

# Otras aproximaciones

*Ma. Laura Cobo*

Métodos Formales para Ingeniería  
Departamento de Ciencias e Ingeniería de la Computación  
Universidad Nacional del Sur  
Argentina

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## Diferentes ramas de la lógica

Para realizar aproximaciones a métodos formales, se utiliza alguna de las siguientes ramas:

1. **Model Theory o Teoría de modelos:** Las propiedades se expresan a través de axiomas, el modelo se deriva de conjuntos. Se relaciona la consecuencia lógica con la consecuencia semántica (conjunto de las posibles secuencias de ejecución). La semántica de las sentencias lógicas está dada por valores de verdad.

**Su simplicidad la convierte en una buena herramienta de especificación. Por ejemplo: Alloy**

2. **Proof Theory o Teoría de prueba:** La semántica de las sentencias lógicas está asociada al conjunto de pruebas que las concluyen (en lugar de su simple valor de verdad)

## Diferentes ramas de la lógica

Para realizar aproximaciones a métodos formales, se utiliza alguna de las siguientes ramas:

3. **Teoría de conjuntos axiomática y teoría de tipos:** considera estructuras matemáticas (conjuntos con operaciones particulares). Esta aproximación ha tenido problemas a la hora de razonar sobre los modelos en forma segura.

**Su simplicidad la convierte en una buena herramienta de especificación. Por ejemplo: Z, VDM (Vienna Development Method), B.**

4. **Teoría de computabilidad:** se basa en el estudio de las funciones computables. Nuevamente esta aproximación ha encontrado dificultades serias a la hora de automatizar las pruebas.

## Key

Los métodos formales son robustos y suficientemente poderosos para las aplicaciones pero .... necesitan volverse más accesibles.

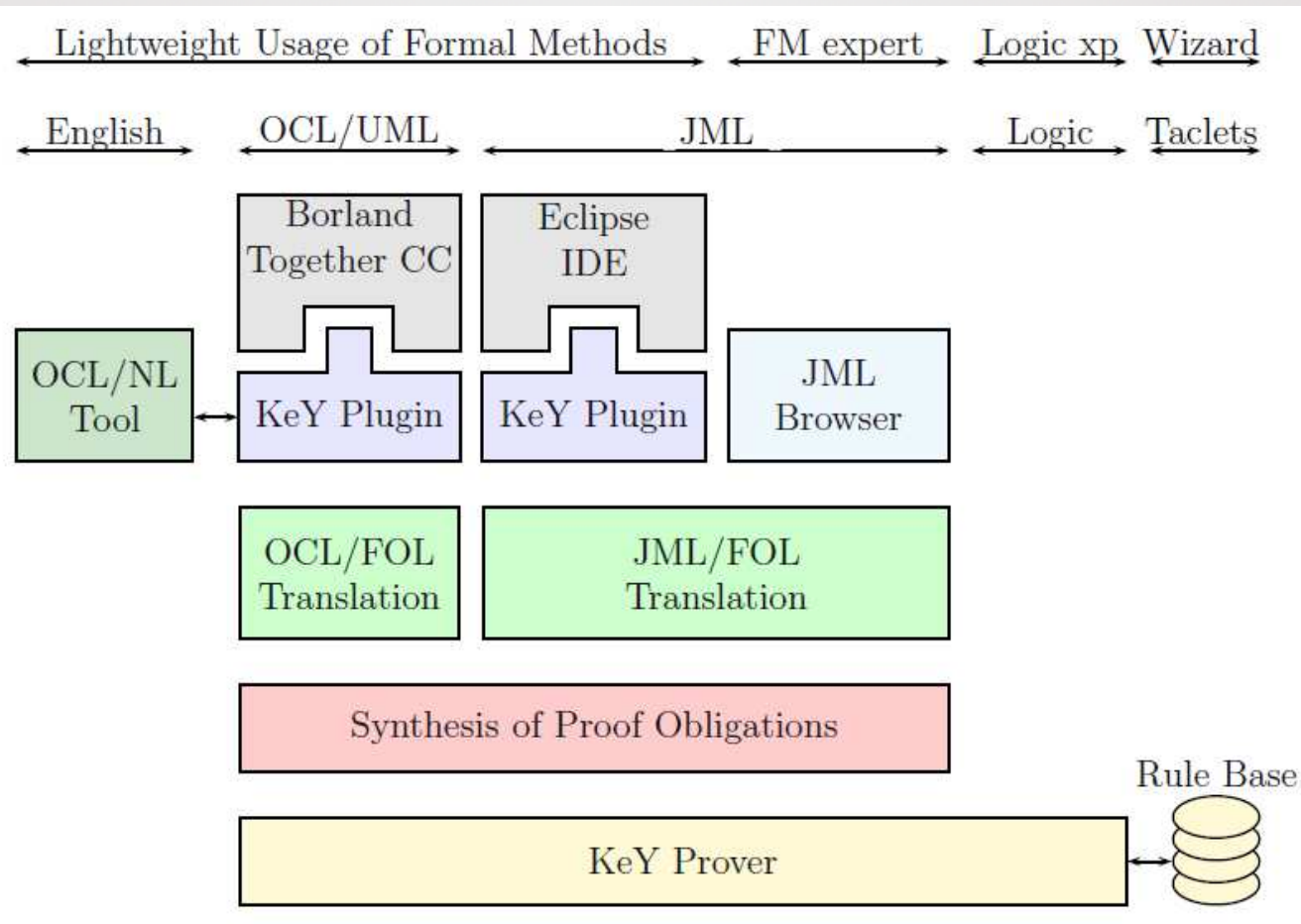
La principal idea detrás del proyecto KeY está en utilizar la lógica y la teoría de prueba.

**No es posible especificar y verificar sistemas complejos de manera automática o por personas sin habilidades adquiridas en métodos formales**

**Lograr esta automatización es tan improbable como la programación automatizada de sistemas complejos**

Página de la herramienta: [www.key-project.org](http://www.key-project.org)

## Arquitectura e interface de Key





## Ejemplo simple

La función de una tarjeta de pago es permitir al dueño pagar cuentas o retirar dinero de las terminales autorizadas por el proveedor de la tarjeta.

Una tarjeta contiene información acerca del balance. El balance no puede ser negativo y no puede exceder el límite dado. El límite no puede modificarse aunque cada tarjeta puede tener un límite diferente.

Cada proveedor posee una operación que actualiza el balance de acuerdo al monto involucrado en la transacción.

**A pesar de la simpleza hay cuestiones sin especificar, como por ejemplo qué sucede si se excede el límite.**

## Especificaciones como contratos

Hacen énfasis en los roles, obligaciones y/o responsabilidades, de una especificación. Análogamente se habla de la metodología de **diseño por contrato**.

### Contrato entre el llamador y el llamado

1. El llamado garantiza proveer el resultado previsto
2. El que realiza la llamada garantiza los pre-requisitos.

Un par de pre/post condiciones para un método  $m$  se satisface para la implementación de  $m$  si:

**Cuando  $m$  es llamado en cualquier estado que satisface la pre-condición entonces en cualquier estado de terminación de  $m$  la post-condición es verdadera.**

# Diseño por contrato

*Ma. Laura Cobo*

Métodos formales para Ingeniería de Software  
Departamento de Ciencias e Ingeniería de la Computación  
Universidad Nacional del Sur  
Argentina

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina



## De modelos a implementaciones

La verificación formal de modelos implica un esfuerzo que se espera impacte en la calidad de la implementaciones

**¿ cómo trasladamos la información del diseño al código ?**

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## De modelos a implementaciones

La verificación formal de modelos implica un esfuerzo que se espera impacte en la calidad de la implementaciones

**¿ cómo trasladamos la información del diseño al código ?**

- Información estática (multiplicidades, invariantes, ...)
- Información dinámica o sobre las operaciones (precondiciones, postcondiciones, condiciones de marco, ...)

## Diseño por contrato

Es un método que hace énfasis en una descripción precisa de la semántica de una **interfaz**

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## Diseño por contrato

Es un método que hace énfasis en una descripción precisa de la semántica de una **interfaz**

- No es solo sintaxis (parte pública de una operación)
- Describe comportamiento, efectos de la invocación a un método, operación o abstracción

## Diseño por contrato

Es un método que hace énfasis en una descripción precisa de la semántica de una **interfaz**

Es un método soportado por herramientas que:

- Permiten una propagación al código de las propiedades semánticas definidas en el diseño (*trasladadas al modelo*)
- Soportan varias formas de validación para esas mismas propiedades.



## La idea básica

Cualquier software puede verse como:

- **Un conjunto de componentes que se comunican**
- **Toda interacción está gobernada por contratos**
- **Los contratos son especificaciones formales de obligación mutua**

**Los contratos son bidireccionales generan obligación en ambas partes**

## Contratos

Siempre hay dos partes:

- el **servidor** (que realiza la tarea)
- el **cliente** (el que requiere la tarea)

**El contrato especifica las obligaciones y beneficios de cada parte**

## Contratos

Especifican lo que debe hacerse.

**Se utilizan los mismos bloques básicos que para el análisis**

- Pre-condiciones
- Post-condiciones
- Condiciones de marco o frame-conditions
- Invariantes

## Especificación o lenguaje de implementación

Se espera una progresión natural del diseño al código

Se busca la posibilidad de utilizar el refinamiento como metodología:

- Progresar del desarrollo de firmas solas
- Desarrollar especificaciones de contratos
- Analizar la consistencia de la interacción cliente-servidor
- Completar detalles de implementación
- Chequear que el código satisface los contratos

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## Especificaciones como contratos

**Un par de pre/post condiciones para un método  $m$  se satisface para la implementación de  $m$  si:**

**Cuando  $m$  es llamado en cualquier estado que satisface la pre-condición entonces en cualquier estado de terminación de  $m$  la post-condición es verdadera.**

Importante:

1. No hay garantías si la precondición no se satisface.
2. La terminación puede estar o no garantizada
3. El estado de terminación puede ser alcanzado por terminación normal o abrupta (ocurrencia de una excepción)



## Especificación formal

**Los contratos deben describirse en un lenguaje preciso y matemático**

Motiva el uso de un lenguaje formal:

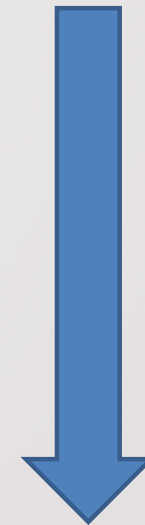
1. Tener un mayor grado de precisión.
2. Eventualmente se pueden automatizar análisis de varios tipos:
  - Chequeo estático
  - Verificación de programas

## Especificación formal

**Los contratos deben describirse en un lenguaje preciso y matemático**

Motiva el uso de un lenguaje formal:

1. Tener un mayor grado de precisión.
2. Eventualmente se pueden automatizar análisis de varios tipos:
  - Chequeo estático
  - Verificación de programas



**En el caso de KeY, el lenguaje de especificación de contratos es JML**

# JML: Java Modelling Language

*Ma. Laura Cobo*

Métodos Formales para Ingeniería  
Departamento de Ciencias e Ingeniería de la Computación  
Universidad Nacional del Sur  
Argentina

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## JML

JML es un lenguaje de especificación.

JML integra

- Especificación
- Implementación

en un solo lenguaje.

1. **La especificación se realiza sobre los archivos JAVA:** aunque la especificación no afecta la implementación pre-existente. En este aspecto es similar a los JavaDocs.
2. **Provee el contexto para diseño por contrato.** Fórmulas de correctitud como las planteadas por la lógica de Hoare.

**JML es Java + lógica de primer orden + pre-post condiciones + .....**

## JML

JML extiende Java con anotaciones. La sintaxis provee capacidad para anotar:

- ✓ precondiciones
- ✓ postcondiciones
- ✓ Invariantes de clase
- ✓ Modificadores adicionales
- ✓ Métodos y campos “solo de especificación”
- ✓ Invariantes de ciclo
- ✓ Etc.

**Estas anotaciones “viven” en comentarios que son ignorados por el lenguaje Java pero que son reconocidos por las herramientas JML.**



## JML

Bajo el signature **modificador** `xxx_behavior` se establece el contrato de cada operación.

La especificación JML se incorpora como comentarios en el archivo .java correspondiente

Un **modificador** posible es **public**.

Cualquier palabra clave que termine con `behavior` abre un caso de especificación. Así por ejemplo: `normal_behavior` establece una especificación donde el método garantiza que no lanzará ninguna excepción.

**Cabe destacar que la especificación dada a través de `normal_behavior` considera la excepciones a nivel superior, si el llamador garantiza las precondiciones dadas en el caso de especificación**

## JML

bajo el signature **modificador** `xxx_behavior` se establece el contrato de cada operación.

La especificación JML se incorpora como comentarios en el archivo .java correspondiente

1. **Comentarios multilínea:** aunque la especificación no afecta la implementación pre-existente. En este aspecto es similar a los JavaDocs.

```
/*@ public normal_behavior
   @ requires !customerauthenticated;
   @ requires pin == insertedCard.correctPIN;
   @ ensures customerAuthenticated;
   @*/
   public void enterPin (int pin) { .....}
```

2. **Comentarios por línea**

```
//@ public normal_behavior
//@ requires !customerauthenticated && pin == insertedCard.correctPIN;
//@ ensures customerAuthenticated;
public void enterPin (int pin) { .....}
```

## JML

El comentario debe comenzar con @ para ser considerado una anotación por una herramienta JML.

No es una anotación JML sino un comentario JML. Los comentarios comienzan con un espacio en blanco no con el símbolo @

```
// @ public normal_behavior  
// @ requires !customerauthenticated &&pin == insertedCard.correctPIN;  
// @ ensures customerAuthenticated;  
public void enterPin (int pin) { .....}
```

```
/* @ public normal_behavior  
   @ requires !customerauthenticated;  
   @ requires pin == insertedCard.correctPIN;  
   @ ensures customerAuthenticated;  
   @*/  
public void enterPin (int pin) { .....}
```

## JML

**Especificación pública:** una especificación pública puede ser

- accedida por todas las clases e interfaces
- Mencionará solo atributos y métodos públicos de esa clase.

```
/*@ public normal_behavior
   @ requires !customerauthenticated;
   @ requires pin == insertedCard.correctPIN;
   @ ensures customerAuthenticated;
   @*/
public void enterPin (int pin) { .....}
```



## JML: precondiciones y postcondiciones

Se utiliza el modificador **requires** acompañado de una expresión. En el ejemplo son expresiones Java booleanas. En general las precondiciones son expresiones JML.

```
/*@ public normal_behavior
   @ requires !customerauthenticated;
   @ requires pin == insertedCard.correctPIN;
   @ ensures customerAuthenticated;
   @*/
public void enterPin (int pin) { .....}
```

La misma idea se aplica a las postcondiciones, sólo que en este caso se utiliza el modificador **ensures** acompañado de una expresión JML booleana



## JML: diferentes especificaciones

Se pueden dar varias especificaciones, las mismas se ligán mediante la palabra clave **also**.

```
/*@ public normal_behavior
   @ requires !customerauthenticated;
   @ requires pin == insertedCard.correctPIN;
   @ ensures customerAuthenticated;
   @
   @ also
   @
   @ public normal_behavior
   @ requires !customerauthenticated;
   @ requires pin != insertedCard.correctPIN;
   @ requires wrongPINCounter < 2;
   @ ensures wrongPINCounter == \old(wrongPINCounter)+1;
   @*/

public void enterPin (int pin) { .....}
```

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## JML: cláusula de “asignabilidad”

Esta cláusula indica las locaciones que pueden cambiar de valor  
**assignable** *lista\_de\_locaciones*.

Su significado establece que ninguna otra locación fuera de las listadas en la cláusula puede ser asignada.

- En caso de que ninguna locación pueda modificarse se agrega la cláusula  
**assignable \nothing;**
- En caso de que todas las locaciones pueda modificarse se agrega la cláusula

**assignable \everything;**

Previene la forma poco feliz de indicar todo lo que no debería cambiar.

**@ ensures** *loc* == **\old**(*loc*)

## JML: cláusula de “asignabilidad”

Se pueden especificar grupos de locaciones asignables utilizando \*

Por ejemplo:

```
@ assignable o.*, a[*]; .
```

En el primer caso se indica que todos los campos del objeto `o` son asignables, mientras que en el segundo caso se indica que todas las locaciones del arreglo `a` lo son.

## JML: modificadores

JML agrega modificadores adicionales. Los más importantes son:

- **spec\_public**
- **pure**



## JML: modificador `spec_public`

Las especificaciones **public** solo pueden “hablar” sobre campos **public**.  
**No es deseable que todos los campos sean públicos.**

Una solución es:

- Mantener los campos **private/protected**..
- Convertir a los que se necesitan para la especificación en **spec\_public**

Ejemplo:

```
private /*@ spec_public @*/ Tipo var = null;  
private /*@ spec_public @*/ int var2 = 0;  
private /*@ spec_public @*/ boolean var3 = false;
```

Otra solución es utilizar `specification-only fields` o campos solo de especificación (este tema no será abordado en el curso)



## JML: modificador pure

Resulta de utilidad utilizar llamadas a métodos en las anotaciones JML. El requerimiento mínimo es que no *tenga efectos colaterales*.

Ejemplos:

```
o1.equals(o2)
li.contiene(elemento)
Lista1.max() < li2.min()
```

En JML se puede especificar un método como **pure**. **Establece como obligación del implementador que el método no produzca efectos colaterales, pero permite su uso en anotaciones JML**

Ejemplo:

```
private /*@ pure @*/ int maximo() {...}
```

**pure** es similar a **assignable \nothing** pero global a métodos

## Expresiones JML booleanas

Se definen en forma recursiva:

- Cada expresión Java de tipo **boolean** libre de efectos colaterales es una expresión JML booleana
- Si **a** y **b** son expresiones JML booleanas y **x** es una variable de tipo **t**, entonces:
  - ✓ **!a** no a.
  - ✓ **a && b** a y b.
  - ✓ **a || b** a o b.
  - ✓ **a ==> b** a implica b.
  - ✓ **a <==> b** a es equivalente a b.
  - ✓ **(\forall t x; a)** para todo x de tipo t tal que a.
  - ✓ **(\exists t x; a)** existe x de tipo t tal que a.
  - ✓ **(\forall t x; a; b)** para todo x de tipo t **que cumple** a, b es verdadera.
  - ✓ **(\exists t x; a; b)** existe x de tipo t **que cumple** a, tal que b es verdadera.

son expresiones JML booleanas

## Expresiones JML booleanas

```
(\forallall t x; a; b)  
(\exists t x; a; b)
```

a recibe el nombre de “*predicado rango*”.

Ambas formas son redundantes, aunque simplifican la lectura de la especificación, ya que se pueden reescribir de la siguiente manera.

```
(\forallall t x; a; b) es equivalente a (\forallall t x; a==>b)  
(\exists t x; a; b) es equivalente a (\exists t x; a&&b)
```

Ejemplo:

```
(\forallall int i,j; 0<=i && i<j && j<10; arr[i]<=arr[j])
```

La idea es utilizar **a** para restringir **x** mas allá del tipo **t**

## Ejemplo

Expresar: “todas las instancias creadas de la clase TarjetaBancaria tienen NumeroTarjeta diferentes”.

```
(\forall TarjetaBancaria p1, p2;  
  \created(p1) && \created(p2);  
  p1 != p2 ==> p1.numeroTarjeta != p2.numeroTarjeta)
```

Observaciones:

- ✓ Los cuantificadores JML toman como rango los objetos no creados también.
- ✓ Lo mismo sucede para los cuantificadores en KeY
- ✓ En JML se puede restringir el análisis solo a los objetos creados con el modificador **\created**

## Valores resultado en las post-condiciones

En las post-condiciones, se puede utilizar el modificador **\result** para hacer referencia al valor retornado por el método

```
/*@ public normal_behaviour
   @ ensures \result == (\exists int i;
   @           0 <= i && i < tamaño;
   @           arr[i] == elem);
   @*/

public /*@ pure @*/ boolean contiene (int elem) {
    /*...*/ }
```



## Ejemplo operación agregar

```
/*@ public normal_behaviour
@ requires tamaño < limite && !contiene(elem);
@ ensures \result == true;
@ ensures contiene(elem);
@ ensures (\forallall int e;
@           e != elem;
@           contiene(e) <==> \old(contiene(e)));
@ ensures tamaño == \old(tamaño)+1;
@
@ also
@
@ <caso de especificación 2>
@*/

public boolean agregar (int elem) {
    /*...*/ }
```

## Ejemplo operación agregar

```
/*@ public normal_behaviour
   @ <caso de especificación 2>
   @
   @ also
   @
   @ public normal_behaviour
   @ requires (tamaño == limite) || contiene(elem);
   @ ensures \result == false;
   @ ensures contiene(elem);
   @ ensures (\forall int e;
   @           contiene(e) <==> \old(contiene(e)));
   @ ensures tamaño == \old(tamaño);
   @*/

public boolean agregar (int elem) {
    /*...*/ }
```

## JML: Invariantes de clase

```
public class conjuntoDeEnterosLimitadoOrdenado {  
  
    public final int limite;  
  
    /*@ public invariant(\forall forall int i;  
        @           0 < i && i < tamaño;  
        @           arr[i-1] <= arr[i])  
    @*/  
  
    private /*@ spec_public @*/ int arr[];  
    private /*@ spec_public @*/ int tamaño = 0;  
  
    // resto de la clase  
  
}
```

De no contar con el invariante la condición debería ser una post y pre condición en todas las operaciones.

Departamento de Ciencias e Ingeniería de la Computación – Universidad Nacional del Sur, Argentina

## JML: Invariantes de clase

Los invariantes pueden colocarse en cualquier lugar de la clase, contrariamente a lo que sucede con el **método contrato** que debe estar ubicado inmediatamente antes del método.

Se recomienda colocar los invariantes de clase inmediatamente antes de la declaración del campo/atributo sobre el cual establece la propiedad/restricción.

## JML: Invariantes de instancia vs. estáticos

**Invariantes de instancia:** pueden hacer referencia a los atributos de instancia de este objeto (no calificados o calificados con el `this`).  
Se utiliza la sintaxis: `instance invariant`

**Invariantes estáticos:** **NO** pueden hacer referencia a los atributos de instancia de este objeto  
Se utiliza la sintaxis: `static invariant`

Ambos pueden hacer referencia a:

- los atributos estáticos de la clase
- atributos de instancia vía referencia explícita como por ejemplo

`o.tamaño`

En clases por defecto los invariantes son de instancia, es decir si el `static` o `instance` se omite el invariante se considera de instancia.



## JML: especificando comportamiento excepcional

**normal\_behavior**: esta forma de especificación establece que la verificación de las precondiciones PROHIBEN al método la señalización de excepciones, pero EXIGEN el cumplimiento de las post-condiciones

**exceptional\_behavior**: esta forma de especificación establece que la verificación de las precondiciones REQUIEREN que el método señale excepciones.

Se utilizan las palabras claves

**signals**  
**signals\_only**

La primera especifica el post-estado dependiendo de la excepción lanzada, la segunda limita el tipo de excepciones lanzadas

## JML: especificando comportamiento excepcional

```
/*@ <caso 1> also <caso 2> also
  @
  @ public exceptional_behavior
  @ requires ...
  @ signals_only   ATMException;
  @ signals   (ATMException) !clienteAutenticado;
  @*/

public void enterPIN (int pin) { ...
```

Por defecto las dos formas de especificación, **normal\_behavior** y **exceptional\_behavior** hacen cumplir la propiedad de terminación.

## JML: permitiendo no terminación

En cada caso de especificación la no terminación puede permitirse vía la clausula

**diverges true;**

**significado:** si las precondiciones se satisfacen en el pre-estado, el método **puede o no terminar.**

## JML: más modificadores

JML extiende los modificadores, tanto

- Los atributo de clase,
- Los parámetros de métodos, como
- El tipo retornado por un método

pueden declararse como

**nullable**: puede o no ser **null**.

**non\_nullable**: no puede ser **null**.

## JML: más modificadores

JML extiende los modificadores, tanto

- Los atributo de clase,
- Los parámetros de métodos, como
- El tipo retornado por un método

pueden declararse como

**nullable**: puede o no ser **null**.

**non\_nullable**: no puede ser **null**.

Hay muchas herramientas que soportan JML, algunas de ellas pueden encontrarse en: [www.jmlspecs.org](http://www.jmlspecs.org)