第九讲：过程抽象与异常处理

OO2019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
  - 掌握基于规格的测试方法

如何证明你的设计是正确的

- 2018年10月以来，美国波音公司旗下737MAX系列客机，发生了两起空难事故。
- 2019年4月初，波音公司首席执行官承认，两起空难都与737MAX系列客机“自动防失速系统”有关，承诺将进行系统软件更新
- 2019年4月17日，波音首席执行官表示，系统软件更新后的737MAX系列客机，已经完成了工程试飞，**这将是史上最安全的客机之一**。
- 问题：你信吗？谁能证明这位CEO的话？！

业界的努力和形成的方法

- 混合了形式化和非形式化的方法
  - **Design by contract（DbC）/契约式设计**
  - **Formal validation/形式化证明**
  - OO and reuse techniques/面向对象和重用技术
  - Global public scrutiny/全局公共对象详查
  - Extensive testing/大量测试
  - Metrics efforts/基于度量的质量分析评价

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)

定义三角函数与实现三角函数

- sin(x)
- 定义（规格描述）：在直角三角形中，∠α（不是直角）的对边与斜边的比叫做∠α的正弦，记作sinα。
- 工程计算实现：
  - 按定义计算：将一个角放入直角坐标系中，使角的始边与X轴的非负半轴重合，在角的终边上取一点A（x，y），过A做X轴的垂线，则 $r=\sqrt{x^2+y^2}$
  - Taylor展开：$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \cdots$（计算精度可预期）

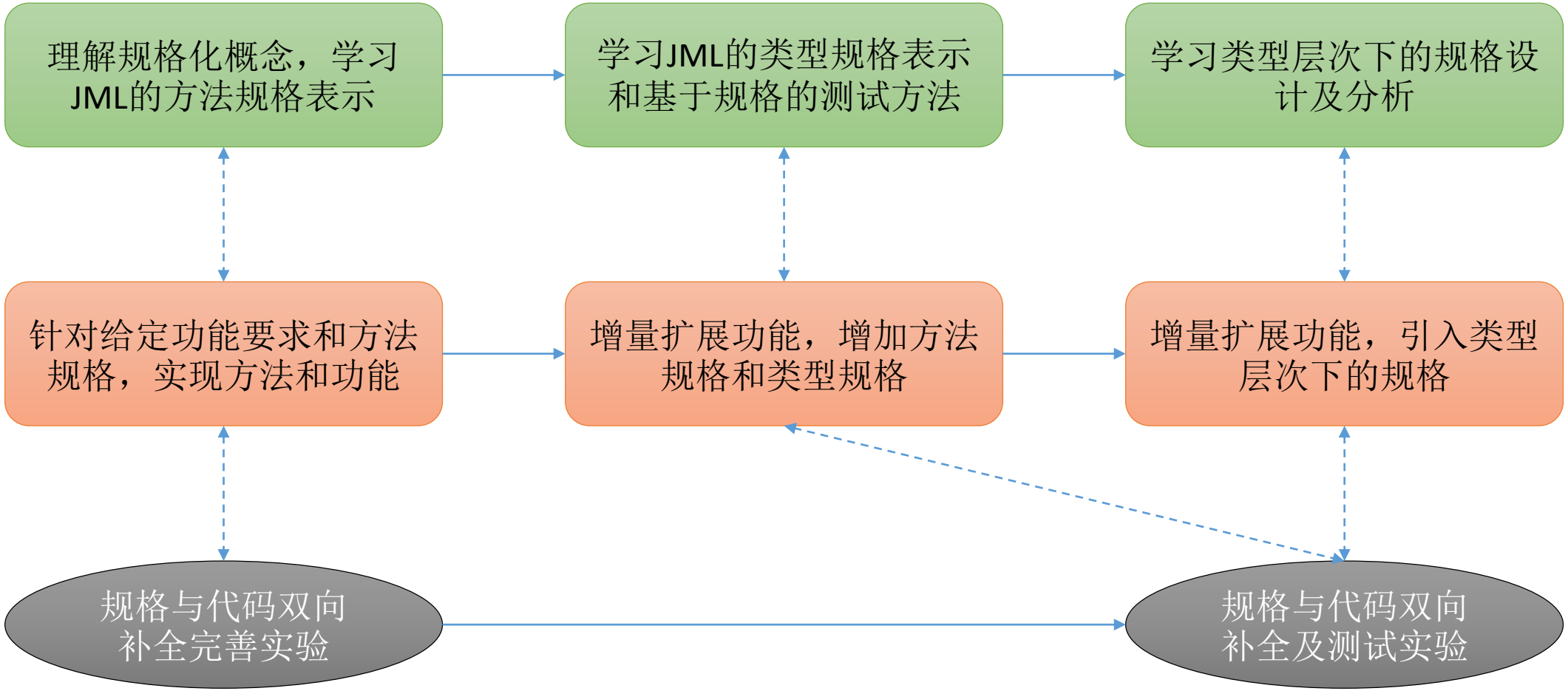
抽象类别

- 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

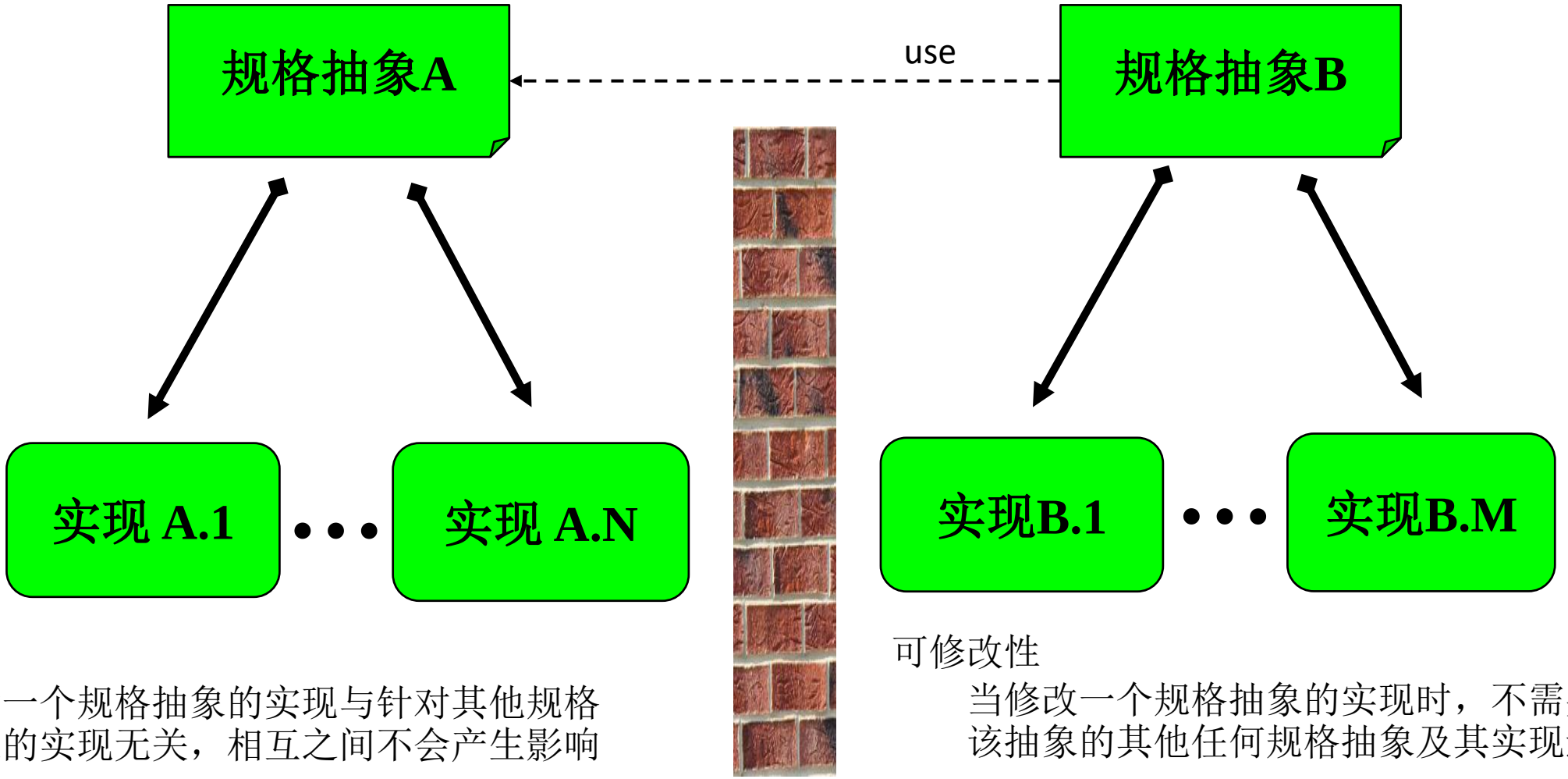
为什么要学习规格？

- 获得回答方法正确性问题的技术手段
  - 满足规格的方法实现就是正确的！！！
- 开展测试设计的依据
  - 不可能只在黑盒层次开展测试
  - 方法或接口测试是频繁使用的手段
- 准确理解一个方法的行为
  - 抓住其规格，而不是代码实现（有时看不到代码实现）
  - 多人协同开发时的交互基础

本单元的理论学习与训练的途径设计



规格抽象



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

如何表示规格抽象

- 有很多研究，比如使用形式化语言
- 课程介绍和使用JML来表示
  - 整合了Java和Javadoc，并引入了必要的形式化表达手段的规格化语言
  - 丰富的工具，可以把规格与实现代码在一起做整合分析、测试，甚至形式验证
- 方法规格抽象
  - 执行前对输入的要求----前置条件(precondition)
  - 执行后返回结果应该满足的约束----后置条件(postcondition)
- 数据规格抽象（类型抽象）
  - 数据状态应该满足的要求----不变式(invariant)
  - 数据状态变化应该满足的要求----约束(constraint)

什么是JML(Java Modeling Language)

- JML是一种形式化的、面向JAVA的行为接口规格语言（behavioral interface specification language）
  - JML允许在规格中混合使用Java语法成分和JML引入的语法成分
- JML拥有坚实的理论基础
- JML使用Javadoc的注释方式
  - 结构化、扩展性强
  - 块注释：/*@ … @*/
  - 行注释：//@
- JML已经拥有了相应的工具链，可以自动识别和分析处理JML规格
  - openjml：http://www.openjml.org/
  - http://www.eecs.ucf.edu/~leavens/JML//index.shtml

Java
Modeling: abstraction(method + type)
Language: concepts, constructs, rules

JML 语法一览

- Precondition
  - /*@ requires P; @*/
- Postcondition
  - /*@ ensures P; @*/
- Side-Effects
  - /*@ assignable list;@*/
- Exception
  - /*@ signal (Exception e) P;@*/
- Invariant
  - /*@ invariant P; @*/
- Constraint
  - /*@ constraint P; @*/
- method result reference
  - \result
- Previous expression value
  - \old(E)

Using private fields in specifications

private /*@ spec_public @*/ Type property;

Fields not null

Private /*@ not_null @*/ Type property;

Declare spec variable

//@ public model Type x;

Quantifiers

Iterating over all variables

(\forall T x; R(x); P(x))

Verifying if exist variables

(\exists T x; R(x); P(x))

Num of elements

(\num_of T x; R(x); P(x))

Sum of expression

(\sum T x; R(x); E)

一定要仔细阅读和理解所给的JML手册！！！

方法规格的组成(非严格表示法)

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

方法规格的组成(JML表示法)

```
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
*/
```

```
 /*@requires (\forall int i,j; 0<=i&&i<j&&j< a.length; a[i]<=a[j]);
   @assignable \nothing;
   @ensures (\exists int i;0<=i&&i<a.length;a[i]==x)
   @         ==>a[\result]==x;
   @ensures (\forall int i;0<=i&&i<a.length;a[i]!=x)
   @         ==>\result == -1;
   @*/
public static int sortedSearch (int[]a, int x)
```

基于规格的方法实现

```
/*@requires (\forall int i,j; 0<=i&&i<j&&j< a.length; a[i]<=a[j]);
   @assignable \nothing;
   @ensures (\exists int i;0<=i&&i<a.length;a[i]==x)
   @          ==>a[\result]==x;
   @ensures (\forall int i;0<=i&&i<a.length;a[i]!=x)
   @          ==>\result == -1;
   @*/
public static int sortedSearch (int[]a, int x)
```

实现是否满足了这个规格？

是否充分利用了规格？

规格本身是否完备和合理？

```
public static int sortedSearch(int[] a, int x){

    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;

}
```

类规格的组成

- 类的组成
  - 数据
  - 方法
- 类规格的组成
  - 对数据状态的要求： invariant, constraint
  - 对方法的要求: method specification
- 类规格完整准确定义了一个类的设计目标和能力
- 方法规格是类规格的组成部分

方法规格抽象

- 方法完成从输入到输出的转换计算
  - 可能需要对输入做出要求
  - 可能会修改输入参数
  - 可能会修改this对象的数据
  - 可能会(显式/隐式)返回结果
  - 可能会抛出异常
- 方法规格
  - 定义执行后获得的效果（后置条件），而不是具体怎么做
  - (满足前置条件) → (满足后置条件)
- 方法实现
  - 不同的实现只是在算法和数据表示方面有差异
  - 一种实现可以被另一种实现替换，调用者无需了解
- 方法测试
  - 满足前置条件场景
  - 不满足前置条件场景
- 调用者只通过方法规格就可确定是否需要，以及如何调用

方法规格抽象

- 很多时候需要对输入进行划分，不同的输入状态对应不同的处理效果
- 例如public int removePath(Path p)
  - p==null　　\result == -1?
  - p!=null, but p is not valid　　\result == -1?
  - p is valid, but p is not contained in this　　\result == -1?
  - p is valid and p is contained in this　　\result >=0 with a[\result] == path
- 这四个不同的输入划分分别对应什么返回结果？
- 三个(\result == -1)是否含义相同？

方法规格抽象

- 为什么我需要设计这个方法？
  - 核心价值：数据处理
    - →正常功能
  - 如果输入偏离了正常范围，导致方法无法进行预期的处理呢？
    - →异常功能
- JML提供了表达机制，分离出正常和异常情况下的规格表示与设计
  - 可以有多个normal_behavior及exceptional_behavior
  - normal_behavior与exceptional_behavior一定不能在输入上有交集→矛盾的规格！

```
@public normal_behavior
    @ requires clause;
    @ assignable clause;
    @ ensures clause;
```

```
@public exceptional_behavior
    @ requires clause;
    @ assignable clause;
    @ signals clause;
```

方法规格抽象

```
@public normal_behavior
    @ requires x>0;
    @ assignable \nothing;
    @ ensures p1;
@public exceptional_behavior
    @ requires x>80;
    @ assignable \nothing;
    @ signals_only **Exception;
```

```
@public normal_behavior
    @ requires x>0;
    @ assignable \nothing;
    @ ensures p1;
@public normal_behavior
    @ requires x<60;
    @ assignable \nothing;
    @ ensures p2;
```

方法规格抽象

- 为了准确定义一个方法的执行效果，有时不得不诉诸于方法需要访问的数据
- 例如public int removePath(Path p)
  - 除了返回值之外，要求执行后this中不再有p
- 从规格角度，设计者一般不会规定具体实现方案
  - 算法和数据存储方案
- 声明规格变量，仅用于说明规格所对应的逻辑条件
  - public model non_null Path[] pList;
- 有时候设计者可能会规定数据存储方案，但是私有化保护
  - private /*@spec_public@*/ ArrayList<Path> pList;

方法规格抽象

- 方法执行效果的规约往往需要在方法执行前的对象状态与方法执行后的对象状态之间建立逻辑联系
- 例如public int removePath(Path p)
  - 执行前所有和p不相同的对象，执行后仍然存在
  - 执行前所有与p相同的对象（if any），执行后不会存在
- 使用\old(E)表达式来记录方法执行前表达式E的取值
  - 就像给E在方法执行前拍个快照一样

```
private /*@spec_public@*/ ArrayList <Path> pList;
public int f(Path p){…;pList.add(p);…}
Q: \old(pList).size()与\old(pList.size())是否有区别？
```

Try：方法规格改写与实现

```
/* @requires All elements of v are not null;
   @assignable v;
   @ensures duplicate elements are removed from v;
@*/
public static void removeDupls (Vector v)
```

```
All elements of v are not null.
(\forall int i; 0<=i&&i<v.size();v.get(i)!=null)

duplicate elements are removed from v.
(\forall int i,j;0<=i&&i<j&&jv.size();
v.get(i)!=v.get(j))
```

Task1：请尝试使用JML来改写规格

Task2：请实现该方法

```
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); }
}
}
```

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)
- 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)
- 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

方法规格的特性

- 最少限度性
  - 只强调使用者关心的要求
- 避免不确定性
  - 不确定性：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，一个方法不应该做太多事情

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ensures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
  - 规格中专门说明exceptional_behavior
  - 实现中抛出异常来触发针对性的处理机制

带有异常终止的过程规格

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

/*@ public normal_behavior

@requires …

@assignable …:须明确当抛出异常时会产生什么副作用

@ensures …: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 exceptional_behavior

@ requires …

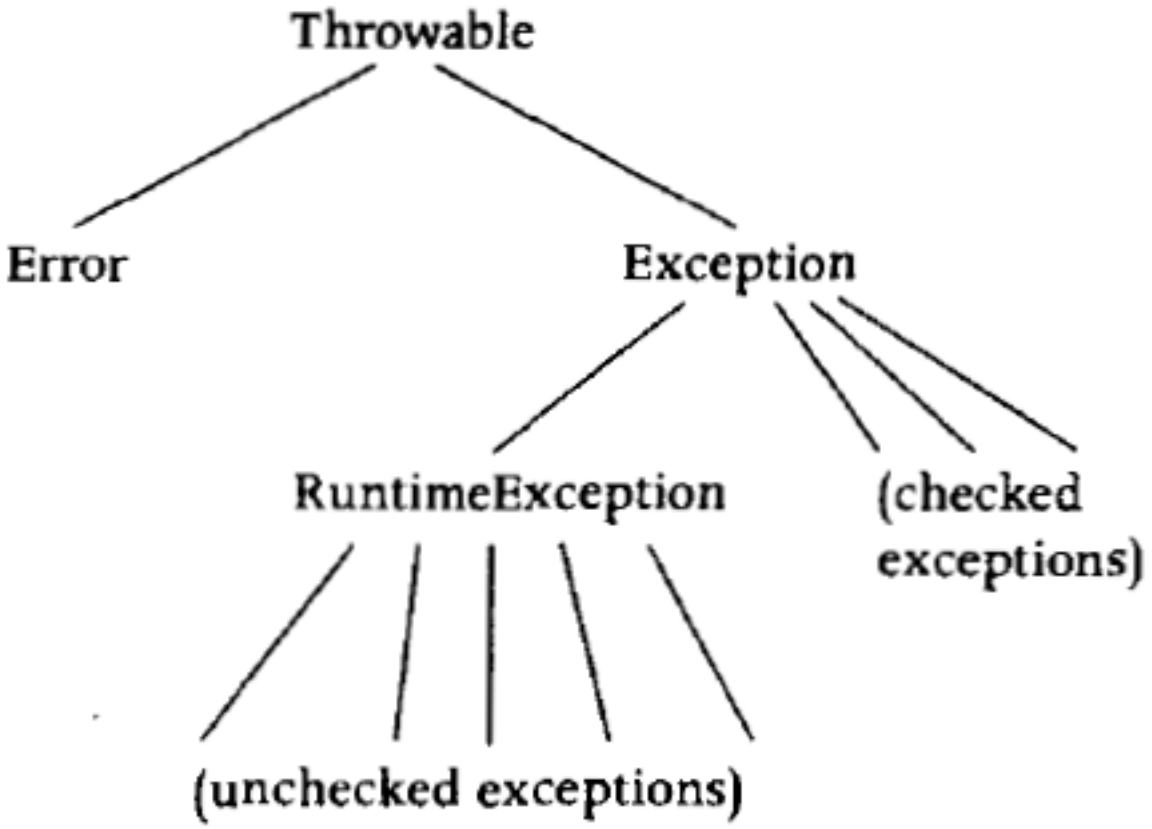
@ assignable …

@ signals (**Exception e) P;

*/

```
/* @ public normal_behavior
   @ requires v != null && (\forall int i;
   @    0<=i&&i<v.size();v.get(i).intValue()
   @    <=x.intValue());
   @ assignable v;
   @ ensures v.contains(x);
   @ public exceptional_behavior
   @ assignable \nothing;
   @ signals (NullPointerException e) v==null;
   @ signals (NotSmallException e)
   @ (\exists int i;0<=i&&i<v.size();
   @ v.get(i).intValue()==x.intValue());
   @*/
public void addMax (Vector v,Integer x) throws
NullPointerException, NotSmallException
```

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());
    }
}
```

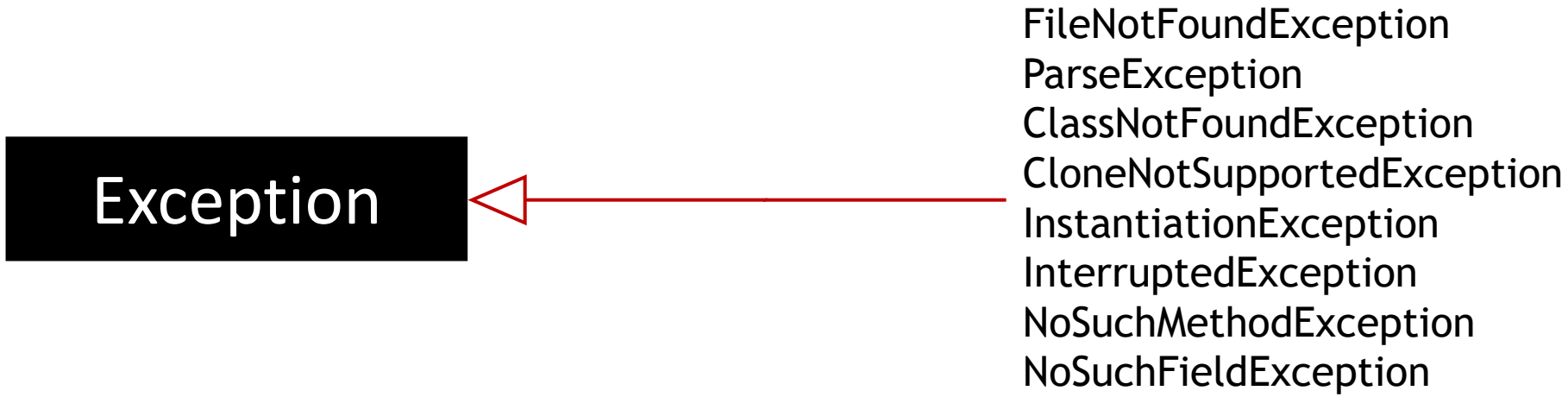
```
Exception in thread "main"
java.lang.NullPointerException
at ExceptionTest.main(ExceptionTest.java:5)
```

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

- 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);
```

异常的抛出与捕捉处理

```
public class Num{
    public static int fact(int n) throws NonPositiveException
    // 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

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 normal_behavior
  @ requires a!=null&&a.length>0;
  @ assignable \nothing;
  @ ensures (\forall int i;0<=i&&
  @                 i<a.length;a[i]>=\result);
  @ exceptional_behavior
  @ assignable \nothing;
  @ signals (NullPointerException e) a==null;
  @ signals (EmptyException e) a.length==0;
  @*/
```

从这个实现推一下其规格？

```
public static int min (int[ ] a) throws NullPointerException,
EmptyException {
   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
  - **p=null, \result == -1?**
- 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？

基于方法规格的代码实现要点

- 首先要准确理解给定的方法规格，特别是前置条件和后置条件
- 代码实现是要注意
  - **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
    - Caller需要注意callee是否会抛出异常
  - 当调用返回时
    - caller检查callee规格中ensures所明确的各种效果
    - 返回有可能直接进入到异常处理部分
  - 方法只能对assignable中规定的变量进行修改
  - caller方法返回时也必须保证满足相应的ensures，或者抛出异常

关于异常抛出规格的理解

- signals (Exception e) p==null;
- signals (Exception e) p.isValid()==false;

- 这不意味要一定要抛出两次异常，只是说这两个条件满足时要抛出异常
- 等价于signals (Exception e) p.isValid()==false || p==null;
- 此外，实现代码时可以抛出自定义的异常(Exception的子类)

本周作业

- 给定了Path和PathContainer接口，要求实现这两个接口中定义的操作，并实现指导书中规定的交互命令（控制台）
  - 接口给出了完整的规格定义，没有自然语言注释
  - 交互命令与接口有对应
- 测试仍然包括公测和互测，通过交互命令来对规格实现效果进行测试
- 要注意规格中有引用其他方法的规格，这不意味你一定要调用相应的方法
- 一定要基于规格来准备自己的测试集！！！
- 下一单元可能会对这两个接口做有趣的扩展，并可能增加新的接口
  - 注意实现的可扩展性