Java Modeling Language

H1 ## Introduzione

> JML è un linguaggio di specifica per il Design by Contract di Java.

H2

Le specifiche scritte in JML hanno diversi vantaggi:

✔ Forniscono una documentazione *esplicita* del contratto

✔ Vengono rese chiare le *assunzioni* fatte a livello di progetto, le quali andranno considerate in fase di implementazione

✔ Rendono più facile la *comprensione* e la *manutenzione* del codice

✔ Possono essere analizzate da *tool automatici*

Sintassi

Vediamo alcuni punti riguardanti la sintassi di JML:

H2

- Il codice JML viene aggiunto al file `.java` come commenti ed è racchiuso tra `/*@ ... @*/` o dopo `//@` .
- Le condizioni sono scritte come *espressioni booleane* con la presenza di operatori aggiuntivi:
  - `\result`
  - `\forall`
  - `\old`
  - `⟹`
  - `...`
- alcune parole chiave:
  - `requires`
  - `ensures`
  - `invariant`
  - `...`

H3 Precondizioni, postcondizioni, invarianti

È possibile aggiungere:

- **precondizioni** ai metodi:
  - `//@ requires <espressione booleana>`
- **postcondizioni** ai metodi:
  - `//@ ensures <espressione booleana>`
- **invarianti** di classe:
  - `//@ invariant <espressione booleana>`

```
/*@ requires amount ≥ 0; ensures true; @*/
public int debit(int amount){}
```

Il metodo `debit()` richiede come precondizione che `amount` sia positivo (`requires amount ≥ 0`) e non garantisce alcuna postcondizione (`ensures true`), che è la *postcondizione di default*.

```
public class Account{
    ...
    /*@ invariant balance ≥ minBalance; @*/
    ...
}
```

Specifico un'invariante: un conto non potrà avere un bilancio inferiore al minimo

---

H3 Altri operatori

Espressione	Significato
`a ⟹ b`	`a` implica `b`
`a ⟸ b`	`a` consegue da `b` (o `b` implica `a`)
`a ⟺ b`	`a` se e solo se `b`

Espressione	Significato
`a <=!=> b`	non (`a` se e solo se `b`)
`\old(E)`	valore di `E` prima della chiamata
`\result`	risultato della chiamata del metodo

```
//@ ensures \result ⟺ j < n;
boolean minore (int j, int n) {
    return j < n;
}
```

Il metodo minore restituisce `true` *se e solo se* `j < n`, quindi in JML scriviamo:
`//@ ensures \result ⟺ j < n;`

H3 Quantificatori

Possiamo inserire dei **quantificatori** nelle espressioni in JML, per potenziare l'espressività delle condizioni imposte:

- *universali*
  - `\forall` : **per ogni** elemento
- *esistenziali*
  - `\exists` : **esiste** un elemento
- *generali*
  - `\sum`
  - `\product`
  - `\min`
  - `\max`
- *numerici*
  - `\num_of`

La sintassi per usare i quantificatori è la seguente:

`<quantificatore> <tipo> <variabile>; <range predicate>; <espressione>`

Tutti gli studenti contenuti nella `Collection` `juniors` hanno `advisor` (dato dal metodo `getAdvisor()`):

`\forall Student s; juniors.contains(s); s.getAdvisor() ≠ null`

Facciamo ora un esempio di contratto usando i quantificatori:

Scriviamo il contratto per un metodo che restituisce il minimo di un array.

```
public static int find_min(int a[])
```

- **PRE**: l'array `a` non è `null` e ha almeno un elemento
  - `requires a ≠ null && a.length ≥ 1;`
- **POST**: l'elemento restituito è effettivamente minore di tutti gli elementi dell'array e appartiene ad esso
  - `ensures (\forall int i; 0 ≤ i && i ≤ a.length; \result ≤ a[i]) && (\exists int i; 0 ≤ i && i ≤ a.length; \result == a[i]);`

H3 ## *non_null , \old, pure*

non_null

Spesso vorremmo imporre che un certo riferimento non sia mai `null` . Per fare ciò usiamo l'operatore `non_null` .

H4

```
public class Directory{
    private /*@ non_null @*/ File[] files;

    void createSubdir(/*@ non_null @*/ String name){
        ...
    };
}
```

\old(variabile)

Nel caso volessimo accedere ad il valore di una variabile prima che un metodo fosse eseguito, possiamo usare l'operatore `\old(variabile)`.
H4 Spesso utile nelle **postcondizioni**, per confrontare il valore di una variabile dopo l'esecuzione del metodo con il valore precedente all'esecuzione.

```
public class Contatore{
    int n;

    //@ ensures n == \old(n) + 1
    public void incremnta() {
        n++;
    }
}
```

---

pure

L'operatore `pure` permette di dichiarare metodi senza *side-effect*.

H4

```
public static /*@ pure @*/ int abs(int x){
    if (x ≥ 0)
        return x;
    else
        return -x;
}
```

H3 Visibilità

JML impone delle regole di **visibilità** (***scope***) simili a quelle di Java. Ad esempio, non è possibile fare riferimento ad una variabile `private` da una specifica di un metodo `public`.

```
public class Bag{
    private int n;

    //@ requires n > 0;     //WRONG!//
    public int extractMin(){
        ...
    };
}
```

La precondizione `requires n > 0` è specificata su un metodo pubblico, ma `n` è privata!

Per risolvere questo problema si usa l'operatore `spec public` da aggiungere alla variabile privata.

```
public class Bag{
    private /*@ spec public @*/ int n;

    //@ requires n > 0;
    public int extractMin(){
        ...
    };
}
```

L'operatore `spec public` su `n` permette alla precondizione `requires n > 0` di accedere ad `n`!

H3 Ereditarietà

Alcuni punti chiave con cui JML gestisce l'ereditarietà:

- La parola chiave `also` va usata per i metodi overridden per ereditare la specifica del metodo `super`
- Le precondizioni della sottoclasse vengono relazionate secondo la logica di un **or disgiuntivo** con le precondizioni della superclasse
- Le postcondizioni della sottoclasse vengono congiunte (**and**) come le postcondizioni della superclasse
- Gli invarianti vengono congiunti (**and**) come le postcondizioni

H3 *assert*

La parola chiave `assert` di JML permette di richiedere se una certa condizione è verificata ad un certo punto del codice.

```
if (i ≤ 0 || j < 0){
    ...
}
else if (j < 5){
    //@ assert i > 0 && 0 ≤ j && j < 5;
    ...
}
else{
    //@ assert i > 0 && j < 5;
}
```

NOTA: la parola chiave `assert` esiste anche in Java "standard", ma quella di JML è più espressiva!