《面向对象设计与构造》
Lec09-方法规格与异常处理

OO2022课程组

北京航空航天大学计算机学院

内容提要

- 设计正确性的表示
- 设计规格
- 基于JML的规格表示
- 异常处理
- 异常类型
- 异常处理方式
- 作业解析

第三单元

- 基于规格的**层次化设计**
  - 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
  - 掌握抽象层次下类规格之间的关系
  - 掌握基于规格的测试方法

第一单元
面向问题分解与归纳的**层次化设计**

第二单元
面向并发控制与安全的**层次化设计**

第四单元
面向复杂数据管理的**层次化设计**

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**
- **2021年，特斯拉被曝多起刹车失灵事故，提供运行时数据以证明软件没有问题**

一切都是为了质量

- 提出和实践了多种方法和技术
  - Extensive testing/大量测试
  - Metrics efforts/基于度量的质量分析评价
  - OO and reuse techniques/面向对象和重用技术
  - **Design by contract（ DbC ）/契约式设计**
  - **Formal validation/形式化验证**
- 很多质量问题都是设计层次的问题
  - 设计阶段能否检测出来
  - 设计时能否有效避免

需要一种设计层次的质量控制手段

➔把质量在设计层次**显式表达**出来

➔**正确性**是最基础的质量特性

设计的正确性

- 设计是以系统为整体对如何完成需求的解决方案规划
  - 数据及其关系
  - 行为及其关系
- 设计正确性的内涵
  - 内在一致性
  - 满足需求
- 设计正确性必须在设计层次表达
  - 建立在设计规格上的约束

设计规格上的正确性表示

- 面向对象语言提供了数据抽象和行为抽象
  - 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
  - 继承、接口实现提供了层次描述抽象
  - 参数化、返回值提供了行为接口描述抽象
- 面向对象语言没有提供规格抽象(specification abstraction)
- 规格是对一个方法/类/程序的**外部可感知行为**的抽象表示
  - 设计正确性➔规格正确性

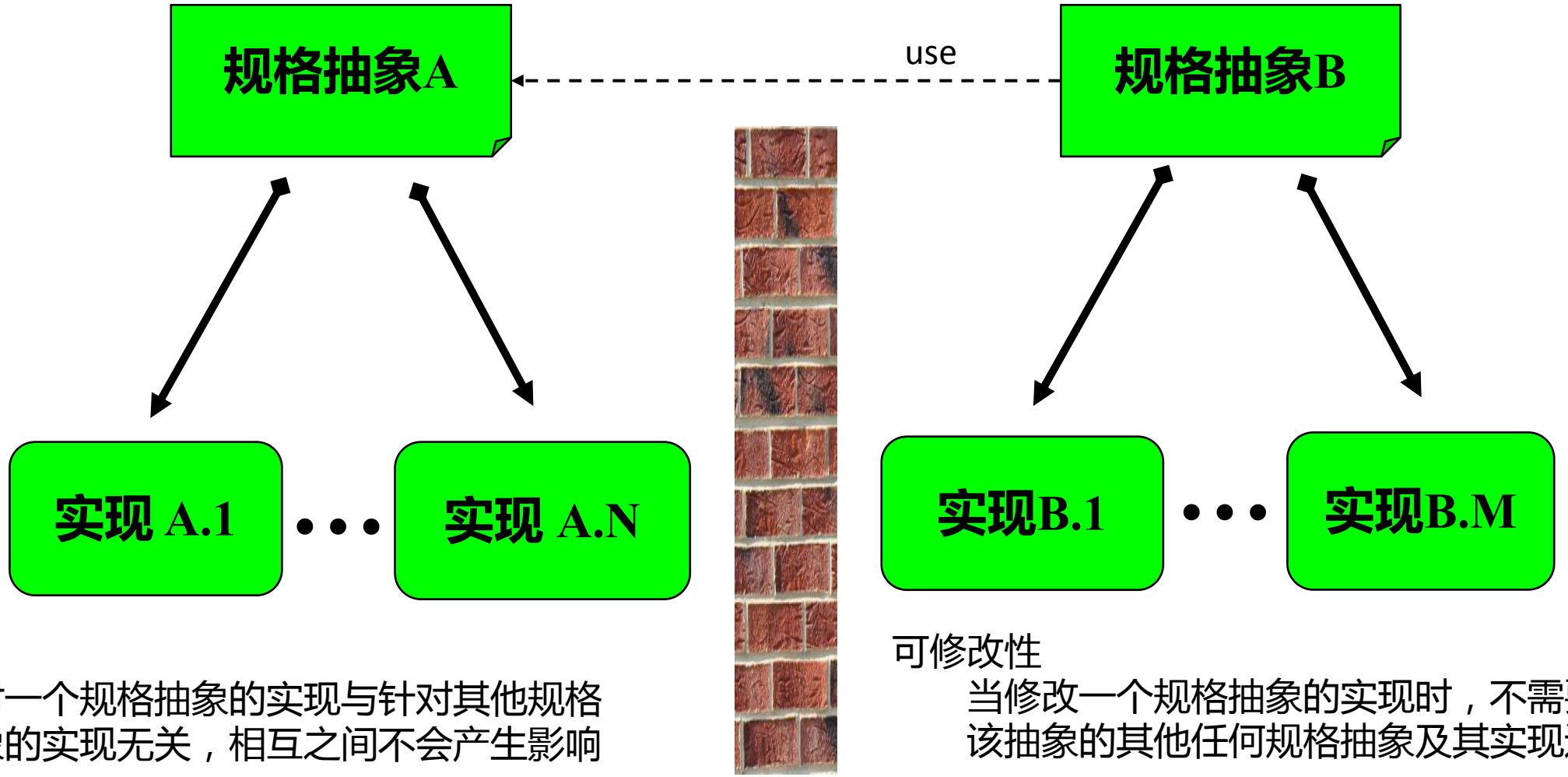
规格表示的类别

- 规格表示的要求
  - 明确规格主体：方法、类
  - 明确规格主体的外部**可感知**行为
  - 使用规范语言加以表示
- 方法规格
  - 定义一个方法或接口的外部可感知行为及其约束
  - 是方法或接口实现的依据
- 数据规格
  - 从外部使用者角度定义一个类所管理的数据及其需要满足的约束
- 类规格
  - 数据规格+方法规格

三角函数的设计规格与实现

- sin(x)
  - 设计规格：直角三角形中∠α（不是直角）的对边与斜边的比
  - 工程计算：
    - 基于规格定义的算法：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), sinα=y/r
    - 基于Taylor展开的算法：sin(x) = x - x^3/3！+ x^5/5！- x^7/7！+……（计算精度可预期）
  - 实现不必采用规格所定义的求解逻辑，但是必须满足规格（正确性要求）
    - 规格不是为了给出实现逻辑，而是表达设计正确性的抽象逻辑

规格抽象间的关系



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

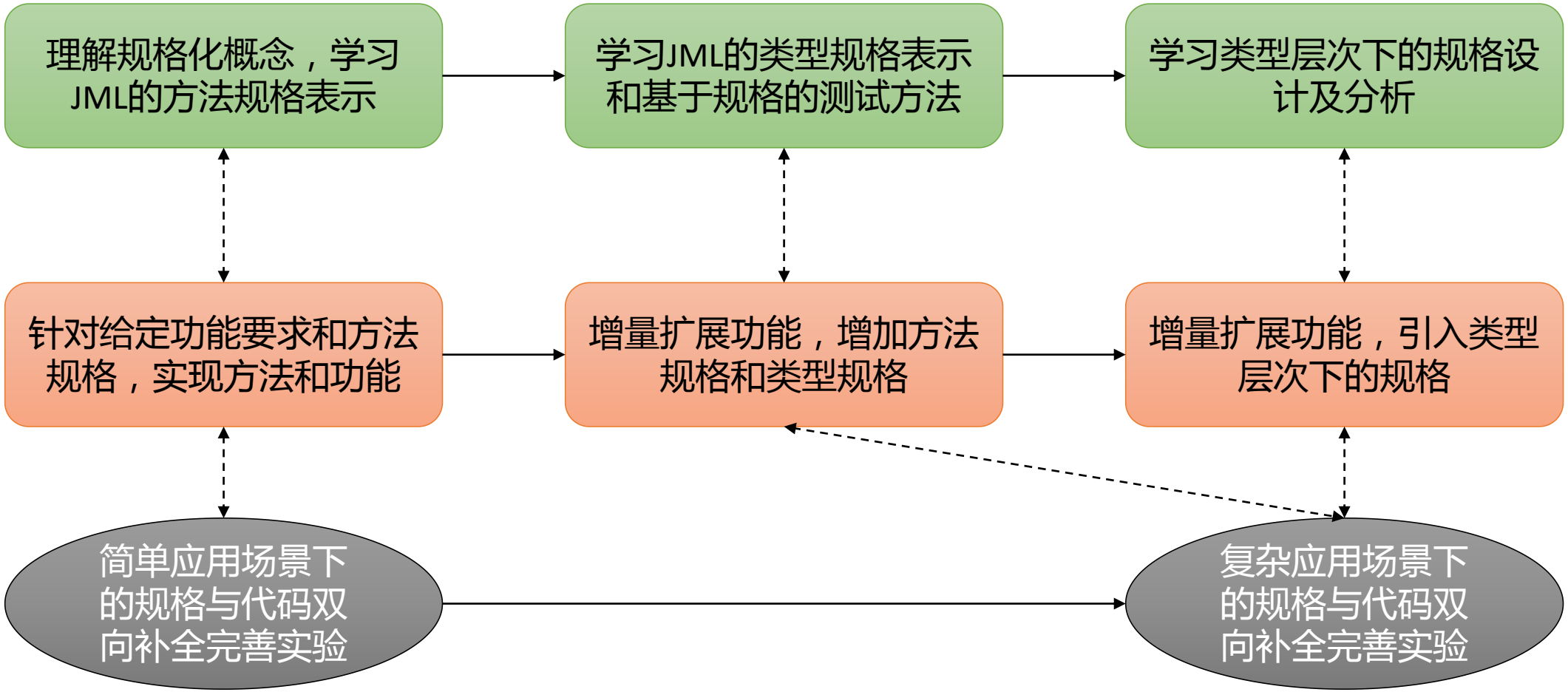
可修改性

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

为什么要学习规格？

- 准确定义和表示方法行为的正确性
  - 从而可以逻辑方式来验证代码实现的正确性
- 开展测试设计的依据
  - 不能只在黑盒层次开展测试
  - 需要对类、方法和接口进行测试
- 把设计与实现相分离
  - 控制架构复杂性

本单元的理论学习与训练的途径设计



如何表示规格

- 有很多研究，形式化表示是共识
  - 课程介绍和使用**JML**来表示，轻量级的形式化表示
  - 整合了Java和Javadoc，易用的工具支持
- 方法规格抽象
  - 执行前对输入的要求----前置条件(precondition)
  - 执行过程中对于环境（参数、所在this）的改变描述----副作用(Side-Effects)
  - 执行后返回结果应该满足的约束----后置条件(postcondition)
  - **方法正确性：前置条件满足→后置条件满足 && 修改不超出副作用范围**
- 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)
  - **数据正确性：对象状态满足不变式 && 对象状态变化满足约束**

什么是JML(Java Modeling Language)

- JML是面向JAVA的行为接口规格语言（behavioral interface specification language）
  - 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，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)

P：谓词；
T：类型；
R(x)：x取值范围；
E：表达式。

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)

仔细阅读和理解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)
```

是否满足规格？

是否充分利用了规格？

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

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)
```

是否满足规格？

是否充分利用了规格？

规格是否完备？

```
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

- 方法规格：定义执行成功的前提条件和成功执行的效果
- 方法实现：完成从输入到输出的转换计算
  - 副作用：可能会修改输入对象或this对象
  - 结果：(显式/隐式)返回结果、抛出异常
  - 一种实现可被另一种实现替换，调用者不关心
    - 不同的实现在算法和数据表示方面有差异
- 方法测试：检查方法实现是否满足方法规格
  - 满足前置条件场景
  - 不满足前置条件场景

方法规格

- 难以统一表示在不同输入情况下的执行效果
  - 输入划分，分情况定义执行效果
- 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

- 为什么我需要设计这个方法？
  - 核心价值是提供数据处理能力→正常处理
  - 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？→异常处理
- JML提供了分离式表达机制，强制区分正常和异常情况下的设计
  - 可以有多个normal_behavior及exceptional_behavior
  - normal_behavior与exceptional_behavior在对应的输入上无交集

@public normal_behavior		@public exceptional_behavior
(@ requires clause;)	**&&**	**(@ requires clause;) == false**
@ assignable clause;		@ assignable clause;
@ ensures clause;		@ signals clause;

方法规格的冲突或不确定性

```
@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<100;
    @ assignable \nothing;
    @ ensures p2;
```

```
@public normal_behavior
    @ requires x<=0;
    @ assignable \nothing;
    @ ensures p2;
@public normal_behavior
    @ requires x>=100;
    @ assignable \nothing;
    @ ensures p1;
@public normal_behavior
    @ requires x>0 && x<100;
    @ assignable \nothing;
    @ ensures p1 && p2;
```

方法规格

- 为了准确定义方法**执行效果**，有时必须借助方法要访问的**数据**
  - 例如：要求removePath(Path p)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，仅用于说明规格所涉及的约束条件
  - 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

- 往往需要在方法**执行前**与**执行后**的**对象状态**间建立逻辑联系，从而准确表达方法执行效果
- 例如removePath(Path p)
  - 执行前所有与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，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
  - 给E在方法执行前拍个快照

\old(pList).contains(p1)与\old(pList.contains(p1))是否有区别？

```
private /*@spec_public@*/ ArrayList <Path> pList;
@ensures (\forall Path p1; p1 != p; \old(pList.contains(p1)) ==> pList.contains(p1))
@ensures (\forall Path p1; p1 == p; \old(pList.contains(p1)) ==> !pList.contains(p1))
```

Try：使用JML改写方法规格

```
/* @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&&j<v.size(); v.get(i)!=v.get(j))
```

该规格有哪些实现方案？

此规格是否有什么问题？

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，等同于限定了相应方法的适用范围
  - 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→职责为先
- 确定性
  - 对于给定的输入，规格可以推导出确定的执行结果
  - removeDupls未明确非重复元素是否会被remove！
- 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
  - 字符串搜索方法的两个规格：规定分隔符 vs 不规定分隔符
- 简单性
  - 规格应该保持简单，一个方法不应该做太多事情→职责单一

部分适用过程隐藏有风险

- 部分适用过程对使用者提出了要求
  - 要求使用者必须清楚被调用方法的前置条件
  - 但是这个适用范围信息常常会被忽略→输入不满足前置条件
- 解决办法
  - 方法负责检查输入，如果不满足，返回**特定的值**告知使用者有例外情况
    - 弊端1：容易出现冗余检查，降低性能
    - 弊端2：使用者未必会对特殊的返回值进行处理，出现鲁棒性问题
  - 转变为全局适用的方法
    - 无需进行额外检查
    - 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
  - 能够判断输入是否满足前置条件
  - 能够**提醒**使用者输入出现了异常情况
- 如何提醒？
  - 使用特殊的返回值，使用者可以忽略
  - 使用异常机制，使用者必须进行处理
- 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

带有异常处理的过程规格

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

/*@ public normal_behavior

@requires …

@assignable …:

@ensures …:当输入满足requires条件时的结果；

@ public exceptional_behavior

@ requires …

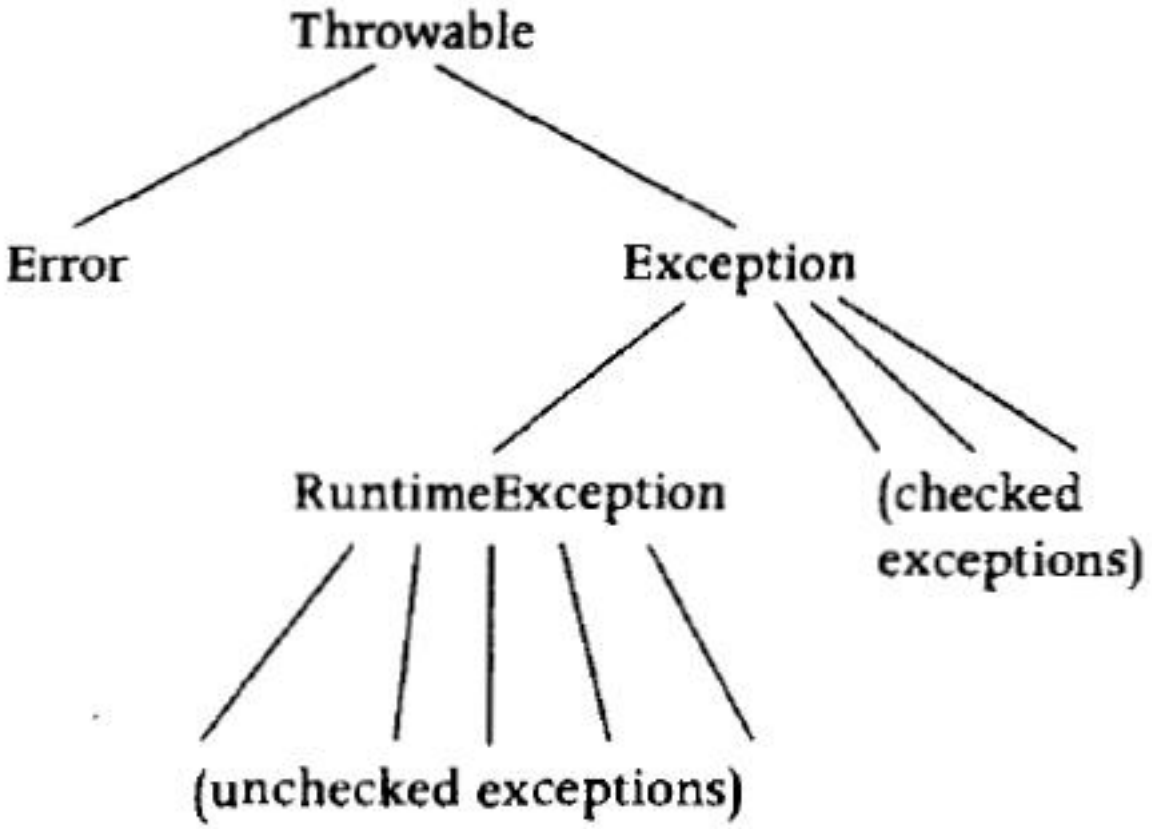
@ assignable …:须明确当抛出异常时会产生什么副作用

@ signals (**Exception e) P;当输入不满足时抛出的异常

*/

```
/* @ 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 (NotMaxException e)
   @ (\exists int i;0<=i&&i<v.size();
   @ v.get(i).intValue()>=x.intValue());
   @*/
public void addMax (Vector v, Integer x)
throws NullPointerException, NotMaxException
```

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 str = null;
        System.out.println(str.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();
}
```

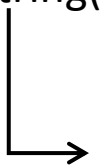
异常类型定义

- 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

- 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, 我搞砸了对你所给数组的访问？！

```
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;
}
```

```
/*@ 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;
   @*/
```

使用可控异常还是不可控异常

- 如果期望不去“干扰”调用者的处理逻辑，即不必捕捉相应的异常，则应使用不可控异常（**隐式**处理）
  - 优点：不可控异常默认逐层“上报”，可以在合适位置集中捕捉和处理
  - 缺点：如果每一层都忘记捕捉处理，一旦抛出异常会导致程序崩溃
- 如果要求调用者必须进行处理，应该使用可控异常（**显式**处理）
  - 优点：通过编译确保异常一定会得到处理
  - 缺点：分散的异常捕捉和处理，容易出现不一致

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
- 代码实现时要注意
  - **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;

- 这不意味一定会抛出两次异常
  - 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;

- 实现代码时可以抛出具体化的异常类型(Exception的子类)

作业解析

- 实现一个社交关系模拟系统
  - JML规格的理解和代码实现
- 实现规定的接口
  - 可以按照各自理解构造中间的对象管理层次
  - 要通过JML规格来准确理解接口的功能要求
  - 需要了解基本的图论及其算法
- 不一样的指导书风格
  - 接口功能：JML
  - 系统功能：自然语言
- 基于规格来准备自己的测试集