第5章 Java异常处理和日志技术

由于在软件设计中存在错误，于1996年6月4日发射的阿丽亚娜5型飞行器501(Ariane 5 Flight 501)在发射后40s内就失败了，损失了5千万美元。原因仅是一个细小的软件错误造成的，这个错误就是一些本来为之前版本的阿丽亚娜4型飞行器编写的程序代码抛出了异常，该程序段在飞行中并不是真正需要的，但是被遗留下来了。该段程序计算出了非常大的数字，并试图保存到很短的数据存储空间中，从而造成了溢出。但程序没有提供处理程序来捕获和处理这种情况。这种特殊情况本应该不会轻易发生，即使提供空的错误处理程序，也将有可能拯救这种失败，但是在缺少错误处理程序的情况下，错误传送到了操作系统，中止了该进程。遗憾的是该程序是火箭的引导程序，火箭就这样自我毁灭了。

从以上的故事中可以看出异常处理的重要性，用户在设计和开发软件时，事先无论如何仔细考虑，或多或少都会出现一些意想不到的问题或错误，这就使得程序员在软件开发过程中越来越重视测试环节。在编程过程中，经常会出现的错误有语法错误、逻辑错误以及异常情况。


视频讲解

5.1 异常的概念和处理机制

5.1.1 异常的定义

异常是一项工作流程中的非正常状况，它会改变事先设计好的流程，导致错误的结果，或者流程无法进行等。在计算机的进程中一旦引发异常，该进程将突然中止，且对CPU的控制权将返回给操作系统，而在发生异常后，此前分配的其他资源都将保留在相同的状态，这将导致资源漏洞。

在程序中试图处理这些异常就称为异常处理。Java语言提供了一套比较完善的异常处理机制，正确地运用这套机制，有助于提高程序的健壮性。所谓程序的健壮性，就是指程序在多数情况下能够正常运行，返回预期的正确结果，如果偶尔遇到异常情况，程序也能采取周到的解决措施。而不健壮的程序则没有事先充分预计到可能出现的异常，或者没有提供强有力的异常解决措施，导致程序在运行时经常莫名其妙地终止，或者返回错误的运行结果，而且难以检测出现异常的原因。

5.1.2 异常的处理机制

Java语言采用了面向对象的思想来处理异常，这使得程序具有更好的可维护性。即正在运行的Java应用程序在发生异常时，会创建异常对象来封装错误信息并将其抛出，程序

的控制权会发生转移,转移到捕获代码处并尝试捕获此异常对象,如果捕获成功,则程序的控制权会转移到此处继续执行,以便进行相应的异常分析和异常处理。如果没有成功捕获异常对象,异常会沿着调用堆栈向上传递,在调用方法中再进行异常的捕获处理,如果还是没能成功捕获和处理,则继续向上传递,直到 JVM 终止退回到操作系统。Java 异常处理机制具有以下优点。

(1) 把各种不同类型的异常情况进行分类,用 Java 类来表示异常情况,这种类被称为异常类。把异常情况表示为异常类,可以发挥类的可扩展性和可重用性的优势。

(2) 异常流程代码和正常流程代码分离,提高了程序的可读性,简化了程序的结构。

(3) 可以灵活地处理异常,如果当前方法有能力处理异常,就捕获异常并处理它,否则只需要抛出异常对象,由方法调用者来处理它。

在 Java 中采用了 5 个关键字来处理异常。

(1) try: 尝试执行的代码段。

(2) catch: 用来捕获异常对象代码段。

(3) finally: 不管是否有异常,最终都要执行的代码段。

(4) throw: 用来明确地抛出异常对象。

(5) throws: 用来声明方法可能会出现的异常。

一般情况下,异常流程由 try-catch-finally 语句来控制。如果程序中还包含 return 和 System.exit()语句,就会使流程变得更加复杂。所以,在异常处理模块中,尽量不要出现 return 或 System.exit()的语句。

5.1.3 程序中的异常分类

在 Java 中,可以抛出和捕获的异常可以分为两大类:

(1) 内部错误(或称重量级异常)是由 Error 类的子类产生,错误发生时程序无法继续执行,它们有时也被称为硬错误。这种类型的典型实例就是 OutOfMemoryError,通常这些类型的致命错误由 Java API 或 Java 虚拟机本身抛出。

(2) 轻量级的异常也被称为非致命错误,即程序出现异常但不是那么严重,并且应用程序在绝大多数情况下是可以解决的。这类异常又可以分为以下 2 种。

① 由 RuntimeException 及其子类产生的异常对象,如 IndexOutOfBoundsException 或 IllegalArgumentException 类型的异常对象,这种异常对象一般只能发生在进程中,也称这些异常为非检查异常,编译器一般不检查这类异常。

② 除 RunTimeException 类系外的其他异常都是检查异常,这些异常是由程序之外的某些外部原因导致的,例如磁盘错误或网络连接中断导致的 IOException。编译器在编译时会检查这类异常,确保它们已经得到了处理,如果程序中有没有处理的检查类异常,程序将无法通过编译。

视频讲解

5.2 Java 语言中的异常类层次

在程序运行中,任何中断正常流程的因素都被认为是异常。按照面向对象的思想,Java 语言用异常类对象来封装错误发生时的现场信息及可能的情况信息。所有异常类的祖先类

为 java.lang.Throwable 类，它指向的实例表示具体的异常对象，可以通过 throw 语句抛出。Throwable 类提供了访问异常信息的一些方法，常用的方法如下。

(1) getMessage()：返回 String 类型的异常信息。

(2) printStackTrace()：打印跟踪方法调用栈而获得的详细异常信息。在程序调试阶段，此方法可用于跟踪错误。

JDK 提供的异常类层次如图 5-1 所示。

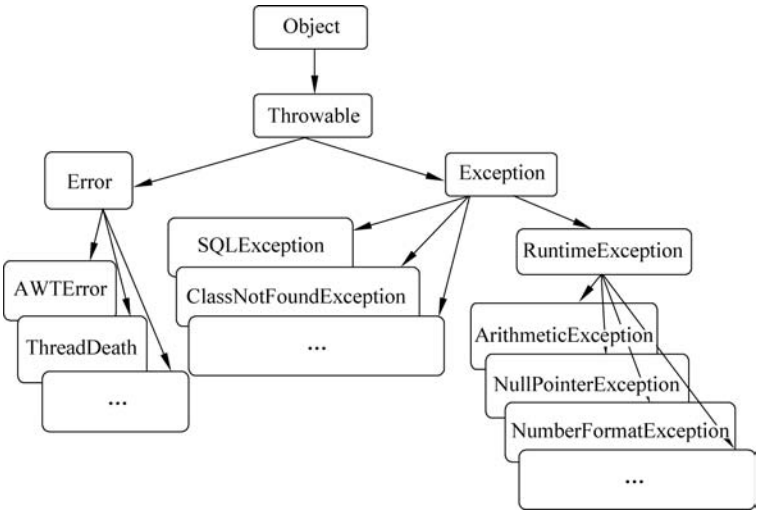

图 5-1　异常类的层次图

可以看到，在 Java 中所有的异常类都是 Throwable 的子类，它派生出两个类，Exception 类表示程序本身可以处理的轻量级异常，即程序运行时会出现这类异常，程序可以捕获并应该尽可能地处理，使程序能够恢复执行；Error 类及其子类表示仅靠程序本身无法修复的重量级异常，例如内存空间不足，或者 JVM 方法调用栈溢出。在大多数情况下，当遇到这样的错误时，建议终止程序运行。表 5-1 列出了 Java 语言中常见的异常类及其说明。

表 5-1　常见的异常类及其说明

异常类名	说　明	备　注
Exception	发生异常	轻量级异常类的父类(熟练掌握)
ClassNotFoundException	找不到类	编译或运行时检查类异常(常用)
InterruptedException	线程被打断	运行时异常(线程中用)
IllegalAccessException	非法访问异常	违法访问规则
NoSuchMethodException	找不到方法	编译或运行时检查类异常
RuntimeException	运行时发生异常	运行时异常的父类
ArithmeticException	数学溢出异常	运行时异常类的子类(熟练掌握)
IllegalArgumentException	非法参赛异常	方法传递的参赛有误
IllegalThreadStateExcepton	线程状态异常	IllegalArgumentException 的子类
IndexOutOfBoundsException	下标越界异常	RuntimeException 的子类
ArrayStoreException	数组存储异常	在读写数组数据时发生异常
NumberFormatException	数字格式异常	字符串转换为数字时发生异常

续表

异常类名	说　明	备　注
ArrayIndexOutOfBoundsException	数组下标越界异常	IndexOutOfBoundsException 的子类(常用、熟练掌握)
NegativeArraySizeException	数组大小为负数异常	RuntimeException 的子类
NullPointerException	空引用异常	RuntimeException 的子类(常用)
SecurityException	安全类异常	RuntimeException 的子类
IOException	输入输出类异常	进行输入输出时的异常
FileNotFoundException	找不到文件异常	运行时异常
SQLException	SQL 查询类异常	数据库操作时发生异常
AWTException	抽象窗口类异常	在图形界面中发生画图异常

注:建议初学者应首先掌握粗体标识的基本异常类。

Java 语言中可用的处理异常方式有以下两种。

(1) 自行处理:将可能引发异常的语句封入在 try 块内,而将处理异常的相应语句封入 catch 块内。

(2) 回避异常:在方法声明中包含 throws 子句,通知潜在调用者,如果发生了异常,必须由调用者处理。

5.2.1 自行异常处理

为了在程序中捕获轻量级的异常,Java 提供了名为 try-catch 的语句结构。将运行时可能产生异常的可疑代码放到 try 代码块中,将处理异常的代码放到对应的 catch 块中,其基本结构的语法如下:

```
try {
    //可能引起异常的代码段
}catch(Exception e) {
    //处理异常 e
}
```

当 try 代码块中出现一个异常时,这个代码块的剩余部分将终止执行,程序会生成并抛出一个异常对象,然后转入 catch 模块捕获异常,当捕获到异常对象时,则转入对应的 catch 代码块执行。如上所示,catch 代码块中的参数 e 类似于方法定义中的形参 e,当 catch 捕获到一个异常对象时,引用 e 就会指向此异常对象,并可以使用相应的方法。

【例 5-1】 演示 try…catch 结构。

```
public class trycatch {
  public static void main(String[] args) {
    int x = 12;int y = 0;
    try {
      x = x/y;
      System.out.println("You won't see this!");
    }catch(Exception e){
        System.out.println("Catch a Exception!");
        e.printStackTrace();                    //打印调用堆栈
    }
```

```
    }
}
```

有时，单个代码片段可能会引起多个异常，这种情况下可提供多个 catch 块分别处理各种异常类型。比如在一个跨国高速公路的出口处，需要同时设置好几个检查站，有海关部门、缉毒部门、刑侦部门、卫生防疫部门等，根据不同的异常情况可由不同的部门进行处理。

【例 5-2】 演示多 catch 结构。

```
import java.util.Scanner;
public class trycatchdemo {
  public static void main(String[] args) {
     int a = 0,b = 0,c = 0;
     try {
       a = Integer.parseInt(args[0]);
       b = Integer.parseInt(args[1]);
       c = a/b;
       System.out.println("c = " + c);
       System.out.println(" ****** 正常执行 ***** ");
       System.out.println("实数的除法不产生数学类异常:3.0/0.0 = " + (3.0/0.0));
     }
     catch(ArrayIndexOutOfBoundsException e1)  {
        System.out.println("此程序要输入两个参数!");
     }
     catch(NumberFormatException e2) {
        System.out.println("必须输入数字! ");
     }
     catch(ArithmeticException e3) {
       System.out.println("除数不能为 0!");
       System.out.print("请重新输入除数:");     //演示如何补救
       Scanner in = new Scanner(System.in);
       b = in.nextInt();
       c = a/b;
       System.out.println("c = " + c);
     }
  }
```

注意：在多个 catch 的情况下，子类异常应该出现在父类异常之前，否则子类异常永远不会捕获到，因为第 4 章讲过，父类引用可以指向子类对象。

5.2.2 回避异常处理

回避异常是指在异常发生的方法中不使用 try-catch 做异常处理，而是将异常交给调用者来处理，通常在定义此方法时要用 throws 声明可能抛出的异常类型，在代码中用 throw 明确地抛出一个异常对象。

【例 5-3】 演示 throw 和 throws 配合处理异常。

```
class ThrowsDemo {
    public static void main(String [] args) {
        Person zhangsan = new Person();
        try     {
```

```
                zhangsan.setName("张三");
                zhangsan.setAge(20);
                zhangsan.display();
            }
            catch(Exception e) {
               System.out.println(e.getMessage());
            }
        }
    }
    class Person {
      private int age;
      private String name;
      public int getAge(){return age;}
      public String getName(){return name;}
      public void setAge(int a) throws Exception {
        if(a<0)
            throw new Exception("出现异常:年龄不能小于零!");
        if(a>150)
            throw new Exception("出现异常:年龄大于 150 的概率很小!");
        age=a;
      }
      public void setName(String nn){name=nn;}
      public void display(){
        System.out.println("我叫"+name+"\n"+"我今年"+age+"岁!");
      }
    }
```

注意:throws 可以抛出多种类型的异常,异常类型之间以逗号分隔。

视频讲解

5.2.3 异常情况下的资源回收和清理工作

前面已经说过,当异常发生时已经分配的资源会保持原来的状态,不能被释放。为了避免这种情况,Java 提供了 finally 关键字用来修饰一个代码块,此代码块不管有没有异常发生都要执行,该代码块主要用来释放和清理有关的资源或善后工作,完整的 try-catch-finally 语法如下所示。

```
try {
    尝试执行的代码块
}catch(异常类 变量名){
    异常处理代码块
}
finally {
    总是要执行的代码块
}
```

可以将 finally 和 try 结合使用而无须 catch 块。换言之,在程序代码中完全可以省略 catch 块。finally 的演示示例如下所示。

【例 5-4】 演示 finally 关键字。

```
class FinallyDemo {
```

```
int no1,no2;
FinallyDemo(String args[])    {
try {
    no1 = Integer.parseInt(args[0]);
    no2 = Integer.parseInt(args[1]);
    System.out.println("相除结果为 " + no1/no2);
}
catch(ArithmeticException i) {
    System.out.println("不能除以 0");
}
catch(ArrayIndexOutOfBoundsException e1) {
        System.out.println("此程序要输入两个参数!");
     }
     catch(NumberFormatException e2) {
       System.out.println("必须输入数字! ");
     }
  finally {
    System.out.println("Finally 已执行,用来做清理工作!");
  }
  }
  public static void main(String args[]) {
    new FinallyDemo(args);
  }
}
```

注意：当代码中的 return 语句被执行时，不影响 finally 块的执行，即在返回调用方法前，finally 块还是要执行的。finally 块中如果出现 break、continue 语句可能会逻辑混乱，即如果有 finally 的情况下，try 中的 break 或 continue 都不会立即执行，程序会将 finally 中的语句执行完，所有一般不要再跨 try 模块使用 break 或 continue。try 块可以有 0 个或多个 catch 块，但最多只能有一个 finally 块。

5.2.4 带资源的 try 语句

为了简化编程，Java SE 7 又给 try 块增加了新的语法，从而提供更简单的方法来清理资源。使用这种新的语法，可在 try 语句中打开资源，当语句执行结束时会自动清理或关闭资源，可在 try 块中包含多个资源，每一个资源用分号隔开，如下例程序所示。

【例 5-5】 演示带资源的 try 语句。

```
import java.io.*;
public class FileCopy {
    public static void main(String[] args){
        try (InputStream fis = new FileInputStream(new File("src.txt"));
            OutputStream fos = new FileOutputStream(new File("dest.txt"))) {
            byte[] buf = new byte[8192];
            int i;
            while((i = fis.read(buf))!= -1){
                fos.write(buf,0,i);
            }
        }catch(Exception e){
```

```
            e.printStackTrace();
        }
    }
}
```


最新版本的 try 语句可以不带 catch 或 finally 子句,可以通过方法声明中的 throws 直接上传抛出的异常对象。

视频讲解

5.3 自定义异常

JDK 提供的异常类不可能涵盖所有的异常类型,所以在特定的问题领域可以通过扩展 Exception 类或其子类来创建自定义的异常类,以适合特定的商业逻辑,例如在 ATM 上取款,单日累计不能超过 20000,计算机不会因为 20000 或 20001 产生错误,这是人的逻辑,所以需要自定义异常。自定义异常类应该包含了和异常相关的信息,有助于负责捕获异常的 catch 代码块正确地分析并处理异常。自定义异常类通常都是通过创建异常对象,然后用 throw 抛出该异常对象。

【例 5-6】 演示通过 Exception 的子类派生的自定义异常类。

```
class ArraySizeException extends NegativeArraySizeException{
   ArraySizeException() {
      super("您传递的是非法的数组大小");
  }
}
class UserExceptionDemo {
  int size, array[];
  UserExceptionDemo(int s) {
  size = s;
   try {     checkSize();     }
   catch(ArraySizeException e) {System.out.println(e);}
}
void checkSize() throws ArraySizeException {
  if(size < 0)      throw new ArraySizeException();
  array = new int[size];
  for(int i = 0; i < size; i++) {
     array[i] = i + 1;
  System.out.print(array[i] + " ");
}}
public static void main(String arg[]) {
   new UserExceptionDemo(Integer.parseInt(arg[0])); }
}
```

【例 5-7】 演示 Exception 类派生的自定义异常类。

```
class myexception extends Exception {
      String mymsg = "我自己定义的异常!";
      double mynum = 2.0;
      myexception(){super("首字母不能为 A!");}
      myexception(String msg){super(msg);}
```

```
    public void displayme(){System.out.println(mymsg);}
    public double mymethod(){return Math.sqrt(mynum);}
}
class exceptiontest {
     public static void main(String[] args) {
        try {
          if(args[0].charAt(0) == 'A') {
                myexception e = new myexception();
                System.out.println("kkkk:" + e.mymethod());
                e.displayme();
                System.out.println(" ********** in try ********* ");
                throw e;
           }
           else if(args[0].charAt(0) == 'B') {
                throw new myexception("第一个字符不应是 B!");
            }else{System.out.println(args[0]);}
          }
          catch(myexception aaaa) {
                System.out.println(aaaa.getMessage());
                aaaa.displayme();
                System.out.println("" + aaaa.mymethod());
          }
          catch(ArrayIndexOutOfBoundsException e){
              System.out.println("命令行参数个数错!");
          }
     }
}
```

从此例可以看出，异常类的设计和普通的类的设计区别不大，只不过异常类对象可以用 throw 关键字抛出，由 catch 捕获而已。

5.4 使用异常的指导原则

如何明智地、有效地使用异常处理，是每一个资深程序员必须要掌握的技巧。异常处理的黄金法则如下。

(1) 要具体。

(2) 早抛出。

(3) 晚捕获。

进行简单的检查就可以避免抛出异常，应尽量使用检查语句。例如，在一个空对象上调用方法会抛出 NullPointerException 异常，用 if(ref! =null) {...}要比用 try-catch 语句有效得多；再比如执行堆栈上的弹出操作时，必须确认堆栈不为空，简单地进行检查将会发现这种情况，这比异常处理的效率更高。

初学者喜欢将 try-catch 块放到程序代码中任何可能的地方，使用过多的 try-catch 将导致代码凌乱，并可能掩盖程序的主要逻辑。在单个 try 块组织所有可疑的语句，并为这个 try 块提供多个 catch 处理程序。如果喜欢，可以抛出异常对象给调用者，然后集中所有的异常处理。

另外,永远不要使用空的异常处理语句,即 catch(Exception e){},因为在任何时候如果出现异常,异常将会被悄悄地忽略。而忽略异常将导致不可预知的程序状态,这将是很难诊断和修复的。

而且不幸的是,Java 的异常在实现上有一个小缺陷。尽管异常是程序中出现危机的迹象,并且不应该被忽略,但是在 Java 中异常也有可能丢失。这种情况在使用 finally 子句的特定配置时可能发生,或者在新的异常对象覆盖旧的异常对象时发生。

如果从构造方法中抛出异常,这些清理行为可能无法正常发生。这意味着在编写构造方法时必须仔细斟酌。

【例 5-8】 异常丢失示例。

```
class VeryImportantException extends Exception {  //重要的异常
  public String toString() {
    return "A very important exception!";
  }
}
class HoHumException extends Exception {           //不太重要的异常
  public String toString() {
    return "A trivial exception";
  }
}
public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) {
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.f();                                    //抛出重要的异常对象
      } finally {
        lm.dispose();                              //抛出不重要的异常对象,原来的异常对象丢失了
      }
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

程序运行时前面先抛出的异常对象丢失了,所以在处理异常时一定要正确地构造和传递异常信息。

5.5 日　　志

每个 Java 程序员都熟悉向程序中插入 System.out.println()调用的过程,用来检查程序的执行过程,但这些会扰乱程序正常的执行过程,当然,如果程序能够正常执行,可以删除这些插入的打印语句。这会带来不小的代码工作开销,并且容易出错,Java 提供日志 API

的设计就是为了解决这个问题，其主要优势如下。

(1) 很容易隐藏所有的日志记录，或者只是那些低于某个级别的日志记录，同样也很容易把它们打开。

(2) 被抑制的日志代码开销非常小，因此在发布的软件系统代码中，遗留的日志记录代码的影响基本上可以忽略不计。

(3) 可以将日志记录定向到不同的处理程序，用于在控制台中显示或写入到文件等。

(4) 日志记录器和处理程序都可以过滤记录，过滤器可以使用用户提供的标准选择来丢弃日志记录。日志记录可以采用不同的格式，例如纯文本或 XML 格式。

(5) 应用程序可以使用多个日志记录器，这些日志记录器具有层次化的名称，例如 com. mycompany. myapp，类似于包名。

(6) 默认情况下，日志配置由配置文件控制，应用程序如果需要，可以替换这个机制。

5.5.1 日志简单使用

对于简单的日志使用，类似于 System. out. println 方法，使用 java. util. logging 包中 Logger 类的静态方法获取一个日志记录器对象，然后调用 info()方法输出日志。

```
Logger.getGlobal().info("简单日志记录");
```

默认情况下，日志记录的输出如下：

```
四月 27, 2020 3: 43: 12 下午 chap05.SimpleLogging main
信息: 简单日志记录
```

但如果设置了日志记录级别，例如 Logger. getGlobal(). setLevel(Level. OFF)，则后面的日志就被忽略了。

【例 5-9】 简单日志演示。

```
import java.util.logging.Level;
import java.util.logging.Logger;
public class SimpleLogging {
    public static void main(String[] args) {
        Logger.getGlobal().setLevel(Level.OFF); //后面日志的记录可用 Level.ON 或 Level.OFF
                                                //控制
        System.out.println("普通输出语句");
        Logger.getGlobal().info("简单日志记录");
    }
}
```

5.5.2 日志高级使用

前面已经讲解了简单的日志记录，下面介绍一下高级日志记录。在专业应用程序中，不希望将所有记录都记录到一个全局记录器中。相反，可以定义自己的日志记录器。调用 getLogger 方法来创建或检索一个日志程序，并给日志对象起一个日志名称，例如“Logger myLogger = Logger. getLogger("com. mycompany. myapp");”与包名类似，日志名也是层次化的。事实上，它们比包更具有层次结构。包和它的父包之间没有语义关系，但是

logger 父包和子包共享某些属性。例如,如果在日志记录器 com 上设置日志级别,子日志记录器就会继承了这个级别,日志级别如表 5-2 所示。

表 5-2 Java 中日志级别

SEVERE	严 重
WARNING	警告
INFO	信息
CONFIG	配置
FINE	良好
FINER	较好
FINEST	最好
ALL	开启所有级别日志记录
OFF	关闭所有级别日志记录

默认情况下,前 3 个级别是被记录的。可以设置一个不同的日志级别,如"logger.setLevel(Level.Fine);"Fine 上的所有级别的日志都会被记录。也可以使用 Level.ALL 打开所有级别的日志记录器。可以采用方法记录日志,所有级别都有对应的日志记录方法,例如 log.warning(message)、logger.fine(message)等,或者可以使用 log 方法并提供级别,例如用 log.log(Level.Fine,message)来记录日志。

默认情况下,日志记录器将记录发送到一个 ConsoleHandler,后者将记录打印到系统中的错误流(System.err)。要将日志记录发送到其他地方,需要添加其他日志处理程序。日志 API 为此提供了两个有用的处理程序:FileHandler 和 SocketHandler。SocketHandler 将记录发送到指定的主机和端口,FileHandler 在文件中收集日志记录。可以简单地将记录发送到一个默认的文件处理程序,例如以下代码:

```
FileHandler handler = new FileHandler();
logger.addHandler(handler);
```

这些记录被发送到用户主目录下的 java%d.log 文件中,其中%d 是使文件唯一的数字。在实际应用中可以指定日志文件的路径,经处理器同意后可以设置日志记录级别,默认的日志文件是 XML 格式,代码如下所示:

```
<record>
  <date>2020-04-27T18:47:12</date>
  <millis>1587984432426</millis>
  <sequence>3</sequence>
  <logger>javasoft.blog</logger>
  <level>SEVERE</level>
  <class>chap05.TestLogging</class>
  <method>main</method>
  <thread>1</thread>
  <message>严重</message>
</record>
```

日志的一个常见的用途是记录意外的异常。有两个方法常用来处理包括异常信息描述和日志中记录异常信息。

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

典型的使用方式如下：

```
try {
  if(...){
    IOException exception = new IOException("....");
    logger.throwing("com.flyhorsespace.www","read",exception);
    throw exception;
  }
  ...

}catch(IOException e){
  logger.("com.flyhorsespace.www").log("Level.WARNING, "装入图像",e);
}
```

throwing 调用日志记录器并抛出一条带有异常信息的异常对象，在捕获异常的处理代码中用 log 方法记录相关信息。以下程序演示了日志的常用 API。

【例 5-10】 日志高级使用演示。

```
import java.io.IOException;
import java.util.logging.ConsoleHandler;
import java.util.logging.FileHandler;
import java.util.logging.Level;
import java.util.logging.Logger;
public class TestLogging {
    public static void main(String[] args) throws IOException {
        Logger log = Logger.getLogger("javasoft");
        log.setLevel(Level.INFO);
        Logger log1 = Logger.getLogger("javasoft");
        System.out.println(log == log1);       // true
        Logger log2 = Logger.getLogger("javasoft.blog");
        ConsoleHandler consoleHandler = new ConsoleHandler();
        consoleHandler.setLevel(Level.FINE);
        log1.addHandler(consoleHandler);
        FileHandler fileHandler = new FileHandler("d:/temp/testlog%g.log");
        log2.addHandler(fileHandler);
        log2.setLevel(Level.FINEST);
        log1.severe("严重");
        log1.warning("警告");
        log1.info("信息");
        log1.config("配置");
        log1.fine("良好");
        log1.finer("较好");
        log1.finest("最好");
        log2.severe("严重");
        log2.warning("警告");
        log2.info("信息");
        log2.config("配置");
        log2.fine("良好");
```

```
            log2.finer("较好");
            log2.finest("最好");
        }
    }
```


5.6 类设计指导原则

到现在为止,类的基本设计原理和 Java 语言的基本语法已经介绍完了,可以尝试开发一些具有实用价值的程序,接下来总结一些类设计中指导原则。

5.6.1 内聚

一个类应该抽象成一个单独的实体类型,并且所有的类操作应该在逻辑上结合在一起以支持一个一致的目的。例如,可以为学生使用一个类,但是不应该将学生和职员合并到同一个类中,因为学生和职员是不同的实体。

一个具有许多职责的实体可以被分成几个类来分割职责。例如,String、Stringbuilder 和 Stringbuffer 类都处理字符串,但它们的职责不同。String 类处理不可变的字符串,Stringbuilder 类用于创建可变字符串,而 StringBuffer 类与 Stringbuilder 类似,只是 StringBuffer 包含用于更新字符串的同步方法。

5.6.2 一致

应遵循标准的 Java 编程风格和命名约定,为类、数据字段和方法选择信息丰富的名称。一种流行的风格是将数据声明放在构造函数之前,将构造函数放在方法之前。

让名称保持一致。为类似的操作选择不同的名称不是一个好的实践。例如,length()方法返回字符串、Stringbuilder 和 Stringbuffer 的大小。如果在这些类中对这个方法使用了不同的名称,那么它将是不一致的。

通常,应该始终如一地提供一个公共的无参数构造函数来构造默认实例。如果一个类不支持无参数构造函数,记录下原因。如果没有显式定义构造函数,则假定为具有空主体的公共默认无参数构造函数。

如果希望阻止用户为类创建对象,可以在类中声明私有构造函数,与 Math 类相同。

5.6.3 封装

类应该使用私有修饰符来隐藏它的数据,不让客户端直接访问,这使得类很容易维护。

只有在希望数据字段可读的情况下才提供 getter 方法,只有在希望数据字段可更新的情况下才提供 setter 方法。例如,在上一章的有理数建模案例中,Rational 类为分子和分母提供了一个 getter 方法,但是没有提供 setter 方法,因为 Rational 对象是不可变的。

5.6.4 清晰

内聚性、一致性和封装是实现设计清晰性的良好指导原则。一个类应该有一个清晰的契约,易于解释和理解。

用户可以在不同的场合中用不同的顺序、不同的方式组合任意组合使用类。因此应该设计一个类，对如何或何时使用没有限制，允许用户可以随意使用它，设计属性时，让用户可以以任何顺序和值的组合来设置其值；设计方法时，方法应该独立于它们调用的顺序。例如，学生类包含属性姓名、性别、年龄等属性，可以按任何顺序设置这些属性的值。

应该直观地定义方法，而不引起混淆。例如，java.lang包中的String类中的substring(int beginindex, int endindex)方法有点令人困惑。该方法将一个子字符串从beginindex返回到endindex-1，而不是返回到endindex。将一个子字符串从beginindex返回到endindex会更直观一些。

不应该声明可以从其他数据字段派生的数据字段。例如，下面的Person类有两个数据birthdate和age。由于年龄可以从出生日期派生，所以不应该将年龄声明为数据字段。

```
public class Person {
    private Date birthdate;
    private int age;                                    //没有必要
    ...
}
```

5.6.5 完整

类是为许多不同的用户设计的。为了在广泛的应用程序中发挥作用，抽象的类应该是自我独立和完整的，可以能通过属性和方法提供各种功能，基本不需要二次编码去实现特定功能。例如，String类包含40多个用于各种应用程序的方法。

5.6.6 合理区分实例和静态

依赖对象的具体属性的变量或方法必须是实例变量或方法。一个类的所有实例对象共享的变量应该声明为静态的。始终从类名(而不是引用变量)中引用静态变量和方法，以提高可读性并避免错误。不要从构造函数中传递参数来初始化静态数据字段，最好使用setter方法来更改静态数据字段。

实例和静态是面向对象编程的组成部分。一个变量或方法是对象层次(实例)或类层次(静态)的，应该依赖于应用场景或需求。不要错误地忽略类方法和类变量设计，当需要定义一个类方法时却定义为实例方法是一个常见的设计错误。例如，计算n的阶乘的factorial(int n)方法应该被定义为静态的，因为它独立于任何特定的实例。

构造函数总是实例方法，因为它用于创建特定的实例。可以从实例方法调用静态变量或方法，但不能直接从类方法调用实例变量或方法，必须先创建实例对象后，通过对象访问。

5.6.7 继承和聚合

继承和聚合之间的区别就是is-a关系和has-a关系之间的区别。例如，苹果是一种水果；因此可以使用继承来建模类Apple和Fruit之间的关系。一个汽车有发动机，因此，应该使用聚合来建模类Car和Engine之间的关系。

5.6.8 接口与抽象类

接口和抽象类都可用于指定对象的公共行为。如何决定是使用接口还是使用类？一般地,一个强的 is-a 关系可以明确地描述父子关系,其应该使用抽象类和继承来建模。例如,由于橘子是水果,它们之间的关系应该使用类继承来建模。弱 is-a 关系,也称为 is-kind-of 关系,表示一个对象具有某种属性。弱 is-a 关系可以使用接口来建模。例如,所有字符串都是可比较的,所以 String 类实现了 Comparable 接口。圆形或矩形是几何对象,因此 Circle 可以设计为 Shape 的子类。圆是不同的,可以根据它们的半径或面积进行比较,所以 Circle 可以实现 Comparable 接口。

接口比抽象类更灵活,因为子类只能扩展一个超类,但可以实现任意数量的接口。但是,接口不能包含具体的方法。抽象类通过创建接口和实现接口,可以将接口和抽象类的优点结合起来。然后可以通过方便的方式使用接口或抽象类。

视频讲解

5.7 程序建模示例

【程序建模示例 5-1】 处理输入错误示例。

假设有一个程序要求用户提供一个文件名。这个文件应当包含数据值,文件的第一行包含数值的个数。其余各行包含具体的数据。一个典型的输入文件如下：

```
3
2.45
-2.5
2.89
```

程序运行时可能会出什么问题？有以下两个主要风险。

(1) 输入的文件可能不存在。

(2) 这个文件中的数据可能格式不正确。

设计的程序应该能检测和处理这些问题。当文件不存在时,Scanner 类构造方法会抛出一个 FileNotFoundException 异常,可以解决第一个问题。对第二个问题,当文件数据的格式不正确时应该抛出一个 BadDataException,这应该是一个定制的检查异常类。之所以使用一个检查异常类是因为数据文件的破坏超出了程序员的控制范围。

```
//DataAnalyzer.java
package chap05;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class DataAnalyzer {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        DataSetReader reader = new DataSetReader();

        boolean done = false;
```

```
        while(!done) {
            try {
                System.out.println("Please enter the file name: ");
                String filename = in.next();

                double[] data = reader.readFile(filename);
                double sum = 0;
                for(double d:data) {sum = sum + d;}
                System.out.println("The sum is " + sum);
                done = true;
            }catch(FileNotFoundException e) {
                System.out.println("File not found.");
            }catch(BadDataException ex) {
                System.out.println("Bad data: " + ex.getMessage());
            }catch(IOException ex) {
                ex.printStackTrace();
            }
        }
      }
}
//DataSetReader.java
package chap05;
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
/**
 * 从文件中读取一个数据集,文件必须有以下格式:
 * numberOfValues
 * value1
 * value2
 *....
 * @author Majun
 */
public class DataSetReader {
    private double[] data;
    public double[] readFile(String filename) throws IOException {
        File inFile = new File(filename);
        try (Scanner in = new Scanner(inFile)){
            readData(in);
            return data;
        }
      }
    private void readData(Scanner in) throws BadDataException {
        // TODO Auto-generated method stub
        if(!in.hasNextInt()) {
            throw new BadDataException("Length expected");
        }
        int numberOfValues = in.nextInt();
        data = new double[numberOfValues];
        for(int i = 0;i < numberOfValues;i++) {
            readValue(in,i);
```

```
            }
            if(in.hasNext()) {
                throw new BadDataException("End of file expected!");
            }
        }
        private void readValue(Scanner in, int i) throws BadDataException {
            // TODO Auto-generated method stub
            if(!in.hasNextDouble()) {
                throw new BadDataException("Data value expected!");
            }
            data[i] = in.nextDouble();
        }
    }
    //BadDataException.java
    package chap05;
    import java.io.IOException;
    public class BadDataException extends IOException {
       public BadDataException() {}
       public BadDataException(String message) {
        super(message);
       }
    }
```

在程序中,可以检查出两个潜在的错误,即文件可能不是以一个整数开始的,或者读取了所有值之后可能还有额外的数据。对应的处理流程如下所述。

(1) DataAnalyzer.main 调用 DataSetReader.readFile 方法。

(2) readFile 调用 readData 方法。

(3) readData 调用 readValue 方法。

(4) readValue 没有找到期望的值,抛出一个 BadDataException 异常对象。

(5) readValue 没有这个异常的处理器,立即终止。

(6) readData 没有这个异常的处理器,立即终止。

(7) readFile 没有这个异常的处理器,关闭 Scanner 对象后立即终止。

(8) DataAnalyzer.main 方法中有 BadDataException 处理器。这个处理器向用户打印一个消息,之后向用户提供另一个机会来输入一个文件名。注意计算值总和的语句。

【程序建模示例 5-2】 试建模一个年历程序,在字符界面输入年份,在屏幕上显示如图 5-2 所示的万年历,该程序的要点是格式控制,图形界面出现之前的早期计算机程序都是采用这种基于字符的输出控制完成程序设计的。

分析:如果直接使用输出语句在屏幕上进行格式控制并打印输出将非常困难,此处采用构造字符串数组的方式,将图 5-2 屏幕上的每一行看成是一行字符。首先在程序中抽象月份类,在类中抽象存储日历的二维数组,该月第一天的星期日历以及该月总的天数等。然后在构造方法中通过传来的参数构造字符串形式的日历数组,并编写一次在屏幕上打印两个月的方法。在主方法中通过 JDK 提供的 Calendar 类取得相关日历数据来构造 12 个月份对象,然后调用相应的打印方法即可,源代码如下。

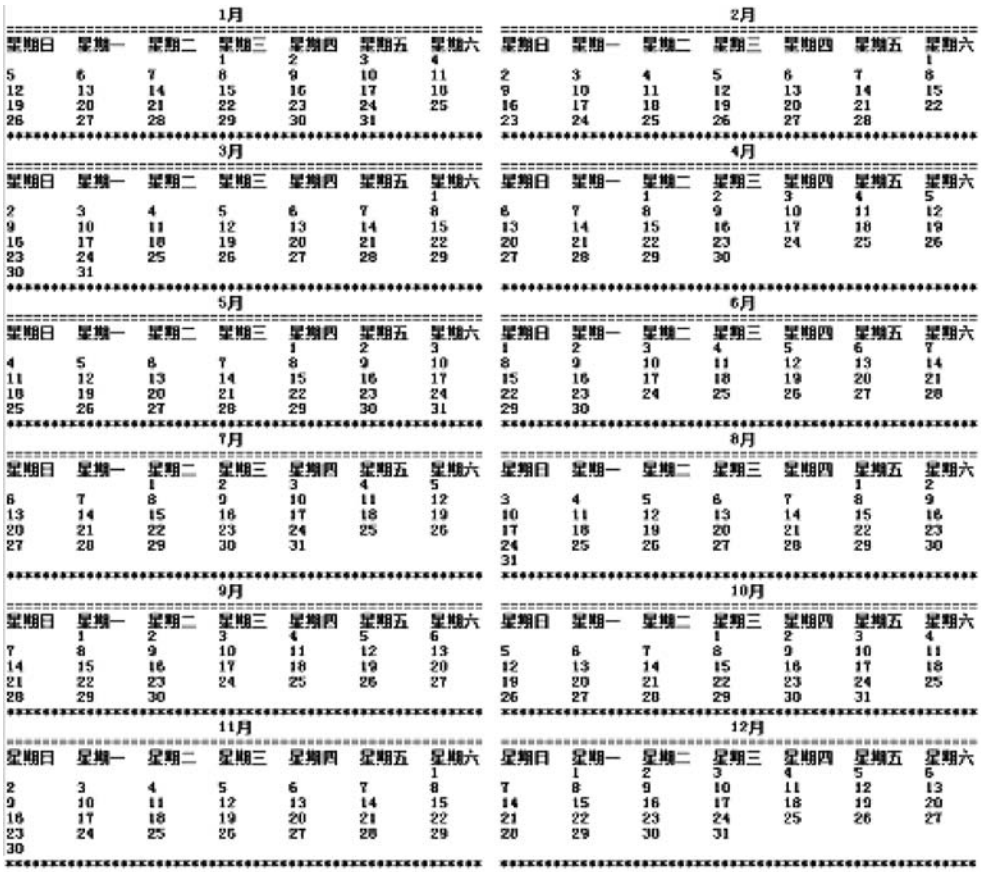

图 5-2　打印年历

```
import java.util. * ;
import java.text.DateFormatSymbols;
class MyMonth{
    private int month;
    private int start_of_week;
    private int days_in_month;
    public static String[ ] weekdayNames = new
            DateFormatSymbols().getShortWeekdays();
    private String[ ][ ] data = new String[7][8];
    public MyMonth(int m,int s,int d){
        month = m;
        days_in_month = d;
        start_of_week = s;
        for(int j = 1;j < 8;j++){
            data[0][j] = new String(MyMonth.weekdayNames[j]);
        }
        int days = 1,day_of_week = start_of_week,r = 1;
        do{
            data[r][day_of_week] = String.valueOf(days);
            days++;
```

```
                day_of_week++;
                if(day_of_week == 8) {
                    day_of_week = 1;
                    r++;
                }
            }while(days <= days_in_month);
        }
        public void display(){
            System.out.println("\t\t\t" + (month + 1) + "月");
System.out.println(" ========================================== ");
                for(int i = 0;i < 7;i++){
                    for(int j = 1;j < 8;j++){
                        if(data[i][j] == null)System.out.print("\t");
                        else System.out.print(data[i][j] + "\t");
                    }
                    System.out.println();
                 }
System.out.println(" ========================================== ");

        }
        public int getMonth(){return month + 1;}
        public int getDaysInMonth(){return days_in_month;}
        public String[][] getData(){return data;}
}
class MyCalendarTest{
        public static void main(String[] args){
            Calendar d =  Calendar.getInstance();
            Scanner keyin = new Scanner(System.in);
            System.out.print("请输入要显示年历的年份:");
            int year = keyin.nextInt();
            d.set(Calendar.YEAR,year);
            MyMonth[] mymonth = new MyMonth[12];
            for(int i = 0;i <= Calendar.DECEMBER;i++){
                d.set(Calendar.MONTH, i);
                d.set(Calendar.DAY_OF_MONTH, 1); //set d to start date of the month
                mymonth[i] = new
        MyMonth(i,d.get(Calendar.DAY_OF_WEEK),d.getActualMaximum(Calendar.DAY_OF_MONTH));
            }
            for(int i = 0;i <= Calendar.DECEMBER;i += 2){
                displayTwoMonth(mymonth[i],mymonth[i + 1]);
            }
        }
        public static void displayTwoMonth(MyMonth mon1,MyMonth mon2){
            System.out.print("\t\t\t" + mon1.getMonth() + "月");
            System.out.println("          \t\t\t\t\t\t  " + mon2.getMonth() + "月");
System.out.print(" ================================================ ");
System.out.println(" ================================================ ");
            String[][] d1 = mon1.getData();
            String[][] d2 = mon2.getData();
            for(int i = 0;i < 7;i++){
                 for(int j = 1;j < 8;j++){
```

```
                if(d1[i][j] == null)System.out.print("\t");
                else System.out.print(d1[i][j] + "\t");
            }
            for(int j = 1;j < 8;j++){
                if(d2[i][j] == null)System.out.print("\t");
                else System.out.print(d2[i][j] + "\t");
            }
            System.out.println();
        }
System.out.print(" ************************************************");
System.out.println(" ************************************************");
    }
}
```

5.8 本章小结

本章介绍了Java程序中异常处理的相关知识。正在运行的程序可能会遇到错误，甚至可能崩溃。在程序中提供异常处理代码可以最大限度地减少发生这种情况的概率。Java将异常分为重量级异常和轻量级异常，重量级异常通常超出了程序员的控制并通常导致应用程序崩溃。轻量级异常则可以被程序员捕获并及时处理。

Java的异常处理使用了5个关键字，即try、catch、throw、throws、finally。异常处理流程由try、catch和finally 3个代码块组成。其中，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。

Java提供了完整的多层次异常类库，Throwable类表示可抛出的异常类的父类，派生出的Error类表示严重的错误，无法单由程序来处理；而Exception类表示程序中出现的轻量级异常，是可由程序捕获并能处理的异常。在无法找到内置的异常类能用来充分说明异常的情况下，程序员可以提供自己的异常类。

Java的异常处理有两种方式，一种是在本方法中通过try－catch中处理，另一种通过throws来声明此方法有可能抛出异常，需要调用者来处理。

Java提供了日志API，它可以详细地记录程序的执行过程，可以分为多个记录级别。

最后介绍了在设计类时常用的指导原则，内聚、封装、清晰和完整是程序软件在设计中追求的主要目标。

第5章 习 题

一、单选题

1. 在Java中需要监测异常的代码放在(　　)。

A. try块　　B. catch块

C. finally块　　D. 以上选项都不正确

2. 在编写异常处理的程序段中，每个catch语句块都应该与(　　)语句块对应，如果捕

获成功则使用该语句块来启动相应的处理流程。

A. if-else　　B. switch　　C. try　　D. throw

3. 在 Java 的异常处理中,不管有没有异常,总要执行的代码块是(　　)。

A. try 块　　B. catch 块　　C. finally 块　　D. throws 块

4. 语句"System.out.println(args[i]);"有可能引发什么异常?(　　)

A. ArithmaticException　　B. ArrayIndexOutOfBoundsException

C. NumberFormatException　　D. FileNotFoundException

5. 下列程序编译或执行的结果是(　　)。

```
public static void main(String[]args){
    try{
        return;
        }finally{System.out.println("Finally"); }
    }
```

A. 程序正常运行,但不输出任何结果

B. 程序正常运行,并输出"Finally"

C. 编译能通过,但运行时会出现一个异常

D. 因为没有 catch 语句块,所以不能通过编译

6. Java 中用来抛出异常的关键字是(　　)。

A. try　　B. catch　　C. throw　　D. finally

7. 关于 Java 中的异常,下列说法正确的是(　　)。

A. 异常是一种对象

B. 一旦程序运行,异常将被创建

C. 为了保证程序运行速度,要尽量避免异常控制

D. 以上说法都不对

8. 下面哪一个类是所有异常类的父类?(　　)

A. Throwable　　B. Error　　C. Exception　　D. AWTError

9. 用 java MultiCatch 执行下列程序,说法正确的是(　　)。

```
1. class MultiCatch {
2.     public static void main(String args[])      {
3.         try {
4.             int a = args.length;
5.             int b = 42/a;
6.             int c[] = {1};
7.             c[42] = 99;
8.             System.out.println("b = " + b);
9.         }catch(ArithmeticException e) {
10.            System.out.println("除 0 异常: " + e);
11.        }catch(ArrayIndexOutOfBoundsException e) {
12.            System.out.println("数组越界异常: " + e);
13.        } catch(Exception e){}
14.    }
15.}
```

A. 程序没有输出

B. 程序在第 10 行出错

C. 程序将输出“除 0 异常：java.lang.ArithmeticException：/ by zero”

D. 程序将输出“数组越界异常：java.lang.ArrayIndexOutOfBoundsException：42”

10. 关于 Java 中日志的使用，下列说法正确的是(　　)。

A. Java 中不支持日志记录

B. Java 中默认的日志类 Logger 在 java.util 包中

C. Java 中默认的日志类 Logger 在 java.util.logging 包中

D. Java 日志只能记录严重错误

二、编程题

1. 输入一个以 24 小时为周期的时间，将其转换为 12 小时为周期的时间，例如 23：12，转换为 11：12PM。如果输入时间非法，则抛出一个异常。

2. 读取一个字符串，字符串中的单词以空格分隔，分离单词并将其转换为整型量，如果单词中出现非 0～9 的字符或者无法转换成整数类型，则抛出异常，然后继续处理后续单词。

3. 编写一个带 throws 子句的检查方法，用来检查字符串是否仅由英文字符和数字组成，如果是空字符串或包含非法字符则抛出异常对象，并编写测试方法进行测试。

4. 设计一个类，提供二进制和十进制的转换方法 bin2Dec(String binaryString)和 dec2Bin(String decimalString)，并定义一个名为 NumberStringException 的自定义异常类。在转换方法中，如果字符串不是二进制字符串或十进制字符串，则抛出 NumberStringExceptioin 异常对象。

三、简答题

1. 什么是异常？什么是重量级异常？什么是轻量级异常？

2. 简述 Java 的异常处理机制。

3. 简述 throw 和 throws 关键字的区别。

4. 如果在一个方法内出现了一个异常并抛出异常，方法内又没有异常处理代码块，将会发生什么情况？

5. 下列程序段的输出结果是什么？

```
public class test{
  public static void main(String args[]){
    int flag = 90;
    try{
      System.out.println("try - catch entered");
      if(flag > = 0)
      throw new Exception("The grade is A");
       System.out.println("Exception is: " + e.getMessage());
       }finally{
          System.out.println("after catch - block");
                }
  }
}
```

输出结果是什么？如果修改 flag 为 80，结果又是什么？

6. 在下面程序段中的合适位置加上 throws 关键字,使程序正确。

```
public static void procedure(int n){
    if(n < 0)   throw new Exception("negative number");
}
```

7. 下面程序抛出了一个“异常”并捕捉它,请在横线处填入适当内容完成程序。

```
class TrowsDemo {
  static void procedure() throws IllegalAccessExcepton  {
    System.out.println("inside procedure");
    throw _________ IllegalAccessException("demo");
  }
public static void main(String args[]) {
     try {
          procedure();
       }_____________________________ {
          System.out.println("捕获: " + e);
       }
    }
```

8. 假设在下面的程序段中,statement2 导致了一个异常,请回答下列问题。

```
try {
 statement1;
 statement2;
 statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}
statement4;
```

(1) 语句 statement3 会执行吗?

(2) 如果异常没有被捕获,语句 statement4 会执行吗?

(3) 如果异常在 catch 中被捕获成功,语句 statement4 还会执行吗?