异常及异常处理

程序错误

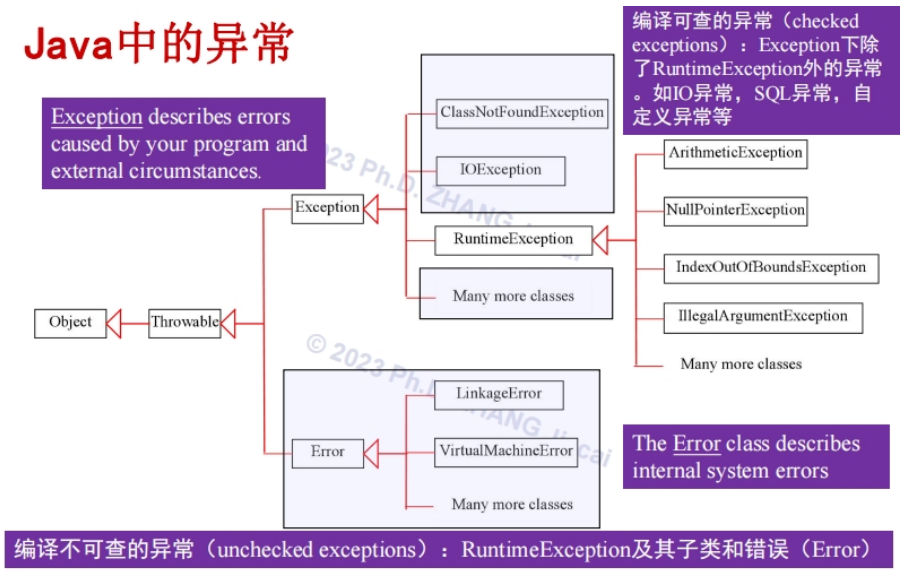
程序运行中出现的问题

- 在进行程序设计时，错误的产生是不可避免的
  - 比如要打开的文件不存在，被除数为0，网络中断
- 程序错误
  - 编译错误：没有遵循Java语言规范，编译器能发现，Javac命令就能提示错误原因和位置，刚接触Java语言最常遇到的问题
  - 逻辑错误：程序没有按预定的逻辑顺序，输入输出关系错误
  - 运行（时）错误：程序执行时，运行环境或变量取值异常导致的错误
- 如何处理运行时错误，如何补救：
  - 把错误交给谁去处理
  - 程序又该如何从错误中恢复
- 异常
  - 程序运行过程中发生的异常事件，比如除0溢出、数组越界、文件找不到
  - 这些事件的发生将阻碍程序的正常运行
  - 为了增加程序的强壮性，程序设计时，必须考虑到可能发生的异常情况并做出相应的处理

Java中的错误类与异常类

- **错误类**
  - 严重的错误情况，一般来说程序无法恢复，只能中止
  - 比如内存错误，虚拟机报错
- **异常类**
  - 试图打开文件不存在
  - 网络中断
  - 数组越界，被除数为0等
  - 找不到要装载的类
- **程序碰到异常**
  - 不一定中止，可以编写代码处理异常状况，如文件不存在，先创建新文件，再打开文件等
  - 一旦发生异常，如果不处理这个异常，虚拟机将终止呈现

Java中的异常



- 举例
  这份代码好吗？

```java
import java.util.Scanner;

public class Quotient {
        public static void main(String[] args) {
                Scanner input = new Scanner(System.in);

                // Prompt the user to enter two integers
                System.out.print("Enter two integers: ");
                int number1 = input.nextInt();
                int number2 = input.nextInt();

                if (number2 != 0)
                        System.out.println(number1 + " / " + number2 + " is " +
(number1 / number2));
                else
                        System.out.println("Divisor cannot be zero ");
        }
}
```

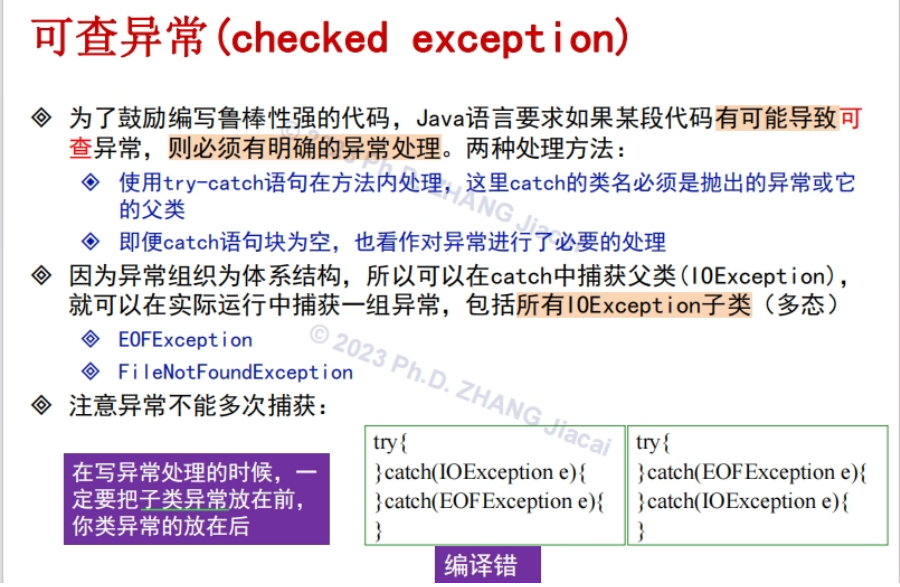
看似给出了除数为0的处理机制，但是不够优雅：
一是没有给程序机会补救，就直接退出了，太强势；
二是程序必须事先知道所有可能出错原因，出错现场没有信息供程序分析原因；
三是正常流程与异常流程混在一起，影响程序可读性

常见异常

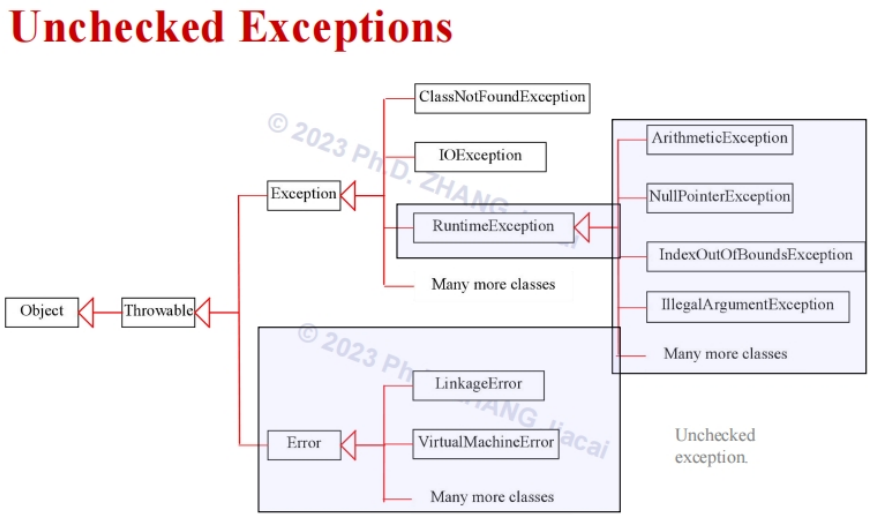
- Java预定义了几个异常，unchecked(不可查，不会在编译时报错)：
  - `ArithmeticException`，整数被0除，浮点数除0产生这个异常。
  - `NullPointerException`，访问一个没有实例的对象的成员。
  - `NegativeArraySizeException`，生成大小为负数的数组。

    - `ArrayIndexOutOfBoundsException`，访问下标超过数组大小的数组元素。
    - `SecurityException`，最典型的抛出是浏览器，当一个小程序（applets）试图进行下面操作时，SecurityManager类抛出SecurityException
- 常见的异常方法
    - `getCause()`：返回此 throwable 的 cause。如果 cause 不存在或未知，则返回 `null`。
    - `getMessage()`：返回此 throwable 的详细消息字符串。返回的字符串包含了关于发生异常的一些详细信息。这些信息通常包括异常的类型和详细信息，但不包括异常的堆栈跟踪信息。
    - `printStackTrace()`：在控制台上打印此 throwable 及其追踪的堆栈轨迹，可以快速地显示异常的堆栈跟踪，这样开发者就可以知道异常是在哪里抛出的，以及异常抛出的上下文是什么。
- *程序处理不了`error`及其子类*

可查异常



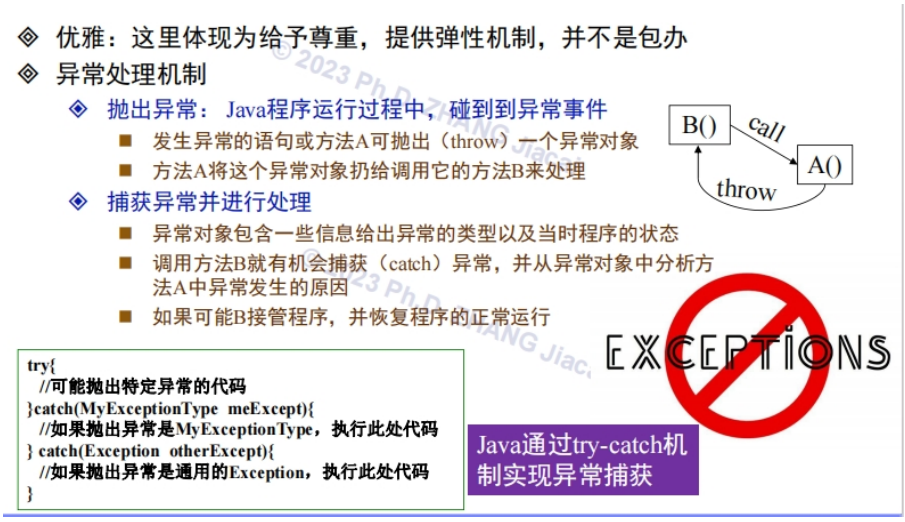
不可查异常



- unchecked exception：`RuntimeException, Error及其子类`，程序不会强制你来处理，但一旦发生可能导致程序终止
- 避免try-catch语句让代码太笨重累赘，Java*允许不处理unchecked exception*
- unchecked exception通常与编程逻辑失误有关
- 其它异常，checked exception必须处理，否则，编译不通过

Java异常处理机制

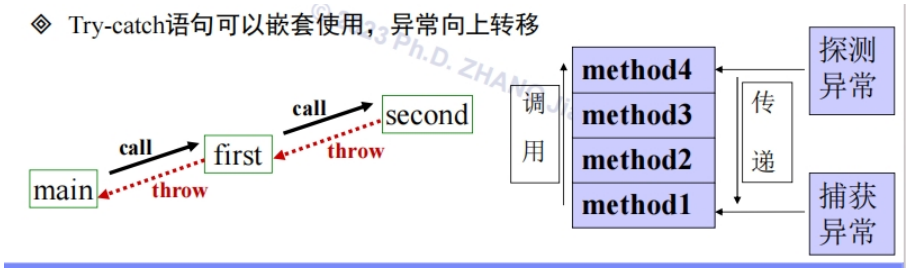
Java为异常准备了特殊的弹性异常处理机制

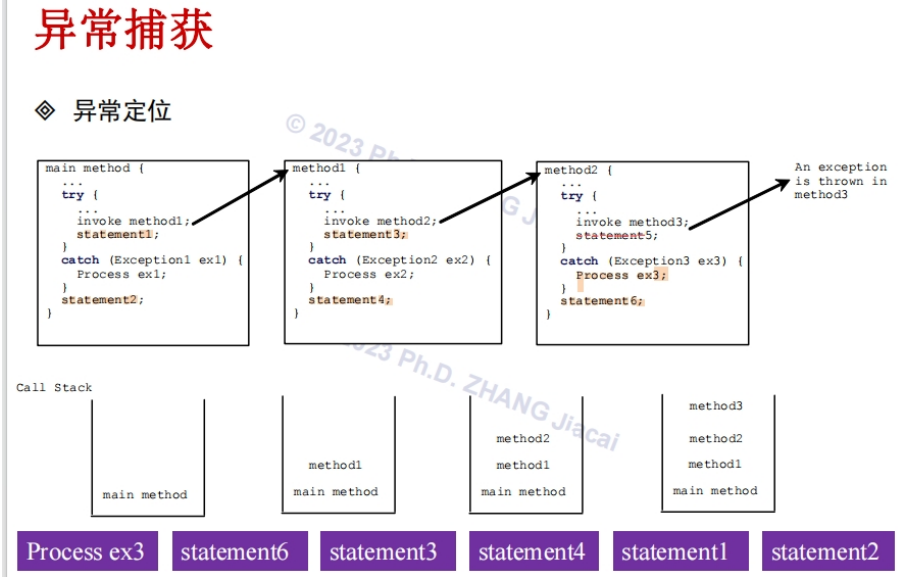


- 某语句可能抛出异常，可用`try{受保护代码块}`语句监测异常，并用`catch(){异常处理代码块}`语句捕获并处理异常
- 如果try代码块中异常发生
  - 程序跳转到对应的`catch`代码块，`try`异常后面的代码不执行
  - 如果有`finally`代码块，执行`finally`代码块
  - `finally`无论有无异常均会执行
  - `catch`只会执行一种类型，如果异常没有对应的`catch`类型，将中止代码块运行
  - 在写异常处理时，一定要把子类写在后，父类写在前

try-catch机制

- Java中的异常处理采用调用**堆栈机制**
  - 如果一个方法A中异常没有在当前的`try`和`catch`语句处理，那么继续将异常抛给上级方法
  - 以此类推。如果直到`main`方法，异常还没有处理，程序将结束
- Java中的异常处理结构是`try和catch`语句，将引发异常的代码放在try语句块中，接下来是一系列catch语句块，每个块对应一个异常
  - 当`try`语句块中抛出异常时，执行对应的`catch`语句块；一个try语句块可以跟随多个`catch`语句，每个`catch`语句处理一个异常

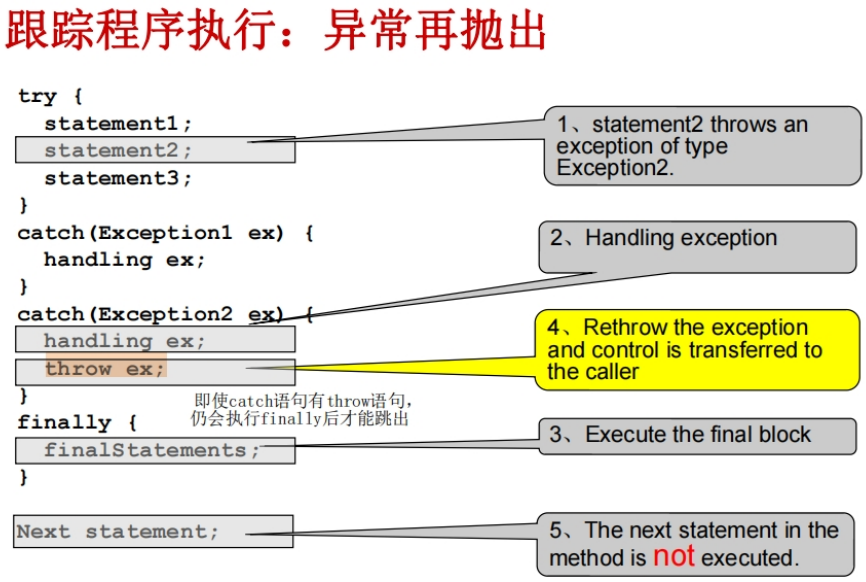




finally机制

`finally`语句定义了一个语句块，无论是否抛出错误，永远都要执行

- `finally`语句为异常提供了一个统一出口，使得程序控制匹配前能够对程序状态进行统一管理
- 只有一种情况`finally`语句不会执行：受保护代码中调用`System.exit()`强行终止程序
- 即使`return`语句在`try`代码块内，仍然会执行`finally`语句
- `finally`最多只有一个



示例

```java
public class HelloException {
        public static void main(String[] args) {
                int i = 0;
                String greetings [] = {"Hello", "Hi", "Going!"};
                try {
                        while(i < 4) {
                                System.out.println(i + ":" + greetings[i]);
                                i++;
                        }
                }catch(ArrayIndexOutOfBoundsException e){
                        System.out.println("Re-setting index value");
                }finally {
                        System.out.println("This is always print and i:" + i);
                }
        }
}
```

输出：

```
0:Hello
1:Hi
2:Going!
Re-setting index value
This is always print and i:3
```

Q：如果把 `while` *放在外面？*

因为`i=3`后`i++`不执行，持续进入`catch`语句，导致死循环

修改方式：

改为`System.out.println(i + ":" + greetings[i++])`，这样到了`i=3`时虽然异常，但仍能进行自加

```
0:Hello
This is always print and i:1
1:Hi
This is always print and i:2
2:Going!
This is always print and i:3
Re-setting index value
This is always print and i:4
```

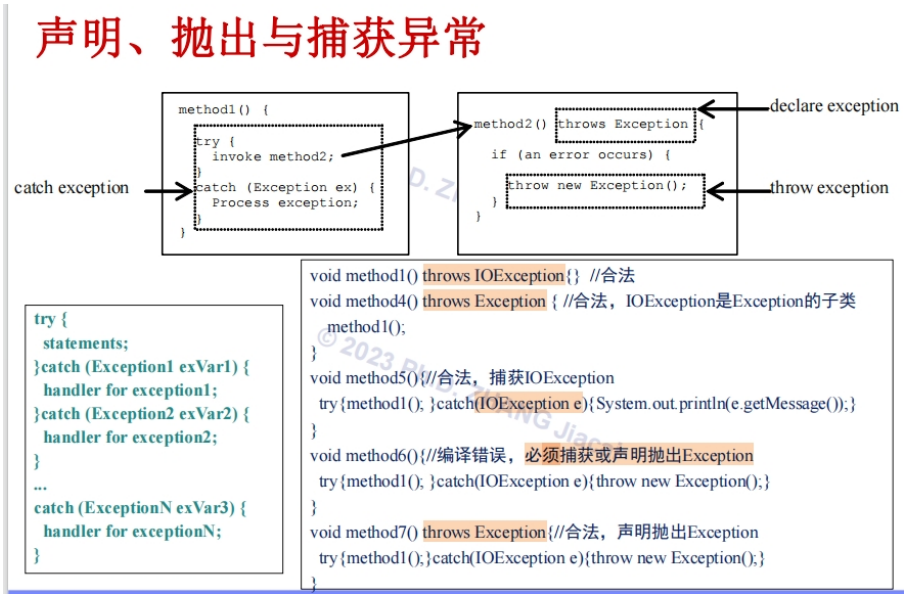
主动抛出异常

> 异常不在当前方法内处理，而是抛给交调用它的方法来处理。这时方法就要声明抛出异常，使得*异常对象可以从调用栈向后传播*，直到有合适的方法捕获它为止

- 程序运行过程中如果出现异常，程序可以处理异常，系统也可抛出异常，根据程序的检测结果用`throw`语句主动抛出异常
- ***注意`throw`和`throws`语句区别***

- 方法声明中`throws`后跟的是异常类型，声明抛弃异常
- 方法内语句`throw`跟的是异常对象

```java
pubic void myMethod() throws IOException;
pubic void myMethod() throws OtherException;

if(count == 0){
        throw new ArithmeticException("数目不正确");
}
if(count < 0){
        throw new MyProjException("计数错误");
}
```



`method5` 方法中使用了一个 `try-catch` 块来处理 `method1` 方法可能抛出的 `IOException`。如果在 `method1` 中的代码抛出了 `IOException`，那么这个异常将被 `catch` 块捕获，然后打印异常信息。

在这种情况下，`method5` 方法不需要使用 `throws` 语句声明该方法可能抛出 `IOException`，因为该异常已经在方法内部被处理了。当你使用 `try-catch` 块捕获异常时，你实际上是在告诉编译器：“我知道这里可能会发生异常，但我已经有了处理它的方法，所以你不必再担心它”。因此，编译器不会要求你在方法签名中使用 `throws` 语句来声明该异常。

总的来说，如果一个方法内部包含了对某种可能抛出的异常的处理（无论是通过 `try-catch` 块还是其他方式），那么该方法就不需要在其签名中使用 `throws` 语句来声明这种异常。

抛出异常可以传递更多信息

JAVA

```java
import java.util.Scanner;

public class QuotientWithException{  // uncheck异常可以不在方法里面throws异常类
        public static int quotient(int number1, int number2){
                if(number2 == 0) throw new AtithmeticException("Divisor cannot be
zero");
                return number1 / number 2;
        }

        public static void main(String[] args){
                Scanner input = new Scanner(System.in);
                System.out.print("Enter two integers: ");
                int number1 = input.nextInt();
                int number2 = input.nextInt();
                try{
                        int result = quotient(number1, number2);
                        System.out.println(number1 + "/" + number2 + "is" + result);
                }catch(ArithmeticException ex){
                        System.out.println(ex.getMessage());
                }
                System.out.println("Execution continues...");
        }
}
```

```
Enter two integers: 2 0
Divisor cannot be zero
Execution continues...
```

- 返回值只能告诉调用方法是否执行成功
- 异常包含更多信息。这样调用方法能更准确捕获程序出现的问题
- `getMessage()`方法返回错误信息的备注
- **算数异常属于unchecked异常**，可以不在方法定义处写`throws`

抛出异常方法的重写

子类重写的方法抛出的异常必须：

- **异常范围不能扩大**。只能抛出父类方法抛出的异常或其子类
- 异常**个数可以少于父类，也可以多于父类**，列举更多父类抛出的异常个数，但**不能抛出父类没有的异常**

示例：

JAVA

```java
import java.io.*;
public class TestMultiA{
        public void methodA() throws IOException, RuntimeException{
                ...; throw new IOException; ...
        }
}
class TestMultiB1 extends TestMultiA{
        public void methodA() throws FileNotFoundException, UTFDataFormatException,
ArithmeticException{}
}

class TestMultiB2 extends TestMultiA{
        // 编译错误，因为父类没有抛出SQLException这个错误
        public void methodA() throws FileNotFoundException,
UTFDataFormatException,ArithmeticException, SQLException{}
}

class TestMultiB3 extends TestMultiA {
        public void methodA() throws java.io.IOException {}
}
```

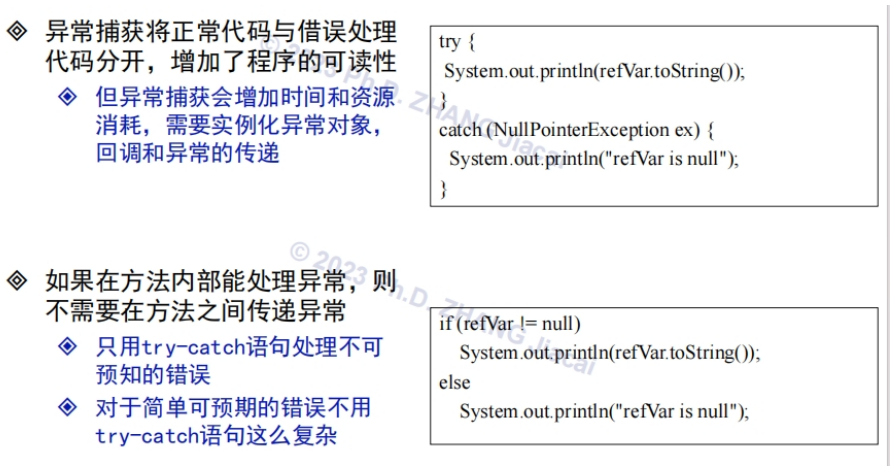
定制异常

- 用户可以定制异常，但必须继承`Exception`类
  - `Exception`类包含异常基本结构
- 使用的时候要抛出异常
  - 抛出`ServerTimeOutException`对象
  - 最好在抛出异常的地方生成异常，因为程序代码位置添加到了异常构造器中，如果在其他代码行生成异常，那么异常中的程序出错信息就是误导

JAVA

```java
package lesson10;

public class MainCatcher {
        public void methodA(int money)throws SpecialException{
                if(--money <= 0) throw new SpecialException("Out of money");
                System.out.println("methodA");
        }

        public void methodB(int money)throws SpecialException{
                methodA(money);
                System.out.println("methodB");
        }

        public static void main(String[] args) {
                try {
                        new MainCatcher().methodB(1);
                        System.out.println("main");
                }catch(SpecialException e) {
                        System.out.println(e.getMessage());
                }
        }
}

class SpecialException extends Exception{  // 定制异常
        public SpecialException(String s) {
                super(s);
        }
}

// 输出:  Out of money
```

注意事项



课堂总结

- 异常处理的**五个关键字**：`try、catch、finally、throws、throw`

- 异常处理流程由`try、catch`和`finally`三个代码块组成
  `try`代码块包含了可能发生异常的程序代码
  `catch`代码块紧跟在`try`代码块后面，用来捕获并处理异常
  `finally`代码块用于释放被占用的相关资源
- `Exception`类表示程序中出现的异常，可分为**可检查异常和运行时异常**

注意：

课堂小问题(3)

在执行tryThis()方法，如果problem()方法抛出Exception，程序将打印什么结果

```
public void tryThis() {
   try {
      System.out.println("1");
      problem();
   } catch (RuntimeException x) {
      System.out.println("2");
      return;
   } catch (Exception x) {
     System.out.println("3");
     return;
   } finally {     System.out.println("4");   }

   System.out.println("5");
}
```

答案：打印如下内容：
1 3 4

没有打印5，原因是catch语句中有return，直接返回