U3推送

本文基于JML Level 0手册编写，并提取出其中笔者认为的重要部分。大学没有真正的reference book，如果有更多需求，请大家和老师同学多多讨论，自行查阅资料。

初识JML——欢迎来到JML荣耀

JML是什么

JML是**一种用于对Java程序进行规格化设计的表示语言**，全称为**J**ava **M**odeling **L**anguage。

从课程教学角度，我们仅针对level 0语言特征，选择其中最核心和最常用的一些要素进行介绍和训练。如果要了解更多请点击这里。需要注意的是，目前这个Reference Manual仍然是草案，其中有些地方的定义和描述尚处于讨论阶段，因此建议同学们主要以课程组提供给大家的内容为主。

JML的作用是什么

JML的主要用途是为Java代码**提供精确的规格描述**，它使用特殊的注释语法来表达这些规格，这些注释可以被特定的工具读取和分析。JML是一种强大的工具，它通过为Java代码提供形式化的规格说明，帮助团队构建更加健壮和可靠的软件系统。

JML的特点是什么

1. **注释方式**：JML使用**javadoc风格的注释**来表达规格，这些注释通常以'@'符号开始。注释可以是行注释（`//@ annotation`）或块注释（`/*@ annotation @*/`），并且通常放置在被注释元素的紧邻位置。相应地，作为注释，JML并不会被java编译器编译或者运行，对程序结果没有影响。
2. **工具支持**：存在专门的工具，如JMLUnit和OpenJML，它们可以**解析JML注释**，并据此生成测试用例或验证代码是否符合其规格。
3. **适用场景**：JML尤其适用于那些需要**高可靠性和精确规格**的软件系统，如航空电子、医疗设备等领域的软件开发。
4. 规格中的**每个子句都必须以分号结尾**，否则会导致JML工具无法解析。
5. 在JML断言中，不可以使用带有赋值语义的操作符，如 ++，--，+= 等操作符，因为这样的操作符会对被限制的相关变量的状态进行修改，产生**副作用**。

往届作业里的一个小例子

下面标号**3**（标号仅仅是为了本例教学说明，合法的JML中不允许也没有这样的标号）后面的JML用来表示NetWork(社交网络)类中addPerson方法的规格，需要同学们根据规格补全java代码。故事背景是存在一个“全局”容器persons用于存放Person类，表示一个NetWork里有多少人。addPerson方法的功能顾名思义：就是把一个Person类对象add到persons容器中。

```
public interface Network {
    1//@ public instance model non_null Person[] persons;

    2/*@ invariant persons != null && (\forall int i,j; 0 <= i && i < j && j
< persons.length; !persons[i].equals(persons[j]));*/

    3/*@ public normal_behavior
      @ requires !(\exists int i; 0 <= i && i < persons.length;
persons[i].equals(person));
```

```
      @ assignable persons[*];
      @ ensures persons.length == \old(persons.length) + 1;
      @ ensures (\forall int i; 0 <= i && i < \old(persons.length);
      @          (\exists int j; 0 <= j && j < persons.length; persons[j] ==
(\old(persons[i]))));
      @ ensures (\exists int i; 0 <= i && i < persons.length; persons[i] ==
person);
      @ also
      @ public exceptional_behavior
      @ signals (EqualPersonIdException e) (\exists int i; 0 <= i && i <
persons.length;
      @                                      persons[i].equals(person));
      @*/
    public void addPerson(/*@ non_null @*/Person person) throws
EqualPersonIdException;

    ......(此处省略NetWork类的其他部分)
}


```

猜你想问（猜助教问过）

1.本来用自然语言写一句话我就能明白这个函数要干什么，甚至我看方法名都看得出来，课程组非要用JML写一大堆让我读，简直是折磨我的眼睛。之前U1和U2我读指导书读得好好的，你把他换了干什么？换汤不换药啊（范志毅式无奈）。What can I say, 难道课程组想毁了我吗😭？

2.学习JML之后，对我有什么好处？我能在哪方面得到提升呢？

3.为什么上面说是全局容器persons而不说是全局数组persons呢？我明明在JML里看到persons用中括号取其中的成员了，容器不应该是用get方法获取元素吗？

4.什么是invariant?

5.如果标号3后面是addPerson是addPerson的JML,那1和2是用来干嘛的？为什么和3分开写，是因为吵架了吗？

6.什么是normal_behavior?什么是exception_behavior?

7.requires，assignable，ensures是什么？我偷看了往届的作业，好像基本上都是按照requires，assignable,ensures的顺序来？

8.怎么有一些单词前面有类似于转义字符的斜杠\？是助教不小心还是故意写上去的？

9.如果\是转义字符，那\old肯定不是普通变量，它有什么特殊含义吗？是说\old后面括号里的东西是个老登？

10.\exists,\forall是什么，以前离散一的时候学过全称量词和特称量词，和这个有关吗？为什么它们后面的部分写得像个for循环，却没有花括号和循环体？

11.also是什么，signal又是什么？

12.看了半天，这个方法是怎么表示把一个Person类对象加入persons容器中的？

如果你没有这么多的问题，大抵说明你比笔者聪明很多，是先天JML圣体。接下来笔者会给大家讲解一些常用且重要的JML规则，对于上述样例中提到的部分，会在相应知识点后面进行解答，请大家带着问题有目的性地学习，学习完后再回头思考上述的12个问题，前2个问题也许需要大家自己慢慢感悟，后10个问题大家看完本文应该能够回答。

JML规则——严格版JAVADOC

此部分书写的逻辑结构是自顶而下的，希望能给同学们梳理清楚JML的层次。

类型规格

定义

类型规格是针对Java程序中定义的**数据类型**所设计的限制规则，一般而言，就是针对类或接口所设计的约束

规则。从面向对象角度来看，类或接口包含**数据成员**和**方法成员**的声明及或实现。

类中规格变量声明

上述例子中，NetWork类中所有方法的规格必须建立在NetWork所管理的数据规格上，因此为了准确说明这两个方面的规格，首先给出了NetWork所管理的数据规格，如序号1后面所述JML。其中的model表示后面的 Person []persons 仅仅是规格层次的描述，并**不是这个类的声明组成部分**，**更不意味该类的实现人员必须提供这样的属性定义**（即你的代码中未必要出现persons这个属性，更不是必须定义一个数组，否则查找性能可能会很差）

按照JML的语法，可以区分两类规格变量，**静态**或**实例**。如果是在Interface中声明规格变量，则要求明确变量的类别。一个类的静态方法**不可以**访问这个类的非静态成员变量（即实例变量）。静态成员**可以**直接通过类型来引用，而实例成员只能通过类型的实例化对象来引用。因此，在设计和表示类型规格时需要加以区分。

从便于理解的角度，可以认为static规格变量与Java中的static变量具有相同的访问规则。

针对上面的例子，如果是**静态规格**变量，则声明为**static model**

```
1 //@public static model non_null Person []persons;
```

如果是**实例规格**变量，则可声明为**instance model**

```
1 //@public instance model non_null Person []persons;
```

如果不声明，**默认是实例型的规格变量。**

二者的其余共性部分中，public表示变量的**可见性**为public，non_null表示该“数组”**对象引用不可为空。**Person []表示这是一个Person类的“数组”。

限制规则

JML针对类型规格定义了多种限制规则。从课程的角度，主要涉及两类，**不变式限制**(invariant)和**约束限制**(constraints)。无论哪一种，类型规格都是**针对类型中定义的数据成员**所定义的限制规则，一旦违反限制

规则，就称相应的状态有错。不变式限制（invariant）在本课程用得更多。

不变式限制（invariant）

要求在所有**可见状态(visible state)**下都必须满足的特性，语法上定义为`invariant P`，其中`invariant`为关键词，`P`为谓词。对于类型规格而言，**可见状态**是一个特别重要的概念，一般指在特定时刻下对一个对象状态的观察。下面所述的几种时刻下对象o的状态都是可见状态：

1.对象的有状态构造方法（用来初始化对象成员变量初值）的执行结束时刻

2.在调用一个对象回收方法(finalize方法)来释放相关资源开始的时刻

3.在调用对象o的非静态、有状态方法(non-helper)的开始和结束时刻

4.在调用对象o对应的类或父类的静态、有状态方法的开始和结束时刻

5.在未处于对象o的构造方法、回收方法、非静态方法被调用过程中的任意时刻

6.在未处于对象o对应类或者父类的静态方法被调用过程中的任意时刻

上面的定义难以全部记住，同学们把握住一点：**凡是**会修改成员变量（包括静态成员变量和非静态成员变量）的方法**执行期间**，对象的状态**都不是**可见状态（其本质原因是对象的状态修改未完成，此时观察到的状态可能不完整）。这里的可见不是一般意义上的能否见到，而是带有**完整可见**的意思。**在会修改状态的方法执行期间，对象状态不稳定，随时可能会被修改，因此在方法执行期间，对象的不变式有可能不满足。**

类型规格强调在任意可见状态下都要满足不变式。为了便于理解，可见状态就是一个**写方法**的**执行前后**（当然这并不严谨），如果你知道算法导论里提到的“循环不变式”，也许会更好理解这一点。

以上述例子中标号2后面的JML为例

```
1   2/*@ invariant persons != null && (\forall int i,j; 0 <= i && i < j && j < persons.length; !persons[i].equals(persons[j]));*/
```

表示在**任何可见状态下**，persons对象都不能为null，且应该满足后面的表达式的内容，虽然我们还没有讲\forall的用法，不过不难看出，\forall对应的表达式返回值应该是一个布尔值。（否则不能用&&和前面连接）

约束限制(constraints)

变是不变的道理，二者对立统一。

如果说invariant描述的是状态不变的规则，那constraints描述的就是对象的状态在变化时满足的约束。

对于对象的状态在变化时往往也要满足的一些约束，这类约束本质上也是一种不变式。

JML为了简化使用规则，规定invariant只针对**可见状态**(即当下可见状态)的取值进行约束，而是用constraint来对**前序**可见状态和**当前**可见状态的关系进行约束。如下面的例子：

```
public class Person{
    private /*@spec_public@*/ long money;
    //@ invariant money >= 0;
    //@ constraint money == \old(money)+1;
}
```

invariant指出**每个可见状态**money总是>=0，而constraint约束**每次修改**money只能加1。虽然这个约束可以在对money进行修改的方法中通过后置条件（ensures里的内容，后面会提到，可以暂时理解为其它限制性的JML语句）来表示，但是每个可能修改money的方法都需要加上这样的后置条件，远不如constraint这样的表示来的方便。不仅如此，invariant和constraint可以直接被子类继承获得。

关于上述代码中spec_public的解释：默认情况下，方法的规格对调用者可见，但是方法所在类的成员变量一般都声明为private,对调用者不可见。有时方法规格不得不使用类的成员变量来限制方法的行为，比如上面例子中的副作用范围限定，这就和类对相应成员变量的私有化保护产生了冲突。为了解决这个问题，JML提供了/*@spec_public@*/来注释一类的私有成员变量，表示在规格中可以直接使用，从而调用者可见

方法规格

方法规格的核心内容包括三个方面，**前置条件**、**后置条件**和**副作用限定**

前置条件(pre-condition)

通过requires子句来表示：`requires P;`。其中requires是JML关键词，P是谓词，顾名思义，表达的意思是**“要求调用者调用方法之前确保P为真”**。注意，方法规格中**可以有多个**requires子句，是**并列关系**，即调用者**必须同时满足所有的并列子句要求(requires子句间是and的关系)。**

如果设计者想要表达逻辑运算，则应该使用一个requires子句，在其中的谓词P中使用逻辑或操作符来表示相应的约束场景：如 // @ requires P1||P2;

后置条件(post-condition)

通过ensures子句来表示：`ensures P;`。其中ensures是JML关键词， P是谓词，表达的意思是**“方法实现者确**

保方法执行返回结果一定满足谓词P的要求，即确保P为真”。同样地，方法规格中可以有多个ensures子句，是

并列关系，即方法实现者必须同时满足所有并列ensures子句的要求。也可以和requires一样表示逻辑运算，如： // @ ensures P1||P2;

副作用范围限定(side-effects)

副作用指方法在执行过程中会修改对象的属性数据或者类的静态成员数据，从而给后续方法的执行带来影响。从方法规格的角度，必须要明确给出副作用范围。JML提供了副作用约束子句，使用关键词 **assignable** 或者 **modifiable** 。从语法上来看，副作用约束子句共有两种形态，一种不指明具体的变量，而是用JML关键词来概括；另一种则是指明具体的变量列表。下面是几种经常出现的副作用约束子句形态：

```
public class IntegerSet{
    private /*@spec_public@*/ ArrayList<Integer> elements;
    private /*@spec_public@*/ Integer max;
```

```
    private /*@spec_public@*/ Integer min;
    /*@
    @ assignable \nothing;
    @ assignable \everything;
    @ modifiable \nothing;
    @ modifiable \everthing;
    @ assignable elements;
    @ modifiable elements;
    @ assignable elements, max, min;
    @ modifiable elements, max, min;
    @*/
}
```

如该例子所述， assignble 表示**可赋值**，而 modifiable 则表示**可修改**。虽然二者有细微的差异，**在大部分情况下，二者可交换使用。**其中 \nothing关键词表示**当前作用域内可见的所有**类成员变量和方法输入对象**都不可以**赋值或者修改；\everything 关键词表示**当前作用域内可见的所有**类成员变量和方法输入对象**都可以**赋值或者修改。也可以指明具体可修改的变量列表，一个变量或多个变量，如果是多个则如上通过逗号进行分隔。

pure方法

设计中会出现某些纯粹访问性（只读）的方法，即不会对对象的状态进行任何改变，也不需要提供输入参数，这样的方法无需描述前置条件，也不会有任何副作用，且执行一定会正常结束。对于这类方法，可以使用简单的（轻量级）方式来描述其规格，即使用 **pure 关键词**：

```
//@ ensures \result.equals(name);
public /*@ pure @*/ String getName()
```

正常功能行为和异常行为

以最初例子为例：

```
    3/*@ public normal_behavior
      @ requires !(\exists int i; 0 <= i && i < persons.length;
persons[i].equals(person));
      @ assignable persons[*];
      @ ensures persons.length == \old(persons.length) + 1;
      @ ensures (\forall int i; 0 <= i && i < \old(persons.length);
      @          (\exists int j; 0 <= j && j < persons.length; persons[j] ==
(\old(persons[i]))));
      @ ensures (\exists int i; 0 <= i && i < persons.length; persons[i] ==
person);
      @ also
      @ public exceptional_behavior
      @ signals (EqualPersonIdException e) (\exists int i; 0 <= i && i <
persons.length;
      @                                    persons[i].equals(person));
      @*/
    public void addPerson(/*@ non_null @*/Person person) throws
EqualPersonIdException;
```

两种behavior开头的 public 指相应的规格在所在包范围内的所有其他规格处都可见。其中 normal_behavior 表示接下来的部分对addPerson方法的正常功能给出规格。所谓正常功能，一般指**输入或方法关联this对象的状态**在**正常范围内**时所指向的功能。与正常功能相对应的是异常功能，即 exceptional_behavior 下面所定义的规格。需要说明的是，如果一个方法没有异常处理行为，则无需区分正常功能规格和异常功能规格，因而也就不必使用这两个关键词。上面例子中出现了一个关键词 also ，它的意思是除了正常功能规格外，还有一个异常功能规格。

需要说明的是，按照JML语言规范定义，有两种使用also的场景：（1）父类中对相应方法定义了规格，子类重写了该方法，需要补充规格，这时应该在补充的规格之前使用also； **（2）一个方法规格中涉及多个功能规格描述，正常功能规格或者异常功能规格，需要使用also来分隔。**

signal子句

signals子句的结构为 signals (Exception e) b_expr ， 意思是当 b_expr 为 true 时，方法会抛出括号中给出的相应异常e。对于上面的例子而言，只要输入满足传入的person已经存在于persons数组，就一定会抛出异常EqualPersonIdException 。需要注意的是，所抛出的既可以是Java预先定义的异常类型，也可以是用户自定义的异常类型。

此外，还有一个注意事项，如果一个方法在运行时会抛出异常，**一定要在方法声明中明确指出（使用Java的 throws 表达式）**，且必须确保signals子句中**给出的异常类型一定等同于方法声明中给出的异常类型，或者是后者的子类型。**

还有一个简化的signals子句，即signals_only子句，后面跟着一个异常类型。signals子句强调在对象状态满足某个条件时会抛出符合相应类型的异常；而signals_only则不强调对象状态条件，强调满足前置条件时抛出相应的异常。

原子表达式

\result表达式

表示一个**非 void 类型**的方法执行所获得的结果，即**方法执行后的返回值。**\result表达式的类型就是方法声明中定义的返回值类型。如针对方法： public boolean equals (Object o) ，\result的类型是 boolean ，任意传递一个 Object 类型的对象来调用该方法，可以使用\result来表示 equals 的执行结果（ true 表示 this 和 o 相等； false 表示不相等）。

\old(expr)表达式

用来表示一个表达式 expr 在相应方法**执行前**的取值。该表达式涉及到**评估 expr 中的对象是否发生变化**，遵从Java的引用规则，即针对一个对象引用而言，**只能判断引用本身**是否发生变化，而不能判断引用所指向的对象实体内容是否发生变化。

假设一个类有属性 v 为 HashMap 类型，假设在方法执行前v的取值为 0x952ab340 ，即指向了存储在该地址的具体 HashMap 对象，则\old(v)的值就是这个引用地址。如果方法执行过程中没有改变 v 指向的对象，则 v 和\old(v)有相同的取值，即便方法在执行过程中对 v 指向的 HashMap 对象执行了插入或删除操作，因此 v.size() 和\old(v). size() 也有相同的结果。很多情况下，我们希望获得 v 在方法执行前所管理的对象个数，这时应使用\old(v.size())。**作为一般规则，任何情况下，都应该使用\old把关心的表达式取值整体括起来。**

\not_assigned(x,y,...)表达式

用来表示括号中的变量**是否在方法执行过程中被赋值**。如果没有被赋值，返回为 true ，否则返回 false 。实际上，该表达式主要用于后置条件的约束表示上，即限制一个方法的实现不能对列表中的变量进行赋值。

\not_modified(x,y,...)表达式

与上面的\not_assigned表达式类似，该表达式限制括号中的变量在**方法执行期间的取值未发生变化**。同学们不需要过于纠结它和\not_assigned的区别，课程作业中不会考察这种细微区别。

\nonnullelements(container)表达式

表示 container 对象中存储的对象中不会有 null 。

\type(type)表达式

返回类型type对应的类型(Class)，如type(boolean)为Boolean.TYPE。TYPE是JML采用的缩略表示，等同于Java中的 java.lang.Class 。

\typeof(expr)表达式

该表达式返回expr对应的准确类型。如\typeof(false)为Boolean.TYPE。

量化表达式

返回布尔值的量化表达式

\forall表达式

全称量词修饰的表达式，表示对于给定范围内的元素，每个元素都满足相应的约束。如：(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j]) ，意思是针对数组a中的任意两个下标在[0,10)之间的不同元素，一定满足排在后面的比排在前面的元素值大。这个表达式如果为真(true)，则表明数组a实际是升序排列的数组。

\exists表达式

存在量词修饰的表达式，表示对于给定范围内的元素，存在某个元素满足相应的约束。(\exists int i; 0 <= i && i < 10; a[i] < 0) ，表示针对0<=i<10，至少存在一个a[i]<0。

总结：上述规律为（\关键字 循环变量声明；确定遍历范围；给出约束要求）。对于\forall，若范围内约束均满足则为真；对于\exists，若范围内有至少一个约束满足则为真。

注意：由于“给出约束要求”，即最后一个分号后面的内容也是一个布尔值，因此\forall和\exisits可以嵌套自己或对方。

返回数字的量化表达式

\sum表达式

返回**给定范围内的表达式的和。** (\sum int i; 0 <= i && i < 5; i) ，这个表达式的意思计算[0,5)范围内的整数i的和，即0+1+2+3+4==10。注意中间的 0 <= i && i < 5 是对i范围的限制，求和表达式为最后面的那个 i 。同理，我们构造表达式 (\sum int i; 0 <= i && i < 5; i*i) ，则返回的结果为0+1+4+9+16。

\product表达式

返回**给定范围内的表达式的连乘结果。** (\product int i; 0 < i && i < 5; i)，这个表达式的意思是针对(0,5)范围的整数的连乘结果，即1* 2* 3 * 4==24。

\max表达式

返回**给定范围内的表达式的最大值。** (\max int i; 0 <= i && i < 5; i)，这个表达式返回[0,5)中的最大的整数，即4。

\min表达式

返回**给定范围内的表达式的最小值。** (\min int i; 0 <= i && i < 5; i)，这个表达式返回[0,5)中的最小的整数，即0。

总结：上述四个表达式规律为（\关键字 循环变量声明；确定遍历范围；给出映射关系），映射关系一般是将循环变量遍历空间映射到取值集合空间，可以再看一遍\sum表达式最后一个例子体会一下。

\num_of表达式

返回**指定变量中满足相应条件的取值个数。** (\num_of int x; 0<x && x<=20;x%2==0)，这个表达式给出(0,20]以内能够被2整除的整数个数，得到的数目为10。一般的，\num_of表达式可以写成 (\num_of T x; R(x);P(x))，其中T为变量x的类型，R(x)为x的取值范围；P(x)定义了x需满足的约束条件。但是其实本质上R(x)和P(x)可以合并为R(x)&&P(x），取值范围本质上就是一种限制条件。从逻辑上来看，该表达式也等价于 (\sum T x;R(x)&&P(x);1) 。

笔者的高级抽象理解

以上所有运算都可以抽象为（\二元运算符确定（表达式类型决定的）+ 变量声明；计算范围；待计算元素）；

其中待计算元素在计算范围内展开之后通过二元运算符连接。

\forall理解为（与运算 + 变量声明，计算范围，布尔值） A&&B&&C&&D

\exists理解为（或运算 + 变量声明，计算范围，布尔值） A||B||C||D

\sum理解为（加法运算 + 变量声明，计算范围，数值） A+B+C+D

\max表达式 （max运算 + 变量声明，计算范围，数值）max(A,max(B,max(C,D)))

其余类似，不一一赘述

操作符

子类型关系操作符

E1<:E2，注意这不是小于号，后面还有一个冒号。如果类型E1是类型E2的子类型(sub type)，则该表达式的结果为真，否则为假。如果E1和E2是相同的类型，该表达式的结果也为真，如 Integer.TYPE<:Integer.TYPE 为真；但Integer.TYPE<:ArrayList.TYPE 为假。相应地，任意一个类X，都必然满足X.TYPE<:Object.TYPE 。

等价关系操作符

b_expr1<==>b_expr2 或者 b_expr1<=!=>b_expr2 ，其中b_expr1和b_expr2都是布尔表达式，这两个表达式的意思是 b_expr1==b_expr2 或者 b_expr1!=b_expr2 。可以看出，这两个操作符和Java中的 == 和 != 具有相同的效果，按照JML语言定义， <==> 比 == 的优先级要低，同样 <=!=> 比 != 的优先级低。

推理操作符

b_expr1==>b_expr2 或者 b_expr2<==b_expr1 。对于表达式 b_expr1==>b_expr2 而言，当 b_expr1==false ，或者 b_expr1==true 且 b_expr2==true 时，整个表达式的值为 true 。可以理解为同学们大一离散学过的蕴含关系p->q，它也等价于!p||q

变量引用操作符

除了可以直接引用Java代码或者JML规格中定义的变量外，JML还提供了几个概括性的关键词来引用相关的变量。\nothing指示一个空集；\everything指示一个全集，即包括当前作用域下能够访问到的所有变量。变量引用操作符经常在assignable句子中使用，如 assignable \nothing 表示当前作用域下每个变量都不可以在方法执行过程中被赋值。

小试牛刀——我现在强得可怕

你能否解释一下最开始的函数每一行JML是什么意思?

```
1  public interface Network {
2      1//@ public instance model non_null Person[] persons;
3
4      2/*@ invariant persons != null && (\forall int i,j; 0 <= i && i < j && j
   < persons.length; !persons[i].equals(persons[j]));*/
5
6
7      3/*@ public normal_behavior
8        @ requires !(\exists int i; 0 <= i && i < persons.length;
   persons[i].equals(person));
9        @ assignable persons[*];
10       @ ensures persons.length == \old(persons.length) + 1;
11       @ ensures (\forall int i; 0 <= i && i < \old(persons.length);
12       @          (\exists int j; 0 <= j && j < persons.length; persons[j] ==
   (\old(persons[i]))));
13       @ ensures (\exists int i; 0 <= i && i < persons.length; persons[i] ==
   person);
14       @ also
15       @ public exceptional_behavior
16       @ signals (EqualPersonIdException e) (\exists int i; 0 <= i && i <
   persons.length;
17       @                                    persons[i].equals(person));
18       @*/
19     public void addPerson(/*@ non_null @*/Person person) throws
   EqualPersonIdException;
20
21     ......(此处省略NetWork类的其他部分)
22 }
```

答案:

```
public interface Network {
    //表示创建一个数据规格persons，属性是pulic,instance model表示其是实例对象，且引用不为null。
    1//@ public instance model non_null Person[] persons;

    //不变式，表示在任意可见状态，persons引用不为null，对于persons内不存在两个相同的对象。当然“相同”的意思取决于
    //Person的equal方法是否重写，如何重写。
    2/*@ invariant persons != null && (\forall int i,j; 0 <= i && i < j && j < persons.length; !persons[i].equals(persons[j]));*/

    //正常行为
    3/*@ public normal_behavior
    //前提是persons内不存在两个相同的对象
      @ requires !(\exists int i; 0 <= i && i < persons.length; persons[i].equals(person));
    // 可以赋值的对象是persons内的任意一个元素
      @ assignable persons[*];
    // 方法运行完之后persons的长度比之前多1
      @ ensures persons.length == \old(persons.length) + 1;
    // 方法运行完之后persons内的每一个元素都可以在方法运行之前persons内找到对应一模一样的元素，即没有删除任何之前的元素
      @ ensures (\forall int i; 0 <= i && i < \old(persons.length);
      @          (\exists int j; 0 <= j && j < persons.length; persons[j] == (\old(persons[i]))));
    // 方法运行完之后addPerson传入的person元素可以在persons中找到
      @ ensures (\exists int i; 0 <= i && i < persons.length; persons[i] == person);
     //异常行为
      @ also
      @ public exceptional_behavior
    // 如果persons内存在两个相同的对象，则抛出异常EqualPersonIdException
      @ signals (EqualPersonIdException e) (\exists int i; 0 <= i && i < persons.length;
      @                                      persons[i].equals(person));
      @*/
     // 传入的person引用不为null
    public void addPerson(/*@ non_null @*/Person person) throws EqualPersonIdException;

    ......(此处省略NetWork类的其他部分)
}
```

助教对JML的简化——神秘妙妙工具

这一部分在往届中并不存在，是笔者在本届初步推行的改变，初衷是为了减少大家阅读JML的负担，但是这样会破坏JML的严格性，因此笔者需要做出一些约定和限制。和往届对比具体的简化内容如下：我们还是先以addPerson方法为例

1.**尽可能使用了已有的读方法进行对象性质的刻画**，而不是每次都是用逻辑语言描述，即不重复造轮子让同学读。

```
    /* ......(其它JML)
    @ also
      @ public exceptional_behavior
      @ signals (EqualPersonIdException e) containsPerson(person.getId());
      @*/
```

其中，containsPerson(int idx)表示NetWork里面有id为idx的人，这个函数之前已经有数理逻辑描述的JML的定义，且经常在后面出现，因此不每次重复说明，使用封装好且易于理解阅读的函数表示。不要小看这一点，积累起来产生的质变还是很客观的，可以大大方便同学们的阅读

2.**针对容器相关的写方法的JML简化（核心部分）**

以addPerson方法为例，我们不难发现，其实我们就只是想加一个Person到persons里面去，真正描述方法功能的核心JML就一条,如下：

```
// 方法运行完之后addPerson传入的person元素可以在persons中找到
// @ ensures (\exists int i; 0 <= i && i < persons.length; persons[i] ==
person);
```

其余的JML都是为了严格起见，为了保证逻辑的绝对严谨，因此对于add类的方法来说，一般都要写如下几条JML：

1.add前容器中的元素add后仍然在容器中

2.add前容器中的元素没有被修改或替换过（\not_modified(A)或者\not_assigned(A)或者A==\old(A)）

3.容器中只add进了传入的元素，一般体现在length的变化上，在1，2的基础上，可以保证这一点。

简化之后：

```
/*@ public normal_behavior
      @ requires !containsPerson(person.getId());
      @ assignable persons[*];
      @ ensures containsPerson(person.getId());
      @ also
      @ public exceptional_behavior
      @ signals (EqualPersonIdException e) containsPerson(person.getId());
      @*/
 public void addPerson(/*@ non_null @*/Person person) throws
EqualPersonIdException;
```

是不是可读性上升了很多？requires需要方法执行前persons里不存在person，现在ensures方法执行后persons里存在person。虽然这样逻辑不严谨，因为没有规定不让删除或者修改之前的元素或者加入其它没有要求加入的元素；不过想必鲜有人会这样自找麻烦。

相应地delete（从容器中删除某个元素）方法和modify（修改容器中某个元素）方法也存在类似的简化空间。但是为例保证JML的严谨性，需要做出一些约定来规范这些简化。

JML简化标准

省流：JML里面让你做的你一定要做（包括JML指令的所有side-effect），JML不让你做的你千万不要做(不能产生任何其它side-effect)。

为了让同学们放心，对于使用了JML简化规则的，在方法名前面都加了safe标签，如下：

```
/*@ safe @*/
```

任何写方法都可能包含add(向容器中加入新元素),delete(从容器中删除已有元素)和modify(修改容器中已有元素的属性)三种过程，上述addPerson方法就只包括add过程，有些方法可能包括以上三种过程中的多种过程，现在给出三种过程的如下定义和简化约定。

add过程

定义：指的是向容器C内新增某个（或某些）对象O，这里只考虑对应容器数量的变化，不考虑容器内原有对象或新增对象本身属性的修改。

如何判断一个方法包括**add过程**：**requires**中指明了某个容器**Container**不包含某个（或者某些）对象**O**，而**ensures**中要求容器**Container**包含某个（或者某些）对象**O**，即**requires**和**ensures**中分别会出现布尔值相反的两个的要求。

如果不简化：

1.add前容器中的对象add后仍然在容器中

2.add前容器中的对象没有被修改或替换过（\not_modified(A)或者\not_assigned(A)或者A==\old(A)）

3.容器中只add进了传入的对象，没有add别的对象，一般体现在length的变化上，在1，2的基础上，可以保证这一点。

简化约定：不可以额外增加**对象O**以外的对象，也不可以删除或修改**容器C**中原有的对象

delete过程

定义：指的是从容器C内删除某个（或某些）对象O，这里只考虑对应容器数量的变化，不考虑容器内原有对象（包括保留的和被删除的对象）本身属性的修改。

如何判断一个方法包括**delete过程**：**requires**中指明了某个**容器C**包含某个（或者某些）**对象O**，而**ensures**中要求**容器C**不包含某个（或者某些）**对象O**，即**requires**和**ensures**中分别会出现布尔值相反（互斥）的两个要求。

如果不简化：

1.delete后容器中的对象都能在delete前的容器中被找到

2.delete前容器中包含的对象（包括保留的和被删除的对象）在delete后没有被修改或替换过（\not_modified(A)或者\not_assigned(A)或者A==\old(A)）

3.只从容器delete了指定的对象（通过传参等方式在JML中体现），没有delete其它对象，一般体现在length的变化上，在1，2的基础上，可以保证这一点。

简化约定：不可以额外删除**对象O**以外的对象，也不可以修改**容器C**中原有的对象（包括保留的和被删除的对象），更不能向容器内增加其它对象。

modify过程

定义：指的是只修改**容器C**中某个（或某些）**对象O**的属性（一般属性或容器属性），**容器C**的size不变，**容器C**中对象的集合不变，即没有对象的迁入或者迁出。

如何判断一个方法包括**modify过程**：JML要求中，方法执行前后容器的size和容器内对象的集合不变（对象引用不变，但是对象属性可能会发生变化），但是对容器内对象的属性（一般属性或容器属性）进行了修改。

如果不简化：

1.modify前后容器长度不变。

2.modify前后除了JML要修改的对象被修改，其余对象都不能被修改。

3.modify前后对象集合不变，这是为了防止有对象被替换成别的对象。

简化约定：不可以向容器C内新增对象或删除容器C内已有的对象，只能按照ML中给出要求对容器内对象的属性（一般属性或容器属性）进行修改，JML没有要求的修改不可进行。

注意，add,delete,modify过程可能同时存在，甚至递归，一个方法对一个对象O中数组对象属性C进行add一个新的对象obj的操作，对于对象O，这是一个modify过程，修改了其属性C对应的数组对象；对于数组对象C，这是一个add过程,被add的对象是obj。

举例：往年的addRelation方法

以方法**addRelation(int id1,int id2,int value)**为例，表示将id1和id2对应的人建立好友联系，联系的值（亲密度）为value。每个person都有acquaintances数组和values数组分别表示其好友和亲密度，values数组中的元素用int表示。

显然，我们在修改id1和id2对应的acquaintance数组和value数组新增相应值即可（这事实上是一个add过程，嵌套在modify过程里面，后续进一步介绍），然而为了说明这一点，我们需要用如下一长串的jml：

```
// id1和id2对应的person应该存在，且二者之前未建立好友关系
1 @ requires containsPerson(id1) && containsPerson(id2) &&
!getPerson(id1).isLinked(getPerson(id2));


* @ assignable persons[*];

// persons数组长度不应该发生改变，因为只涉及到关系的增加，不涉及人员变动
2 @ ensures persons.length == \old(persons.length);

// persons数组里面仍然是之前的persons，persons中元素的值没有修改，即没有修改对象本身（在2的基础上排除了删除一个对象然后新增一个对象的情况）
3 @ ensures (\forall int i; 0 <= i && i < \old(persons.length);
\not_modified(\old(persons[i])));


// 对于id1和id2之外对应的person，不要对其进行赋值（修改），将写的范围限制在id1和id2，排除了对其它person进行修改的可能
4 @ ensures (\forall int i; 0 <= i && i < persons.length &&
\old(persons[i].getId()) != id1 &&
        \old(persons[i].getId()) != id2; \not_assigned(persons[i]));

// 要求方法执行之后id1和id2互为关联（好友）
5 @ ensures getPerson(id1).isLinked(getPerson(id2)) &&
getPerson(id2).isLinked(getPerson(id1));

// 要求id1对id2的亲密度为value
6 @ ensures getPerson(id1).queryValue(getPerson(id2)) == value;

// 要求id2对id1的亲密度为value
7 @ ensures getPerson(id2).queryValue(getPerson(id1)) == value;

// id1对应person与旧好友的亲密度不变（顺序也不变，为后面一一对应铺垫）
8 @ ensures (\forall int i; 0 <= i && i <
\old(getPerson(id1).acquaintance.length);
@
not_assigned(getPerson(id1).acquaintance[i],getPerson(id1).value[i]));
```

```
30
31  // id2对应person与旧好友的亲密度不变（顺序也不变，为后面一一对应铺垫）
32  9 @ ensures (\forall int i; 0 <= i && i <
    \old(getPerson(id2).acquaintance.length);
33  @
    not_assigned(getPerson(id2).acquaintance[i],getPerson(id2).value[i]));
34
35  // 保证id1中acquaintance和value数组是一一对应的，即id1对应person与其
    acquaintance[i]的亲密度是value[i]
36  10 @ ensures getPerson(id1).value.length ==
    getPerson(id1).acquaintance.length;
37
38  // 保证id2中acquaintance和value数组是一一对应的，即id2对应person与其
    acquaintance[i]的亲密度是value[i]
39  11 @ ensures getPerson(id2).value.length ==
    getPerson(id2).acquaintance.length;
40
41  // 保证id1只增加了一个好友，也就是id2
42  12 @ ensures \old(getPerson(id1).value.length) ==
    getPerson(id1).acquaintance.length - 1;
43
44  // 保证id1只增加了一个好友，也就是id1
45  13 @ ensures \old(getPerson(id2).value.length) ==
    getPerson(id2).acquaintance.length - 1;
46
47  public /*@ safe*/ void addRelation(int id1, int id2, int value) throws
48              PersonIdNotFoundException, EqualRelationException;
```

不难发现，描述方法功能的jml是5，6，7三条。1是前提条件，2，3，4可以归纳为**modify过程**的限制，即只修改id1和id2对应对象的属性（acquaintance和value数组），而不修改persons容器的大小和元素集合,也不修改persons容器中id1和id2对应对象之外的元素。

8，12和9，13一样，是一个**add过程**（因为id1之前acquaintance和value数组里面没有id2对应的值，现在有了，符合add过程的特征）的限制。10和11事实上可以删去，因为acquaintance和value数组都是从0开始的，而12和13保证了它们的同步增长。

因此这事实上是一个modify和add嵌套的过程

对于容器persons,涉及到一个**modify过程**，而对于getPerson(id1或id2)属性组中的acquaintance和value容器，则分别涉及到**add过程**

其实从assignable就可以看出来，persons[*]对应的是每一个person元素，因此persons容器本身不需要修改长度或元素集合，故对于persons容器而言是一个modify过程。具体到person对象，则涉及到其属性中容器的add过程。当然，如果此时修改的person的属性不是容器，是一个普通变量（int,long.. ）或单个对象(假如每个人只有一个Head)则不涉及jml的简化。

简化后，我们只保留5，6，7,这足以正确实现这个方法。

```
1 /*@ requires containsPerson(id1) && containsPerson(id2) &&
!getPerson(id1).isLinked(getPerson(id2));

* @ assignable persons[*];

5 @ ensures getPerson(id1).isLinked(getPerson(id2)) &&
getPerson(id2).isLinked(getPerson(id1));

6 @ ensures getPerson(id1).queryValue(getPerson(id2)) == value;

7 @ ensures getPerson(id2).queryValue(getPerson(id1)) == value;

.... (还有其他部分的JML)*/
public /*@ safe @*/ void addRelation(int id1, int id2, int value) throws
            PersonIdNotFoundException, EqualRelationException;
```

当然，实际操作中同学们不需要分析这么透彻就能直接从简化后的JML中获取这些必要的信息以正确完整地完成方法功能，上述看似繁琐的约定是为了保证简化的绝对严谨。

JML进阶——哎哟你干嘛

之前的JML较为基础，如果你觉得你掌握了，不妨阅读下列代码，这是往届作业中笔者认为较难的JML，及时无法读懂，也请同学们不要慌张，熟练之后这个也非常简单。

```
/*@ public normal_behavior
      @ requires contains(id) && (\exists Person[] path;
      @         path.length >= 4;
      @         path[0].equals(getPerson(id)) &&
      @         path[path.length - 1].equals(getPerson(id)) &&
      @         (\forall int i; 1 <= i && i < path.length; path[i -
1].isLinked(path[i])) &&
      @         (\forall int i, j; 1 <= i && i < j && j < path.length;
!path[i].equals(path[j])));
      @ ensures (\exists Person[] pathM;
      @         pathM.length >= 4 &&
      @         pathM[0].equals(getPerson(id)) &&
      @         pathM[pathM.length - 1].equals(getPerson(id)) &&
      @         (\forall int i; 1 <= i && i < pathM.length; pathM[i -
1].isLinked(pathM[i])) &&
      @         (\forall int i, j; 1 <= i && i < j && j < pathM.length;
!pathM[i].equals(pathM[j]));
      @         (\forall Person[] path;
      @         path.length >= 4 &&
      @         path[0].equals(getPerson(id)) &&
      @         path[path.length - 1].equals(getPerson(id)) &&
      @         (\forall int i; 1 <= i && i < path.length; path[i -
1].isLinked(path[i])) &&
      @         (\forall int i, j; 1 <= i && i < j && j < path.length;
!path[i].equals(path[j]));
```

```
      @         (\sum int i; 1 <= i && i < path.length; path[i -
1].queryValue(path[i])) >=
      @         (\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i]))) &&
      @         \result==(\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i])));
      @ also
      @ public exceptional_behavior
      @ signals (PersonIdNotFoundException e) !contains(id);
      @ signals (PathNotFoundException e) contains(id) && !(\exists Person[]
path;
      @         path.length >= 4;
      @         path[0].equals(getPerson(id)) &&
      @         path[path.length - 1].equals(getPerson(id)) &&
      @         (\forall int i; 1 <= i && i < path.length; path[i -
1].isLinked(path[i])) &&
      @         (\forall int i, j; 1 <= i && i < j && j < path.length;
!path[i].equals(path[j])));
      @*/
public int queryLeastMoments(int id) throws PersonIdNotFoundException,
PathNotFoundException;
```

阅读JML技巧与步骤

前情提要：

1.contains（idx）表示NetWork里面是否包含id为idx的Person。

2.isLinked可以理解为两个Person是否有社交联系。

3.person1.queryValue(person2)表示person1对person2的社交亲密度（具有对称性）。

看不懂没关系，把人理解为点，关系理解为边，value理解为边权。

阅读分析JML步骤

1.**看方法名和返回值**了解大概，但是这里看不出来(往届助教起名比较有想象力)，我们只知道返回一个int。

2.**拆分behavior，先看异常exceptional_behavior**，因为一般exceptional_behavior只有类似于requires的条件谓词部分，没有副作用等，相对容易阅读。

2.1**异常行为**

第一个异常：当传入的id不存在于NetWork里面时，报异常，异常名字很清楚，PersonId无法找到。

第二个异常：不存在任何一种路径path，使得路径长度>=4，路径从传入id对应的这个Person开始，也在传入id对应的这个Person结 束，且路径path的前一个人和后一个人有社交联系，并且path中除了首尾是同一个人，其余成员各不相同。翻译翻译，就是说**不存在以id对应Person为起点的路径长度>=4的社交环。**

2.2**正常行为**

正常行为的requires一般是异常行为条件的补集，正如我们所料，第一个requires可以不用看了。ensures括号嵌套过于复杂，需要进一步拆分。

3.**划分逻辑部分。**不难发现，描写pathM的部分好像似曾相识，就是描述了之前所说**以id对应Person为起点的路径长度>=4的社交环**，把这个特征记作P，即**P:以id对应Person为起点的路径长度>=4的社交环**，则ML可以划分为:

```
/* (\exists Person[] pathM; pathM在满足P的范围内 ; (\forall Person[] path;
      @         path.length >= 4 &&
      @         path[0].equals(getPerson(id)) &&
      @         path[path.length - 1].equals(getPerson(id)) &&
      @         (\forall int i; 1 <= i && i < path.length; path[i -
1].isLinked(path[i])) &&
      @         (\forall int i, j; 1 <= i && i < j && j < path.length;
!path[i].equals(path[j]));
      @         (\sum int i; 1 <= i && i < path.length; path[i -
1].queryValue(path[i])) >=
      @         (\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i]))) &&
      @         \result==(\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i])))*/
```

不难发现，后面的这一部分描述\forall的部分也似曾相识，也有一部分满足P，进一步划分：

```
/* @  (\exists Person[] pathM; pathM在满足P的范围内 ;
   @         (\forall Person[] path; path在满足P的范围内 ; 并且满足
   @         (\sum int i; 1 <= i && i < path.length; path[i -
1].queryValue(path[i])) >=
   @         (\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i]))) &&
   @         \result==(\sum int i; 1 <= i && i < pathM.length; pathM[i -
1].queryValue(pathM[i])))*/
```

下述语句表示环路上相邻两人之间亲密度的和，可以理解为边权和，下述语句就是**环路边权和的值。**

```
// (\sum int i; 1 <= i && i < path.length; path[i - 1].queryValue(path[i]))
```

再化简：

```
/* @  (\exists Person[] pathM; pathM在满足P的范围内 ;
   @         (\forall Person[] path; path在满足P的范围内 ; 并且满足path边权和
>=pathM边权和) &&
   @ \result==pathM边权和)*/
```

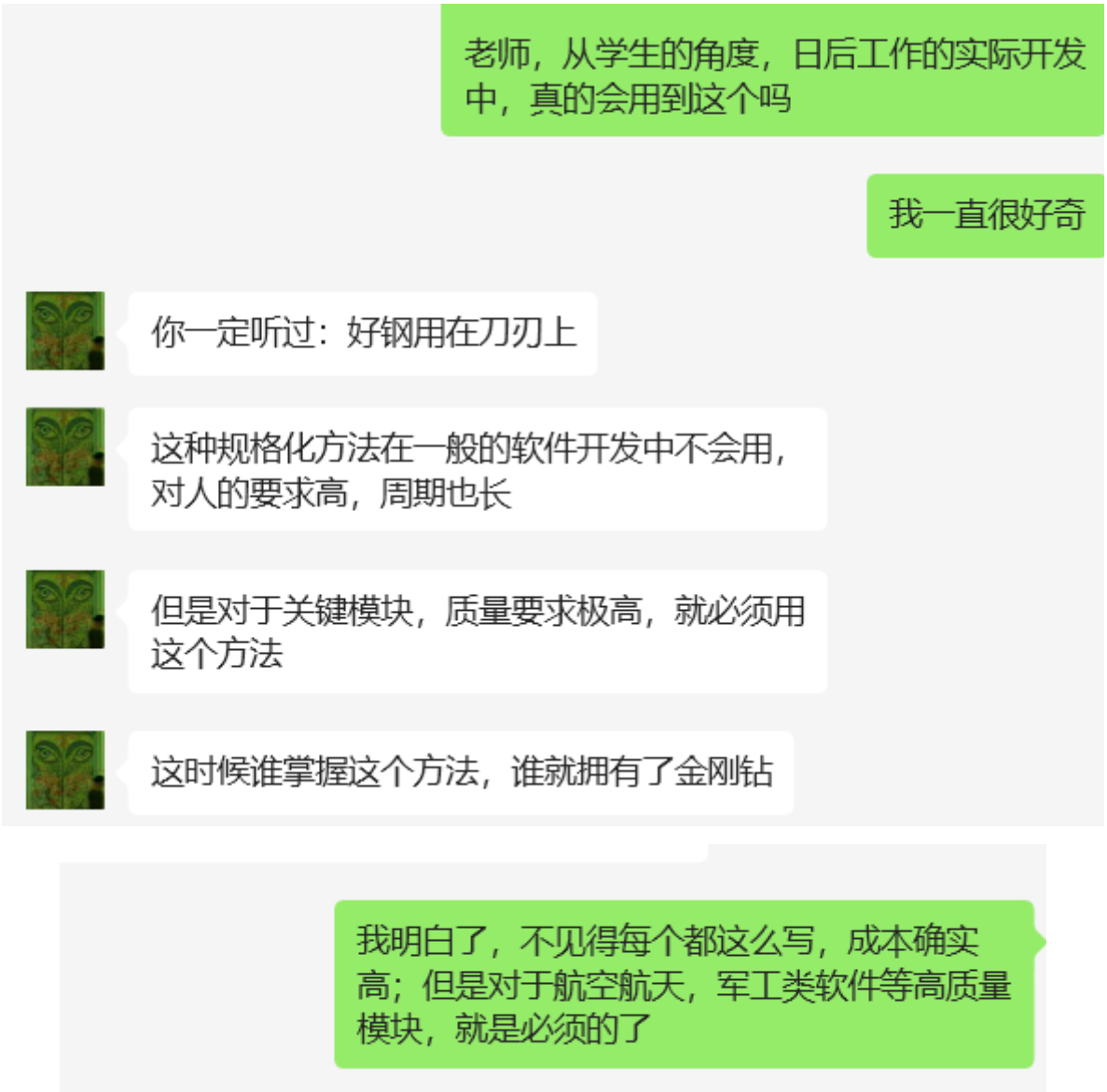
现在已经一目了然，这个方法就是要返回**所有以id对应Person为起点的路径长度>=4的社交环路中边权和的最小值**

写在最后——笔者寄语

相信大家对之前的12个问题已经有所体会，看完本篇文章，笔者希望大家对后10个问题心里都有答案，因为笔者已经详细解释过。至于第1个问题和第2个问题，笔者有如下话要和大家倾心分享。

作为上一届过来人，在结束U1U2单元的洗礼后，恰逢OS压力增大，笔者在U3最后一次作业中还阳性了一周，差点没有完成有效作业，不过就OO而言最后结局依然不错,不然也没有资格和能力为大家写下这篇推送，希望以自己的故事激励大家。笔者衷心希望大家能耐心学透JML的语法，熟练阅读甚至自己编写JML，不要急于写题目或者自己不看，直接问别的同学这个方法是什么意思。即使你以后不会用到JML，这段时间的训练对你的代码逻辑，单元测试思想和数理逻辑能力都是很大的提升，甚至还会让你学到一些算法小知识。

U3的思维量和U1、U2相比略小，希望同学能耐得住寂寞。从实际开发的角度，对于JML的使用，不会每个方法每个类都写JML，但是对于重要模块，JML的地位还是很高的。课程组出题让大家每个方法都阅读JML，是为了给大家更多的训练机会，即使是这样，在笔者简化JML之后，给大家的负担也已经较相对少（不信大家可以看看往届作业的JML）。JML在历届曾受到一些质疑，有些同学不明白为何要学习JML，加上大二下课业压力较大，部分同学情绪起伏较大，对JML甚至课程组产生排斥心理。在这里，笔者祝愿大家在U3能有所收获，保持身体和心理的健康，这也是笔者和课程组的夙愿，我们都希望大家越来越好。以下是笔者作为一名学生和吴际老师关于JML的对话，但愿能给大家一些启发。



是的。

> 旺仔：我明白了，不见得每个都这么写，成本确实高；但是对于航空航天，军工类软件等高质...

本质上，北航计算机学院是要培养能拿住金刚钻的人才，但确实不见得每个人将来都有机会拿金刚钻