



# Introducción al lenguaje de especificación JML

**Elena Hernández Pereira**  
**Óscar Fontenla Romero**

**Tecnología de la Programación**  
– Octubre 2006 –

*Departamento de Computación*  
*Facultad de Informática*  
*Universidad de A Coruña*

## Bibliografía

- JML Home Page:
  - <http://www.jmlspecs.org>
- Documento sobre diseño preliminar de JML:
  - [http://www.cs.iastate.edu/~leavens/JML/prelimdesign/pr elimdesign\\_toc.html](http://www.cs.iastate.edu/~leavens/JML/prelimdesign/pr elimdesign_toc.html)
- Manual de referencia:
  - <http://www.cs.iastate.edu/~leavens/JML/jmlrefman/>
- Código libre JML y herramientas (incluye el compilador):
  - <http://sourceforge.net/projects/jmlspecs/>

## Introducción (I)

- JML (Java Modeling Language): lenguaje de especificación para programas Java
- Las especificaciones se introducen en los ficheros fuente de Java
- Las especificaciones JML hacen una descripción formal del comportamiento de clases y métodos
- Añadir especificaciones JML a un programa ayuda a:
  - Comprender qué función debe realizar un método o una clase
  - Encontrar errores en los programas: comprueba si un método cumple su especificación cada vez que se ejecuta

## Empezando a especificar (I)

- **Aserción** o **anotación**: cláusula lógica insertada en un programa
- Objetivo de una aserción: representar una condición que debe ser cierta en un punto del programa
- Esta condición puede ser una condición típica de Java, escrita usando los operadores lógicos && y ||
- La cláusula de JML `assert` sirve para indicar estas condiciones o aserciones en cualquier lugar de un programa
  - Si la condición no fuese cierta entonces el programa se detiene y se visualizara un error en tiempo de ejecución indicando que dicha aserción no se cumple.

## Empezando a especificar (II)

- Ejemplo:

```
// Este código almacena en z el mayor de x e y
if (x >= y)
    z = x;
else
    z = y;
/*@ assert z >= x && z >= y;
```

- La condición que debe ser cierta se indica a continuación de la cláusula **assert**, y debe estar terminada con el carácter **;**
- Delante de la cláusula aparecen los caracteres **//@** que indican al compilador que la línea contiene una cláusula de JML
- *¿Es suficiente la condición del assert?:* **NO**

## Empezando a especificar (III)

- Si se cambia el código por:

```
// Este código almacena en z el mayor de x e y
z = x*x + y*y;
/*@ assert z >= x && z >= y;
```

- La condición se haría cierta, y sin embargo z no contiene el mayor de x e y
- *¿Cómo mejorar la aserción para solucionar el problema?*

## Empezando a especificar (IV)

- Posible solución:

```
// Este código almacena en z el mayor de x e y

if (x >= y)
    z = x;
else
    z = y;

/*@ assert z >= x && z >= y && (z == x || z == y);
```

- Hay dos aserciones que reciben un nombre particular:
  - Aserción *precondición*: condición que debe cumplirse para que un método pueda ejecutarse
  - Aserción *postcondición*: condición que debe cumplirse al acabar la ejecución de un método

## Empezando a especificar (V)

- **Ejemplo**: para poder hacer una operación de división
  - *Precondición*: divisor distinto de cero
  - *Postcondición*:  $\text{dividendo} == \text{divisor} * \text{cociente} + \text{resto}$
- Cláusulas JML para la precondición y postcondición de un método: *requires* y *ensures*
  - Deben ir antes de la declaración del método
- JML recoge en la variable **\result** el valor devuelto por el método

## Empezando a especificar (VI)

- **Ejemplo:** postcondición para el método del máximo

```
/* Método maximo:
   Devuelve el mayor de x e y */

//@ ensures \result >= x && \result >= y && (\result == x || \result == y);
public static int maximo(int x, int y)
{
    int z;

    if (x >= y)
        z = x;
    else
        z = y;

    return z;
}
```

## Empezando a especificar (VII)

- También es posible indicar varias precondiciones o postcondiciones: **todas ellas se deben cumplir**
- Ejemplo

```
/* Método maximo: Devuelve el mayor de x e y */
//@ requires true;
//@ ensures \result >= x && \result >= y;
//@ ensures \result == x || \result == y;
public static int maximo(int x, int y)
{
    int z;
    if (x >= y)
        z = x;
    else
        z = y;
    return z;
}
```

## Empezando a especificar (VIII)

- Si hay varias líneas de aserciones se pueden emplear las marcas `/*@` y `@*/`
- Ejemplo:

```
/* Método maximo: Devuelve el mayor de x e y */
/*@ requires true;
   @ ensures \result >= x && \result >= y;
   @ ensures \result == x || \result == y;
   @*/
public static int maximo(int x, int y)
{
    int z;
    if (x >= y)
        z = x;
    else
        z = y;
    return z;
}
```

## Expresiones de especificación (I)

- **Expresión de especificación:** expresión similar a un predicado de lógica que se evalúa a cierto o falso
  - Permiten describir condiciones que deben ser ciertas en determinados puntos durante la ejecución de un programa
  - Permiten especificar el comportamiento de los métodos y también de la clase
  - Se puede indicar con una condición que utilice:
    - Los operadores lógicos de Java como `&&`, `||`, `!`, etc.,
    - Los operadores de comparación `==`, `!=`, `<`, `<=`, etc.
    - Cuantificadores más comunes (universal, existencial, etc.)
    - Otros operadores lógicos

## Expresiones de especificación (II)

- Operadores y cuantificadores:

<b>Operadores</b>	<code>==&gt;</code>	Implicación
	<code>&lt;==</code>	Contraimplicación
	<code>&lt;==&gt;</code>	Equivalencia
	<code>&lt;!=&gt;</code>	No equivalencia
<b>Cuantificadores</b>	<code>\forall</code>	Para todo
	<code>\exists</code>	Existe
	<code>\sum</code>	Suma
	<code>\product</code>	Producto
	<code>\num_of</code>	Número de
	<code>\max</code>	Máximo
	<code>\min</code>	Mínimo

## Expresiones de especificación (III)

- Todos los cuantificadores tienen la misma sintaxis:
  - Definición de una variable ligada
  - Rango de la variable
  - Expresión sobre la que se aplica el cuantificador
- Ejemplos:

Expresión	Equivale a
<code>(\sum int I; 0 &lt;= I &amp;&amp; I &lt; 5; I)</code>	$0 + 1 + 2 + 3 + 4$
<code>(\product int I; 0 &lt; I &amp;&amp; I &lt; 5; I)</code>	$1 * 2 * 3 * 4$
<code>(\max int I; 0 &lt;= I &amp;&amp; I &lt; 5; I)</code>	4
<code>(\min int I; 0 &lt;= I &amp;&amp; I &lt; 5; I-1)</code>	-1
<code>(\num_of int I; 0 &lt;= I &amp;&amp; I &lt; 5; I*2 &lt; 6)</code>	3

## Expresiones de especificación (IV)

- JML también define dos pseudovariantes que pueden ser utilizadas en las postcondiciones de los métodos:
  - **\result**: valor devuelto por el método
  - **\old(E)**: valor que tomaba la expresión E al comenzar la ejecución del método
- En las aserciones no está permitido:
  - Utilizar operadores que modifiquen variables: ++, --, =, +=, y todos los operadores que contienen =
  - Llamadas a métodos que modifiquen variables no locales: métodos con efectos laterales

## Expresiones de especificación (V)

- **Ejercicio 1**: especificación de un método que devuelve la suma de todos los elementos de un array de enteros
  - *Precondiciones*:
    - El array no es nulo (está creado)
  - *Postcondiciones*:
    - Resultado método =  $\sum_{i=0}^{N-1} array[i]$
    - Los elementos del array no se modifican



## Expresiones de especificación (VI)

- **Ejercicio 1:** especificación de un método que devuelve la suma de todos los elementos de un array de enteros

```
/*@
 @ requires array != null;
 @ ensures \result == (\sum int I; 0 <= I && I < array.length; array[I]);
 @ ensures (\forall int I; 0 <= I && I < array.length;
 @         array[I] == \old(array[I]));
 @*/

public static int sumaDeArray(int[] array) {

    // Cuerpo del método

}
```

## Expresiones de especificación (VII)

- **Ejercicio 2:** especificación de un método que inserte un elemento en una posición dada de una matriz sin modificar el resto de elementos

```
/*@ requires matriz != null;
 @ requires Posx >= 0 && Posy >= 0;
 @ requires Posx < matriz.length && Posy < matriz[0].length;
 @ ensures (\forall int I; 0 <= I && I < matriz.length;
 @         (\forall int J; 0 <= J && J < matriz[0].length;
 @         (I==Posx && J==Posy && matriz[I][J]==Dato) ||
 @         matriz[I][J]==\old(matriz[I][J]) ));
 @*/

public static int insertarElemento(int matriz[][], int Posx, int Posy, int Dato)
{

    // Cuerpo del método

}
```

¿Es correcta esta especificación?: NO

## Expresiones de especificación (VIII)

- **Ejercicio 2** (una de las posibles soluciones correctas):  
especificación de un método que inserte un elemento en una posición dada de una matriz sin modificar el resto de elementos

```
/*@ requires matriz != null;
@ requires Posx >= 0 && Posy >= 0;
@ requires Posx < matriz.length && Posy < matriz[0].length;
@ ensures matriz[Posx][Posy] == Dato &&
@     (\forallall int I; 0 <=I && I < matriz.length;
@     (\forallall int J; 0<=J && J < matriz[0].length;
@     (I!=Posx || J!=Posy) ==> matriz[I][J]==\old(matriz[I][J])
@     ));
@*/
public static int insertarElemento(int matriz[][], int Posx, int Posy, int Dato)
{
    // Cuerpo del método
}
```

## Expresiones de especificación (IX)

- Comentarios para aclarar el objetivo de cada aserción:
  - (\* esto es un comentario \*)
  - Son tratados como una aserción más, por tanto deben ir unidas a otras aserciones utilizando el operador &&
  - Siempre serán evaluadas como ciertas
  - Ejemplo:

```
/*@ requires array != null;
@ ensures (* devuelve la suma de todos los elementos del array *)
@     && \result == (\sum int I; 0 <= I && I < array.length; array[I]);
@ ensures (* no modifica el array *)
@     && (\forallall int I; 0 <= I && I < array.length;
@     array[I] == \old(array[I]));
@*/
```

## Expresiones de especificación (X)

- **Ejercicio 3:** especificación de un método que intercambie las posiciones del elemento menor y mayor de un array (no hay elementos repetidos)
  - *Precondiciones:*
    - El array no es nulo (está creado)
    - No hay elementos repetidos
  - *Postcondiciones:*
    - Se intercambian los elementos en las posiciones ***j*** y ***k*** tal que  $\text{array}[j] = \min(\text{array})$  y  $\text{array}[k] = \max(\text{array})$
    - El resto de los elementos del array no se modifican

## Expresiones de especificación (XI)

- **Ejercicio 3:** especificación de un método que intercambie las posiciones del elemento menor y mayor de un array (no hay elementos repetidos)

```
/*@ requires array != null                && (* array no nulo *);
   @ requires (\forall int I; 0 <= I && I < array.length-1;
   @           (\forall int J; I+1 <= J && J < array.length;
   @           array[I] != array[J] ))    && (* no hay elementos repetidos *);
   @ ensures (* postcondicion *) &&
   @ (\exists int POSMAX; 0 <= POSMAX && POSMAX < array.length;
   @ (\exists int POSMIN; 0 <= POSMIN && POSMIN < array.length;
   @   \old(array[POSMAX]) == (\max int I; 0 <= I && I < array.length; \old(array[I])) &&
   @   \old(array[POSMIN]) == (\min int I; 0 <= I && I < array.length; \old(array[I])) &&
   @   array[POSMIN] == \old(array[POSMAX]) &&
   @   array[POSMAX] == \old(array[POSMIN]) &&
   @   (\forall int J; 0 <= J && J < array.length && J != POSMAX && J != POSMIN;
   @     array[J] == \old(array[J]) )
   @ ));
   @*/
public static void intercambiaMinMax(int[] array) {
    // Cuerpo del método
}
```

## Especificación de varios comportamientos

### (I)

#### ■ Ejercicio:

```
/* Método intDivide()
   Devuelve: el cociente de la division entera entre dividendo y
   divisor, siempre que el divisor sea mayor que cero. */

/*@
  @ requires divisor > 0;
  @ ensures divisor*\result <= dividendo
  @      && divisor*(\result +1) > dividendo;
  @*/
public static int intDivide(int dividendo, int divisor)
{
  // Cuerpo del método
}
```

## Especificación de varios comportamientos

### (II)

- Si se quiere modificar el método:
  - Además del comportamiento anterior, también devuelva un cero cuando el divisor sea cero
- Esto supone que este método ahora tiene dos comportamientos diferentes:
  - Cuando el divisor es positivo
  - Cuando el divisor es cero
- Cada comportamiento diferente se indica con la cláusula **normal\_behavior**
- Si hay varios comportamientos se deben unir mediante la cláusula **also**

## Especificación de varios comportamientos (III)

### ■ Ejemplo:

```
/* Método intDivide()
   Devuelve:
   - El cociente de la division entera entre dividendo y divisor,
   si el divisor es mayor que cero.
   - Cero, si el divisor es cero. */
/*@ public normal_behavior
   @ requires divisor > 0;
   @ ensures divisor*\result <= dividendo
   @       && divisor*(\result+1) > dividendo;
   @
   @ also
   @ public normal_behavior
   @ requires divisor == 0;
   @ ensures \result == 0;
   @*/
public static int intDivide(int dividendo, int divisor)
{
    // Cuerpo del método
}
```

## Especificación de excepciones (I)

- Para especificar un comportamiento excepcional:  
cláusula **exceptional\_behavior**
  - En este caso el método devuelve una excepción
  - Se genera la excepción usando **signals**
- Se pueden combinar comportamientos normales y excepcionales empleando la cláusula **also**

## Especificación de excepciones (II)

### ■ Ejemplo:

```
/* Método intDivide()
   Devuelve:
   - El cociente de la division entera entre dividendo y divisor,
   si el divisor es mayor que cero.
   - La excepción ArithmeticException, si el divisor es cero. */
/*@ public normal_behavior
   @ requires divisor > 0;
   @ ensures divisor*\result <= dividendo
   @          && divisor*(\result+1) > dividendo;
   @ also
   @ public exceptional_behavior
   @ requires divisor == 0;
   @ signals (ArithmeticException);
   @*/
public static int intDivide(int dividendo, int divisor)
  throws ArithmeticException
{
  // Cuerpo del método
}
```

## Especificaciones públicas y privadas (I)

### ■ Especificación pública:

- Sólo pueden hacer referencia a variables o métodos accesibles por el usuario (públicas) y a los argumentos de los métodos
- Se indican empleando la cláusula **public**

### ■ Especificación privada:

- Además de las variables y métodos públicos también pueden incluir los privados
- Están dirigidas al propio programador
- Se indican empleando la cláusula **private**

## Especificaciones públicas y privadas (II)

- Las cláusulas `public` y `private` preceden a:
  - `normal_behavior`
  - `exceptional_behavior`
- Si no se indica nada entonces se asume que es una especificación pública

■ Ejemplo:

```
/*@ public normal_behavior
@   ...
@ also
@ private normal_behavior
@   ...
@*/
public void miMetodo()
{
    ...
}
```

## Especificaciones externas (I)

- Es posible realizar las especificaciones en un fichero distinto del `.java`
- Posibles extensiones: `.jml`, `.refines-java`, `refines-spec`, etc.
- Hay que incluir en el fichero `.java` el nombre del fichero de especificación:
  - Es necesario emplear la cláusula `refine`
  - Ejemplo: `//@ refine "especificacion.jml";`

## Especificaciones externas (II)

- Ejemplo (sin error de especificación):

```
//@ refine "prueba.jml";

public class prueba
{
    int dato;

    public int copiar(int a)
    {
        return 2*a;
    }
}
```

*prueba.java*

```
public class prueba
{

    //@ requires a > 0;
    //@ ensures \result == 2*a;
    public int copiar(int a);
}
```

*prueba.jml*

Introducción a JML

31

## Especificaciones externas (III)

- Ejemplo (con error de especificación):

```
//@ refine "prueba.jml";

public class prueba
{
    int dato;

    public int copiar(int a)
    {
        return 3*a;
    }
}
```

*prueba.java*

```
public class prueba
{

    //@ requires a > 0;
    //@ ensures \result == 2*a;
    public int copiar(int a);
}
```

*prueba.jml*

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
by method prueba.copiar regarding specifications at
File "prueba.jml", line 5, character 23 when
'a' is 2
'\result' is 6
at
prueba.checkPost$copiar$prueba(prueba.java:245)
at prueba.copiar(prueba.java:322)
at p.main(p.java:11)
```

Introducción a JML

32



## Invariantes y bucles (I)

- JML distingue dos tipos de invariantes: de bucle y de clase
- Invariantes de bucle:
  - Predicado cierto durante *todas* las iteraciones del bucle
  - Además de la invariante un bucle bien construido tiene asociado una **cota**:
    - Expresión que se decrementa cada iteración y está acotada inferiormente
    - Asegura que el bucle finaliza en algún momento

## Invariantes y bucles (II)

- Invariante del bucle: cláusula **maintaining**
- Cota del bucle: cláusula **decreasing**
- Estas expresiones deben indicarse antes del bucle
- Son válidas para todos los tipos de bucles de Java
- Ejemplos:

```
/*@ maintaining ....  
/*@ decreasing ....  
for ( .... ; .... ; .... ) {  
  
}  
  
/*@ maintaining ....  
/*@ decreasing ....  
while ( .... ) {  
  
}
```

## Invariantes y bucles (III)

- Ejemplo 1:

```
public long sumaArray (int [] a) {
    long suma = 0;
    int i = a.length-1;

    /*@ maintaining suma == (\sum int J; 0<=J &&
        @                               J<a.length && i<J; a[J]);
        @ decreasing i;
    @*/
    while (i >= 0)
    {
        suma = suma + a[i];
        i = i-1;
    }
}
```

## Invariantes y bucles (IV)

- Ejemplo 2:

```
/*@ maintaining 0 <= i && i<= n;
    @ decreasing n-i;
@*/
for (i=0; i<n; i++) {
    // Aquí estaría el código del for
}
```

## Invariantes de clase (I)

- Invariantes de clase:
  - Predicado que expresa propiedades que son ciertas durante toda la vida de los objetos de una clase
  - Es cierta en los momentos en los que el objeto es estable
    - Está en un estado inestable cuando se está ejecutando alguno de sus métodos
  - Por ello, la comprobación de la invariante se realiza
    - Antes y después de la llamada a un método
    - Por tanto, es como una postcondición a todos los métodos de la clase

## Invariantes de clase (II)

- Se especifica mediante la cláusula **invariant** ó **constraint**:
  - Pueden incluirse en cualquier parte de la clase excepto dentro de un método
- Ejemplo:

```
class EjemploInvariant
{
  ...
  // declaraciones de variables y/o métodos
  ...

  /*@ public invariant ...
    @*/
}
```

## Invariantes de clase (III)

- Diferencias:

- **invariant**: propiedad cierta dentro del "estado actual" de la clase
  - No permite el uso del operador `\old()`
  - Ejemplo: `//@ private invariant dato1 == 2*dato2;`
- **constraint**: propiedad cierta que puede relacionar el estado anterior y actual de la clase
  - Permite el uso del operador `\old()`
  - Ejemplo: `//@ private constraint dato==\old(dato)+5;`

## Invariantes de clase (IV)

- Ejemplo:

```
class EjemploInvariant
{
    //@ invariant v != null;
    //@ constraint v.length >= \old(v.length);
    int[] v;

    //@ invariant a >= b + c;
    int a, b, c;
}
```

## Métodos puros

- En una especificación de JML está permitido llamar a métodos:
  - *Restricción*: los métodos llamados no pueden tener efectos laterales
  - Estos métodos se denominan *puros*
  - Deben tener el modificador **pure** en su declaración
  - Ejemplo: 

```
/*@ pure @*/ public int xxx()  
{  
    // Código del método  
}
```
  - Si no se incluye este modificador el compilador de JML producirá un error

## Compilación y ejecución

- Compilación:
  - **jmlc** *NombreClase.java*
  - Al compilar se visualizan todos los módulos que se van analizando. Para evitarlo:
  - **jmlc -Q** *NombreClase.java*
- Ejecución:
  - **jmlrac** *NombreClase*