第九讲：过程抽象与异常处理

OO2018课程组

北京航空航天大学计算机学院

第三单元

- 规格化的面向对象设计方法
  - 理解规格的概念
  - 掌握方法的规格及其设计方法
  - 掌握类的规格及其设计方法
  - 掌握继承层次下类规格之间的关系
  - 掌握线程类的规格
- 内容主要来自于参考教材
  - 课后一定要看书
  - 我们做了一定的研究扩展

内容提要

- 什么是抽象
- 抽象类别
- 过程抽象的定义
- 依据抽象规格的方法实现
- 异常处理
- 异常类型
- 异常处理方式

什么是抽象

- 抽象是一种过程
  - 忽略个体的具体差异(不感兴趣)，把诸多个体的共性特征(感兴趣)抽取出来的过程
  - 把一组个体的特征进行归纳的过程
  - 例：数据结构、面向对象、数学分析有哪些共性特征？
- 抽象是一种结果
  - 是对抽取出的共性特征的表示结果
  - 学位课是什么？
  - 有序数组如何表示？

什么是抽象

- 面向对象方法是一个抽象过程
  - 提取类（概括了一系列对象共性特征后形成的类型）
  - 定义类的操作(概括了一系列对象的行为能力)
  - 定义类的属性(概括了一系列对象的状态空间)
- 面向对象语言(如Java)提供了实现层次的抽象表示
  - 类、接口、继承与实现、多态、…
- 向上是抽象过程
- 向下是具体化过程

抽象类别

- 面向对象语言提供了结构抽象和行为抽象
  - 类从结构上把数据和操作聚合在一起
  - 接口把一组类的公共行为抽象出来
  - 通过继承、实现可形成不同的抽象层次
  - 参数化、控制流程、返回值等提供了行为描述抽象
- 然而，面向对象语言没有提供规格抽象(specification abstraction)

抽象类别

- 什么是规格？
  - 对一个方法/类/程序的外部**可感知**行为(语义)的抽象表示
  - 内部细节无需在规格中表示
  - 规格把设计与实现有效分离
- 过程抽象
  - 过程(即类中的方法或者接口)规格提炼的结果
  - 过程抽象是对方法进行实现的依据
- 数据规格
  - 数据的组成及其生命周期内应该满足的约束
- 类规格
  - 数据规格+方法规格+迭代器规格

提取规格抽象的必要性

```
// search upwards
found = false;
for (int i = 0; i < a.length; i++)
   if (a[i] == e) {
      z = i;
      found = true;
   }
```

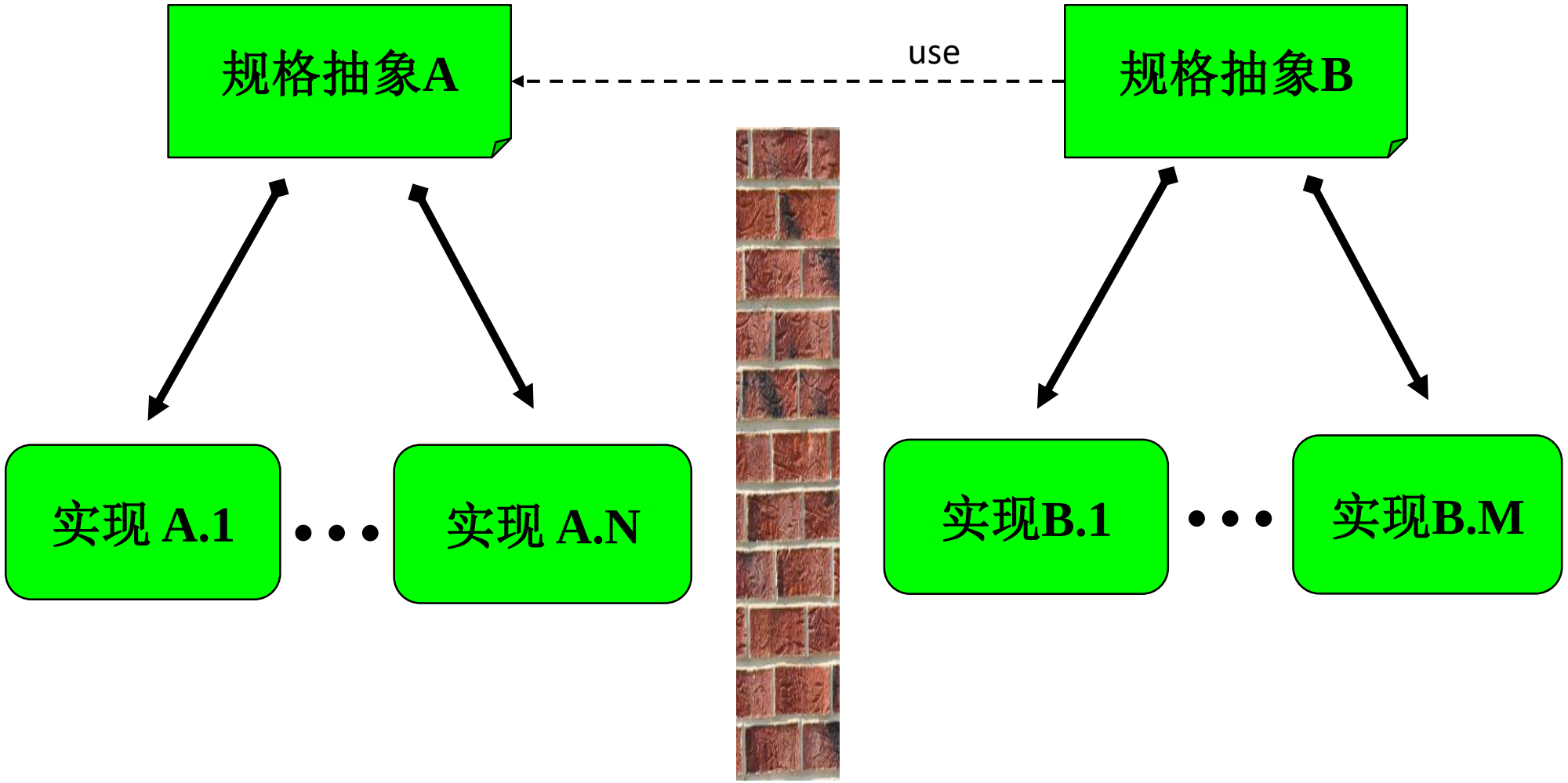
```
// search downwards
found = false;
for (int i = a.length-1; i >= 0; i--)
   if (a[i] == e) {
      z = i;
      found = true;
   }
```

比较这两个搜索程序的共同点和不同点

使用规格抽象获得的好处

- 局部性
  - 针对一个规格抽象的实现与针对其他规格抽象的实现无关，相互之间不会产生影响
- 可修改性
  - 当修改一个规格抽象的实现时，不需要对使用该抽象的其他任何规格抽象及其实现进行调整

规格抽象



如何表示规格抽象

- 有很多研究，比如使用形式化语言
- 这里我们通过Javadoc风格的结构化注释来表示
  - 重点是在程序设计时首先明确规格，然后实现代码
- 方法规格抽象
  - 执行前对输入或系统状态的要求----前置条件(precondition)
  - 执行后返回结果或系统状态应该满足的约束----后置条件(postcondition)
- 数据规格抽象
  - 数据状态变化应该满足的要求----不变式(invariant)
- 迭代规格抽象
  - 迭代遍历集合中元素时满足的要求
- 其他规格抽象？
  - 如线程？

过程规格的组成

标题

后置条件

```
public static int sortedSearch (int[]a, int x)
/**@Requires:  a is sorted in ascending order
   @Modifies: none
   @Effects:   if x is in a returns an index where
               x is stored; otherwise, returns -1
*/
```

前置条件

副作用

规格描述模板：标题+对结果的描述

标题：定义了过程的形式。f: input → output

前置条件(Requires)：定义了过程对输入的约束要求

副作用(Modifies)：过程在执行过程中对Input的修改

后置条件(Effects)：定义了过程在所有未被Requires排除的输入下给出的执行效果

过程规格的组成

```
public static int sortedSearch (int[]a, int x)
/**@Requires: a is sorted in ascending order
   @Modifies: none
   @Effects:  if x is in a returns an index where
              x is stored; otherwise, returns -1
*/
```

这个实现是否满足这个规格？

```
found = false;
for(int i=a.length-1;i>=0;i--){
   if(a[i] == x){
        z = i;
        found = true;
   }
}
if(found) return z;
else return -1;
```

类规格的组成

```
public class Arrays {
/** @OVERVIEW: This class provides a number of standalone procedures that are useful for manipulating arrays of ints.
*/
public static int search (int [ ] a, int x)
/** @EFFECTS ： If x is in a, returns an index where x is stored; otherwise, returns -1.
*/
public static int search
/** @REQUIRES: a is
   @EFFECTS： If x is
*/
public static void sort (
/** @MODIFIES: a
   @EFFECTS: Rearranges the elements of a into ascending order, e.g., If a  = [3, 1, 6, 1], before  the call, on return
                a_post = [1, 1, 3, 6].
*/
```

```
public static int[ ] boundedArray (int [] a, int n)
/** @EFFECTS: Returns a new array containing the elements of a
in the order they appear in a except that any elements of a that are
greater than n are replaced by n.
*/
```

基于这样的规格描述模板，一个类的设计目标和行为能力得到了准确定义

使用隐含输入输出的过程规格

- 有些过程除了使用形式参数输入，还使用隐含的输入输出
  - 全局变量
  - 系统设备输入(文件、流)
  - 用户交互输入(事件数据)
- 隐含输入如果不在规格中明确，就会破坏规格的两个特性
  - 局部性
  - 可修改性

使用隐含输入输出的过程规格

```
public static void copyLine( )
/**@REQUIRES: System. in contains a line of text
    @MODIFIES: System. in and System. out
    @EFFECTS: A line of text from System.in would be consumed and the cursor would be
moved to the end of the line, and the line of text consumed would be written on System.out.
*/
```

过程规格抽象

- 过程完成从输入到输出的转换计算
  - 可能会修改输入参数
  - 可能会产生副作用
  - 可能会显式/隐式返回结果
- 过程的实现应该不受在何处使用的影响
- 过程规格抽象
  - 描述过程执行后获得的效果（后置条件），而不是具体怎么做
  - 不同的实现只是在细节方面有差异
  - 一种实现可以被另一种实现替换

过程规格设计

- 课堂练习
  - 二叉排序树BinarySortedTree，任意一个节点node，node中存储一个整数，满足node.left.value <= node.value <=node.right.value，node.left和node.right指向node的左子节点和右子节点
    - public void insert(BinarySortedTree tree, int x): 把x插入到树中，保持树的有序性
    - public int remove(BinarySortedTree tree, int x): 如果树中节点的值为x，则从树中移除该节点，返回1；否则返回-1。保持树的有序性
  - 请使用10分钟写出这两个方法的规格抽象

过程规格设计

public void insert(BinarySortedTree tree, int x)
/** @Requires: tree is sorted in ascending order from left to right
    @Modifies: tree
    @Effects: if tree is null, an empty tree created. A new node x_node containing x created inserted into the right place such that x_node.value =x, and if x_node.left!=null, x_nodel.left.value <=x, and if x_node.right !=null, x_node.right.value >= x
*/

public int remove(BinarySortedTree tree, int x)
/** @Requires: tree is sorted in ascending order from left to right
    @Modifies: tree
    @Effects: if x in tree, the node containing x removed, returns 1; otherwise returns -1. And tree is sorted in ascending order.
*/

依据过程规格抽象的实现

- 给定一个方法的规格抽象
  - 只能对Modifies子句中明确的变量进行修改
  - 如果输入满足Requires的要求，则输出必须满足Effects中的要求

```
public static int sortedSearch (int[]a, int x)
/** @Requires: a is sorted in ascending order
    @Modifies: none
    @Effects:  if x is in a returns an index where x is stored;
               otherwise, returns -1
*/
if(a == null) return -1;
for (int i=0;i<a.length;i++)
     if(a[i] == x) return i; else if(a[i]>x)return -1;
return -1;
}
```

Think as a user

依据过程规格抽象的实现

```
public static void removeDupls (Vector v)
/** @Requires: All elements of v are not null.
    @Modifies: v
    @Effects:  duplicate elements are removed from v.
*/
   if (v == null) return;
   for (int i = 0; i < v.size( ); i++) {
      Object x = v.get(i);
      int j = i + 1;
      // remove all dupls of x from the rest of v
      while (j < v.size( ))
         if (!x.equals(v.get(j))) j++;
         else { v.set(j, v.lastElement( ));
            v.remove(v.size( )-1); }
   }
}
```

请实现该方法

多线程设计下的过程规格

- 在多线程设计情况下，任何一个类都可能被多个线程共享访问
- 需要在方法规格层次明确相应的要求
  - Requires：如果Modifies中涉及类成员属性变量或全局变量，则需要明确是否让调用者对该方法的执行进行锁控制，如果不明确则表示调用者无需操心
  - Modifies：与使用它的线程无关
  - Effects：如果Modifies中涉及类成员属性变量或全局变量，且Requires中未对调用者做出锁控制要求，则需要明确是否thread-safe。

该对使用者要求多少？

- 规格抽象中的Requires本质上是对使用者提出要求
- 如果提出了具体要求，等同于限定了一个方法的适用范围
  - 部分适用过程(partial procedure)：部分适用的过程
- 如果没有任何具体要求，等同于相应方法在任何情况下都适用
  - 全局适用过程(total procedure)：全部适用的过程
- 从设计角度来看，一个规格应尽可能减少对使用者的要求
  - 一个规格对使用者的约束越少，相应方法就越易于使用
  - 如果不要求使用者进行锁控制，则就是一个thread-safe的方法

规格中蕴含的不确定性

- 对于public static int searchSorted(int[] a, int x),如果a中包括不止一个x会怎么样？
  - 从0开始的线性查找会给出最后一个x的位置
  - 从a.length-1开始的线性查找会给出第一个x的位置
  - 二分查找呢？
- 对于public static void removeDupls(Vector v)
  - 如果v是有序的怎么办？
  - 如果用户不希望v中元素次序被改变怎么办？

过程规格的特性

- 最少限度性
  - 只强调使用者关心的要求
- 避免不确定性
  - 如果对于给定的输入，规格产生的结果依赖于具体实现
  - 设计者应避免产生不确定行为
- 一般性
  - 如果规格A比规格B能处理更多可能输入，则规格A更具有一般性
  - 搜索方法的两个规格：仅支持在定长数组中搜索 vs 支持在不定长数组中搜索
- 简单性
  - 规格应该保持简单，一个过程不应该做太多事情

部分适用过程隐藏有风险

- 虽然部分适用过程对使用者的要求具有合理性，但是无法保证使用者总是清楚有相应的要求，以及如何满足相应的要求
- 解决办法
  - 要么在方法中对输入进行检查，如果不满足，返回特定的值告知使用者进行处理
    - 弊端1：大量的冗余检查，降低性能
    - 弊端2：使用者未必会对特殊的返回值进行处理，降低程序的鲁棒性
  - 要么使之成为全局适用过程
    - 不进行额外检查，如何处理不满足要求的输入情况？

部分适用过程隐藏有风险

- 我们需要的处理手段
  - 能够识别出输入是否满足要求
  - 即使返回值都在规格规定的Effects范围内，也能够提醒使用者出现了异常情况
  - 为了避免使用者疏忽检查返回值，要求这种‘提醒’采用不同于控制流调用返回的方式
- 异常机制

带有异常终止的过程规格

visibility type procedure (args) throws <list of exception_types>

/**@Requires …

@Modifies …:须明确当抛出异常时会产生什么副作用

@Effects …:当输入满足Requires条件时的结果；当输入不满足时抛出的异常

*/

```
public static void addMax (Vector v, Integer x) throws
NullPointerException, NotSmallException
/** @Requires: All elements of v are Integers.
    @Modifies: v
    @Effects:  If v is null throws NullPointerException; if v contains an
element larger than x throws NotSmallException;else v=\old (v) +{x}.
*/
```

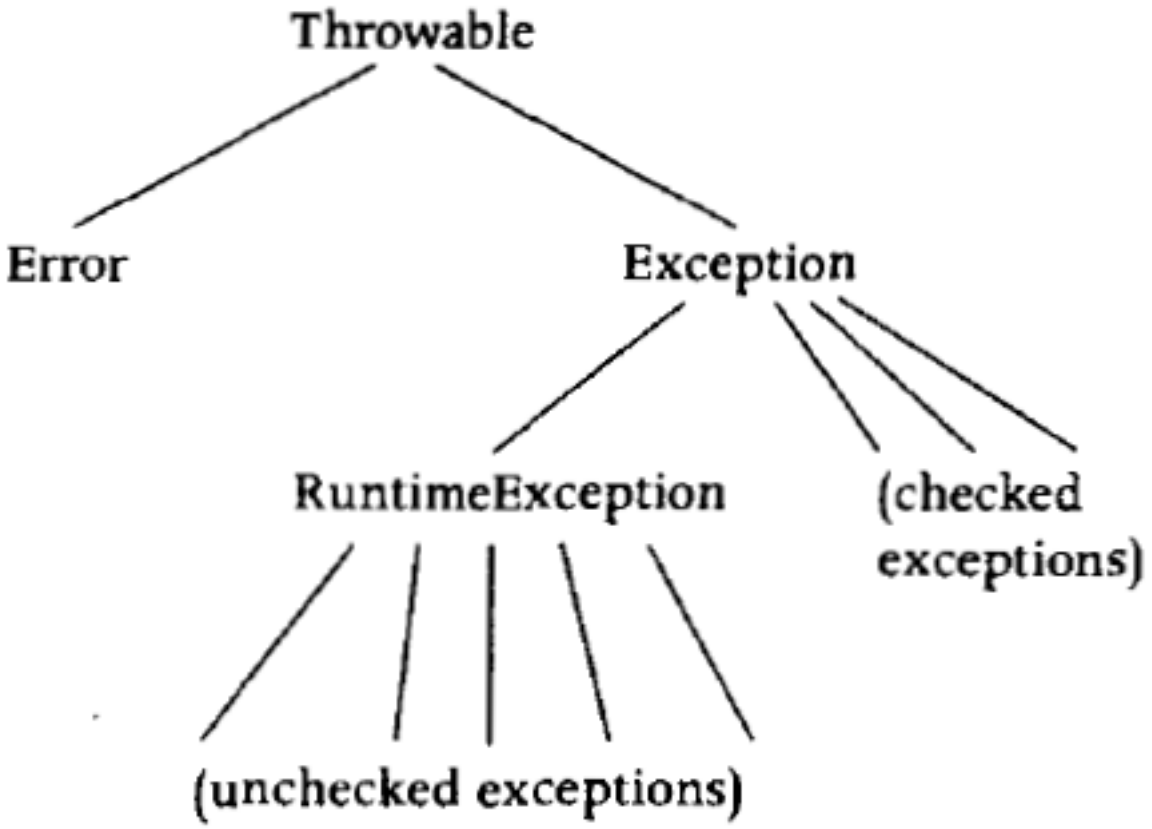
带有异常终止的过程规格

- 学生成绩管理Student类
  - public boolean courseSelected(Course c)
  - public float getCourseMark(Course c)
  - public boolean selectCourse (Course c)
  - public boolean unselectCourse (Course c)
  - public boolean recordMark(Course c, float m)

```
public class Student {
    public boolean courseSelected(Course c){…}
    public StuKind getStuKind(){…}
    public String getName(){…}
    public float getTotalCredit(){…}
    public float getCourseMark(Course c){…}
    public boolean selectCourse(Course c){…}
    public boolean unselectCourse(Course c){…}
    public boolean recordMark(Course c, float m){…}
}
```

courseSelected(Course c) throws NullPointerException, NotExistedCourseException
getCourseMark(Course c) throws NullPointerException, NotExistedCourseException, NotSelectedCourseException
selectCourse(Course c) throws NullPointerException, NotExistedCourseException //如果c已经被选择了，返回什么？True
unSelectCourse(Course c) throws NullPointerException, NotExistedCourseException //如果c未被选择，返回什么？True
recordMark(Course c, float m) throws NullPointerException, NotExistedCourseException, NegativeMarkException, NotSelectedCourseException

异常类型



checked exception (by compiler): 可控异常，要求必须在方法声明中列出来，否则无法通过编译。继承自Exception

unchecked exception (by compiler): 不可控异常，可以不在方法声明中列出。继承自RuntimeException

建议阅读：http://beginnersbook.com/2013/04/java-checked-unchecked-exceptions-with-examples/

不可控异常类型

```
public class ExceptionTest {
    public static void main(String[] args) {
         int i = 10/0;
    }
}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at ExceptionTest.main(ExceptionTest.java:5)
```

```
public class ExceptionTest {
    public static void main(String[] args) {
        int arr[] = {'0','1','2'};
        System.out.println(arr[4]);
    }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
at ExceptionTest.main(ExceptionTest.java:6)
```

```
import java.util.ArrayList;
public class ExceptionTest {
    public static void main(String[] args) {
        String string = null;
        System.out.println(string.length());
    }
}
```

```
Exception in thread "main"
java.lang.NullPointerException
at ExceptionTest.main(ExceptionTest.java:5)
```

不可控异常类型


RuntimeException
ArithmeticException
ArrayStoreException
BufferOverflowException
BufferUnderflowException
CannotRedoException
CannotUndoException
ClassCastException
CMMException
ConcurrentModificationException
DOMException
EmptyStackException
IllegalArgumentException
IllegalMonitorStateException
IllegalPathStateException
IllegalStateException
ImagingOpException
IndexOutOfBoundsException
MissingResourceException
NegativeArraySizeException
NoSuchElementException
NullPointerException
ProfileDataException
ProviderException
RasterFormatException
SecurityException
SystemException
UndeclaredThrowableException
UnmodifiableSetException
UnsupportedOperationException

可控异常类型

```
try {
    String input = reader.readLine();
    System.out.println("You typed : "+input); // Exception prone area
    } catch (IOException e) {
      e.printStackTrace();
}
```

异常类型定义

- 选择扩展Exception或RuntimeException
- 只需定义构造函数

```
public class NewKindOfException extends Exception {

   public NewKindOfException( ) { super( ); }
   public NewKindOfException(String s) { super(s); }
}
```

Exception e1=new NewKindOfException(“this is the reason”);
String s = e1.toString();

“NewKindOfException: this is the reason”

异常的抛出与捕捉处理

```
public class Num{
    public static int fact(int n) throws NonPositiveException
    /**@Effects: If n is non-positive, throws NonPositiveException, else returns the factorial of n
    */
    {
        if(n<=0) throw new NonPositiveException("n in Num.fact");
        …
    }
}
```

```
try{ x=Num.fact(y);}
catch(NonPositiveException e){
    System.out.println(e);
}
```

屏蔽式处理

```
try { ... ;
    try { x= Arrays.search(v, 7);}
    catch (NullPointerException e) {
        throw new NotFoundException( ); }
} catch (NotFoundException b) { . . . }
```

反射式处理

异常的抛出与捕捉处理

- 如果一个方法m没有使用try…catch来捕捉和处理可能出现的异常，则会产生如下两种情况
  - 如果抛出的是不可控异常，则Java会自动把该异常扩散至m的调用者
  - 如果抛出的是可控异常，且在m的标题中列出了该异常或者该异常的某个父类异常，则Java自动把该异常扩散至m的调用者
- 由于不可控异常的产生在运行时才能确定，因此需要格外小心其捕捉与处理

```
try { x=y[n];}
catch (IndexOutDfBoundsException e) {
    //handle IndexOutOfBoundsException from the array access y[n]
}
 i=Arrays.search(z, x);
```

关于异常的处理方式

为什么p抛出不同于e1的异常？

- 反射
  - 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后抛出另一个异常e2给其调用者
  - “我”处理了一种意外情况，根据软件需求，这种情况也需要报告给”上层”
- 屏蔽
  - 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后不再抛出异常给其调用者
  - “我”处理了一种意外情况，根据软件需求，没必要让”上层”知道是否发生了这种意外

关于异常的处理方式

- 对于在给定数组中搜索某个元素而言，考虑数组对象为null，或者对数组访问越界两种意外情况
  - NullPointerException需要通知调用者
    - Hey, 你给了一个不存在的数组！

  - IndexOutOfBoundsException呢？
    - Hey, 你给的数组我没访问好？！

```
public static int min (int[ ] a) throws
NullPointerException, EmptyException {
/**@EFFECTS: if a is null throws NullPointerException
else if a is empty throws EmptyException else returns
the minimum value of a
*/
int m;
   try { m = a[0]; }
   catch (IndexOutOfBoundsException e) {
         throw new EmptyException (”Arrays.min"); }
   for (int i=1; i < a.length; i++)
        if (a[i] < m) m = a[i];
   return m;
}
```

何时使用异常

- 当需要使用一个特殊返回值来告知调用者输入有异常情况时
  - 特殊返回值和异常相比，所表达的语义模糊，且容易被忽略
- 当一个方法期望可以在多处被重用时
  - 全局适用过程
- Requires规格中关于输入的某些有效性检查易于实施
  - 进行检查，一旦不满足抛出异常

Requires规格中有些要求是为了提高方法效率，或者方法自身检查难度大。这就要求调用者必须确保。

使用可控异常还是不可控异常

- 如果期望调用者提供的输入数据不会引发异常，就应该使用不可控异常 //隐式处理
  - 不可控异常默认逐层”上报”，确保会有方法进行处理，否则会中断程序的运行
  - 不可控异常对于调用者要求较高，如果忘记捕捉，会带来潜在的程序崩溃风险
- 如果不对调用者做特殊要求，应该使用可控异常 //显式处理
  - 能够提高程序的健壮性

防御编程(Defensive Programming)

- 异常处理机制提供了一种在主流程处理之外的程序防护能力
  - 确保主流程逻辑的清晰性
  - 通过异常类型有效管理程序需要关注的各种意外情况
  - 反射和屏蔽机制为异常处理带来了灵活性
- 在设计类的方法时，需要问如下问题
  - 有哪些输入？
  - 输入会出现哪些“例外”情况？
  - 这些“例外”情况如何通知调用者？

契约式编程

- 契约式编程(contract based programming, design by contract)
- 课外阅读材料与实践建议
  - Eiffel language and programming
  - JML (Java Modeling Language) and the Eclipse plugin
- 我们在教材基础上设计实现了JSF(Java Semi-Formal Specification Language)，使用Javadoc样式，并定义了一系列逻辑量词和表达式来描述前置条件、后置条件
  - 前置条件必须是一个可判定的布尔表达式
  - 后置条件必须是一个可判定的布尔表达式
- 本次作业的一个重点是撰写和检查过程规格

过程规格对于实现的重要性

- 首先书写规格，从整体上把握住一个方法**对外部的要求(Requires)**、**对外部的承诺(Effects)**和**方法要修改的系统状态和对象状态(Modifies)**
- 然后在规格基础上进行代码实现
  - **方法是否需要对照Requires检查输入？**
  - 当调用一个方法时
    - caller确保满足callee规格中Requires要求
    - caller需要注意callee是否修改传入的对象
  - 当调用返回时
    - caller检查callee规格中Effects所明确的各种效果
  - 方法只能对Modifies中规定的变量进行修改
  - 方法返回时必须保证满足Effects

规格的撰写

- 严格使用JSF来撰写，我们会有工具进行检查
- 一定不能在Effects部分撰写控制流程或算法流程，而是方法执行后用户获得的效果
- Effects一定要概括方法在不同输入形态下的多种执行效果
- 正向过程
  - 不是写完代码后再来补写规格
- 测试检查
  - 前置条件是否为布尔表达式？
  - 后置条件是否为对方法执行结果的归纳，而不是算法流程？
  - 方法实现是否与规格一致？

JSF简要介绍

- 打开JSF Guideline简要介绍几个例子
- TIPS
  - 如果MODIFIES不为空，则在EFFECTS要严格区分访问或操作的变量是方法执行之前的状态（\old(…)），还是之后的。不使用\old则一定是指方法执行之后的，即用户调用返回后看到的结果
  - 可以像书写代码那样在规格中声明变量，并调用变量的**访问型**方法，甚至忽略方法的参数也是可以的(v.size)
  - 注意量词的使用\all, \exist
  - \result表示方法返回值
  - Normal_behavior和exceptional_behavior(***Exception)要严格分开书写

JSFTool征集令

- 2017年胡婧韬同学设计并实现了JSF规范和相应的检查工具
  - 基于Javadoc风格，扩展Javadoc实现了对规格文本的解析和处理
  - 在2017年的OO课上进行了初步使用，取得了预期效果
  - 我们会把工具发给大家自行检测
- 我们会在公测中自动扫描大家所写的规格，不允许还出现JSFTool已经报告的问题
- 征集有战斗力、激情的10位以内同学对JSFTool进行优化和功能扩展，我们使用你改进的JSFTool来扫描同学们所写的规格，并根据所发现的不符合规范情况，给予适当奖励，特别会在学期末颁奖中加以考虑。请直接通过助教报名。

作业

- 新增功能
  - 道路临时关闭/打开功能
  - 增加按照流量的道路选择
- 设计要求
  - 针对所有方法书写过程规格。对新增方法，应先写规格再实现代码；对已有方法，应确保规格和实现的一致性
- 测试要求
  - 仍然使用测试线程：模拟乘客请求、模拟道路的关闭/打开
  - 检查设计要求的满足情况
    - 检查规格书写的规范性
    - 检查规格与代码实现的一致性