

III

# Java Modeling Language

M1

# Introduction

- JML (Java Modeling Language) a été conçu par Gary Leavens et ses collègues à l'Université de l'Iowa (Iowa State University)
- JML est un langage de spécification pour Java, qui permet de spécifier différentes assertions :
  - des invariants pour des classes ;
  - des pré- et des post-conditions pour des méthodes ;
  - des assertions au milieu de code Java.
- JML est un langage inspiré des assertions Eiffel, et des langages de spécifications orientés modèles comme VDM et Larch.

# Java Modeling Language (JML)

- Les spécifications sont ajoutées au code en Java (en commentaire)
- Il existe des logiciels (Esc/Java) vérifiant que le code Java réalise la spécification

# Assertions JML

- Des assertions JML portant sur un programme Java peuvent être incluses dans ce programme Java, sous l'une des trois formes de commentaires spéciaux suivantes :

```
/**  
 * <pre><jml>  
 *  
 * invariant condition ;  
 *  
 * </jml></pre>  
 */  
/*@ invariant condition ; */  
//@ invariant condition ;
```

- Les conditions sont écrites en Java, sous forme d'expressions booléennes. Ces expressions doivent être purement fonctionnelles, c'est-à-dire sans effet de bord.

# Assertions JML

- On peut spécifier qu'une méthode Java est purement fonctionnelle en plaçant l'assertion

```
/*@ pure */
```

juste avant le code de cette méthode. Cette méthode peut alors être utilisée dans des assertions JML.

- Les assertions JML peuvent également être définies dans un fichier spécial portant l'extension .jml.
- JML utilise la notion d'état visible.

# Assertions JML

- JML utilise la notion d'état visible.
- Un état visible est un état du système avant ou après l'exécution d'une méthode.
- Les assertions JML doivent être vérifiées dans un état visible, et non à un point quelconque du programme.

# Assertions associées à des classes

À une classe Java peuvent être associés

- des invariants et
- des contraintes d'historique.

# Notion d'invariant

- Un invariant est une condition qui porte sur les attributs de la classe et qui doit être vérifiée dans tout état visible.
- Comme en Java, on distingue en JML les invariants d'instance et les invariants de classe. Les invariants de classe sont préfixés par le mot réservé `static` et ne peuvent faire référence qu'à des constituants de classe (attributs et méthodes `static`).
- Comme les attributs et méthodes en Java, un invariant peut en JML avoir plusieurs niveaux de visibilité : `public` (`public`), protégé (`protected`), paquetage (visibilité par défaut), et privé (`private`).



# Notion d'invariant

- Un invariant public peut uniquement faire référence à des constituants publics de la classe.
- Un invariant protégé peut uniquement faire référence à des constituants publics et protégés de la classe.
- Un invariant ayant la visibilité par défaut ne peut pas faire référence à des constituants privés de la classe.
- Un invariant privé peut faire référence à tous les constituants de la classe.

# Contrainte d'historique

- Une contrainte d'historique permet de poser une condition d'évolution.
- Il s'agit d'une condition qui porte sur l'état avant l'appel d'une méthode quelconque et l'état après l'appel de cette méthode.
- On peut faire référence à une expression `exp` évaluée dans l'état avant l'appel de la méthode avec la construction : **`\old(exp)`**

# Contrainte d'historique

- Comme pour les invariants, on distingue les contraintes d'historique de classe, préfixées par `static`, qui ne font référence qu'à des constituants de classe, et les contraintes d'historique d'instance.
- Comme pour les invariants, on distingue quatre niveaux de visibilité pour les contraintes d'historique : `public` (public), `protected` (protégé), `package` (visibilité par défaut), et `private` (privé).

# Exemple

- On peut par exemple définir une classe Compteur comme suit.

```
class Compteur {  
    /**  
    * <pre><jml>  
    *  
    * private invariant val >= 0 ;  
    *  
    * private constraint val >= \old(val) ;  
    *  
    * </jml></pre>  
    */  
    // Valeur du compteur  
    private int val ;  
  
    ...  
}
```

# Exemple

- La classe Compteur comporte un invariant et une contrainte d'historique.
- L'invariant **private invariant val >= 0** ; qui spécifie que la valeur du compteur doit toujours être positive ou nulle.
- Il s'agit d'un invariant d'instance privé car il porte sur l'attribut val qui est un attribut d'instance privé.
- La contrainte d'historique **private constraint val >= \old(val)** ; qui spécifie qu'aucune méthode ne peut faire diminuer la valeur du compteur.
- Il s'agit d'une contrainte d'historique d'instance privée car elle porte sur l'attribut val qui est un attribut d'instance privé.

# Assertions associées à des méthodes

- À une méthode peuvent être associées des **pré-conditions**, des **post-conditions** et des **post-conditions exceptionnelles**.

## Pré-condition

- Une pré-condition pour une méthode est une condition portant sur
  - l'état de l'objet qui invoque la méthode,
  - les paramètres de la méthode,qui doit être satisfaite avant chaque appel de cette méthode.
- Une pré-condition est introduite en JML par **pre** ou **requires** (les deux mots clés sont équivalents).
- Exemple : `/*@ requires x >= 0 (* x is positive *); */`

# Assertions associées à des méthodes

## Post-condition

- Une post-condition pour une méthode est une condition portant sur
  - l'état de l'objet avant l'exécution de la méthode,
  - l'état de l'objet après l'exécution de la méthode,
  - les paramètres de la méthode,
  - le résultat de la méthode,qui doit être satisfaite après chaque appel de cette méthode qui termine normalement, c'est-à-dire dans le cas d'absence d'exception.
- Une post-condition est introduite en JML par **post** ou **ensures** (les deux mots clés sont équivalents).
- Exemple : `/*@ ensures \result >= 0 */`

# Assertions associées à des méthodes

## Post-condition exceptionnelle

- Une post-condition exceptionnelle est une condition qui porte sur les mêmes éléments qu'une post-condition, et qui doit être satisfaite après chaque appel de méthode qui termine en levant une exception spécifiée.
- Une post-condition exceptionnelle est introduite en JML par **signals** ou **exsures** (les deux mots clés sont équivalents).
- **Exemple** : `/* @ signals (IllegalArgumentException e) x < 0;`  
`@*/`



## Exemple

Pour continuer l'exemple précédent, on peut spécifier deux méthodes dans la classe Compteur :

- la méthode **int getVal()** retourne la valeur courante du compteur ;
- la méthode **void incr(int n) throws Debordement** permet d'incrémenter le compteur de n unités. Cette méthode lève l'exception **Debordement** si la nouvelle valeur est trop grande pour être stockée dans un entier de type int.

# Exemple

- On définit l'exception Debordement :

```
/**  
 * Une exception levée en cas de débordement d'un compteur.  
 */  
public class Debordement extends Exception { }
```

Le code de la méthode getVal() est le suivant :

```
/**  
 * Retourne la valeur de ce compteur.  
 */  
public /*@ pure */ int getVal() {  
    return val ;  
}
```

NB: On spécifie que la méthode getVal() est « pure », ce qui signifie qu'elle n'effectue pas d'effet de bord. Cela permet d'utiliser cette fonction dans une spécification JML, en particulier dans les post-conditions de la méthode incr.

# Exemple

```
/**
 * Incrémente la valeur de ce compteur de n.
 * <pre><jml>
 * requires n >= 0 ;
 * ensures getVal() == \old(getVal()) + n ;
 * signals (Debordement deb) getVal() == \old(getVal()) ;
 * </jml></pre>
 */
public void incr(int n) throws Debordement {
    int nouvelle_valeur = val + n ;
    if (nouvelle_valeur >= 0) {
        val = nouvelle_valeur ;
    } else {
        throw new Debordement() ;
    }
}
```

# Exemple

- La pré-condition **requires  $n \geq 0$**  ; spécifie qu'on ne peut pas incrémenter le compteur d'une valeur négative.
- La post-condition **ensures  $\text{getVal()} == \backslash\text{old}(\text{getVal()}) + n$**  ; spécifie que si la méthode `incr` termine normalement (c'est-à-dire sans lever d'exception) alors la valeur du compteur est incrémentée de  $n$ .
- La post-condition exceptionnelle **signals (Debordement deb)  $\text{getVal()} == \backslash\text{old}(\text{getVal()})$**  ; spécifie que si la méthode `incr` termine en levant une exception de type `Debordement`, alors la valeur du compteur n'est pas modifiée.

# Assertions dans le corps des méthodes

- Il est possible de placer des assertions JML dans le corps de méthodes Java. Ces assertions sont introduites par le mot clé `assert`.
- Par exemple, on peut ajouter une assertion dans le corps de la méthode `incr`.

```
public void incr(int n) throws Debordement {  
    int nouvelle_valeur = val + n ;  
    if (nouvelle_valeur >= 0) {  
        val = nouvelle_valeur ;  
        //@ assert val >= 0 ;  
    } else {  
        throw new Debordement() ;  
    }  
}
```

- L'assertion **`assert val >= 0 ;`** spécifie qu'à ce point du programme, l'attribut `val` est positif ou nul.

# Ecriture des assertions

- Les assertions sont écrites en Java, mais :
  - ne peuvent avoir d'effet de bord
  - Pas de =, ++, --, etc.
- Appel uniquement de méthodes pures
  - **pure** est une annotation JML ajoutée aux méthodes, les déclarant ainsi sans effet de bord
- Les assertions peuvent utiliser des extensions propres à JML :

Syntaxe	Signification
<code>\result</code>	résultat de la méthode
<code>a ==&gt; b</code>	a implique b
<code>a &lt;== b</code>	b implique a
<code>a &lt;==&gt; b</code>	a ssi b
<code>a &lt;!=&gt; b</code>	!(a <==> b)
<code>\old(E)</code>	valeur de E à l'entrée

# Compilateur JML (jmlc)

- Le compilateur JML (jmlc) permet de compiler des classes Java accompagnées de spécifications JML en bytecode Java.
- Ce compilateur produit, à partir de fichiers Java (extension .java) et de fichiers purement JML (extension .jml) des fichiers en bytecode (extension .class) qui comportent du code qui vérifie à l'exécution les assertions JML.

# Compilateur JML (jmlc)

- Pour chaque appel de méthode, les actions suivantes sont effectuées :
  - vérification de la pré-condition de la méthode ;
  - vérification de l'invariant de la classe ;
  - appel de la méthode ;
  - vérification de la post-condition, si la méthode a terminé normalement ; vérification de la post-condition exceptionnelle, si la méthode a terminé en levant une exception spécifiée ;
  - vérification de l'invariant de la classe ;
  - vérification de la contrainte d'historique de la classe.
- Si une assertion n'est pas vérifiée, une exception spécifique est levée.



# Exemple de classe avec spécification JML

```
public class Person {  
  private /*@ spec_public non_null @*/  
    String name;  
  private /*@ spec_public @*/  
    int weight;  
  /*@ public invariant !name.equals("")  
    @ && weight >= 0; @*/  
  //@also  
  //@ensures \result != null;  
  public String toString();  
  //@also
```

# Exemple de classe avec spécification JML

```
public /*@ pure @*/ int getWeight();  
/*@also  
@requires kgs >= 0;  
@requires weight + kgs >= 0;  
@ensures weight == \old(weight + kgs);  
@*/  
  
public void addKgs(int kgs);  
/*@also  
@requires n != null && !n.equals("");  
@ensures n.equals(name)  
@&& weight == 0; @*/  
public Person(String n);  
}
```

# Pour les TP

- OpenJML
- Etapes :
  1. Spécifier le code (JML)
  2. Ecrire le code (Java)
  3. Exécuter JML (Esc/Java) pour vérifier que le code Java correspond à la spécification.
    - message d'erreurs si le code ne réalise pas la spécification
    - Message d'alerte, impossible de vérifier que le code réalise la spécification

Fin