



DeepL

订阅DeepL Pro以翻译大型文件。

欲了解更多信息，请访问www.DeepL.com/pro。

JC2002 Java 程序设计

第 6 天：例外情况（cs）

JC2002 Java 程序设计

第 5 天，第 1 课时：Java 中的异常处理

参考文献和学习对象

- 今天的课程主要基于 **Java：如何编程**、第 7 章和 **Java 简要介绍**
- 今天的课程结束后，您应该能够
 - 在 Java 代码中使用 try...catch 结构处理异常
 - 定义和使用自己的自定义异常

异常处理

- 异常是指在程序执行过程中发生的事件，它扰乱了程序指令的正常流程
- **异常处理程序**是可以处理异常的代码块
 - Java 允许将异常处理代码从正常代码中分离出来提高可读性
 - 在找到异常处理程序之前，异常会在调用栈中传播，因此开发人员可以选择在哪一级处理异常

- 对于如何编写和处理异常，每个组织都有自己的风格

异常处理

- 在 **Throwable** 类之上，还有 **Error** 和 **Exception** 子类
- 错误和异常又分为子类
 - 错误表示更严重的问题，通常无法解决运行时（内存不足、未找到类等）

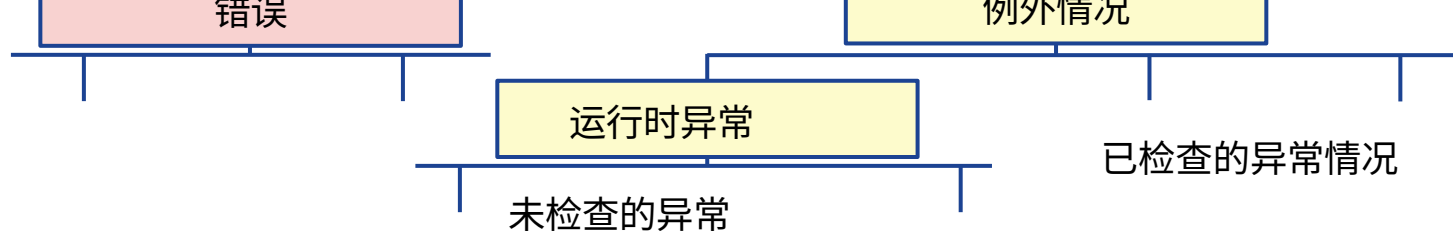
错误

例外情况

运行时异常

未检查的异常

已检查的异常情况



错误示例（无限递归）

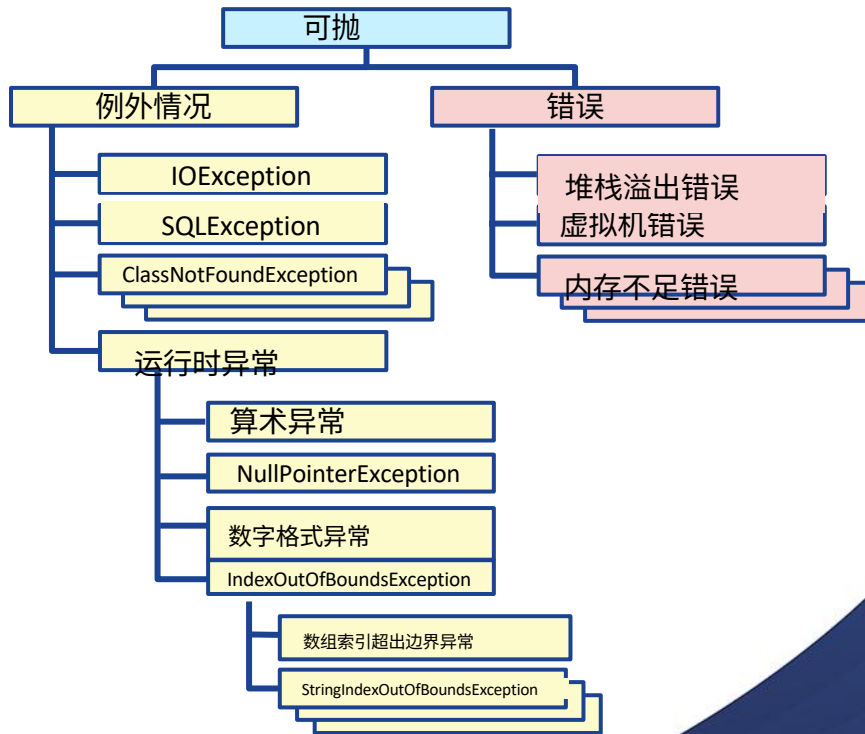
```
1 公共类 SimpleRecursion2 {
2      public static void recursiveLoop(int i, int max){
3          System.out.print(i + " ");
4          if(i < max) {
5              递归循环 (i, 最大);
6          }
7      }
8      public static void main(String[] args){
9          recursiveLoop(1,10);
10         System.out.println();
11     }
12 }
```

索引变量没有
堆栈溢出!

索引变量没有变化，所以最终堆栈溢出！

例外类别

- 预定义的异常几乎涵盖了实际 Java 程序中所有可能出现的错误情况
- 也可以通过子类化现有的类来创建自定义异常
 - 例外情况的等级并不固定



已选中和未选中的例外情况

- 被检查的豁免都是 **Exception** 的子类，除了未检查异常的 **RuntimeException** 子类
- 未检查的异常通常是编程问题造成的
- 许多程序员反对捕获未检查的异常，因为这些异常是无法预测的，如果发生了，就说明代码设计有问题，应该加以修正，防止错误发生
- 已检查的异常必须用 **throws** 关键字声明、否则编译器将返回错误信息

已检查异常示例

```
1  import java.io.*;
2
3  公共类 ExceptionTest1 {
4      public static void main(String args[]) {
5          FileInputStream inputStream = null;
6          inputStream = new FileInputStream("file.txt");
7
8          int m;
9          while ((m = inputStream.read()) != -1) {
10             System.out.print((char) m);
11         }
12     }
```

文件未找到异常

inputStream

```
$ javac ExceptionTest1.java
```

```
ExceptionTest1.java:5: error: unreported exception FileNotFoundException;
```

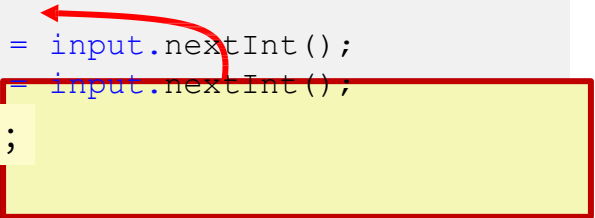

带抛掷的已检查异常示例

```
1  import java.io.*;
2  公共类 ExceptionTest2 {
3      public static void main(String args[]) throws IOException {
4          FileInputStream inputStream = null;
5          inputStream = new FileInputStream("file.txt");
6          int m;
7          while ((m = inputStream.read()) != -1) {
8              System.out.print((char) m)
9              ;
10         }
11         inputStream.close();
12     }
```

```
$ javac ExceptionTest2.java
```


未检查异常示例

```
1 import java.util.*;
2
3 公共类 ExceptionTest3 {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in)
6
7         ;
8         System.out.print("Give x: "); int x = input.nextInt();
9         System.out.print("Give y: "); int y = input.nextInt();
10        System.out.println("x / y = " + x/y);
```



```
$ javac ExceptionTest3.java
```

```
$ java ExceptionTest3
```

```
Give x: 10
```

```
给 y0
```

主线程 "java.lang.ArithmeticException "中出现异常: /为零

\$

使用 try ... catch 进行异常处理

- 默认情况下，程序会在出现异常时停止运行
- 不过，可以使用 **try...catch** 来处理异常。
结构

只有在有
是个例外

计划在此继续、
照例

```
try {  
    造成异常  
}  
catch (Exception e) {  
    如果出现异常，就这样做  
}  
    在此正常继续该计划
```

try ... catch 的示例

```
1  import java.util.*;
2
3  公共类 TryCatchTest {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in)
6
7          ;
8          System.out.print("Give x: "); int x = input.nextInt();
9          System.out.print("Give y: "); int y = input.nextInt();
10         try {
11             System.out.println("x / y = " + x/y);
12         }
13         catch (Exception e) {
14             System.out.println ("y 不可能为零! ");
15         }
16     }
17 }
```

The diagram illustrates the execution flow of the try-catch block. A red arrow originates from the division operation `x/y` on line 11, where a `RuntimeException` (specifically `ArithmeticException`) is thrown. The arrow points to the `catch (Exception e)` block on line 14, indicating that the exception is caught and handled. The `catch` block contains the message `"y 不可能为零! "`, which is printed to the console. The entire code is enclosed in a red-bordered box, and the `try` and `catch` blocks are highlighted in yellow.

```
} }
```

给 x: 10

给 y0

y 不可能为零!

异常处理程序

这会抛出一个异常
(除以零)，如果 $y=0$!

使用关键字抛出


- 在前面的例子中，异常是由 JVM
- 您也可以在方法中抛出一个新的异常对象

```
public class Person {  
    受保护的 int age  
    public void setAge(int age) {  
        if (age < 0) {  
            抛出新的 IllegalArgumentException("Age can't be negative!");  
        }  
        this.age = age;  
    }  
}
```

使用关键字 `finally`

- `finally` 代码块通常用来释放 `try` 代码块中获取的资源（如数据库连接、打开的文件）。
- `finally` 代码块保证执行，除非 `try` 代码块或 `catch` 代码块会调用 `System.exit()`，从而停止 Java 解释器。
- 避免将可能产生异常的代码放在最终代码块中
- 如果需要这样的代码，请将代码括在 `try ... catch` 块中

处理多个异常



您可以在 try 代码块后使用多个 catch 代码块来捕获不同的异常

嵌套的 try ... catch 块

- 可以使用 *嵌套的* try...catch 块
 - 通常最好避免和尝试寻找其他解决方案!

```
1 import java.util.*;
2
3 公共类 TryCatchTest2 { 公共静态 void
4     divide() {
5         System.out.print("Give x ");
6         ...
7         int x = input.nextInt();
8         System.out.print("Give y ");
9         ...
10        int y = input.nextInt();
11        System.out.println("x / y = " + x/y);
12    }
```

```
11 public static void main(String[] args) {
12     try {
13         divide();
14     }
15     catch(Exception e1) {
16         System.out.println("y can't be zero!");
17         System.out.println("Try again.");
18     }
19     try {
20         divide();
21     }
22     catch(Exception e2) {
23         System.out.println("y 还是不可能为零!");
24         System.out.println("Try again.");
25     }
26 }
27 }
```

嵌套的 try...catch

给 x: 6
给 y0
y 不可能为零! 再试一
次
Give x: 7
给 y0

有问题或意见？

JC2002 Java 程序设计

第 6 天，第 2 课时：用户定义的异常情况

为什么是用户自定义异常？

- 内置异常涵盖了编程中几乎所有的常规异常类型
- 不过，在某些情况下，自定义例外也是有益的：
 - 捕获现有 Java 异常的特定子集
 - 处理与程序错误无关的 "业务逻辑异常"、
但例如，应用程序特有的数据错误
 - 自定义异常允许在程序的特定级别进行处理
- 用户定义的异常只需继承现有异常即可创建

用户定义异常示例 (1)

```
1  import java.util.*;
2  类 IntOverflowException 扩展异常 {
3      public IntOverflowException(String str) {
4          super(str);
5      }
6  }
7  公共类 TestCustomException {
8      静态 int fact(int x)
9          throws IntOverflowException {
10         int y=1;
11         for(int i=1; i<=x; y *= i++) {
12             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13                 抛出新的 IntOverflowException ("整数溢出");
14             }
15         }
16         return y;
17     }
18     静态 int computeC(int n, int r)
```

```
23     public static void main(String args[])
24     {
25         try {
26             computeC(50,10); // 计算 C(n,k)
27         }
28         catch (IntOverflowException ex) {
29             System.out.println(ex.getMessage());
30         }
31     }
32     System.out.println("continue...");
33 }
```

```
19         throws IntOverflowException {  
20     int res = fact(n)/(fact(r)*fact(n-r));  
21     return res;  
22 }
```

用户定义异常示例 (2)

```
1  import java.util.*;
2  类 IntOverflowException 扩展异常 {
3      public IntOverflowException(String str) {
4          super(str);
5      }
6  }
7  公共类 TestCustomException {
8      静态 int fact(int x)
9          throws IntOverflowException {
10
11          int y=1;
12          for(int i=1; i<=x; y *= i++) {
13              if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
14                  抛出新的 IntOverflowException ("整数溢出");
15              }
16          }
17          return y;
18      }
19      静态 int computeC(int n, int r)
```

```
23  public static void main(String args[])
24  {
25      try {
26          computeC(50,10); // 计算 C(n,k)
27      }
28      catch (IntOverflowException ex) {
29          System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32  }
33  }
```

```
19          throws IntOverflowException {
20              int res = fact(n)/(fact(r)*fact(n-r));
21              return res;
22          }
```

用户自定义异常。注意
构造函数和调用 `super()`
并不是必须的，但它有
助于实现默认功能。

用户定义异常示例 (3)

```
1  import java.util.*;
2
3  class IntOverflowException extends Exception {
4      public IntOverflowException(String str) {
5          super(str);
6      }
7  }
8  公共类 TestCustomException { 静态 int
9      fact(int x)
10         throws IntOverflowException {
11
12         int y=1;
13         for(int i=1; i<=x; y *= i++) {
14             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
15                 抛出新的 IntOverflowException ("整数溢出")
16             }
17         }
18         return y;
19     }
20     静态 int computeC(int n, int r)
21         throws IntOverflowException {
22         int res = fact(n)/(fact(r)*fact(n-r));
23         return res;
24     }
```

```
23  public static void main(String args[])
24  {
25      try {
26          computeC(50,10); // 计算 C(n,k)
27      }
28      catch (IntOverflowException ex) {
29          System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32  }
```

您需要使用关键字 `throws` 来指明哪些方法会抛出自定义异常。或者，您也可以从 `RuntimeException` 继承异常。

用户定义异常示例 (4)

```
1  import java.util.*;
2
3  class IntOverflowException extends Exception {
4      public IntOverflowException(String str) {
5          super(str);
6      }
7  }
8  公共类 TestCustomException { 静态 int
9      fact(int x)
10         throws IntOverflowException {
11
12         int y=1;
13         for(int i=1; i<=x; y *= i++) {
14             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) { 引发新
15                 的 IntOverflowException("integer overflow");
16             }
17         }
18     }
19     return y;
20 }
21
22 静态 int computeC(int n, int r)
        throws IntOverflowException {
```


```
23  public static void main(String args[])
24  {
25      try {
26          computeC(50,10); // 计算 C(n,k)
27      }
28      catch (IntOverflowException ex) {
29          System.out.println(ex.getMessage());
30      }
31      System.out.println("continue...");
32  }
```

如果变量 y (int) 将在下一轮中溢出，则抛出异常

```
int res = fact(n)/(fact(r)*fact(n-r));  
return res;  
}
```

用户定义异常示例 (5)

```
23     public static void main(String args[])
24     {
25         try {
26             computeC(50,10); // 计算 C(n,k)
27         }
28         catch (IntOverflowException ex) {
29             System.out.println(ex.getMessage());
30         }
31         ;
32     }
33 }
System.out.println("continue...");
}
```



```

1  import java.util.*;
2
3  class IntOverflowException extends Exception {
4      public IntOverflowException(String str) {
5          super(str);
6      }
7  }
8
9  公共类 TestCustomException { 静态 int
10     fact(int x)
11         throws IntOverflowException {
12         int y=1;
13         for(int i=1; i<=x; y *= i++) {
14             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
15                 抛出新的 IntOverflowException ("整数溢出");
16             }
17         }
18         return y;
19     }
20
21     静态 int computeC(int n, int r)
22         throws IntOverflowException {
23         int res = fact(n)/(fact(r)*fact(n-r));
24         return res;
25     }
26 }

```

我们的 try...catch 块。根据经验，我们知道计算 $C(50,10)$ 会导致 int 溢出。

用户定义异常示例 (6)

```
1  import java.util.*;
2  类 IntOverflowException 扩展异常 {
3      public IntOverflowException(String str) {
4          super(str);
5      }
6  }
7  公共类 TestCustomException {
8      静态 int fact(int x)
9          throws IntOverflowException {
10         int y=1;
11         for(int i=1; i<=x; y *= i++) {
12             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
13                 抛出新的 IntOverflowException ("整数溢出");
14             }
15         }
16         return y;
17     }
18     静态 int computeC(int n, int r)
```

```
23     public static void main(String args[])
24     {
25         try { computeC(50,10)
26             ;
27         }
28         catch (IntOverflowException ex) {
29             System.out.println(ex.getMessage());
30         }
31         System.out.println("continue...");
32     }
33 }
```

```
$ java TestCustomException
整数溢出 继续...
$
```

```
19         throws IntOverflowException {  
20     int res = fact(n)/(fact(r)*fact(n-r));  
21     return res;  
22 }
```

自定义未选中异常示例

```
1 import java.util.*;
2 类 IntOverflowException 扩展 RuntimeException {
3 }
4 公共类 TestCustomException2 {
5     static int fact(int x) {
6         int y=1;
7         for(int i=1; i<=x; y *= i++) {
8             if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
9                 抛出新的 IntOverflowException();
10            }
11        }
12        return y;
13    }
14    静态 int computeC(int n, int r) {
```

无构造函数的简化类继承自
RuntimeException

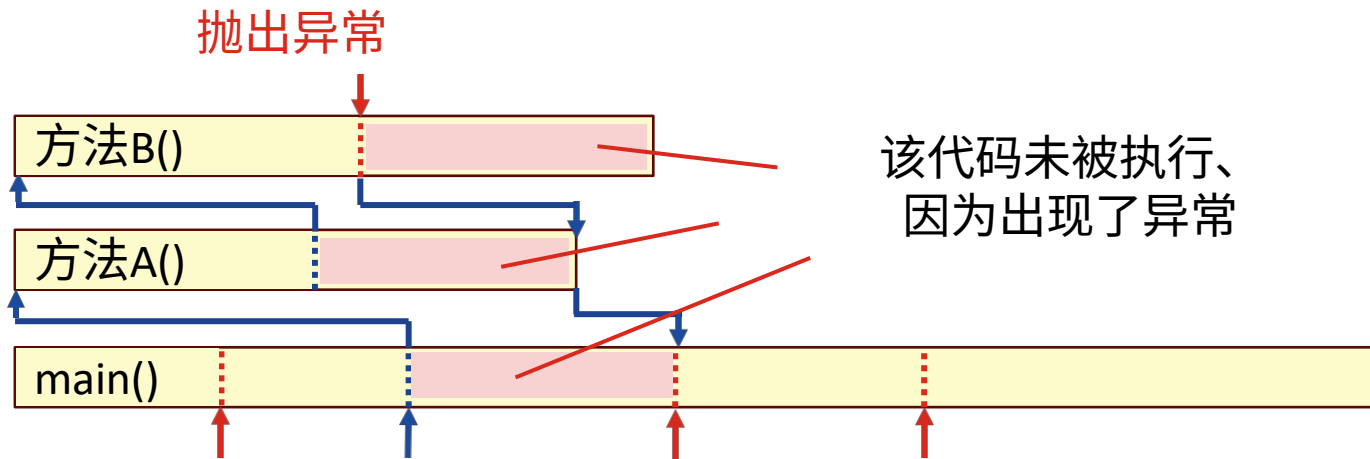
```
18     public static void main(String args[])
19     {
20         try { computeC(50,10)
21             ;
22         }
23         catch (IntOverflowException ex) {
24             System.out.println("int overflow...");
25         }
26         System.out.println("continue...");
27     }
28 }
```

```
$ java TestCustomException2
int overflow
继续...
$
```

未选中的异常不需要在 throws
throws

由于出现异常，代码被忽略

- 请注意，当异常抛出时，它会在调用堆栈中传播，直到异常得到处理：有些数据可能没有被正确初始化！



尝试
开始

方法A
援引

捕获块
开始

捕获块
结束

未分配有效数据的变量

```
1 import java.util.*;
2
3 类 IntOverflowException 扩展 RuntimeException {
4
5 }
6
7 公共类 TestCustomException3 { 静态 int
8     fact(int x) {
9         int y=1;
10        for(int i=1; i<=x; y *= i++) {
11            if(i<x && (long)y*(long)i>Integer.MAX_VALUE) {
12                抛出新的 IntOverflowException();
13            }
14        }
15        return y;
16    }
17 }
```

```
静态 int computeC(int n, int r) {
    int res = fact(n) / (fact(r) * fact(n-r));
    return res;
}
```

因为出现了异常、
int C 没有有效值！

```
18 静态 int C;
19 public static void main(String args[])
20 {
21     try {
22         C = computeC (10,5) ;
23     }
24     catch (IntOverflowException ex) {
25         System.out.println("int overflow")
26     }
27     ;
28 }
29 System.out.println("C = " + C);
```

```
$ java TestCustomException3
int overflow
C = 0
$
```

通用异常处理程序的注意事项

- 捕捉 Exception 超类的通用异常处理程序可能会提供有关潜在问题的误导信息
 - 你不应该期望总是为同理!
- 有时，在代码捕获标准异常后会抛出自定义异常
 - 您应该提供一个构造函数，以保留标准例外的错误

被误解的例外情况

```
1 import java.util.*;
2
3 公共类 TryCatchTest2 {
4
5     公共静态 void divide() {
6         Scanner input = new Scanner(System.in);
7
8         System.out.print("Give x: ");
9         int x = input.nextInt();
10        System.out.print("Give y: ");
11        int y = input.nextInt();
12        System.out.println("x / y = " + x/y);
13    }
14    public static void main(String[] args) {
15        try {
16            divide();
17        }
18        catch (Exception e1) {
19            System.out.println("y 不能为零! ");
20        }
21    }
22 }
```

给 x: 5 给 y:

abc

y 不可能为零!

在这种情况下，输入不是数字，抛出的异常是输入不匹配异常（`InputMismatchException`），而不是算术异常（`ArithmeticException`）。

带原因的自定义异常示例 (1)

```
1 import java.util.*;
2 class DivisionException extends Exception {
3     public DivisionException(String msg、
4                             Throwable cause){
5         super(msg + cause.toString());
6     }
7 }
```

```
8 公共类 TestCustomException4 {
9     公共静态 void divide()
10         throws DivisionException {
11     try {
12         Scanner input = new Scanner(System.in)
13         ;
14         System.out.print("Give x: "); int x = input.nextInt();
15         System.out.print("Give y: "); int y = input.nextInt();
16         System.out.println("x / y = " + x/y);
17     }
```

```
21 public static void main(String[] args) {
22     try {
23         divide();
24     }
25     catch(DivisionException e) {
26         System.out.println(e.getMessage())
27     }
28     ;
29 }
```

```
19     catch(Exception e) {  
20     }     throw new DivisionException("division() failed due to ", e);  
        }
```

带原因的自定义异常示例 (2)

```
1 import java.util.*;
2 class DivisionException extends Exception {
3     public DivisionException(String msg、
4                               Throwable cause){
5         super(msg + cause.toString());
6     }
7 }
8 }
```

```
9 公共类 TestCustomException4 { 公共静态
```

```
10 void divide()
11     throws
```

```
12 try {
```

```
13     扫描仪输入端
```

```
14     System.out.p
```

```
15     System.out.p
```

```
16     System.out.p
```

```
17 }
```

```
18 catch (Exceptio
```

```
19     throw new Di
```

```
20 }
```

```
21 public static void main(String[] args) {
22     try {
23         divide();
24     }
25     catch (DivisionException e) {
26         System.out.println(e.getMessage());
27     }
28 }
29 }
```

定义构造函数，保留异常原因（捕获的原始一般异常）

```
);
);
```

```
o ", e)
;
```


带原因的自定义异常示例 (3)

```
1 import java.util.*;
```

捕捉一般异常并抛出自定义
(业务) 异常

```
    删除 {  
        msg、  
        le cause){
```

```
8 公共类 TestCustomException4 { 公共静态  
9  
10 void divide()  
11     throws DivisionException {  
12     try {  
13         Scanner input = new Scanner(System.in);  
14         System.out.print("Give x: "); int x = input.nextInt();  
15         System.out.print("Give y: "); int y = input.nextInt();  
16         System.out.println("x / y = " + x/y);  
17     }  
18     catch (Exception e) {  
19  
20
```

```
21 public static void main(String[] args) {  
22     try {  
23         divide();  
24     }  
25     catch (DivisionException e) {  
26         System.out.println(e.getMessage());  
27     }  
28 }  
29
```

```
    throw new DivisionException("division() failed due to ", e);  
}  
}
```

带原因的自定义异常示例 (4)

```
1 import java.util.*;
2
3 class DivisionException extends Exception {
4     public DivisionException(String msg、
5                               Throwable cause){
6         super(msg + cause.toString());
7     }
8 }
9
10 公共类 TestCustomException4 { 公共静态
11     void divide()
12         throws DivisionException {
13         try {
14             Scanner input = new Scanner(System.in);
15             System.out.print("Give x: "); int x =
16             input.nextInt(); System.out.print("Give y: "); int y =
17             input.nextInt();
18             System.out.println("x / y = " + x/y);
19         }
20         catch (Exception e) {
21             throw new DivisionException("division() failed due to ", e);
22         }
23     }
24 }
```

```
21 public static void main(String[] args) {
22     try {
23         divide();
24     }
25     catch (DivisionException e) {
26         System.out.println(e.getMessage());
27     }
28 }
29 }
```

捕捉自定义（业务）异常
并打印出异常的根本原因

}

带原因的自定义异常示例 (5)

```
1  import java.util.*;
2  class DivisionException extends Exception {
3      public DivisionException(String msg、
4                              Throwable cause){
5          super(msg + cause.toString());
6      }
7  }
8  公共类 TestCustomException4 {
9      公共静态 void divide()
10         throws DivisionException {
11      try {
12          Scanner input = new Scanner(System.in);
13          System.out.print("Give x: "); int x =
14          input.nextInt();
15          System.out.print("Give y: "); int y =
16          input.nextInt();
17          if (y == 0)
18              throw new DivisionException("除数不能为0");
19          System.out.println("x / y = " + x/y);
20      } catch (DivisionException e) {
21          System.out.println(e.getMessage());
22      }
23  }
```

```
21  public static void main(String[] args) {
22      try {
23          divide();
24      }
25      catch (DivisionException e) {
26          System.out.println(e.getMessage());
27      }
28  }
```

```
$ java CustomExceptionTest4
Give x: 56
给 y0
由于 java.lang.ArithmeticException 导致除法失败: / 除以 0
```

```
20    } catch (Exception e) {  
        抛出新的除法异常 ("除法 () 失败"  
    }  
}
```

```
$ java CustomExceptionTest4  
Give x: abc  
除法因 java.util.InputMismatchException 异常而失败
```

摘要

- 在 Java 中，错误和其他特殊情况会抛出异常
 - 异常可通过 try...catch 结构处理
 - 已检查的异常必须用关键字 throws 或在异常处理程序中处理、
 - 未检查的异常通常是由编程错误引起的，不需要处理（相反，应该修复代码！）。
- 还可定义和处理用户定义的（自定义）异常情况

- 有助于在代码的同一部分执行不同的异常处理程序

有问题或意见？