JML(Level 0)使用手册

JML(Java Modeling Language) 是用于对 Java 程序进行规格化设计的一种表示语言。JML 是一种行为接口规格语言（Behavior Interface Specification Language，BISL），基于 Larch 方法构建。BISL 提供了对方法和类型的规格定义手段。所谓接口即一个方法或类型外部可见的内容。JML 主要由 Leavens 教授在 Larch 上的工作，并融入了 Betrand Meyer， John Guttag 等人关于 Design by Contract 的研究成果。近年来，JML 持续受到关注，为严格的程序设计提供了一套行之有效的方法。通过 JML 及其支持工具，不仅可以基于规格自动构造测试用例，并整合了 SMT Solver 等工具以静态方式来检查代码实现对规格的满足情况。

一般而言，JML 有两种主要的用法：

（1）开展规格化设计。这样交给代码实现人员的将不是可能带有内在模糊性的自然语言描述，而是逻辑严格的规格。

（2）针对已有的代码实现，书写其对应的规格，从而提高代码的可维护性。这在遗留代码的维护方面具有特别重要的意义。

JML 的设计考虑到了未来扩展需要，把语言分成了几个层次。其中 level 0 是最核心的语言特征，要求所有的 JML 工具都要支持。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。建议同学们通过 http://www.jmlspecs.org/ 来了解更多细节。

1. 注释结构

JML 以 javadoc 注释的方式来表示规格，每行都以 @ 起头。有两种注释方式，行注释和块注释。其中行注释的表示方式为 `//@annotation`，块注释的方式为 `/* @ annotation @*/`。按照 Javadoc 习惯， JML 注释一般放在被注释成分的紧邻上部，如下面的例子所示。其中有效的 Java 代码为 line1，line 3，line 15，line18 和 line19。

```
1  package org.jmlspecs.samples.jmlrefman;                // line 1
2                                                         // line 2
3  public abstract class IntHeap {                        // line 3
4                                                         // line 4
5      //@ public model non_null int [] elements;        // line 5
6                                                         // line 6
7      /*@ public normal_behavior                        // line 7
8        @   requires elements.length >= 1;              // line 8
9        @   assignable \nothing;                        // line 9
10       @   ensures \result                             // line 10
11       @        == (\max int j;                        // line 11
12       @               0 <= j && j < elements.length;  // line 12
13       @               elements[j]);                   // line 13
14       @*/                                             // line 14
15     public abstract /*@ pure @*/ int largest();       // line 15
16                                                        // line 16
17     //@ ensures \result == elements.length;          // line 17
18     public abstract /*@ pure @*/ int size();          // line 18
19 };                                                    // line 19
```

在上面的例子中，定义了一个抽象类 `IntHeap` 。该类提供了两个抽象方法，`largest()` 和 `size()`。第 15 行和第 18 行的 JML 规格表示这两个方法都是纯粹查询方法（`/*@ pure @ */`），即方法的执行不会有任何副作用。这两个方法的规格必须建立在 `IntHeap` 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 `IntHeap` 所管理的数据规格，如第 5 行所示。其中的 `model` 表示后面的 `int[] elements` 仅仅是规格层次的描述，并不是这个类的声明组成部分，此外也不意味该类的实现人员必须提供这样的属性定义，`non_null` 的意义是 `elements` 这个数组对象引用不能为 `null`。有了这个基础， 第 17 行给出了 `size` 方法的后置条件 (post-condition)，即任何时候该方法的执行都会返回 `IntHeap` 中存储的元素个数(`elements.length`)，其中的 `\result` 是 JML 的关键词，表示方法的执行返回结果。`largest` 的规格通过从第 7 行到第 14 行的注释块来定义，包括三个部分：

（1）`requires` 子句定义该方法的前置条件(pre-condition)，`elements.length>=1`，即 `IntHeap` 中管理着至少一个元素；

（2）副作用范围限定，`assignable` 列出这个方法能够修改的类成员属性，`\nothing` 是个关键词，表示这个方法不对任何成员属性进行修改，所以是一个 `pure` 方法。

（3）`ensures` 子句定义了后置条件，即 `largest` 方法的返回结果等于 `elements` 中存储的所有整数中的最大的那个(`\max` 也是一个关键词)。

需要注意的是，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。相比较而言，largest 方法的规格复杂，而 `size` 方法的规格则相对简略。在 JML 中对应着两类规格写法，前者适用于前置条件不是恒真的场景，后者则适用于无需描述其前置条件的场景。

最后还要补充说明一下规格变量的声明。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。针对上面的例子，如果是静态规格变量，则声明为 `//@public static model non_null int []elements`；如果是实例规格变量，则可声明为 `//@public instance model non_null int []elements`。

2. JML表达式

JML 的表达式是对 Java 表达式的扩展，新增了一些操作符和原子表达式。同样 JML 表达式中的操作符也有优先级的概念。请参见 JML 语言手册 12.3 节 (Expression) 获得完整的优先级列表。需要提醒的是，JML 相对于 Java 新增的表达式成分仅用于 JML 中的断言 (assertion) 语句和其他相关的注释体。特别需要提醒，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。

2.1 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`，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，即使方法在执行过程中对 `v` 指向的 `HashMap` 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。

`\nonnullelements(container)` 表达式：表示 `container` 对象中存储的对象不会有 `null`，等价于下面的断言，其中 `\forall` 是 JML 的关键词，表示针对所有 `i`。

```
container != null &&
(\forall int i; 0 <= i && i < container.length;
                container[i] != 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`。

2.2 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。`(\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])`，意思是针对任意 `0 <= i < j < 10，a[i] < a[j]`。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`。

`\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 `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` 。

`\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`。

`\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` 需要满足的约束条件。从逻辑上来看，该表达式也等价于 `(\sum T x; R(x) && P(x); 1)`。

2.3 集合表达式

集合构造表达式：可以在 JML 规格中构造一个局部的集合（容器），明确集合中可以包含的元素。`new JMLObjectSet {Integer i | s.contains(i) && 0 < i.intValue() }` 表示构造一个 `JMLObjectSet` 对象，其中包含的元素类型为 `Integer`，该集合中的所有元素都在容器集合 `s` 中出现（注：该容器集合指 Java 程序中构建的容器，比如 `ArrayList`），且整数值大于 `0` 。集合构造表达式的一般形式为：`new ST {T x | R(x) && P(x)}`，其中的 `R(x)` 对应集合中 `x` 的范围，通常是来自于某个既有集合中的元素，如 `s.has(x)` ，`P(x)` 对应 `x` 取值的约束。

2.4 操作符

JML 表达式中可以正常使用 Java 语言所定义的操作符，包括算术操作符、逻辑预算操作符等。此外，JML 专门又定义了如下四类操作符。

(1) 子类型关系操作符：`E1<:E2`，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。需要指出的是，任意一个类 `X` ，都必然满足 `X.TYPE<:Object.TYPE`。

(2) 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。

(3) 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`。

(4) 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。

3. 方法规格

方法规格是 JML 的重要内容，本手册仅涉及最基础的部分，而不引述 JML 扩展部分的内容。方法规格的核心内容包括三个方面，前置条件、后置条件和副作用约定。其中前置条件是对方法输入参数的限制，如果不满足前置条件，方法执行结果不可预测，或者说不保证方法执行结果的正确性；后置条件是对方法执行结果的限制，如果执行结果满足后置条件，则表示方法执行正确，否则执行错误。副作用指方法在执行过程中对输入对象或 `this` 对象进行了修改（对其成员变量进行了赋值，或者调用其修改方法）。课程区分两类方法：全部过程和局部过程。前者对应着前置条件恒为真，即可以适应于任意调用场景；后者则提供了非恒真的前置条件，要求调用者必须确保调用时满足相应的前置条件。从设计角度，软件需要适应用户的所有可能输入，因此也需要对不符合前置条件的输入情况进行处理，往往对应着异常处理。从规格的角度，JML 区分这两种场景，分别对应正常行为规格 (normal_behavior) 和异常行为规格 (expcetional_behavior)。

- 前置条件 (pre-condition)

前置条件通过 `requires` 子句来表示：`requires P;`。其中 `requires` 是 JML 关键词，表达的意思是“要求调用者确保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。如果设计者想要表达或的逻辑，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：`requires P1||P2;`。

- 后置条件 (post-condition)

后置条件通过 `ensures` 子句来表示：`ensures P;`。其中 `ensures` 是 JML 关键词，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，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有所并列 `ensures` 子句的要求。如果设计者想要表达或的逻辑，这应该在在一个 `ensures` 子句中使用逻辑或 （`||`）操作符来表示相应的约束场景：`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` 和 `\everything` 是两个关键词，前者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；后者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，如果是多个则通过逗号分隔，如 `@ assignable elements, max, min;`。

```
注1：JML不允许在副作用约束子句中指定规格声明的变量数据，因为这样的声明只是为了描述规格，并不意味实现者一定要实现这样的数据。
注2：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@*/ 来注释一个类的私有成员变量，表示在规格中可以直接使用，从而调用者可见。
```

设计中会出现某些纯粹访问性的方法，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：

```
public /*@ pure @*/ String getName();
//@ ensures \result == bachelor || \result == master;
public /*@ pure @*/ int getStatus();
//@ ensures \result >= 0;
public /*@ pure @*/ int getCredits();
```

针对上面的三个例子，`getName` 甚至不需要做任何限定，是一种极简的场景；`getStatus` 例子则限定了返回值 `\result` 要么 `==bachelor`，要么 `==master`；`getCredits` 的例子则限定了返回值必须大于等于 `0`：`\result >=0`。

在方法规格中，有些前置条件可以引用 `pure` 方法返回的结果：

```
/*@ requires c >= 0;
@ ensures getCredits() == \old(getCredits()) + c;
@*/
public void addCredits(int c);
```

有时候，前置条件或后置条件需要对不止一个变量进行约束，往往是需要对一个容器中的所有元素进行约束，这时就需要使用 `\forall` 或者 `\exists` 表达式：

```
/*@ requires size < limit && !contains(elem);
@ ensures \result == true;
@ ensures contains(elem);
@ ensures (\forall int e;
@ e != elem;
@ contains(e) <==> \old(contains(e)));
@ ensures size == \old(size) + 1;
@*/
public boolean add(int elem) {/*...*/}
```

上面的这个例子要求调用 `add` 方法之前，调用者必须确保当前对象管理的元素（整数）数目不能超过限制(`limit`)，同时不能重复加入相同的整数（`!contains(elem)`）。该方法的规格有三个并列的后置条件，`ensures contains(elem)` 要求 `add` 方法一定要把参数 `elem` 对应的整数加入到容器中；`ensures size == \old(size) + 1`要求容器中管理的整数数目增加一个；中间的那个后置条件要求原来容器中包含的整数仍然在容器中，使用 `\forall`表达式来表示。

更进一步，假设带有规模限制的整数容器还提供了一个 `remove` 方法，这个方法要求：（1）如果输入参数在容器中，则移除该整数；（2）任何情况下，都不能移除容器中不等于输入参数的任何整数。要对这个方法的后置条件进行设计，需要从两个方面进行限制，一个是方法运行结果中不包含输入的整数，同时容器原来不等于输入参数的整数仍然还在容器中；另一方面需要对象容器的规模进行限制，如果输入参数在容器中，则容器存储的整数数目减少一个，否则保持不变。按照这个设计，可以得到如下完整的方法规格：

```
/*@ ensures !contains(elem);
@ ensures (\forall int e;
@ e != elem;
@ contains(e) <==> \old(contains(e)));
@ ensures \old(contains(elem)) ==> size == \old(size) - 1;
@ ensures !\old(contains(elem)) ==> size == \old(size);
@*/
public void remove(int elem) {/*...*/}
```

如前所述，为了有效的区分方法的正常功能行为和异常行为，JML 提供了这两类行为的区分机制，可以明确按照这两类行为来分别描述方法的规格，如下所示：

```
/*@ public normal_behavior
@ requires z <= 99;
@ assignable \nothing;
@ ensures \result > z;
@ also
@ public exceptional_behavior
@ requires z < 0;
@ assignable \nothing;
@ signals (IllegalArgumentException e) true;
@*/
public abstract int cantBeSatisfied(int z) throws IllegalArgumentException;
```

其中 `public normal_behavior` 表示接下来的部分对 `cantBeSatisfied(int z)` 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 `public exceptional_behavior` 下面所定义的规格。其中的 `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。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。

我们仔细看上面的例子，实际上存在逻辑矛盾，即正常功能的前置条件蕴含了异常功能的前置条件（`{z <= 99}` 与 `{z < 0}` 有交集），因此对于这个例子的规格而言，任何实现都不可能满足该规格。这样的矛盾规格是严重的设计错误，要避免。作为一种重要的设计检查原则，同一个方法的正常功能前置条件和异常功能前置条件一定不重叠。对于上面的例子而言，如果正常功能前置条件修改为 `z >= 0` 就能满足这个要求。可以看出不论是正常功能规格，或者是异常功能规格，都包括前置条件、后置条件和副作用声明。不同的是，异常功能规格中，后置条件常常表示为抛出异常，使用 `signals` 子句来表示。

- signals 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 `z < 0`，就一定会抛出异常 `IllegalArgumentException`。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。

有时候，为了更加明确的区分异常，会针对输入参数的取值范围抛出不同的异常，从而提醒调用者进行不同的处理。这时可以使用多个 exceptional_behavior：

```
public abstract class Student {
/** A specification that can't be satisfied. */
  //@ public model non_null int[] credits;
  /*@ normal_behavior
    @ requires z >=0 && z <= 100;
    @ assignable \nothing;
    @ ensures \result == credits.length;
```

```
    @ also
    @ exceptional_behavior
    @ requires z < 0;
    @ assignable \nothing;
    @ signals_only IllegalArgumentException;

    @ also
    @ exceptional_behavior
    @ requires z > 100;
    @ assignable \nothing;
    @ signals_only OverFlowException;
    @*/
public abstract int recordCredit(int z) throws IllegalArgumentException,
OverFlowException;
}
```

上面这个例子是针对 `Student` 类提供的 `recordCredit(int z)` 方法，从规格角度定义了一个规格数据 `int[] credits`，并提供了三个功能规格，使用两个 `also` 进行了分隔。注意看三个功能规格的 `requires` 子句，在一起覆盖了方法输入参数的所有看取值范围，而且彼此没有交叉。这是功能规格设计的基本要求，同学们一定要小心这一点。其中两个异常功能规格使用 signals_only 子句分别抛出相应的异常。需要指出的是，在异常功能规格中，除了抛出异常，也一样可以正常使用 ensures 子句来描述方法执行产生的其他结果。

4. 类型规格

类型规格指针对 Java 程序中定义的数据类型所设计的限制规则，一般而言，就是指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。不失一般性，一个类型的成员要么是静态成员 (static member)，要么是实例成员 (instance member)。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。

JML 针对类型规格定义了多种限制规则，从课程的角度，我们主要涉及两类，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 invariant

不变式（invariant）是要求在所有**可见状态**下都必须满足的特性，语法上定义 `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 (visible state)** 是一个特别重要的概念。下面所述的几种时刻下对象 `o` 的状态都是可见状态：

- 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
- 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
- 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
- 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
- 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
- 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

由上面的定义可知，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。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。换句话说，在方法执行期间，对象的不变式有可能不满足。因此，类型规格强调在任意可见状态下都要满足不变式。

```
public class Path{
    private /*@spec_public@*/ ArrayList <Integer> seq_nodes;
    private /*@spec_public@*/ Integer start_node;
    private /*@spec_public@*/ Integer end_node;
    /*@ invariant seq_nodes != null &&
      @ seq_nodes[0] == start_node &&
      @ seq_nodes[seq_nodes.length-1] == end_node &&
      @ seq_nodes.length >=2;
      @*/
}
```

如上面的例子所示，`Path` 类的不变式定义了 `seq_nodes` 不能为 `null` ，且任意一个 `path` 对象至少包括两个节点，一个起始节点 (`start_node`) 和一个终止节点 (`end_node`) 。一个类可以包括多个不变式，相互独立。如果一个对象的可见状态不满足不变式，则称该对象的状态有错。实际工程中，如果一个类中有两个产生逻辑矛盾的不变式（即二者不可能同时为真），则出现了规格设计缺陷。需要指出的是，不变式中可以直接引用 `pure` 形态的方法。

对应类成员变量有静态和非静态之分，JML 区分两类不变式，静态不变式 (static invariant) 和实例不变式 (instance invariant)。其中静态不变式只针对类中的静态成员变量取值进行约束，而实例不变式则可以针对静态成员变量和非静态成员变量的取值进行约束。可以在不变式定义中明确使用 `instance invariant` 或 `static invariant` 来表示不变式的类别。

- 状态变化约束 constraint

对象的状态在变化时往往也许满足一些约束，这种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 ServiceCounter{
    private /*@spec_public@*/ long counter;
    //@ invariant counter >= 0;
    //@ constraint counter == \old(counter)+1;
}
```

类 `ServiceCounter` 拥有一个成员变量 `counter` ，包含一个不变式和一个状态变化约束。不变式指出 `counter >= 0` ，而 constraint 不同，约束每次修改 `counter` 只能加 `1` 。虽然这个约束可以在可能对 `counter` 进行修改的方法中通过后置条件来表示，但是每个可能修改 `counter` 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。

和不变式一样，JML 也根据类的静态成员变量区分了两类约束：static constraint 和 instance constraint。其中static constraint 只涉及类的静态成员变量，而 instance constraint 则可以涉及类的静态成员变量和非静态成员变量。同样，也可以在规格中通过关键词来明确加以区分：`static constraint P` 和 `instance constraint P` 。

- 方法与类型规格的关系

如果一个类是不可变类，其实就没必要定义其不变式，只需要在构造方法中明确其初始状态应该满足的后置条件即可。当然，也可以反过来，定义不变式，而不定义构造方法的后置条件。事实上，在大部分情况下，一个类有几种不同类别的方法：静态初始化（不是方法，但也是一种行为）、有状态静态方法、有状态构造方法、有状态非静态方法。下表给出了两类不变式与这些方法的关系：

	静态成员初始化	有状态静态方法	有状态构造方法	有状态非静态方法
static invariant	建立	保持	保持	保持
instance invariant	(无关)	(无关)	建立	保持，除非是finalizer方法

注："建立"的含义是静态成员建立了满足相应不变式的类或对象状态。"保持"的含义是如果方法执行前不变式满足，执行后还应该满足相应的不变式。

同理，JML 也对 constraint 与方法之间的关系进行了约定：

	静态成员初始化	有状态静态方法	有状态构造方法	有状态非静态方法
static constraint	(无关)	遵从	遵从	遵从
instance constraint	(无关)	(无关)	(无关)	遵从

注："遵从"的含义是成员变量的当前取值和上一个取值之间的关系满足 constraint 的规定，即"遵从规定"。

5. 一个完整例子

```
public class Student {
    private /*@ spec_public @*/ String name;
    //@ public invariant credits >= 0;
    private /*@ spec_public @*/ int credits;
    /*@ public invariant credits < 180 ==> !master &&
      @ credits >= 180 ==> master;
      @*/
    private /*@ spec_public @*/ boolean master;

    /*@ requires sname != null;
      @ assignable \everything;
      @ ensures name == sname && credits == 0 && master == false;
      @*/
    public Student (String sname) {
        name = sname;
        credits = 0;
        master = false;
    }

    /*@ requires c >= 0;
      @ ensures credits == \old(credits) + c;
      @ assignable credits, master;
      @ ensures (credits >= 180) ==> master
      @*/
    public void addCredits(int c) {
        updateCredits(c);
        if (credits >= 180) {
            changeToMaster();
        }
```

```
30      }
31
32      /*@ requires c >= 0;
33        @ ensures credits == \old(credits) + c;
34        @ assignable credits;
35        @*/
36      private void updateCredits(int c) {
37          credits += c;
38      }
39
40      /*@ requires credits >= 180;
41        @ ensures master;
42        @ assignable master;
43        @*/
44      private void changeToMaster() {
45          master = true;
46      }
47
48      /*@ ensures this.name == name;
49        @ assignable this.name;
50        @*/
51      public void setName(String name) {
52          this.name = name;
53      }
54
55      /*@ ensures \result == name;
56        @*/
57      public /*@ pure @*/ String getName() {
58          return name;
59      }
60  }
```

上面是一个针对 `Student` 类的完整例子。该类提供了三个私有成员变量：`private String name`、`private int credits` 和 `private boolean master`。如前所述，因为这三个是私有成员，规格中是不可见的，JML 通过使用 `/*@ spec_public @*/` 来解决这个问题。

对比看一下 `addCredits(c)` 方法和 `updateCredits(c)` 方法的规格，二者的共同效果是 `credits == \old(credits) + c`（如果 `c >= 0`）。`(credits >= 180) ==> master` 给出了 `addCredits(c)` 方法的差异化功能，即不仅仅改变学分，还会判断学分情况，从而及时改变 `master` 变量的取值。如果忽略掉了这个后置条件，实际上就隐藏了设计错误。注意方法的规格不对什么情况调用什么方法做出规定（这实际上是非常细节的实现问题），虽然 JML 提供了 callable 子句来规定一个方法在执行过程可能会调用哪些方法，但是并不会对方法执行的后置条件产生实质影响。

针对从给定方法实现中来提取规格的场景，实践中往往会有方法调用，这时需要把被调用方法的后置条件附加到当前方法，并增加关于该方法被调用条件的约束。可以很清楚的看到，`addCredits` 规格合并了 `changeToMaster` 的后置条件，但增加了相应的调用条件约束（`credits >= 180`）。

通过本单元的学习，同学们会感受到通过规格设计，可以更容易获得简洁和职责单一的设计和实现（复杂方法会导致直接写不清楚相应的规格）。另一方面，大家经过实验和作业训练会发现，在很多情况下设计和撰写规格要比编写代码难。一旦规格确定了，其实实现代码就变成了一个相对简单的事情，除非涉及复杂的算法要求。

预告一下：本单元仍然采用增量式作业方式。从代码实现的算法要求和功能要求角度来看，显著的比第一单元和第二单元都有所降低。规格的理解、设计实现和测试能力是本单元训练的重点，自然评测机还会一如既往的为大家服务。祝大家有个愉快而充实的规格化设计之旅。