**3.1、认识异常（了解）**

异常是程序之中导致程序中断的一种指令流，下面，通过两个程序来进行异常产生问题的对比。

**范例：**不产生异常的程序

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  int result = 10 / 2 ;  System.out.println("2、除法计算结果：" + result) ;  System.out.println("3、除法计算结束。") ;  }  } |
| 1、除法计算开始。  2、除法计算结果：5  3、除法计算结束。 |

**范例：**产生异常的程序

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  **int result = 10 / 0 ; // 会出现错误**  System.out.println("2、除法计算结果：" + result) ;  System.out.println("3、除法计算结束。") ;  }  } |
| 1、除法计算开始。  Exception in thread "main" java.lang.**ArithmeticException**: / by zero  at TestDemo.main(TestDemo.java:4) |

这时候程序出现了错误，那么出现错误之后的程序不执行了，而是直接进行了错误信息的输出，并且直接结束程序。但是，出现了错误，应该去处理才对，但是现在没有处理。

**3.2、处理异常（重点）**

现在，如果希望程序出现了异常之后，程序依然可以正常的完成的话，那么就可以使用如下的格式进行异常的处理：

|  |
| --- |
| try {  可能出现异常的语句 ;  } [ catch (异常类型 异常对象) {  处理异常 ;  } catch (异常类型 异常对象) {  处理异常 ;  } ... ] [finally {  不管是否出现异常，都执行此代码 ;  }] |

**范例：**使用以上的操作处理异常

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  try {  **int result = 10 / 0 ; // 异常**  System.out.println("2、除法计算结果：" + result) ; // 之前语句有异常，此语句不再执行  } catch (ArithmeticException e) {  System.out.println(e) ; // 异常处理：输出错误信息，java.lang.ArithmeticException: / by zero  }  System.out.println("3、除法计算结束。") ;  }  } |
| 1、除法计算开始。  java.lang.ArithmeticException: / by zero  3、除法计算结束。 |

可以发现，加入了异常处理之后，程序之中即使有了异常，程序也可以正常的执行完毕，但是现在发现，异常处理时的错误输出信息和之前相比，出错的信息不明确了，那么为了让错误的信息更加的完整，一般而言，都会调用printStackTrace()方法进行异常信息的打印，这个方法打印的异常信息是最完整的：

|  |
| --- |
| java.lang.ArithmeticException: / by zero  at TestDemo.main(TestDemo.java:5) |

**范例：**加入finally

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  try {  int result = 10 / 1 ;  System.out.println("2、除法计算结果：" + result) ;  } catch (ArithmeticException e) {  e.printStackTrace() ;  } **finally {**  **System.out.println("不管是否出现异常都执行") ;**  **}**  System.out.println("3、除法计算结束。") ;  }  } |

对于finally的实际应用必须结合之后的异常处理的标准格式来看，单独看，没意义。

但是，对于之前的程序现在又有了问题：现在执行数学计算的两个参数，都是由程序默认提供，那么如果说现在两个计算的参数通过初始化参数传递呢？

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  try {  **int x = Integer.parseInt(args[0]) ; // 接收参数**  **int y = Integer.parseInt(args[1]) ; // 接收参数**  **int result = x / y ;**  System.out.println("2、除法计算结果：" + result) ;  } catch (ArithmeticException e) {  e.printStackTrace() ;  } finally {  System.out.println("不管是否出现异常都执行") ;  }  System.out.println("3、除法计算结束。") ;  }  } |

这个时候，数据由外部传送，那么在这种情况下，就有可能出现以下几类问题：

· 执行时不输入参数（java TestDemo），ArrayIndexOutOfBoundsException，未处理；

· 输入的参数不是数字（java TestDemo a b），NumberFormatException，未处理；

· 被除数为0（java TestDemo 10 0），ArithmeticException，已处理。

可以发现，以上的程序实际上是存在三种异常，而程序之中只能够处理一种，而对于不能处理的异常，发现程序依然会直接中断执行。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  try {  int x = Integer.parseInt(args[0]) ;  int y = Integer.parseInt(args[1]) ;  int result = x / y ;  System.out.println("2、除法计算结果：" + result) ;  } **catch (ArithmeticException e) {**  **e.printStackTrace() ;**  **} catch (ArrayIndexOutOfBoundsException e) {**  **e.printStackTrace() ;**  **} catch (NumberFormatException e) {**  **e.printStackTrace() ;**  **}** finally {  System.out.println("不管是否出现异常都执行") ;  }  System.out.println("3、除法计算结束。") ;  }} |

此时，问题就来了，如果按照以上的方式一次一次的测试来进行异常类型的推断，还不如直接编写多个if..else。

**3.3、异常的处理流程（重点）**

以上已经完成了异常的基本处理流程，但是也可以发现问题，所有的异常都像之前那样一条条的判断似乎是一件不可能完成的任务，因为日后肯定会接触到一些不常见的异常信息，那么下面就必须首先研究异常的流程和结构。

先查看两个异常类的继承结构：

|  |  |
| --- | --- |
| **ArithmeticException：** | **ArrayIndexOutOfBoundsException：** |
| java.lang.Object  |- **java.lang.Throwable**  |- **java.lang.Exception**  |- java.lang.RuntimeException  |- java.lang.ArithmeticException | java.lang.Object  |- **java.lang.Throwable**  |- **java.lang.Exception**  |- java.lang.RuntimeException  |- java.lang.IndexOutOfBoundsException  |- java.lang.ArrayIndexOutOfBoundsException |

可以发现所有的异常类型最高的继承类是Throwable，通过doc文档可以发现在Throwable下有两个子类：

（**面试题：**请解释Error和Exception的区别？）

· Error：指的是JVM错误，这个时候的程序并没有执行，无法处理；

· Exception：指的是程序之中出现的错误信息，可以进行异常处理，主要关心Exception。

那么通过继承关系可以发现，肯定在进行日后异常处理的时候是以Exception为主，而这个时候就可以形成以下的异常处理流程（**面试题：**请解释java之中的异常处理流程。）

1、 如果程序之中产生了异常，那么会自动的由JVM根据异常的类型，实例化一个指定异常类的对象；

2、 如果这个时候程序之中没有任何的异常处理操作，则这个异常类的实例化对象将交给JVM进行处理，而JVM的默认处理方式就是进行异常信息的输出，而后中断程序执行；

3、 如果程序之中存在了异常处理，则会由try语句捕获产生的异常类对象；

4、 与try之后的每一个catch进行匹配，如果匹配成功，则使用指定的catch进行处理，如果没有匹配成功，则向后面的catch继续匹配，如果没有任何的catch匹配成功，则这个时候将交给JVM执行默认处理；

5、 不管是否有异常都会执行finally程序，如果此时没有异常，执行完finally，则会继续执行程序之中的其他代码，如果此时有异常没有能够处理（没有一个catch可以满足），那么也会执行finally，但是执行完finally之后，将默认交给JVM进行异常的信息输出，并且程序中断；

通过以上的分析，可以发现，实际上catch捕获异常类型的操作，就和方法接收参数是一样的，那么按照之前所学习过的对象多态性来讲，所有的异常类都是Exception的子类，那么这个时候，实际上所有的异常都可以使用Exception进行接收。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  System.out.println("1、除法计算开始。") ;  try {  int x = Integer.parseInt(args[0]) ;  int y = Integer.parseInt(args[1]) ;  int result = x / y ;  System.out.println("2、除法计算结果：" + result) ;  } catch (**Exception** e) {  e.printStackTrace() ;  } finally {  System.out.println("不管是否出现异常都执行") ;  }  System.out.println("3、除法计算结束。") ;  }  } |

那么这个时候就应该可以感受到异常处理所带来的好处了。但是这种操作也存在一种问题。如果在一些异常处理要求严格的项目之中，异常必须分别处理，如果现在异常的处理要求不是很严格，直接编写Exception就足够了。

**3.4、throws关键字（重点）**

thrwos关键字主要是在方法定义上使用的，表示的是此方法之中不进行异常的处理，而交给被调用处处理。

|  |
| --- |
| class MyMath {  public int div(int x,int y) throws Exception {  return x / y ;  }  } |

现在的div()方法之中抛了一个异常出来，表示的是，所有的异常交给被调用处进行处理。

|  |
| --- |
| class MyMath {  public int div(int x,int y) throws Exception {  return x / y ;  }  }  public class TestDemo {  public static void main(String args[]) {  try {  System.out.println(new MyMath().div(10,0)) ;  } catch (Exception e) {  e.printStackTrace() ;  }  }  } |

在调用throws声明方法的时候，一定要使用异常处理操作进行异常的处理，这个是属于强制性的处理，而现在主方法本身也属于方法，那么实际上在主方法上也可以继续使用throws进行异常的抛出。

|  |
| --- |
| class MyMath {  public int div(int x,int y) throws Exception {  return x / y ;  }  }  public class TestDemo {  public static void main(String args[]) **throws Exception** {  System.out.println(new MyMath().div(10,0)) ;  }  } |

这个时候表示的是将异常继续向上抛，交给JVM进行异常的处理。

**3.5、throw关键字（重点）**

之前的所有异常类对象都是由JVM自动进行实例化操作的，而现在用户也可以自己手工的抛出一个实例化对象，就通过throw完成了。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]){  try {  **throw** new Exception("抛着玩的，你管不着。") ;  } catch (Exception e) {  e.printStackTrace() ;  }  }  } |

理论上，对于异常应该躲，现在属于不躲反而自己抛了，有病啊？

**3.6、异常处理的标准格式（重点）**

现在觉得有两个内容实在是没有用处：finally、throw。

现在要求定义一个div()方法，而这个方法有如下的一些要求：

· 在进行除法操作之前，输出一行提示信息，可以理解为大爷开门；

· 在除法操作执行完毕后，输出一行提示信息，可以理解为大爷关门；

· 如果中间产生了异常，则应该交给被调用处来进行处理，譬如：xxOver了。

|  |
| --- |
| class MyMath {  // 出现异常要交给被调用处出，使用throws  **public int div(int x,int y) throws Exception {**  **System.out.println("===== 计算开始 =====") ; // 资源打开**  **int result = 0 ;**  **try {**  **result = x / y ; // 除法计算**  **} catch (Exception e) {**  **throw e ; // 向上抛**  **} finally {**  **System.out.println("===== 计算结束 =====") ; // 资源关闭**  **}**  **return result ;**  **}**  }  public class TestDemo {  public static void main(String args[]){  try {  System.out.println(new MyMath().div(10,0)) ;  } catch (Exception e) {  e.printStackTrace() ;  }  }  } |

而对于以上的操作代码也可以按照如下的简化格式编写。

|  |
| --- |
| class MyMath {  // 出现异常要交给被调用处出，使用throws  public int div(int x,int y) throws Exception {  System.out.println("===== 计算开始 =====") ; // 资源打开  int result = 0 ;  **try {**  **result = x / y ; // 除法计算**  **} finally {**  **System.out.println("===== 计算结束 =====") ; // 资源关闭**  **}**  return result ;  }  }  public class TestDemo {  public static void main(String args[]){  try {  System.out.println(new MyMath().div(10,0)) ;  } catch (Exception e) {  e.printStackTrace() ;  }  }  } |

以上不带有catch的操作在开发之中并不建议使用。永远的标准格式：try…catch、finally、throws、throw一起使用。

**3.7、RuntimeException（重点）**

在讲解此操作之前，首先来观察如下的一段代码：

|  |
| --- |
| public class TestDemo {  public static void main(String args[]){  String str = "123" ;  int num = Integer.parseInt(str) ;  System.out.println(num \* num) ;  }  } |

这个程序就是将一个字符串变为了基本数据类型，而后执行乘法操作，但是下面来看一下parseInt()方法定义：

|  |
| --- |
| public static int parseInt(String s) throws NumberFormatException |

发现这个方法上抛出了一个NumberFormatException的异常，按照之前所讲，如果存在了throws，则必须使用try…catch进行处理，可是现在却没有强制要求处理，来观察一下NumberFormatException的继承结构：

|  |
| --- |
| java.lang.Object  |- java.lang.Throwable  |- java.lang.Exception  |- java.lang.**RuntimeException**  |- java.lang.IllegalArgumentException  |- java.lang.NumberFormatException |

发现NumberFormatException属于RuntimeException的子类，而在Java之中明确规定了，对于RuntimeException的异常类型可以有选择性的来进行处理，在开发之中，如果没有处理，那么出现异常之后将交给JVM默认进行处理。

**面试题：**请解释一下RuntimeException和Exception的区别？请列举出几个常见的RuntimeException；

· RuntimeException是Exception的子类；

· Exception定义了必须处理的异常，而RuntimeException定义的异常可以选择性的进行处理；

· 常见的RuntimeException：NumberFormatException、ClassCastException、NullPointerException、ArithmeticException、ArrayIndexOutOfBoundsException。

**3.8、断言：assert（了解）**

断言指的是程序执行到某行之后，其结果一定是预期的结果，而在JDK 1.4之后增加了一个assert关键字。

**范例：**使用断言

|  |
| --- |
| public class TestDemo {  public static void main(String args[]){  int x = 10 ;  // 假设经过了若干操作  **assert x == 30 : "x的内容不是30" ;**  System.out.println(x) ;  }  } |

默认情况下，Java之中的断言，不会在正常执行的代码中出现，如果要想启用断言，则应该增加一些选项：

|  |
| --- |
| java -ea TestDemo |

不过断言在开发之中几乎是不使用的技术，知道这个关键字就完了。

**3.9、自定义异常类（了解）**

在Java之中本身已经提供了大量的异常类型，但是在开发之中，这些异常类型根本就不能满足于开发的需要。所以在一些系统架构之中往往会提供一些新的异常类型，来表示一些特殊的错误，而这种操作就称为自定义异常类，而要想实现这种自定义异常类，那么可以让一个类继承Exception或RuntimeException。

**范例：**自定义异常类

|  |
| --- |
| class MyException extends Exception { // 自定义异常类  public MyException(String msg) {  super(msg) ;  }  }  public class TestDemo {  public static void main(String args[]) throws Exception {  throw new MyException("自己的异常类") ;  }  } |

如果以后见到了一些特殊的没见过的异常类，基本上都表示自定义的。

**4、总结**

1、 异常的处理流程，出现错误之后的程序不执行；

2、 异常处理的标准结构：try、catch、finally、throw、throws的使用；

3、 RuntimeException和Exception的区别。