**Java异常**

## 一、概念

异常看起来是程序中的错误的意思，有时候会导致歧义，如果程序中出现错误，就是你写的程序有问题，为什么不去修复它，把它变成一个没有异常的程序，这是值得去推敲的。

首先程序不是万能的，它有一定的逻辑和规定，而且程序为了灵活性，往往是动态接收数据。比如一个算数算法，它往往只能接收整型数据，但是非要传入一些其它类型的数据，比如字符串等，为了健壮性考虑，程序往往会返回一些信息，告知调用者是不接收这类型数据的。而早期往往是返回一些标志位，因为函数的返回类型只能有一个，所以很可能通过”-1”等标识表示“类型错误”等含义。但是这里面可能存在的一些问题就是：一方面对于返回标识的判断和业务逻辑判断放置在一起很容易混淆，不便于程序阅读性，而且返回标识位的方式，也不利于调用者的记忆和使用；另外，如果调用者在自身的程序不对这些“错误”做处理，往往出现问题，调用者会认为他的程序本身没有问题，是提供的接口存在错误，所以可以出现强制规定，避免这种随意性，最后，如果程序中出现多处调用都可能存在错误，那么你需要分别在这些地方加上判断，这样代码看起来非常的臃肿，如果能捕获一段代码中所有错误，并且可以转到特定区域去处理，那么业务和错误就进行了分离，这样即简单代码结果又使结构清晰了，而这些目的就是“异常”的特性。所以，“异常”可以看作是一些语言提出来对于编程中“错误”处理的一种体系结构，使它更加清晰和便利被开发者去使用。

## 二、Java异常基本常识

首先了解异常的一个特点：就是异常发生时，表示程序出现问题，当前程序执行步骤会终止，一般需要返回到上一级环境，就比如以前通过【return】返回。

Java中关于异常的知识点：

**1. 异常也是对象，需要在内存堆中创建实例。**

**2. Java所有异常的基类是【Exception】，JDK提供很多预定义的异常类，它们都是Exception的子类。**

**3. 异常发送时，当前程序执行终止，Java通过throw关键字抛出异常对象**,比如：

if (obj = null){

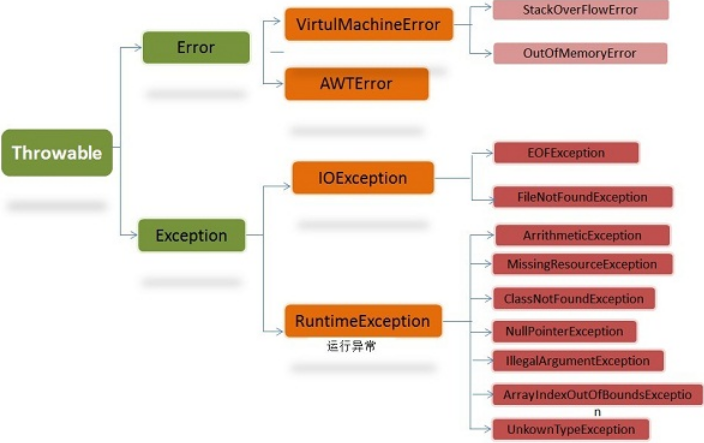
throw new NullPointerException();

//obj.method();

}

代码的含义：对象引用变量obj如果还未初始化，那么它默认值为null，就不能调用方法，这时候需要抛出这个异常，异常抛出后到上一个环境，这是由jvm去处理，至于具体哪里后面具体介绍规则。

**4. 异常是对象，它有两种形式的构造函数，一个是无参数的默认构造函数**；还有一种就是接收单个字符串的。异常的信息基本都是通过异常的名称去传递的，比如NullPointerException类名的含义就是所谓的“空指针异常”。



**5 Throwable：** 有两个重要的子类：Exception（异常）和 Error（错误），二者都是 Java 异常处理的重要子类，各自都包含大量子类。

**6 Error（错误）:**是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

**7 Exception（异常）**:是程序本身可以处理的异常。

Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

**8 通常，Java的异常(包括Exception和Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。**

**9 可查异常（编译器要求必须处置的异常）：**正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

**10 不可查异常(编译器不要求强制处置的异常):**包括运行时异常（RuntimeException与其子类）和错误（Error）。Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

**11 运行时异常：**都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

**12 非运行时异常 （编译异常）：**是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

**13 runtimeException子类:**

（1）java.lang.ArrayIndexOutOfBoundsException

数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。

（2）java.lang.ArithmeticException

算术条件异常。譬如：整数除零等。

（3）java.lang.NullPointerException

空指针异常。当应用试图在要求使用对象的地方使用了null时，抛出该异常。譬如：调用null对象的实例方法、访问null对象的属性、计算null对象的长度、使用throw语句抛出null等等

（4）java.lang.ClassNotFoundException

找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPAH之后找不到对应名称的class文件时，抛出该异常。

（5）java.lang.NegativeArraySizeException 数组长度为负异常

（6）java.lang.ArrayStoreException 数组中包含不兼容的值抛出的异常

（7）java.lang.SecurityException 安全性异常

（8）java.lang.IllegalArgumentException 非法参数异常

**14.IOException**

IOException：操作输入流和输出流时可能出现的异常。

EOFException 文件已结束异常

FileNotFoundException 文件未找到异常

**15 其他**

ClassCastException 类型转换异常类

ArrayStoreException 数组中包含不兼容的值抛出的异常

SQLException 操作数据库异常类

NoSuchFieldException 字段未找到异常

NoSuchMethodException 方法未找到抛出的异常

NumberFormatException 字符串转换为数字抛出的异常

StringIndexOutOfBoundsException 字符串索引超出范围抛出的异常

IllegalAccessException 不允许访问某类异常

InstantiationException 当应用程序试图使用Class类中的newInstance()方法创建一个类的实例，而指定的类对象无法被实例化时，抛出该异常

## 三、捕获和处理异常

**1 异常以throws的方式抛出时**，当前执行步骤将不会继续向下进行，而是返回到上一环境，如果你想程序继续运行，java提供一种捕获异常的方式，语法形式：

try{

// Code that might generate exception

}

**2 首先它是一种尝试捕获，因为你也不确定是否一定会出现异常，另外它是用{}圈起来的，表示它是监督一段区域，而不是一行代码，所以用try这个词。**

捕获的异常必须在某处得到处理。这个“地点”就是异常处理程序，可以针对每一个捕获异常，准备相应的处理。异常处理程序是语法结构：

try{

// Code that might generate exceptions

}catch(Type1 id1){

// Handle exceptions of Type1

}catch(Type2 id2){

//Handle exceptions of Type2

} catch(Type3 id3){

//Handle exceptions of Type3

}

**3 在try的监控区域中发现异常，那么就依次操作catch语句，如果异常类型和catch的中Type1相同，或者是Type1的类型包含出现的异常（是Type1子类），则去执行内部的语句。**

注意：如果catch语句中没有终止程序的语句，将跳出try-catch语句，进行向下执行。

public class ExceptionMethods {

public static void main(String[] agrs) {

try{

throw new RuntimeException("异常提示信息");

}catch(RuntimeException e) {

e.printStackTrace();

}catch(Exception e) {

System.out.println("不会执行的异常捕获");

}

System.out.println("异常被吞咽");

}

}

**4 虽然在异常发生时，我们可以抛出或者捕获异常，但是往往捕获异常后，我们并不能做出有意义工作。**因为如果你自身编写的程序抛出异常后，你再试试去捕获它，做一些修复是不太可能的；如果是别人程序的抛出异常，如果你要捕获修复它，那么就要详细了解对方的代码，增加了耦合（也就是你高度依赖别人的程序），假设别人的代码做出修改，你可能也要跟进相关变动。所以Java支持终止模式，异常抛出，就表示错误无法挽回，不需要继续执行。当然这不代表捕获异常就是没有必要事情，因为你捕获异常后，也可以进行抛出异常，在抛出之前你可以打印一些信息等。当然try-catch使用比较多的，主要在一些需要关闭资源的程序，比如IO编程，后面举例说明。

## 四、创建定义异常

除了上面的预定义异常，也就是Jdk已经提供的，可以根据自己需要创建自定义异常，创建它，必须要从已有的异常类继承。

因为异常对象的构造函数有两个，所以分两种形式：

class MyException extends Exception

{

publicMyException(){}

publicMyException(String msg){super(msg);}

}

## 五、异常说明

Java鼓励你们把方法可能抛出的异常告知使用此方法的客户端程序员，这样可以让调用者确切的指定写什么代码捕获所有潜在的异常。当然，如果你提供了源代码，客户端程序远可以在源代码中查找throw语句来获知相关信息，但是并不是往往别人是不会提供源代码给你的。为了预防这种问题，Java提供了相应的语法，是你可以以其它方式告知客户端程序员某个方法可能会抛出的异常类型，这就是异常说明，它属于方法声明的一部分，紧跟在形式参数列表之后。异常说明使用了关键字throws，注意需要跟上面throw区别，它多一个s，区别的方式就是异常声明throws可以跟多个异常类型，而throw是抛出单个异常对象。

没有异常说明：void f()

增加异常说明：void f() throws TooBig, TooSmall,DivZero{}

throws和throws的区别：

voidf()throws TooBig, TooSmall,DivZero{

throw new TooBig();

throw new TooSmall ();

throw new DivZero ();

}

异常说明的强制性：

**第一点：如果方法内部产生异常，那么你**必须对异常进行处理，比如捕获异常处理，如果不捕获异常或者捕获异常处理时继续抛出异常，那么你必须要有异常声明，否则编译报错。

通过的说方法中出现throws异常时，要么你用try-catch语句处理掉这个异常，要么通过异常声明throws出去告知调用者。

**第二点：如果外部调用这个方法，**这个方法包含异常声明，那么你的调用程序必须可以出现的异常做出处理，处理方式参照一条；如果不处理，则编译报错。

## 六、捕获所有的异常

可以只写一个异常处理程序来捕获所有类型的异常，通过捕获异常类型的基类Exception就可以做到这有点：

catch(Exception e) {

System.out.println(“Caught an exception”);

}

## 七、异常方法

异常的主要信息是透过异常的名称来体现的，所以异常类的方法比较少，但是还是有几个要了解的，Exception方法继承之Throwable类：

一、printStackTrace()把异常信息打印到标准错误流，可以在控制台查看。

public class ExceptionMethods {

public static void main(String[] agrs){

try{

throw newException("异常提示信息");

}catch(Exception e) {

e.printStackTrace();

}

}

}

我们经常遇到异常信息，但是对于具体信息的组成和由来可能没有更多了解。第一行可以通过toString()方法返回，他其实有两部分组成：类名+初始化异常对象的字符串信息，内部实现中关于类名可以通过getClass()或者.class属性获取，而“初始化异常对象的字符串信息”可以通过getMessage()或者getLocalizedMessage()返回，这两个方法默认的返回值相同。复写toString()或者getMessage()可以改变控制台第一行的打印信息。

public classExceptionMethods {

public static void main(String[] agrs){

try{

throw newException("异常提示信息");

}catch(Exception e) {

System.out.println("printStackTrace():");

e.printStackTrace();

System.out.println("toString():"+e);

System.out.println("getMessage():"+e.getMessage());

System.out.println("getLocalizedMessage():"+e.getLocalizedMessage());

}

}

}

class ExceptionByMySelfextends Exception{

private String str;

ExceptionByMySelf(){

}

ExceptionByMySelf(String msg){

super(msg);

this.str = msg;

}

public String toString(){

if("".equals(getMessage()))

return "mystyle: " +getClass().getName();

return "my style: "+getClass().getName()+ ": " +getMessage();

}

public String getMessage(){

if (str==null)

return "";

return str;

}

}

classExceptionDemo{

public static void main(String[] args){

try{

throw newExceptionByMySelf();

}catch(ExceptionByMySelfe){

e.printStackTrace();

}

try{

throw newExceptionByMySelf("自定义的异常");

}catch(ExceptionByMySelf e){

e.printStackTrace();

}

}

}

接着看看从第二行开始就是异常在每一个调用方法中轨迹，主要是方法名和行数；最上面是最后调用函数，最下面是开始的调用的方法。它其实是可以通过getStackTrace()返回，它是一个数值，类型为StackTraceElement，代码示例：

public classWhoCalled {

static void f(){

try{

throw newException();

}catch(Exception e){

for(StackTraceElementste : e.getStackTrace()){

System.out.println(ste.getMethodName());

}

}

}

static void g(){f();}

static void h(){g();}

public static void main(String[] args){

f();

System.out.println("----------------");

g();

System.out.println("----------------");

h();

}

}

以h()方法举例，首先虚拟机找到main方法，main方法入栈 ,然后调用h()，h方法压栈，接着调用g()，g方法压栈，接着f方法压栈。

最上面是元素的0角标。

## 八、特例：RuntimeException

关于RuntimeException的我还不能很好的理解，但是关于它的由来大致可以这样理解：

比如t.f()，为了保证调用成功我们必须做健壮性判断

if( t == null) {

Throw new NullPointerException

}

再比如Person p = (Person)obj，同样为了健壮性必须判断

If( obj instanceof Person ){

Person p = (Person)Obj;

}

但是这里面存在一个问题就是这样的操作(方法调用和类型转换)在我们开发过程会经常出现，为了程序的健壮性考虑，我们可能会把代码变的非常臃肿。所以java为了解决这类问题，就是提倡可以不做这些重复判断，系统本身会去做这些检查，如果发现异常系统本身会把这些异常抛出来，而它们都是RuntimeException的子类；

RuntimeException有一个特性：就是它属于程序运行时，系统自身去检查的异常，然后抛出，那么是不需要编程者对它进行处理的，所以你可以不需要去捕获或者进行异常声明，当然它本身是允许你去捕获，但是不建议。

public class NeverCaught{

staticvoid f(){

thrownew RuntimeException("From f()");

}

staticvoid g(){

f();

}

publicstatic void main(String[] args){

g();

}

}

结果：即使你不对抛出的异常进行捕获或者声明，编译依然通过。

**总结：个人觉得RuntimeException在概念的边界上是很难区分的，而且往往会陷入是创建一个Exception和RuntimeException子类对象的纠结中，我的理解就是如果你为了明确告知别人方法可能抛出的具体类型，用Exception子类对象，这样好处通过异常声明明确了异常的类型，但是这样也是有弊端的，就是别人必须要异常进行处理，一般用作接口程序。而RuntimeException的好处就在，假如这个程序完全由我编写，出现异常的完全了解，直接抛出RuntimeException，最终抛回给main函数，通过虚拟机在控制台中打印出信息就可以，就避免了后续方法调用时强制对异常进行捕获和声明代码。其实两者完全都是可以代替的，但是RuntimeException更加简单和偷懒一点。**

## 九、使用finally进行清理

对于一些代码，可能希望无论try块中的异常是否抛出，他们都能得到执行，通常使用与内存回收之外的情形，为了达到这个效果，需要在异常处理程序后面加上finally。

try{

}catch(A a){

}catch(B b){

}catch(C c){

}finally{

}

示例1：

class ThreeException extends Exception {}

public class FinallyWorks {

staticint count = 0;

publicstatic void main(String[] args) {

while(true){

try{

if(count++==0){

thrownew ThreeException();

}

System.out.println("Noexception");

}catch(ThreeExceptione){

System.out.println("ThreeException");

}finally{

System.out.println("Infinally cluse");

if(count== 2) break;

}

}

}

}

示例2：

没有Finally子句：

class Switch {

privateboolean state = false;

publicboolean read(){return state;}

publicvoid on(){state=true;System.out.println(this);}

publicvoid off(){state=false;System.out.println(this);}

publicString toString(){ return state? "on" : "false";}

}

class OnOffException1 extends Exception{}

class OnOffException2 extends Exception{}

public class OnOffSwitch{

privatestatic Switch sw = new Switch();

publicstatic void f() throws OnOffException1,OnOffException2 {}

publicstatic void main(String[] args) {

try{

sw.on();

f();

sw.off();

}catch(OnOffException1e){

System.out.println("OnOffException1");

sw.off();

}catch(OnOffException2e){

System.out.println("OnOffException2");

sw.off();

}

}

}

有Finally子句：

public class WithFinally {

staticSwitch sw = new Switch();

publicstatic void main(String[] args){

try{

sw.on();

OnOffSwitch.f();

}catch(OnOffException1e){

System.out.println("OnOffException1");

}catch(OnOffException2e){

System.out.println("OnOffException2");

}finally{

sw.off();

}

}

}

当涉及到break、continue、return语句，finally都会得到执行。

实例3：

public class MultipleReturns{

publicstatic void f(int i){

System.out.println("Initializationthat requires cleanup");

try{

System.out.println("Point1");

if(i== 1) return;

System.out.println("Point2");

if(i== 2) return;

System.out.println("Point3");

if(i== 3) return;

System.out.println("End");

return;

}finally{

System.out.println("Performingcleanup");

}

}

publicstatic void main(String[] args){

for(inti = 1; i <= 4; i++){

f(i);

}

}

}

示例4：异常被吞咽和丢失

吞咽：

public class ExceptionMethods{

public static void main(String[] agrs){

try{

throw new Exception("异常提示信息");

}catch(Exception e) {

e.printStackTrace();

}

System.out.println("异常被吞咽");

}

}

因为捕获异常后，我们只是打印了异常信息，这样导致的问题就是：明明我们try中语句出现问题了，但是程序即没有终止，也没修复，而是进行向下执行。但是向下执行已经没有意义。

丢失：

class VeryImportantException extendsException {

publicString toStrin(){

return"A very important exception";

}

}

class HoHumException extends Exception{

publicString toString(){

return"A trivial exception";

}

}

public class LostMessage {

voidf() throws VeryImportantException {

thrownew VeryImportantException();

}

voiddispose() throws HoHumException{

thrownew HoHumException();

}

publicstatic void main(String[] args){

try{

LostMessagelm = new LostMessage();

try{

lm.f();

}finally{

lm.dispose();

}

}catch(Exceptione){

System.out.println(e);

}

}

}

lm.f()异常没有被处理，这些都是不正确的异常处理方式。

## 十、异常的限制

当我们继承类，会出现方法覆盖，超类（子类）只能抛出基类（父类）列出的异常，这种限制的目的是为了解决超类在向上转型为基类时，本来为基类设计的代码，超类也能使用。

比如我们常用多态的示例

void f(Person p){

try{

p.getAge();

}catch(TooSmall e) {//它只抛出一个年龄太小的错误

}

}

class Teacher extends Perosn {

publicint getAge() throw TooSmall,TooBig {//方法复写抛出比基类多的异常

}

}

f(new Teacher());//把子类对象传给f方法，子类向上转型，这是典型的多态，但是出现的问题，子类方法的异常声明有两个，而原来的代码只捕获了一个，这会造成错误。

1.子类方法的异常声明不能多于父类，可以少或者无，另外子类的异常类型可以与父类相同，或者是其子类，因为子类异常也可以向上转型与父类相同，但不能向下转型。

2. 子类的构造函数必须包含父类构造函数的异常，当然子类构造函数可以有自己的异常，因为子类构造函数必须调用父类的构造函数。

3. 子类对象向上转型为父类的引用，通过父类引用调用发放，抛出的异常以父类的为准，而不是子类。

示例：

class BaseballException extends Exception{}

class Foul extends BaseballException{}

class Strike extends BaseballException{}

abstract class Inning {

publicInning() throws BaseballException{}

publicvoid event() throws BaseballException{}

publicabstract void atBat() throws Strike,Foul;

publicvoid walk(){}

}

class StormException extends Exception {}

class RainedOut extends StormException {}

class PopFoul extends Foul {}

interface Storm {

publicvoid event() throws RainedOut;

publicvoid rainHard () throws RainedOut;

}

public class StormyInning extends Inningimplements Storm {

publicStormyInning() throws RainedOut,BaseballException{}

publicStormyInning(String s) throws Foul,BaseballException{}

//publicvoid walk() throws PopFoul{}

//publicvoid event() throws RainedOut{}

publicvoid rainHard() throws RainedOut{}

publicvoid event(){}

publicvoid atBat() throws PopFoul{}

publicstatic void main(String[] agrs) {

try{

StormyInningsi = new StormyInning();

si.atBat();

}catch(PopFoule){

System.out.println("Popfoul");

}catch(RainedOute){

System.out.println("RainedOut");

}catch(BaseballExceptione){

System.out.println("Genericbaseball exception");

}

try{

Inningi = new StormyInning();

i.atBat();

}catch(Strikee){

System.out.println("Strike");

}catch(Foule){

System.out.println("Foul");

}catch(RainedOute){

System.out.println("Rainedout");

}catch(BaseballExceptione){

System.out.println("Genericbaseball exception");

}

}

}

## 十一、构造器

在使用finally进行资源关闭时候，需要注意的就是，被清理的对方不一定在构造时就是成功的，比如我们熟悉的IO流操作。这时候我们往往需要使用try-catch嵌套，这里就不展看说明，因为需要好好的总结一下，由于时间问题，暂时放置两段代码：

import java.io.\*;

public class InputFile {

privateBufferedReader in;

publicInputFile(String fname) throws Exception {

try{

in= new BufferedReader(new FileReader(fname));

}catch(FileNotFoundExceptione){

System.out.println("Couldnot open " + fname);

throwe

}catch(Exceptione){

try{

in.close();

}catch(IOExceptione2){

System.out.println("in.close() unsuccessful");

}

throwe;

}finally{

}

}

publicString getLine(){

Strings;

try{

s= in.readLine();

}catch(IOException){

thrownew RuntimeException("readLine() failed");

}

returns;

}

publicvoid dispose(){

try{

in.close();

System.ou.println("dispose()successful");

}catch(IOExceptione2){

thrownew RuntimeException("in.close() failed");

}

}

}

import java.io.\*;

public class InputFile {

privateBufferedReader in;

publicInputFile(String fname) throws Exception {

try{

in= new BufferedReader(new FileReader(fname));

}catch(FileNotFoundExceptione){

System.out.println("Couldnot open " + fname);

throwe

}catch(Exceptione){

try{

in.close();

}catch(IOExceptione2){

System.out.println("in.close() unsuccessful");

}

throwe;

}finally{

}

}

publicString getLine(){

Strings;

try{

s= in.readLine();

}catch(IOException){

thrownew RuntimeException("readLine() failed");

}

returns;

}

publicvoid dispose(){

try{

in.close();

System.ou.println("dispose()successful");

}catch(IOExceptione2){

thrownew RuntimeException("in.close() failed");

}

}

}

## 十二、异常处理的可选方式

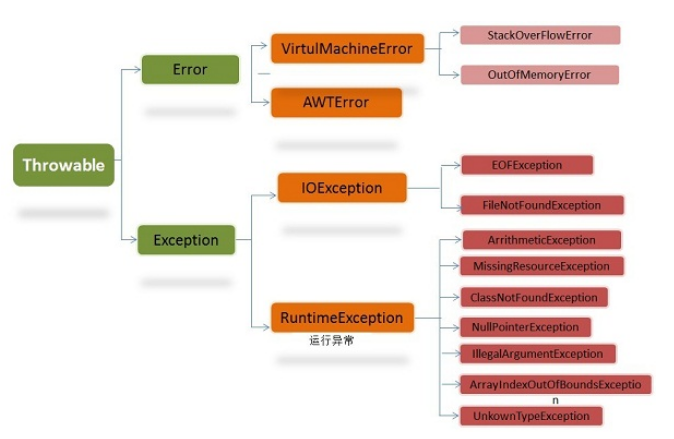
主要有两种方式：

1、 把异常传递给控制台，就是通过层层抛出的方式。

2、把“被检查的异常”转换为“不检查的异常”，就是把非RuntimeException变成RuntimeException。方式：new RuntimeException(e)，e非RuntimeException异常对象。

而且可以通过getCause()返回它原来的异常类型。

## 十三、异常机制概述



**1 异常机制**

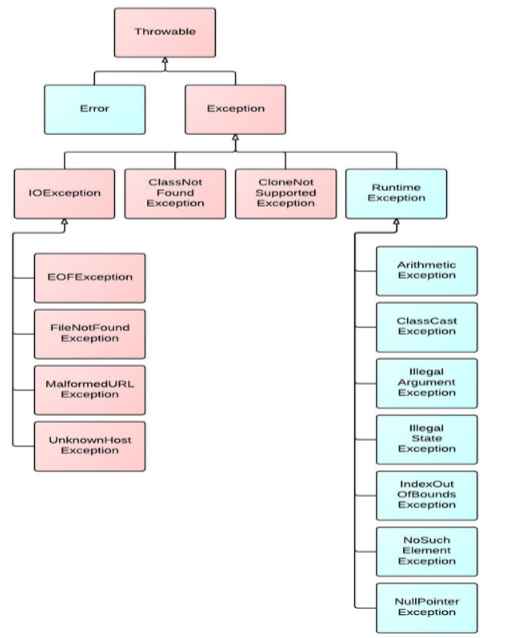
指当程序出现错误后，程序如何处理。具体来说，异常机制提供了程序退出的安全通道。当出现错误后，程序执行的流程发生改变，程序的控制权转移到异常处理器。

**2 异常的处理流程。**

当程序中抛出一个异常后，程序从程序中导致异常的代码处跳出，java虚拟机检测寻找和try关键字匹配的处理该异常的catch块，如果找到，将控制权交到catch块中的代码，然后继续往下执行程序，try块中发生异常的代码不会被重新执行。如果没有找到处理该异常的catch块，在所有的finally块代码被执行和当前线程的所属的ThreadGroup的uncaughtException方法被调用后，遇到异常的当前线程被中止。

**3 异常的体系结构：**

 异常的继承结构：Throwable为基类，Error和Exception继承Throwable，RuntimeException和IOException等继承Exception。Error和RuntimeException及其子类成为未检查异常（unchecked），其它异常成为已检查异常（checked）。



 所有异常类型都是内置类Throwable的子类。因此，Throwable在异常类层次结构的顶层。紧接着Throwable下面的是两个把异常分成两个不同分支的子类。一个分支是Exception（该类用于用户程序可能捕捉的异常情况。它也是你可以用来创建你自己用户异常类型子类的类。在Exception分支中有一个重要子类RuntimeException。 该类型的异常自动为你所编写的 程序定义并且包括被零 除和非法数组索引这样的错误）。另一类分支由Error作为顶层（    Error定义了在通常环境下不希望被程序捕获的异常。Error类型的异常用于Java运行时系统来显示与运行时系统本身有关的错误。堆栈溢出是这种错误的一例。）

## 十四、常见的异常

**1 Eroor异常**

Error表示程序在运行期间出现了十分严重、不可恢复的错误，在这种情况下应用程序只能中止运行，例如JAVA 虚拟机出现错误。Error是一种unchecked Exception，编译器不会检查Error是否被处理，在程序中不用捕获Error类型的异常。一般情况下，在程序中也不应该抛出Error类型的异常。

**2 Exception异常**

Exception异常包括RuntimeException异常和其他非RuntimeException的异常。RuntimeException 是一种Unchecked Exception，即表示编译器不会检查程序是否对RuntimeException作了处理，在程序中不必捕获RuntimException类型的异常，也不必在方法体声明抛出RuntimeException类。RuntimeException发生的时候，表示程序中出现了编程错误，所以应该找出错误修改程序，而不是去捕获RuntimeException。

**未经检查的异常（unchecked exceptions ）。因为编译器不检查它来看一个方法是否处理或抛出了这些异常未经检查的异常,如果你本人不用catch处理，那么系统就会自动处理**。



**3 Checked Exception异常**

 检查的异常：你必须自己处理，必须对该异常,行catch.自行处理才没有问题 不然系统就不会运行。

 Checked Exception异常，这也是在编程中使用最多的Exception，所有继承自Exception并且不是RuntimeException的异常都是checked Exception，上图中IOException      和ClassNotFoundException。

JAVA 语言规定必须对checked Exception作处理，编译器会对此作检查，要么在方法体中声明抛出checked Exception，要么使用catch语句捕获checked Exception进行处      理，不然不能通过编译

Exception类自己没有定义任何方法。当然，它继承了Throwable提供的一些方法。因此，所有异常，包括你创建的，都可以获得Throwable定义的方法。这些方法显示在表           10-3中。你还可以在你创建的异常类中覆盖一个或多个这样的方法。



**4 在声明方法时抛出异常：**

   语法：throws（略）

 （1）为什么要在声明方法抛出异常？

 方法是否抛出异常与方法返回值的类型一样重要。假设方法抛出异常却没有声明该方法将抛出异常，那么客户程序员可以调用这个方法而且不用编写处理异常的代码。那么，一旦出现异常，那么这个异常就没有合适的异常控制器来解决。

（2）为什么抛出的异常一定是已检查异常？

RuntimeException与Error可以在任何代码中产生，它们不需要由程序员显示的抛出，一旦出现错误，那么相应的异常会被自动抛出。遇到Error，程序员一般是无能为力的；遇到RuntimeException，那么一定是程序存在逻辑错误，要对程序进行修改；只有已检查异常才是程序员所关心的，程序应该且仅应该抛出或处理已检查异常。而已检查异常是由程序员抛出的，这分为两种情况：客户程序员调用会抛出异常的库函数；客户程序员自己使用throw语句抛出异常。覆盖父类某方法的子类方法不能抛出比父类方法更多的异常，所以，有时设计父类的方法时会声明抛出异常，但实际的实现方法的代码却并不抛出异常，这样做的目的就是为了方便子类方法覆盖父类方法时可以抛出异常。。

  如果一个方法可以导致一个异常但不处理它，它必须指定这种行为以使方法的调用者可以保护它们自己而不发生异常。做到这点你可以在方法声明中包含一个throws子句。一个 throws 子句列举了一个方法可能抛出的所有异常类型。这对于除Error或RuntimeException及它们子类以外类型的所有异常是必要的。一个方法可以抛出的所有其他类型的异常必须在throws子句中声明。如果不这样做，将会导致编译错误。

    下面是包含一个throws子句的方法声明的通用形式：

type method-name(parameter-list) throws exception-list{

    // body of method

}

这里，exception-list是该方法可以抛出的以有逗号分割的异常列表。

**5 在方法中抛出异常：**

   语法：throw（略）

（1）抛出什么异常？

 对于一个异常对象，真正有用的信息是异常的对象类型，而异常对象本身毫无意义。比如一个异常对象的类型是ClassCastException，那么这个类名就是唯一有用的信息。所以，在选择抛出什么异常时，最关键的就是选择异常的类名能够明确说明异常情况的类。

1. 异常对象通常有两种构造函数

一种是无参数的构造函数；

另一种是带一个字符串的构造函数，这个字符串将作为这个异常对象除了类型名以外的额外说明

1. 程序可以用throw语句抛出明确的异常。

Throw语句的通常形式如下：

     throw ThrowableInstance;

     这里，ThrowableInstance一定是Throwable类类型或Throwable子类类型的一个对象。简单类型，例如int或char，以及非Throwable类，例如String或Object，不能用作异常。有两种可以获得Throwable对象的方法：在catch子句中使用参数或者用new操作符创建。

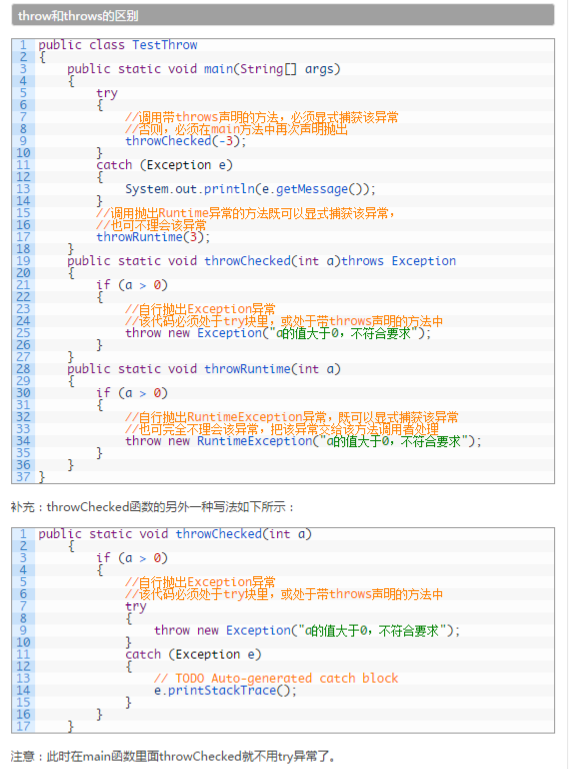
 （4）怎样创建Java的标准异常对象，特别注意下面这一行：

   throw new NullPointerException("demo");

这里，new用来构造一个NullPointerException实例。所有的Java内置的运行时异常有两个构造函数：一个没有参数，一个带有一个字符串参数。当用到第二种形式时，参数指定描述异常的字符串。如果对象用作 print( )或println( )的参数时，该字符串被显示。这同样可以通过调用getMessage( )来实现，getMessage( )是由Throwable定义的。

**6 为什么要创建自己的异常：**

 当Java内置的异常都不能明确的说明异常情况的时候，需要创建自己的异常。需要注意的是，唯一有用的就是类型名这个信息，所以不要在异常类的设计上花费精力



**处理原则：**捕捉并处理哪些知道如何处理的异常，而传递哪些不知道如何处理的异常使用finally块释放资源。finally关键字保证无论程序使用任何方式离开try块，finally中的语句都会被执行。在以下三种情况下会进入finally块：

（1） try块中的代码正常执行完毕。

（2） 在try块中抛出异常。

（3） 在try块中执行return、break、continue。

    因此，当你需要一个地方来执行在任何情况下都必须执行的代码时，就可以将这些代码放入finally块中。当你的程序中使用了外界资源，如数据库连接，文件等，必须将释放这些资源的代码写入finally块中。

    必须注意的是：在finally块中不能抛出异常。JAVA异常处理机制保证无论在任何情况下必须先执行finally块然后再离开try块，因此在try块中发生异常的时候，JAVA虚拟机先转到finally块执行finally块中的代码，finally块执行完毕后，再向外抛出异常。如果在finally块中抛出异常，try块捕捉的异常就不能抛出，外部捕捉到的异常就是finally块中的异常信息，而try块中发生的真正的异常堆栈信息则丢失了。

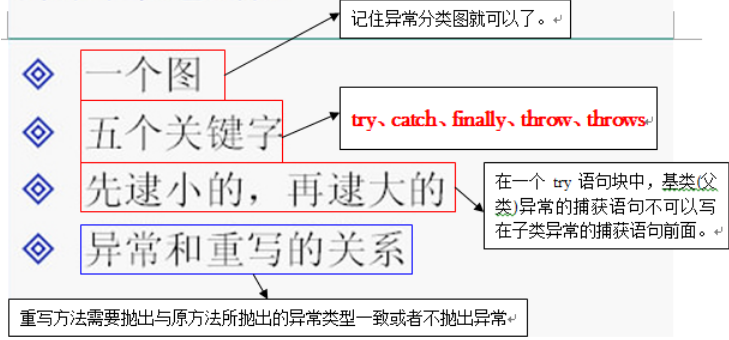
**7 finally创建一个代码块。**

该代码块在一个try/catch 块完成之后另一个try/catch出现之前执行。finally块无论有没有异常抛出都会执行。如果异常被抛出，finally甚至是在没有与该异常相匹配的catch子句情况下也将执行。一个方法将从一个try/catch块返回到调用程序的任何时候，经过一个未捕获的异常或者是一个明确的返回语句，finally子句在方法返回之前仍将执行。这在关闭文件句柄和释放任何在方法开始时被分配的其他资源是很有用的。finally子句是可选项，可以有也可以无。然而每一个try语句至少需要一个catch或finally子句。任何不是被你程序捕获的异常最终都会被该默认处理程序处理。默认处理程序显示一个描述异常的字符串，打印异常发生处的堆栈轨迹并且终止。

    尽管由Java运行时系统提供的默认异常处理程序对于调试是很有用的，但通常你希望自己处理异常。这样做有两个好处。第一，它允许你修正错误。第二，它防止程序自动终止。大多数用户对于在程序终止运行和在无论何时错误发生都会打印堆栈轨迹感到很烦恼（至少可以这么说）程序。

    当你用多catch语句时，记住异常子类必须在它们任何父类之前使用是很重要的。这是因为运用父类的catch语句将捕获该类型及其所有子类类型的异常。这样，如果子类在父类后面，子类将永远不会到达。而且，Java中不能到达的代码是一个错误。

## 十五、异常处理总结



**养成良好的编程习惯，不要把错误给吞噬掉（即捕获到异常以后又不做出相应处理的做法，这种做法相当于是把错误隐藏起来了，可实际上错误依然还是存在的）， 也不要轻易地往外抛错误，能处理的一定要处理，不能处理的一定要往外抛。往外抛的方法有两种，一种是在知道异常的类型以后，方法声明时使用throws把 异常往外抛，另一种是手动往外抛，使用“throw+异常对象”你相当于是把这个异常对象抛出去了，然后在方法的声明写上要抛的那种异常。**