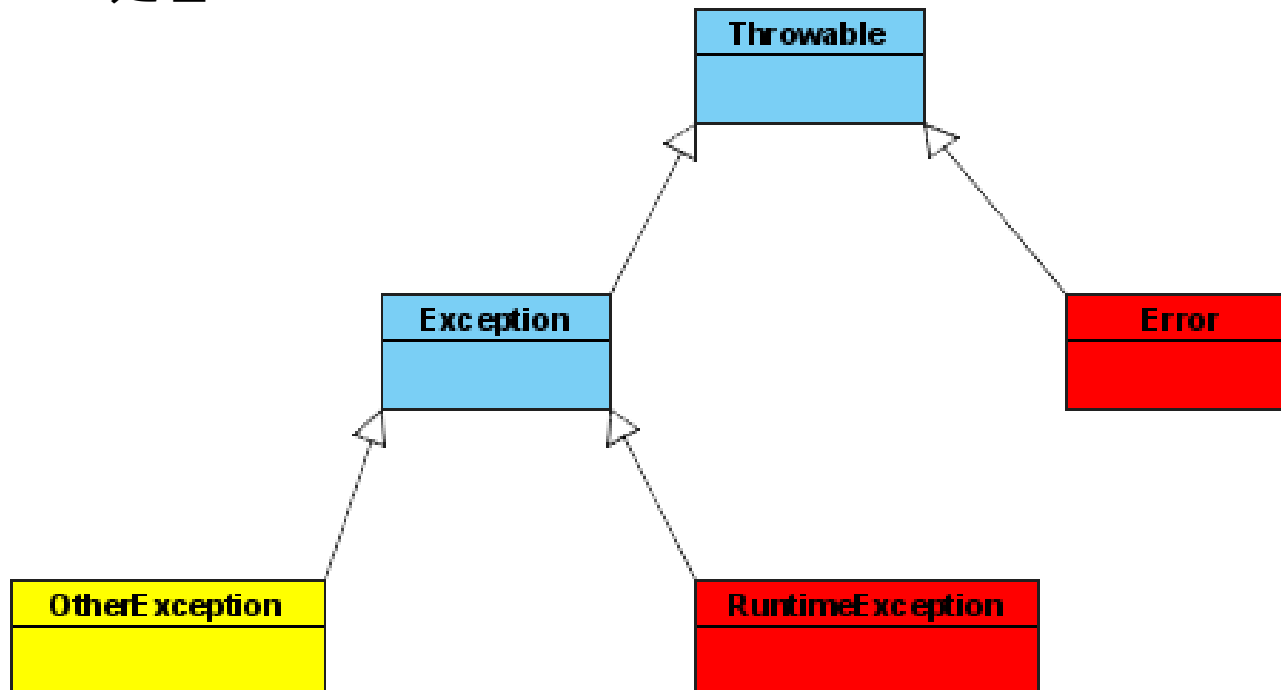
- 异常概述
- 异常处理机制
- 自定义异常
- 断言

异常处理机制

- 对于异常Java中提供了两种常见的方式：
 - 捕获异常
 - 抛出异常

异常处理机制

- Java可以灵活的处理异常
 - 捕获异常 (try—catch—finally)
 - 若当前方法有能力处理异常，就捕获并处理它
 - 抛出异常 (throw、throws)
 - 若当前方法没有能力处理异常，则只需抛出异常，交由方法调用者来处理

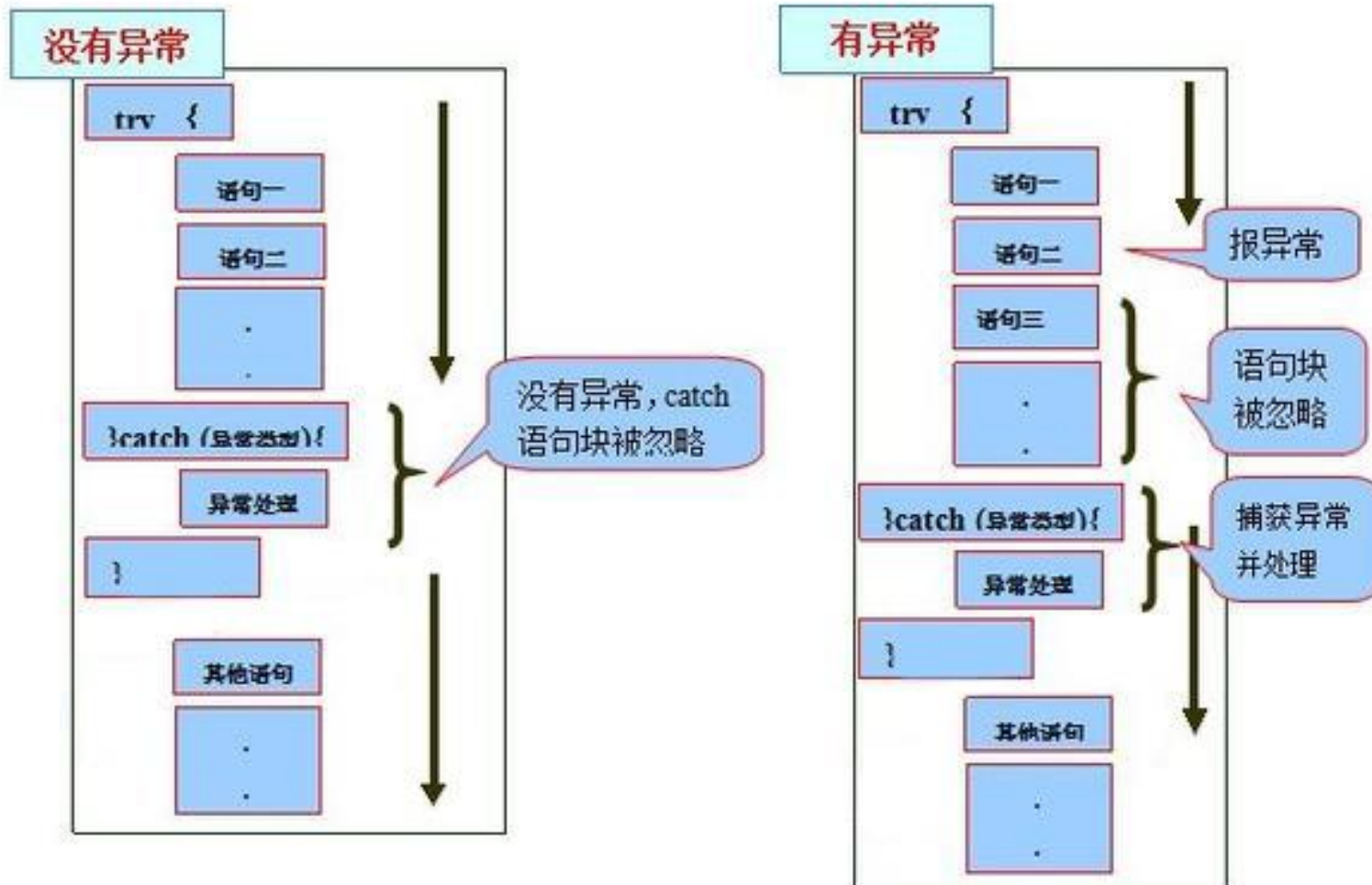


捕获异常

- 异常流程的代码和正常流程的代码分离，提高了程序的可读性，简化了程序的结构
- 使用try和catch捕获异常
 - try {
 - //接受监视的程序块,在此区域内发生的异常,
 - //由catch中指定的程序处理;
 - }catch(要处理的异常种类和标识符) {
 - //处理异常;
 - }

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
} catch (ClassNotFoundException e) {  
    e.printStackTrace();  
}
```

try、catch处理流程



try、catch处理流程

- try语句块中没有抛出任何异常
 - 程序会跳过catch子句
- try语句块中抛出了catch子句中说明的异常
 - 程序跳过try语句块中的其余代码
 - 程序执行catch子句中的异常处理代码



多重异常

- 一段代码可能会生成多个异常
- 当引发异常时，会按顺序来查看每个 catch 语句，并执行第一个类型与异常类型匹配的语句
- 执行其中的一条 catch 语句之后，其他的 catch 语句将被忽略
- 使用多重 catch 语句时，异常子类一定要位于异常父类之前

```
try{  
    .....  
} catch(ArrayIndexOutOfBoundsException e) {  
    .....  
} catch(Exception e) {  
    .....  
}
```

多重异常

- 一个catch可以捕获多个异常
- 多个异常类型之间用 “|” 分隔开
- 只有1个异常类型的标识符
- 多个异常类型之间不存在父子继承关系

```
try {  
    i = a / b;  
    System.out.println("try block");  
} catch (IndexOutOfBoundsException | ArithmeticException e1) {  
    System.out.println("发生异常， 请处理该异常！");  
}
```


try、catch使用注意事项

- try语句块只能有一个，而catch语句块可以有任意多个
- catch语句块紧跟在try语句块之后
- 建议对捕获的异常做适当的处理，而不仅仅是打印异常信息



finally语句块

- finally语句定义一个总是被执行的代码块，而不考虑是否出现异常
 - 无论try、catch是否执行，finally必定执行
- 不执行finally语句块的特殊情况
 - 在执行finally之前首先执行了 “System.exit(0);”
- finally语句块典型应用
 - 回收资源

finally语句块

- 语法结构
- try{
-
- }catch(Exception e){
-
- }
-
- finally{
- //资源回收
- }

课堂练习

- ExceptionFinallyDemo

```
public static void main(String[] args) {  
    System.out.println("return value of test(): " + test(10,5));  
    System.out.println("return value of test(): " + test(10,0));  
}  
public static int test(int a, int b){  
    int i = 0;  
    System.out.println("the previous statement of try block");  
    try {  
        i = a / b;  
        System.out.println("try block");  
    }catch(Exception e){  
        System.out.println("发生异常，请处理该异常！");  
    }finally {  
        System.out.println("finally block,系统资源被释放！");  
    }  
    return i;  
}
```

try-catch-finally使用注意事项

- try、catch、finally通常结合使用，需要注意以下事项
 - 无 catch 时 finally 必须紧跟 try

```
try{  
    //受监视的代码块  
    System.out.println( "这里是try块" );  
}  
//finally语句块  
finally {  
    System.out.println( "无论是否异常总会执行" );  
}
```

- catch 与 finally 不能同时省略
 - try、catch 和 finally 语句块之间不能插入任何代码（注释除外）

判断除数为0

- 分支结构

```
int division(int a, int b){  
    if(b == 0){  
        System.out.println("ERROR: 除数为0");  
    }else{  
        System.out.println("result: "+ a/b);  
    }  
}
```

- 异常处理(DivisionZeroExceptionDemo)

```
void division(int a, int b){  
    try{  
        System.out.print(a/b);  
    }catch(ArithmeticException e){  
        System.out.print("ERROR : 除数为0 ! ");  
    }  
}
```

异常处理的优势

- Java采用异常类表示异常
 - 把异常情况表示成异常类，可以充分发挥类的可扩展和可重用的优势
- 异常流程的代码和正常流程的代码分离，提高了程序的可读性，简化了程序的结构
- Java可以灵活的处理异常，如果当前方法有能力处理异常，就捕获并处理它，否则只需抛出异常，交由方法调用者来处理

异常处理机制

- 对于异常Java中提供了两种常见的方式：
 - 捕获异常
 - 抛出异常

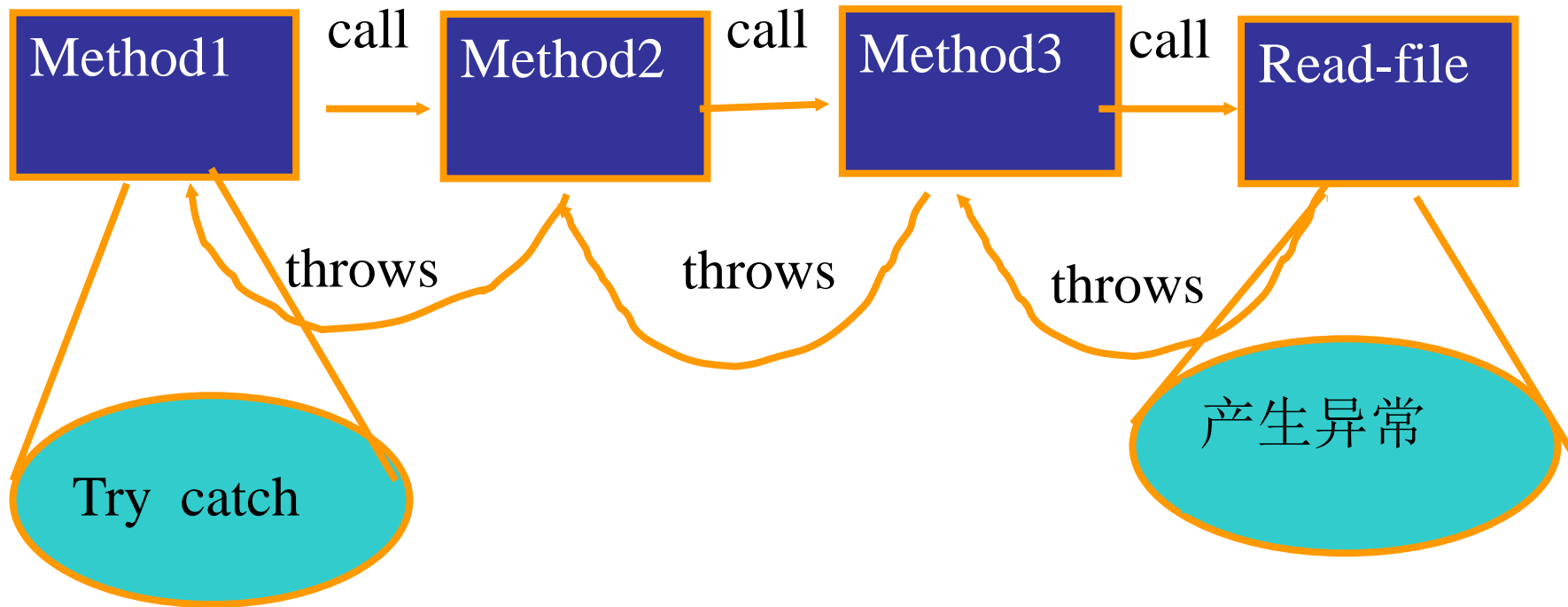
引入

```
void doA(int a) throws Exception1,Exception3{
    try{
        .....
    }catch(Exception1 e){
        throw e ;
    }catch(Exception2 e){
        System.out.println("出错了！");
    }
    if(a!=b){
        throw new Exception3("自定义异常");
    }
}
```

抛出异常

- 一个方法不处理它产生的异常,而是沿着调用层次向上传递,由调用它的方法来处理这些异常,叫抛出异常
- 声明异常使用throws关键字
 - throws是方法可能抛出异常的声明
 - 用在声明方法时,表示该方法可能要抛出异常
 - 语法
 - [(修饰符)](返回值类型)(方法名)([参数列表])[throws(异常类)]{.....}
 - 例如: `public void doA(int a) throws Exception1,Exception3{.....}`

抛出异常



声明异常示例

- 程序运行的结果？

```
public class Calculator{  
    void division(int opt1, int opt2) throws ArithmeticException{  
        int result = opt1/opt2 ;  
        System.out.println("result:" + result);  
    }  
    public static void main(String[] args){  
        Calculator cal = new Calculator();  
        try{  
            cal.division(10,5);  
            cal.division(10,0);  
        }catch(ArithmeticException e){  
            System.out.println("发生异常");  
        }  
    }  
}
```

抛出异常

- 抛出异常: 不是出错产生, 而是人为地抛出
 - 语法: `throw` (异常对象);
 - 如: `throw new ArithmeticException();`
- 如何抛出异常?
 - 确定要抛出的异常类
 - 系统提供的异常类
 - 自定义的异常类
 - 创建这个类的对象
 - 将该对象抛出

抛出异常

- 抛出异常示例
 - Calculator
- 结果

result:2
发生异常



```
void division(int opt1, int opt2) throws
    ArithmeticException{
    if(opt2==0){
        throw new ArithmeticException();
    }
    System.out.println("result:" + opt1/opt2);
}

public static void main(String[] args){
    Calculator cal = new Calculator();
    try{
        cal.division(10,5);
        cal.division(10,0);
    }catch(ArithmeticException e){
        System.out.println("发生异常");
    }
}
```

使用异常机制的建议

- 避免过大的try块，不要把不会出现异常的代码放到try块里面，尽量保持一个try块对应一个或多个异常
- 细化异常的类型，不要不管什么类型的异常都写成Excetpion
- catch块尽量保持一个块捕获一类异常，不要忽略捕获的异常，捕获到后要么处理，要么转译，要么重新抛出新类型的异常。
- 不要把自己能处理的异常抛给别人
- 不要用try...catch参与控制程序流程，异常控制的根本目的是处理程序的非正常情况

- 系统自动抛出的异常
 - 所有系统定义的编译和运行异常都可以由系统自动抛出，称为标准异常，并且 Java 强烈地要求应用程序进行完整的异常处理，给用户友好的提示，或者修正后使程序继续执行
- 语句抛出的异常
 - 用户程序自定义的异常和应用程序特定的异常,必须借助于throws 和 throw 语句来定义抛出异常
- throw和throws的区别
 - throws语句用在方法声明后面，表示该方法会抛出哪些异常，使它的调用者知道要捕获这些异常。
 - throw语句用在方法体内，表示抛出异常，是具体向外抛异常的动作，它抛出一个异常实例。
 - throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常。

讲授思路

- 异常概述
- 异常处理机制
- 自定义异常
- 断言

自定义异常


- 常见异常
- 自定义异常

NullPointerException

- 常见原因：
 - 试图访问一个空对象时
- 常见解决方法：
 - 通过断点先确定哪个对象为空，再找到该对象没实例化的原因

NumberFormatException

- 常见原因：
 - 程序员认为变量是数值 “123” ，但那实际内容可能是 “abc”
- 常见解决方法：
 - 根据控制台报的异常，找到出错的位置，看异常中数据是什么，确定是否有问题。



```
java.lang.NumberFormatException: For input string: "a"  
FormatException.forInputString (NumberFormatException.java:48)  
r.parseInt (Integer.java:447)  
r.parseInt (Integer.java:497)
```

ClassNotFoundException

- 常见原因：
 - 当使用第三方Jar包时，可能由于没导入包，或者包的版本不对，而导致该包中没有程序员用到的类
- 常见解决方法：
 - 导入该包
 - 可以打开第三方Jar包，确认是否有该类，如果没有，可以更改包的版本。

ArrayIndexOutOfBoundsException

- 常见问题：
 - 访问超过数组或集合最大索引值的数据
- 常见解决方法：
 - 通过断点等方法，找到是否是循环超出了范围等原因

ClassCastException

- 常见问题：
 - 在集合中存入某对象，但是取出时却强转成其他对象
- 常见解决方法：
 - 通过加断点的方法，查看集合中存入的是什么对象，再和自己要强转的对象类型比较便知。
 - 泛型，已经限制了存入类型，就不会出现转换异常了。

自定义异常

- 常见异常
- 自定义异常

自定义异常

- 如果JDK提供的异常类型不能满足需求的时候，程序员可以自定义一些异常类来描述自身程序中的异常信息.
- 程序员自定义异常必须是Throwable的直接或间接子类.
- 在程序中获得异常信息一般会调用异常对象的getMessage，printStackTrace，toString方法，所以自定义异常一般会重写以上三个方法.

自定义异常示例

```
public class SpecialException extends Exception {  
    private static final long serialVersionUID = 1L;  
    @Override  
    public String getMessage() {  
        return "SpecialException三角形构造失败";  
    }  
    @Override  
    public void printStackTrace() {  
        System.out.println("三角形构造失败，异常类型：  
"+this.getClass().getName());  
    }  
    @Override  
    public String toString() {  
        return "三角形构造异常，类型为：  
"+this.getClass().getName();  
    }  
}
```

自定义异常使用示例

```
public class Triangle{
    Double a;
    Double b;
    Double c;
    public Triangle(Double a, Double b, Double c) throws
    SpecialException {
        if (a + b < c) {
            throw new SpecialException();
        } else if (a + c < b) {
            throw new SpecialException();
        } else if (b + c < a) {
            throw new SpecialException();
        }
        this.a = a;
        this.b = b;
        this.c = c;
    }
}
```

讲授思路

- 异常概述
- 异常处理机制
- 自定义异常
- 断言

断言

- JavaSE1.4中引入的新特性，用于检查程序的安全性。
- 关键字assert、java.lang.AssertError类
- 当需要在一个值为FALSE时中断当前操作的话，可以使用断言
 - 可看做是异常处理的高级形式，断言的布尔状态为true则没问题，如果为false，则抛出AssertError异常。

断言的使用方法

- 两种形式
 - `assert Expression1`
 - `assert Expression1:Expression2`
- 断言在默认情况下是关闭的，启用断言验证假设须用到java命令的参数-ea
- 在Eclipse中的run configuration中添加默认参数-ea

断言示例

```
public class AssertTest {  
    public static void main(String[] args) {  
        AssertTest at = new AssertTest();  
        at.assertMe(true);  
        at.assertMe(false);  
    }  
    private void assertMe(boolean boo){  
        assert boo?true:false;  
        System.out.println("true condition");  
    }  
}
```

true condition

Exception in thread "main" java.lang.AssertionError
 at AssertTest.assertMe(AssertTest.java:13)
 at AssertTest.main(AssertTest.java:7)

断言示例

```
public class AssertTest2 {  
    public static void main(String[] args){  
        AssertTest2 at = new AssertTest2();  
        at.assertMe(true);  
        at.assertMe(false);  
    }  
    private void assertMe(boolean boo){  
        String s = null;  
        assert boo?true:false:s = "hello world";  
        System.out.println("true condition");  
    }  
}
```

true condition

Exception in thread "main" java.lang.AssertionError: hello world
at AssertTest.assertMe(AssertTest.java:14)
at AssertTest.main(AssertTest.java:7)

使用断言的情况

- 可以在预计正常情况下程序不会到达的地方放置断言：
`assert false`
- 断言可以用于检查传递给私有方法的参数。（对于公有方法，因为是提供给外部的接口，所以必须在方法中有相应的参数检验才能保证代码的健壮性）
- 使用断言测试方法执行的前置条件和后置条件
- 使用断言检查类的不变状态，确保任何情况下，某个变量的状态必须满足。（如age属性应大于0小于某个合适值）

不使用断言的情况

- 断言语句不是永远会执行，可以屏蔽也可以启用
 - 不要再public的方法里面检查参数是不是为null之类的操作
 - 例如
 - ```
public int get(String s){
 assert s != null;
 }
```
    - 假如需要检查也最好通过if s = null 抛出NullPointerException来检查
  - 不要使用断言作为公共方法的参数检查，公共方法的参数永远都要执行
  - 断言语句不可以有任何边界效应，不要使用断言语句去修改变量和改变方法的返回值

# 总结

- 异常概述
- 异常处理机制
- 自定义异常
- 断言



**Thank You**