

Tecnología de la Programación

JML

David Cabrero Souto

Facultad de Informática
Universidade da Coruña

Curso 2007/2008



- Java Modeling Language.
 - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
 - Precondiciones
 - Postcondiciones
 - Invariantes de clase
- <http://www.jmlspecs.org>
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:

```
//@ keyword aserción
```

```
/*@ ... @*/
```

- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.



- Java Modeling Language.
 - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
 - Precondiciones
 - Postcondiciones
 - Invariantes de clase
- <http://www.jmlspecs.org>
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:

```
//@ keyword aserción
```

```
/*@ ... @*/
```

- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.



- Java Modeling Language.
 - A formal behavioral interface specification language for Java.
- Diseño por contrato en Java.
 - Precondiciones
 - Postcondiciones
 - Invariantes de clase
- <http://www.jmlspecs.org>
- Los contratos se incrustan en el código en forma de anotaciones/aserciones:
 - //@ keyword aserción*
 - /*@ ... @*/*
- La aserción combina expresiones java (sin efectos colaterales) y términos lógicos.
- JML convierte los contratos en código java que los verifican en tiempo de ejecución.



- Calcular el máximo de x e y .

```
if (x >= y)
    z = x;
else
    z = y;
//@ assert z >= x && z >= y;
```



- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: $z = x + y + 1$ es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: $z = x*x + y*y$.
- Especificación correcta:

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



- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: $z = x + y + 1$ es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: $z = x*x + y*y$.
- Especificación correcta:

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



- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: $z = x + y + 1$ es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: $z = x*x + y*y$.
- Especificación correcta:

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



- ¿ Cómo construir la especificación ?
- En el ejemplo anterior: $z = x + y + 1$ es correcto respecto a la especificación.

```
//@ assert z >= x && z >= y;
```

- No es cierto, pero sí: $z = x*x + y*y$.
- Especificación correcta:

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



requires = precondición

ensures = postcondición

```
public final static double eps = 0.0001;

/*@ requires x >= 0.0;
   @ ensures JMLDouble.approximatelyEqualTo(x,
   @           \result * \result, eps);
   @*/
public double sqrt(double x) { ... }
```



- Afectan a todos los métodos.

```
public class CuentaCorriente {  
    public static final int MAX_BAL = 100000;  
    private int _balance;  
    /*@ invariant 0 <= _balance &&  
        _balance <= MAX_BAL  
    @*/  
}
```



- Expresiones Java.
 - No puede tener efectos colaterales (`++`, `--`, ...)
 - Sólo llamadas a *métodos puros*:
 - Sin efectos colaterales.
 - Anotados como tales: `/*@ pure */ public int foo() { ... }`
- Formulas lógicas:
 - Operadores lógicos (\implies , \iff , ...).
 - Cuantificadores (\forall , \exists , ...).
- Variables especiales:
 - `\result` Valor de retorno del método.
 - `\old(x)` Valor inicial de la variable `x`.



- Expresiones Java.
 - No puede tener efectos colaterales ($++$, $--$, ...)
 - Sólo llamadas a *métodos puros*:
 - Sin efectos colaterales.
 - Anotados como tales: `/*@ pure */ public int foo() { ... }`
- Formulas lógicas:
 - Operadores lógicos (\implies , \iff , ...).
 - Cuantificadores (\forall , \exists , ...).
- Variables especiales:
 - `\result` Valor de retorno del método.
 - `\old(x)` Valor inicial de la variable x .



- Expresiones Java.
 - No puede tener efectos colaterales ($++$, $--$, ...)
 - Sólo llamadas a *métodos puros*:
 - Sin efectos colaterales.
 - Anotados como tales: `/*@ pure */ public int foo() { ... }`
- Formulas lógicas:
 - Operadores lógicos (\implies , \iff , ...).
 - Cuantificadores (\forall , \exists , ...).
- Variables especiales:
 - `\result` Valor de retorno del método.
 - `\old(x)` Valor inicial de la variable x .



- Expresiones Java.
 - No puede tener efectos colaterales (`++`, `--`, ...)
 - Sólo llamadas a *métodos puros*:
 - Sin efectos colaterales.
 - Anotados como tales: `/*@ pure */ public int foo() { ... }`
- Formulas lógicas:
 - Operadores lógicos (\implies , \iff , ...).
 - Cuantificadores (\forall , \exists , ...).
- Variables especiales:
 - `\result` Valor de retorno del método.
 - `\old(x)` Valor inicial de la variable `x`.



- Usar `jmlc` en lugar de `javac`.
- Usar `jmlrac` en lugar de `java`.
- Usar `jmldoc` en lugar de `javadoc`.
- Usar `jmlunit` en lugar de `junit`.




```
public class Persona {
    private /*@ spec_public non_null @*/ String _nombre;
    private /*@ spec_public @*/ int _peso;
    //@ public invariant !_nombre.equals("") && _peso >= 0;

    /*@ also
       @ ensures \result != null;
       @*/
    public String toString() { ... }

    //@ ensures \result = _peso
    public int getPeso() { ... }

    //@ ensures kgs >= 0 && _peso == \old(kgs * _peso);
    public void añadirKgs(int kgs) { ... }

    /*@ requires !n.equals("");
       @ ensures n equals( _nombre) && _peso == 0.
```



Ejemplo \forall

```
(\forall Student s;  
    juniors.contains(s) ==> s.getAdvisor() != null)
```

- `\forall`
- `\exists`
- `\sum`
- `\product`
- `\max`
- `\min`
- `\num_of`



sumaArray

```
/* sumaArray()
   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 array == \old(array) &&
  @   (\forall int I; 0 <= I &&
  @           I < array.length;
  @           array[I] == \old(array[I]));
  @*/
public int sumaArray(int[] array) { ... }
```

- **Sintaxis:** (* ... *)
- Semántica: expresión booleana.
- Siempre se evalúa a cierto.
- Uso: comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos  
del array *)
```



- **Sintaxis:** (* ... *)
- **Semántica:** expresión booleana.
- **Siempre se evalúa a cierto.**
- **Uso:** comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos  
del array *)
```



- Sintaxis: (* ... *)
- Semántica: expresión booleana.
- Siempre se evalúa a cierto.
- Uso: comentarios de las especificaciones.

```
@ ensures (* devuelve la suma de los elementos  
del array *)
```



- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
`requires divisor > 0 ∨ divisor = 0`
`ensures divisor > 0 ⇒ ... ∧ divisor = 0 ⇒ ...`
- Podemos usar `also`.
Una diferentes comportamientos.



- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
requires `divisor > 0 ∨ divisor = 0`
ensures `divisor > 0 ⇒ ... ∧ divisor = 0 ⇒ ...`
- Podemos usar `also`.
Une diferentes comportamientos.



- Ejemplo: método `divide` para números enteros. Tiene dos comportamientos, cuando el divisor es mayor que cero o cuando el divisor es cero.
- Podríamos hacer
requires `divisor > 0` \vee `divisor = 0`
ensures `divisor > 0` \implies ... \wedge `divisor = 0` \implies ...
- Podemos usar `also`.
Une diferentes comportamientos.



```
/*@ public normal_behavior
   @   requires divisor > 0;
   @   ensures divisor * \result <= dividendo &&
   @           divisor * (\result+1) > dividendo;
   @
   @ also
   @ public exceptional_behavior
   @   requires divisor == 0;
   @   signals (ArithmeticException);
   @*/
public int divide(int dividendo, int divisor)
    throws ArithmeticException
{ ... }
```

- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
- La cota se decrementa en cada iteración.
- Asegura la terminación.



- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
- La cota se decrementa en cada iteración.
- Asegura la terminación.



- Cierta antes de comenzar el bucle.
- Cierta después de cada iteración.
- Puede tener una *cota*.
- La cota se decrementa en cada iteración.
- Asegura la terminación.



Ejemplo de invariante de bucle

```
int i = n;
int s = 0;
/*@ loop_invariant i+s == n
   *@ decreases i;
while (i >= 0)
{
    i = i-1;
    s = s+1;
}
```



- Las especificaciones tb. se heredan.
- La especificación no se puede “relajar”.
- Las postcondiciones se unen en una conjunción.
- Las precondiciones se unen en una disjunción.



- Las especificaciones tb. se heredan.
- La especificación no se puede “relajar”.
- Las postcondiciones se unen en una conjunción.
- Las precondiciones se unen en una disjunción.



- Las especificaciones tb. se heredan.
- La especificación no se puede “relajar”.
- Las postcondiciones se unen en una conjunción.
- Las precondiciones se unen en una disjunción.



- Las especificaciones tb. se heredan.
- La especificación no se puede “relajar”.
- Las postcondiciones se unen en una conjunción.
- Las precondiciones se unen en una disjunción.



```
class Parent {  
    ...  
    //@ invariant invParent;  
    ...  
}
```

```
class Child extends Parent {  
    ...  
    //@ invariant invChild;  
    ...  
}
```

La invariante de la clase hijo es `invChild && invParent`



```
class Parent {  
    ...  
    //@ invariant invParent;  
    ...  
}
```

```
class Child extends Parent {  
    ...  
    //@ invariant invChild;  
    ...  
}
```

La invariante de la clase hijo es `invChild & invParent`



- Ejemplo *naive*.
- `also` nos permite extender la especificación heredada.

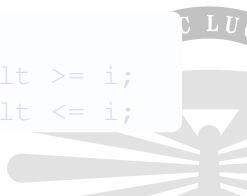
```
class Parent {
    //@ requires i >= 0;
    //@ ensures \result >= i;
    int m(int i){ ... }
}
class Child extends Parent {
    //@ also
    //@   requires i <= 0
    //@   ensures \result <= i;
    int m(int i){ ... }
}
```

- La especificación resultante del método `m()` en `Child`.

```
//@ requires i >= 0;  
//@ ensures \result >= i;  
//@ also  
//@ requires i <= 0  
//@ ensures \result <= i;  
int m(int i){ ... }
```

- Equivalente a:

```
//@ requires i >= 0 || i <= 0;  
//@ ensures \old(i) >= 0 ==> \result >= i;  
//@ ensures \old(i) <= 0 ==> \result <= i;
```

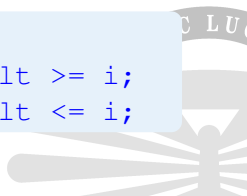


- La especificación resultante del método `m()` en `Child`.

```
//@ requires i >= 0;  
//@ ensures \result >= i;  
//@ also  
//@ requires i <= 0  
//@ ensures \result <= i;  
int m(int i){ ... }
```

- Equivalente a:

```
//@ requires i >= 0 || i <= 0;  
//@ ensures \old(i) >= 0 ==> \result >= i;  
//@ ensures \old(i) <= 0 ==> \result <= i;
```



- Función de ordenación.

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @   \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @   c.compare(\result.itemAt(i-1),
  @             \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- ¿ Correcto ?
- (* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro



- **Función de ordenación.**

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @   \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @   c.compare(\result.itemAt(i-1),
  @             \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- **¿ Correcto ?**

- (* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro

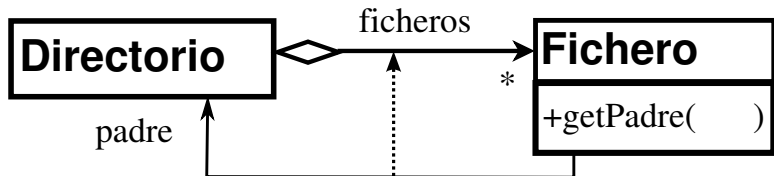


- Función de ordenación.

```
/*@ ensures
  @ (\forall int i; 0 <= i && i < \result.size();
  @   \result.itemAt(i) instanceof Integer)
  @ &&
  @ (\forall int i; 0 < i && i < \result.size();
  @   c.compare(\result.itemAt(i-1),
  @             \result.itemAt(i)) <=0);
  @*/
public static List sort(List l, Comparator c) {
  ...
}
```

- ¿ Correcto ?
- (* \result contiene exactamente los mismos elementos que \old(l), posiblemente en otro

- Incluir restricciones del diseño.

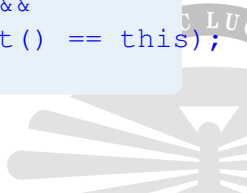


```
public class Directorio {
    private Fichero[] _ficheros;

    /*@ invariant

    _ficheros != null
    &&
    (\forall int i; 0 <= i && i < _ficheros.length;
        _ficheros[i] != null &&
        _ficheros[i].getParent() == this);

    @*/
```



- Sólo se pueden usar en las especificaciones.

```
class SimpleProtocol {
  //@ boolean ghost started;
  //@ requires !started;
  //@ assignable started;
  //@ ensures      started;
  void start() {
    ...
    //@ set started = true;
  }
  //@ requires      started;
  //@ assignable started;
  //@ ensures      !started;
  void stop() {
    ...
  }
}
```



- Pero se pueden mezclar con otros atributos.

```
class SimpleProtocol {  
  private Stack _stack;  
  //@ boolean ghost started;  
  //@ invariant started <==> (_stack != null);  
  ...  
}
```

- En el resto queda oculta la implementación de *started*.

```
//@ requires !started;  
//@ assignable started;  
//@ ensures    started;  
void start() { ... }  
...
```

- Pero se pueden mezclar con otros atributos.

```
class SimpleProtocol {  
  private Stack _stack;  
  //@ boolean ghost started;  
  //@ invariant started <==> (_stack != null);  
  ...  
}
```

- En el resto queda oculta la implementación de *started*.

```
//@ requires !started;  
//@ assignable started;  
//@ ensures    started;  
void start() { ... }  
...
```

- Tb. son atributos sólo de la especificación.
- Se le asocia un valor.

```
class SimpleProtocol {
private ProtocolStack st;
/*@ boolean model started;
/*@ represents started <-- (st!=null);

/*@ requires !started;
/*@ assignable started;
/*@ ensures started;
void startProtocol() { ... }
...
}
```


- ¿ Si quiero extender/añadir un contrato sin modificar el fichero con el código fuente ?
- Solución: usar `refine`.

Persona.jml

```
package org.jmlspecs.samples.jmltutorial;

/*@ refine "Persona.java";

public class Persona {
    private /*@ spec_public non_null @*/ String _nombre;
    private /*@ spec_public @*/ int _peso;

    /*@ public invariant !_nombre.equals("")
        @          && _peso >= 0; @*/

    /*@ also
    /*@ ensures \result != null;
```



- ¿ Si quiero extender/añadir un contrato sin modificar el fichero con el código fuente ?
- Solución: usar `refine`.

Persona.jml

```
package org.jmlspecs.samples.jmltutorial;

/*@ refine "Persona.java";

public class Persona {
    private /*@ spec_public non_null @*/ String _nombre;
    private /*@ spec_public @*/ int _peso;

    /*@ public invariant !_nombre.equals("")
        @          && _peso >= 0; @*/

    /*@ also
    /*@ ensures \result != null;
```

