Java软件设计基础

中国科学技术大学计算机科学与技术学院

7. 异常处理

1. 异常处理机制概述

- 异常概述
  - 异常(Exception)就是程序在运行过程中所发生的异常事件，即不可预测的非正常情况。没有异常捕获和处理代码的程序会非正常终止，并可能引起严重问题。
  - Java语言提供的异常处理机制主要是用来处理程序执行过程中产生的各种错误，使用异常对程序给出一个统一和相对简单的抛出和处理错误的机制。
  - 如果一个方法本身能抛出异常，当所调用的方法出现异常时，调用者可以捕获异常使之得到处理；也可以回避异常；而非简单粗暴的给出错误终止的结果。

目前主流的编程语言，如Java、C#、Ruby、Python等都提供了成熟的异常处理机制。

- 例6.1 一个简单的程序

```
import java.util.Scanner;
public class  ScannerNoException{
        public static void main(String[] args) {
                Scanner s=new Scanner(System.in);
                System.out.println("Please input an integer:");
                int i=s.nextInt();
                System.out.println("The number is:"+i);
        }
}
```

```
Please input an integer:
1.2
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at ScannerNoException.main(ScannerNoException.java:8)
```

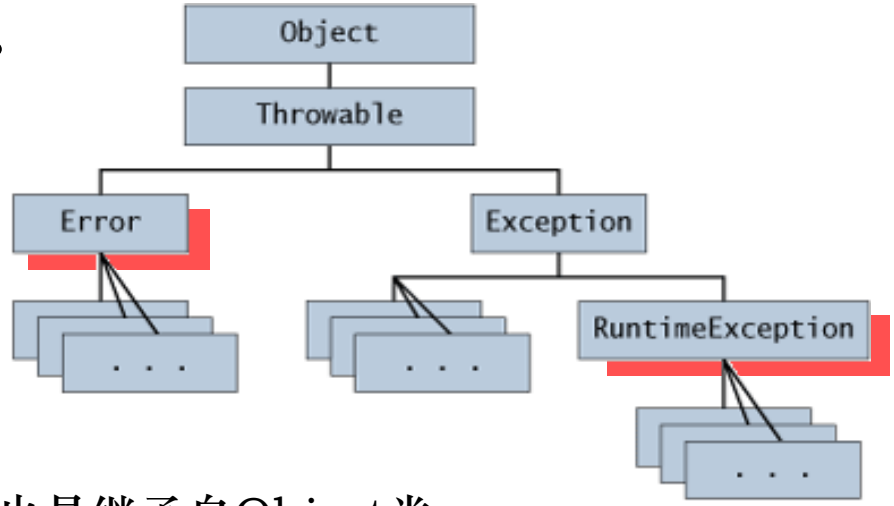
可看出main方法中出现异常

- 例6.2 带有异常处理的程序

```
import java.util.*;
public class  ScannerException{
      public static void main(String[] args) {
            boolean pin=true;
            int i;
            Scanner s=new Scanner(System.in);
            System.out.println("Please input an integer:");
            do{
               try{
                              i=s.nextInt();
                              System.out.println("The number is:"+i);
                              pin=false;
               }catch(InputMismatchException e){
                              System.out.println("Invalid input,try again:");
                              s.nextLine();
               }
            }while(pin);
      }
}
```

```
Please input an integer:
1.1
Invalid input,try again:
1
The number is:1
```

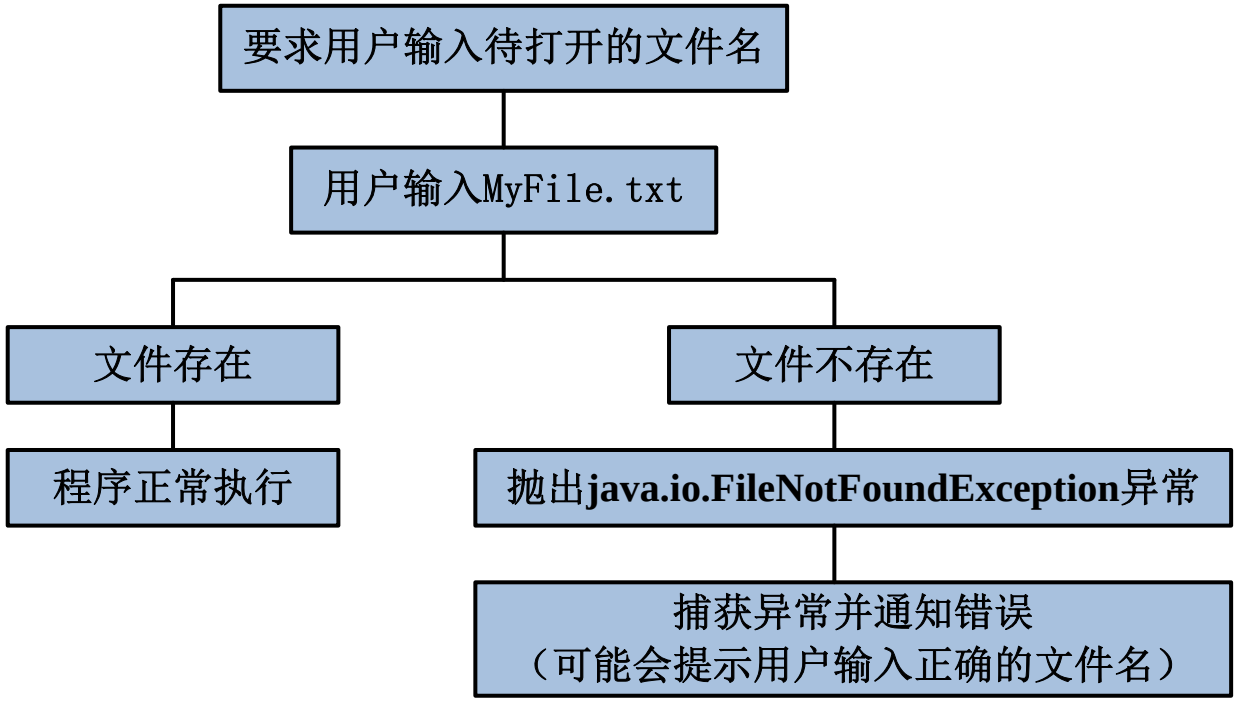
- 异常层次结构
  - Java中的异常完全按照类的层次结构进行组织；
  - Java将异常看作一个类，并且按照层次结构来区别不同的异常。
  - 异常类定在java.lang包中。



  - 说明
    - 根结点为Throwable，当然也是继承自Object类；
    - Throwable类包含在java.lang中，它的子类包含在不同的包中。
      - **与GUI相关的错误包含在包java.awt中；**
      - **与数值有关的异常大多包含在java.lang中；**
      - ……

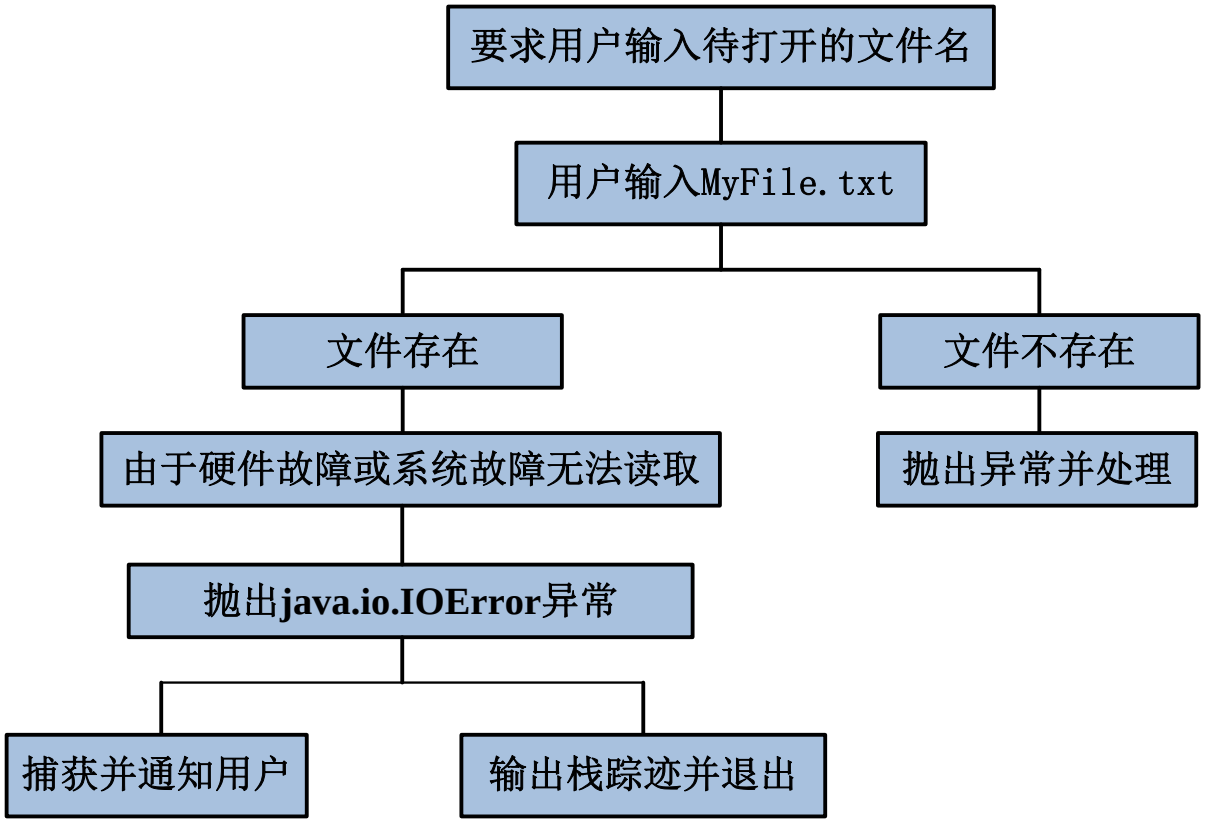
2. 异常的类型

- 可控异常(Checked)
  - 可控异常又称必检异常，Java认为这类异常都是可以别处理（修复）的异常，指编译器会强制程序员显式的检查并处理它们。除了RuntimeException类及其子类以外的Exception类及其子类属于可控异常。



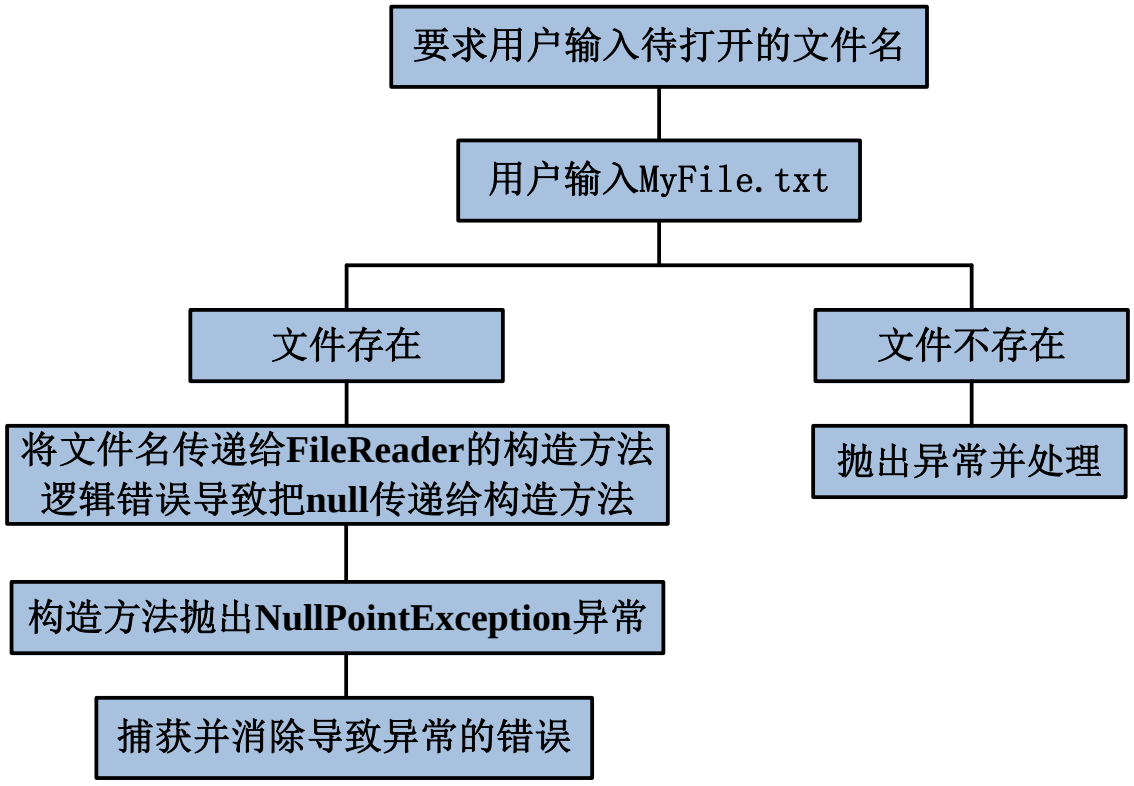
- 不可控异常(unchecked exception)
  - 系统错误（Error）
    - 一般是指与虚拟机相关的问题，如系统崩溃、虚拟机错误、动态链接失败等，无法恢复或不可能捕获，将导致应用程序中断：
    - 例如：

异常类名	说明
LinkageError	一个类对另一个类有某种依赖关系，前者编译后，后者做了不相容的修改
VirtualMachineError	Java虚拟机被终端或者没有必须的资源可用，不能继续运行
AWTError	GUI实时系统的严重错误


要求用户输入待打开的文件名
用户输入MyFile.txt
文件存在
文件不存在
由于硬件故障或系统故障无法读取
抛出异常并处理
抛出java.io.IOError异常
捕获并通知用户
输出栈踪迹并退出

- 运行异常（RuntimeException）
  - 是应用程序内部的异常情况，应用程序通常不能预测它们并且不能从中恢复。这些异常情况通常表示编程错误，比如逻辑错误、API的不正确应用、不合适的转换、访问一个越界数组或数值错误等。运行异常通常由Java虚拟机抛出。
  - **应用程序能够捕获这种异常，但是消除导致异常的错误可能更有意义。**
  - 例如：

异常类名	说明
IllegalAccessException	非法访问错误异常
IndexOutOfBoundsException	索引越界异常
ArithmeticException	算数错误异常，如除数为0
NullPointerException	访问空对象的方法或变量是产生的异常


要求用户输入待打开的文件名
用户输入MyFile.txt
文件存在
文件不存在
将文件名传递给FileReader的构造方法
逻辑错误导致把null传递给构造方法
抛出异常并处理
构造方法抛出NullPointException异常
捕获并消除导致异常的错误

类名Error、Exception和RuntimeException容易引起混淆，其实这三种类型都是异常，这里讨论的错误都是运行错误。

在大多数情况下，免检异常反应程序设计中不可重获的逻辑错误，它们是程序中必须纠正的逻辑错误。免检异常可能在程序任何地方出现，而且在典型的程序中，这种异常的数量可能非常非常大，必须在每个方法声明中都加上运行时异常会降低程序的清晰度。为了避免过多的使用try-catch语句块，Java语言不要求方法捕获或指定免检异常。

基本原则：如果可以期望客户合理的从异常中恢复，就使它为可控异常，若客户无法从异常中恢复，就使它为不可控异常（免检异常）。

3. 处理异常

- 异常采用了一种**面向对象**的处理机制：
  - 每当发生此类事件时，Java即自动创建一个异常对象(Exception object)，它包含关于异常的信息、类型和异常发生时程序的状态。
    - 系统：如果程序员不指定产生某种类型的异常之后如何进行处理，则系统会在程序运行过程中产生异常的时候自动抛出异常，执行系统默认的程序，显示异常信息，而后程序结束；
    - 程序员：当需要在异常产生的时候进行相应的动作或有特定的要求，则可以由程序员编写相应的代码对异常进行处理。
- Java的异常处理模型基于三种操作

- 在方法声明中说明可能会抛出的异常
  - 在Java中，当前执行的语句属于某个方法。因此每个方法都必须说明它可能抛出的可控异常类型，以便通知方法的调用者(Error和RuntimeException除外)。
  - 通常情况下，异常是由系统自动捕获的。在有些情况下，一个方法并不需要处理它所生成的异常，而是向上传递，由调用该方法的其他方法来捕获该异常。
  - 如果该方法不捕获其中发生的可控异常，则必须表明它可以抛出这些异常。

- throws子句
  - throws子句是为了指定某方法可以抛出的异常，需要在该方法声明中添加的子句。
  - 格式：

    [修饰符] 返回值类型 方法名([参数列表]) throws 异常类型清单 {方法体}

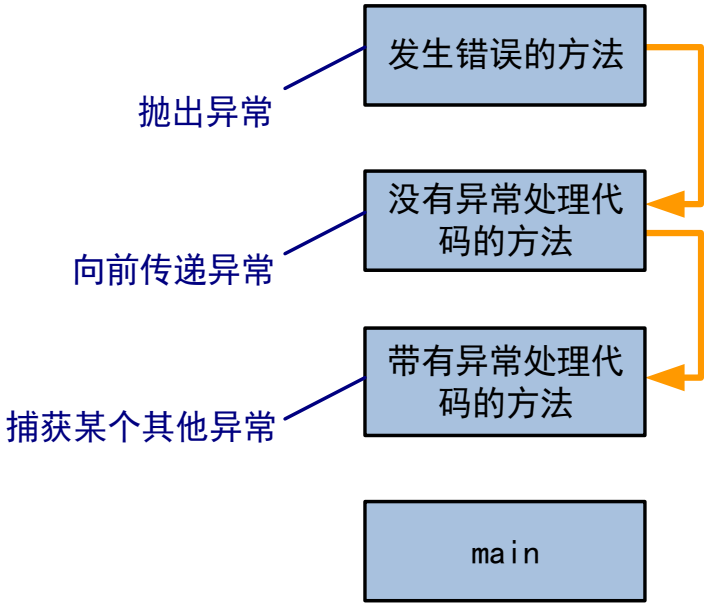
    - throws关键字和后面的异常清单表示该方法可能会抛出的异常；
    - 异常清单以逗号分隔；

    如果在父类中方法没有声明异常，那么在子类中不能对其进行覆盖以声明异常。

- 抛出异常
  - 创建异常对象并将它交给运行时系统被称为抛出异常(throw an exception)。
  - 在能捕获异常之前，必须有代码抛出异常。
  - 在方法抛出一个异常之后，运行时系统尝试寻找对此异常处理的某些机制。对异常进行处理的这一套“机制”是一系列有序的方法，是为了到达发生错误的方法而调用的一系列方法，这个方法列表被称为“调用栈”(call stack)。
  - 不管是什么代码抛出了异常，都是使用throw语句抛出的。

```
throw 异常对象;
```

- 捕获和处理异常
  - 选择合适的异常处理代码(exception handler)被称为捕获异常(catch the exception)。如果运行时系统彻底搜索了调用栈中的所有方法，但没有找到合适的异常处理代码，那么运行时系统和程序就会终止。



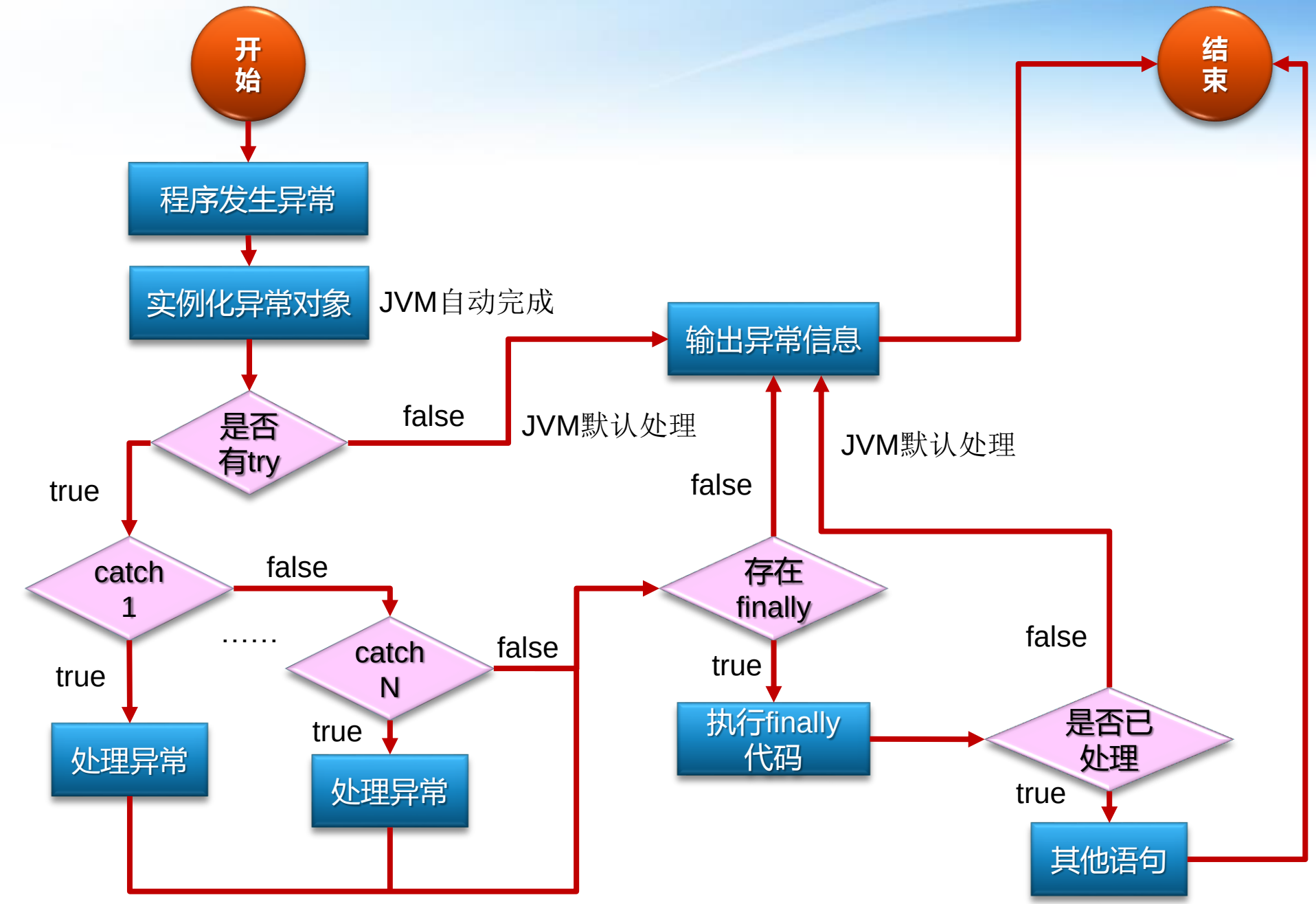
- 搜索从发生错误的方法开始
- 按照与方法被调用次序相反的次序遍历call stack
- 当找到合适的处理器时，运行时系统将异常传递给此处理器
- 如果被抛出的异常对象类型和某个异常处理代码可以处理的类型匹配，那么此处理器就被认为是合适的

- try-catch语句格式

异常处理的语法结构

```
try{
        可能出现异常的程序执行体
}
catch(异常类型1 异常对象1)
        {异常类型1对应的异常处理程序体1}
catch(异常类型2 异常对象2)
        {异常类型2对应的异常处理程序体2}
……
[finally
        {异常处理结束前的执行程序体}]
```

- 处理步骤：
  - 程序运行过程中，try后面的各catch块不起作用。
  - 如果try块内出现了异常，系统将终止try块代码的运行，自动跳转到对应的catch块中，执行该块的代码。
  - 异常处理结束之后，程序从try块语句代码之后继续执行。


开始
程序发生异常
实例化异常对象
JVM自动完成
是否有try
false
JVM默认处理
true
catch 1
false
……
catch N
false
true
处理异常
true
处理异常
输出异常信息
结束
false
JVM默认处理
存在finally
false
true
执行finally代码
是否已处理
false
true
其他语句

- 例7.3

指出当前存在的异常

```
public class ThrowsExceptionEx{
        public static int Sum() throws NegativeArraySizeException{
                int s=0;
                int x[]=new int[-8];
                for(int i=0;i<4;i++){
                        x[i]=2*i;
                        s=s+x[i];
                        }
                return s;
                }
        public static void main(String args[]){
                try{
                        System.out.println(Sum());
                }catch(NegativeArraySizeException e){
                        System.out.println("异常信息:"+e.toString()+"数组负下标异常");
                }
        }
}
```

异常信息

```
异常信息:java.lang.NegativeArraySizeException数组负下标异常
```

- 例程

抛出异常语句

```
public class ThrowsEx{
        static void throwmethod() throws IllegalAccessException{
                System.out.println("throw an exception in throwmethod");
                throw new IllegalAccessException();
        }
        public static void main(String args[]){
                try{
                        throwmethod();
                }catch(IllegalAccessException e){
                        System.out.println("catch an exception:"+e+"非法访问错误异常");
                }
        }
}
```

异常信息

```
throw an exception in throwmethod
catch an exception:java.lang.IllegalAccessException非法访问错误异常
```

- try语句块
  - try语句用于指明可能产生异常的程序代码段，其中所写的为被监视的代码段，一旦发生异常，则由catch代码进行处理；
  - try语句中包含一行或多行能抛出异常的语句。
- catch语句块
  - catch为等待处理的异常事件及其处理代码，在try语句之后。一个try语句可以有若干个catch语句与之相匹配，用于捕捉异常。
  - 每一个要捕捉的异常类型对应一个catch语句，该语句包含着异常处理的代码；
  - catch语句的作用域仅仅局限于其前的try语句制定的代码段，若在try语句之前已经产生了异常，则后面的所有代码包括try语句和catch语句本身将不被执行，而是采用默认的异常处理机制进行处理。因此一定把可能产生异常的语句包含在try语句内部。

异常在try语句之前发生

```
a=20;
b=0;
c=a/b;
try{a=2/b;}
catch(ArithmeticException e){
        System.out.println("除数为0");}
System.out.println("a="+a);
```

- 用catch语句捕捉异常时，若找不到相匹配的catch语句，则系统将执行默认的异常处理，这与不处理异常相同。

没有匹配的catch语句

```
a=20;b=0;
try{int c=a/b;}
catch(ArrayIndexOutOfBoundsException e){System.out.println("数组下标越界");}
```

- 当有多个catch语句时，系统将依照先后顺序逐个对其进行检查，执行第一个匹配的catch语句，其余的语句将不再执行。因此需要注意类型之间的层次关系。一般来说处理子类异常的catch语句必须位于父类异常的catch语句之前。

在catch块中指定异常的顺序是非常重要的。如果父类的catch块出现在子类的catch块之前，就会导致编译错误。

错误

```
try{
        …
}catch(Exception ex){
        …
}catch(RuntimeException ex){
        …
}
```

正确

```
try{
        …
}catch(RuntimeException ex){
        …
}catch(Exception ex){
        …
}
```

- 除了输出错误消息或者终止程序之外，异常处理代码能够进行更多操作，它们可以进行错误恢复、提示用户作出决定，或者使用链式异常把错误传递给更高级别的处理器。

- finally语句块
  - finally为最终处理的代码段，是个可选项。如果包含有finally块，无论异常是否发生，或者即使出现未预料到的异常，也都必须执行finally的代码块。
  - 除了处理异常之外，finally块还可以避免因清理代码而偶然被return、continue或者break绕过。

异常处理的语法结构

```
try{
        打开资源;
        可能引发异常类型1的语句;
        可能引发异常类型2的语句;}
catch(异常类型1 异常对象1){异常类型1对应的异常处理程序体1}
catch(异常类型2 异常对象2){异常类型2对应的异常处理程序体2}
finally{
        异常处理结束前的执行程序体
        关闭资源;}
```

- 在上述的例子中，可能有三种方法退出try块：
  - 抛出异常1；
  - 抛出异常2；
  - 所有语句成功执行，try块正常退出。
- 不管以哪种方式退出，都应该关闭已经打开的资源。finally语句保证无论try块中的语句正常执行，还是发生两种类型的异常，系统总会执行finally块中的语句。因此，finally块是执行清理工作的理想位置。
- finally块是防止资源泄露的关键工具，当关闭文件或者通过其他方式回收资源时，在finally块中加入代码，以便确保资源总是被回收。

在finally语句块中定义了return或throw语句会导致try、catch语句块中的return或throw语句失效。

- Java 7增强了try语句的功能，允许在try关键字后紧跟一对圆括号，圆括号内可以声明、初始化一个或多个资源，try语句在该语句结束时自动关闭这些资源。

```
try(
        ……  //初始化资源语句
){
        ……//使用资源的语句，可能会出现异常；
}
```

包含了隐式的关闭资源的finally块

从Java 7开始，一个catch块可以捕获多种类型的异常，多种异常类型之间用竖线“|”隔开，捕获多种类型异常时，异常变量有隐式的final赋值，因此不能对异常变量重新赋值。

```
catch(IndexOutOfBoundsException|NumberFormatException ie){
        ……  //异常处理语句
}
```

- catch和throw同时使用
  - 在实际应用中，单靠某个方法可能无法完全处理该异常，必须由几个方法协作才可以完全处理，也就是说，在异常出现的方法中，程序只对异常做部分处理，还有些处理需要在该方法的调用者中才能完成，所以应该再次抛出异常。

```
public f() throws Exception{
        try{
                ……
        }catch(Exception e){
                部分异常处理语句；
                throw e;
        }
}
```

4. 自定义异常

- 创建异常类
  - 选择要抛出异常的类型时，当发生以下情况时可以编写自己的异常类：
    - 需要的异常类型无法用Java平台中的异常类表示；
    - 让用户区分你要编写的异常类有别于其他开发者编写的异常类是有好处的；
  - 格式：
    - 类的声明

```
[修饰符] class 自定义异常类名 extends Exception{异常类体;}
```

    - 对象的创建

```
异常类型 异常对象名 = new 异常构造函数([参数列表]);
```

- 实例
  - 异常类的定义

异常类的定义

```
class UseException extends Exception{
        int n=0;
        UseException(){n++;}
        UseException(String s){
                super(s);
                n++;
        }
        String show(){return “selfDefinedExceptionObj:”+n;}
}
```

  - 异常类的使用

使用自定义的异常

```
public class testException{
        static void Test() throws UseException{
                UseException e = new UseException(“selfDefinedExceptionObj”);
                throw e;
        }
        public static void main(String args[]){
                try{Test();}
                catch(UseException e){System.out.println(e.show());}
        }
}
```

- 辅助调试方法
  - 在程序中添加输出变量的信息
    - 这是一种常用的程序调试方法，通过向代码中添加大量的输出语句，观察输出项的值，判断程序的出错范围。
  - 在非静态方法中，通过this输出当前对象的状态。
    - 注意在静态方法中不能使用this。
  - 栈踪迹(stack trace)方法
    - 栈踪迹方法提供当前线程的执行历史信息，并且列出当出现异常时被调用的类和方法的名称。当异常被抛出时，栈踪迹是有用的调试工具。
      - **采用printStackTrace()方法输出异常对象调用栈的信息；**
      - **采用getMessage()方法获取异常信息；**
      - **采用getClass()和getName()方法获取异常类名。**

- 实例

栈踪迹实例

```
class useException extends Exception{
        public useException(){
                super("自定义异常");
        }
}
public class getMessage{
        public static void m1() throws useException{
                m2();
        }
        public static void m2() throws useException{
                throw new useException();
        }
        public static void main(String args[]){
                try{
                        m1();
                }catch(useException e){
                        System.out.println(e.getMessage());
                        e.printStackTrace();
                        System.out.println("异常类型:"+e.getClass().getName());
                }
        }
}
```

异常信息

```
自定义异常
useException: 自定义异常
    at getMessage.m2(getMessage.java:11)
    at getMessage.m1(getMessage.java:8)
    at getMessage.main(getMessage.java:15)
异常类型:useException
```

5. 异常的优点

- 优点一：把错误处理代码和“常规”代码分离开
  - 异常提供了把非正常情况下的处理代码与程序的主逻辑分离的途径。
  - 在传统的程序设计中，错误的检测、报告和处理经常导致代码混乱，如以下的伪代码：

将文件读入内存的伪代码

```
readFile{
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
}
```

如果无法打开文件，会发生什么情况？

如果无法判断文件的长度，会发生什么情况？

如果无法分配足够的内存，会发生什么情况？

如果读操作失败，会发生什么情况？

如果无法关闭文件，会发生什么情况？

- 为了处理以上的情况，必须手动的添加更多的代码进行错误的检测、报告和处理。可能会演变成下面的伪代码：

将文件读入内存的伪代码

```
errorCodeType readFile {
        initialize errorCode = 0;
        open the file;
        if (theFileIsOpen) {
                determine the length of the file;
                if (gotTheFileLength) {
                        allocate that much memory;
                        if (gotEnoughMemory) {
                                read the file into memory;
                                if (readFailed) {errorCode = -1;}
                        }
                        else {errorCode = -2;}
                }
                else {errorCode = -3;}
                close the file;
                if (theFileDidntClose && errorCode == 0) {errorCode = -4;}
                else{errorCode = errorCode and -4;}
        }
        else {errorCode = -5;}
        return errorCode;
}
```

- 可以看出，这么多的错误检测、报告和返回的错误代码，程序变得非常复杂，代码的逻辑流程也变得非常不清楚，层层嵌套的if-else语句也难以判断代码逻辑是否正确。

- 采用异常机制的错误处理技术应该是下面这样的代码：

将文件读入内存的伪代码

```
readFile{
        try{
                open the file;
                determine its size;
                allocate that much memory;
                read the file into memory;
                close the file;
        }catch(fileOpenFailed){doSomething;}
         catch(sizeDeteminationFailed){doSomething;}
         catch(memoryAllocationFailed){doSomething;}
         catch(readFailed){doSomething;}
         catch(fileCloseFailed){doSomething;}
}
```

- 优点二：把错误沿调用栈向上传递
  - 假设上面的例子是主程序一系列嵌套的方法调用中的第四个方法：

嵌套方法调用

```
method1{
        call method2;
}
method2{
        call method3;
}
method3{
        call readFile;
}
```

  - 假设只有method1对readFile中可能发生的错误感兴趣。
  - 传统的错误通知技术迫使method2和method3将readFile返回的错误编码沿调用栈向上传递，直到最终到达method1。
  - 为了让method1获得返回的错误代码，伪代码如下：

嵌套方法调用

```
method1{
        errorCodeType error;
        error=call method2;
        if(error) doErrorProcessing;
        else proceed;}
errorCodeType method2{
        errorCodeType error;
        error=call method3;
        if(error) return error;
        else proceed;}
errorCodeType method3{
        errorCodeType error;
        error=call readFile;
        if(error) return error;
        else proceed;}
```

- Java运行时环境会沿调用栈往回搜索，寻找可以处理特定异常的方法。一个方法可以不理会其中抛出的任何异常，因为异常会沿调用栈向上传递直至被捕获。伪代码如下：

嵌套方法调用

```
method1{
        try{
        call method2;
        }catch (exception e){
                doErrorProcessing;
        }
}
method2 throws exception{
        call method3;
}
method3 throws exception{
        call readFile;
}
```

- 只有关心错误情况的方法才必须为检测异常操心。
- 如上面的伪代码所示，回避异常要求在中间的方法中做一些工作。一个方法中可抛出的任何可控异常都必须在这个方法的throws子句中指定。

- 优点三：对错误类型进行分组和区分
  - 程序内抛出的所有异常都是对象，因此类层次结构的一个自然结果就是对异常进行分组和区分。
    - 方法可以编写特殊化的处理器，以便处理非常特殊的异常；
    - 方法可以在catch语句中指定任何异常的超类来设置异常组或一般类型进而根据异常组或一般类型捕获异常；
  - 通过以上两种情况，可以以一般化的方式处理异常；或使用特定的异常类型来区分异常，并以更有针对性的方式处理异常。

6. 正确的使用异常

- 何时使用异常
  - 由于异常处理需要初始化新的异常对象，并重新返回调用堆栈，并且通过方法调用链传播异常，以便搜寻异常处理器，所以，通常情况下异常处理需要更多的时间和资源。
  - 在代码中，当必须处理不可预料的错误时应该使用try-catch块处理异常，而不要用其处理简单的、可预测的情况。如：

使用异常

```
try{
        System.out.println(refVar.toString());
}catch(NullPointerException ex){
        System.out.println("refVar is null!");
}
```

不使用异常

```
if(refVar!=null)
        System.out.println(refVar.toString());
else
        System.out.println("refVar is null!");
```

- 使用异常的几点建议：
  - 在可以使用简单的测试就能完成的检查中，不要使用异常来代替简单的逻辑判断；
    - 例如：
      ```
      if(ins!=null){使用ins引用对象的语句}
      ```
  - 不要过细的使用异常。
  - 不要捕获了一个异常而又不对它做任何处理；
    - 例如：

捕获而不处理

```
try{
        可能产生异常的代码块
}catch(Exception e){
}
```

    - 将异常保留给方法的调用者并非不好的做法，有些异常可以交给方法的调用者去处理，这是一种更好的处理办法。