# Tutorial on Java Modelling Language

An introductive tutorial

Balint Armand Alexandru,
Barna Tudor Cristian,
Screciu Alin Constantin

West University of Timisoara

06.02.2022

## What is JML?

- JML, short for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and methods.

## What is JML?

- JML, short for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and methods.
- It is heavily based on syntactic sugar to make various notations

## What is JML?

- JML, short for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and methods.
- It is heavily based on syntactic sugar to make various notations
- It uses mathematical concepts in order to describe the behavior of a module without providing the concrete implementation details

## What is JML?

- JML, short for "Java Modelling Language", is a behavioral interface specification language designed to specify Java classes and methods.
- It is heavily based on syntactic sugar to make various notations
- It uses mathematical concepts in order to describe the behavior of a module without providing the concrete implementation details
- It can be written either inside the code or in the documentation.

## Why should you use JML?

- It is easy to read and deduce its meaning!

## Why should you use JML?

- It is easy to read and deduce its meaning!
- Provides runtime assertion checking, which also helps in automating parts of testing

## Why should you use JML?

- It is easy to read and deduce its meaning!
- Provides runtime assertion checking, which also helps in automating parts of testing
- Helps in verifying if the described specifications are fulfilled.

## Why should you use JML?

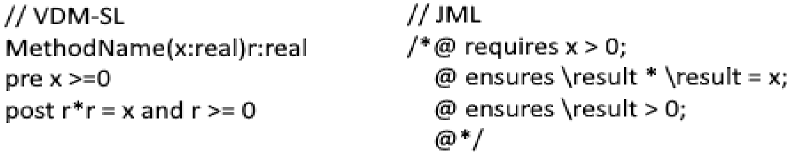
- It is easy to read and deduce its meaning!
- Provides runtime assertion checking, which also helps in automating parts of testing
- Helps in verifying if the described specifications are fulfilled.
- Even its lightweight specifications can help bug-seeking tools

## Advantages of JML

- Besides the ease of read, JML is made exclusively for Java
- JML gives the important benefit of being able to specify exceptions; a vital feature for Java code
- Methods can be called from assertions ( e.g. $//@\backslash ensures(members < getMaxMembers()$ )
- It can be written directly into the source code

## VDM-SL vs JML

Below will be 2 different code snippets, namely VDM-SL and JML



```
// VDM-SL
MethodName(x:real)r:real
pre x >=0
post r*r = x and r >= 0
```

```
// JML
/*@ requires x > 0;
  @ ensures \result * \result = x;
  @ ensures \result > 0;
  @*/
```

As we can see, both code snippets refer to the same squaring method for real numbers and JML is much tidier than VDM-SL

## What are JML tools good for?

Pertaining to JML tools, there is a plethora of options, yet all of them share some of the features, those being:

- typo-checker
- a type-checker to find any assertions which refer to fields which no longer exist together
- annotation parsing

## What are JML tools good for?

And for some more specific tools:

- **jmlc** - runtime assertion checking & assertion violation guard
- **jmlunit** - similar but combined with unit testing
- **jmlspec** - helps the user generate JML specifications
- **Daikon** - detects possible invariants
- **Houdini** - uses Java to deduce annotations for code

## Bare-bones structure

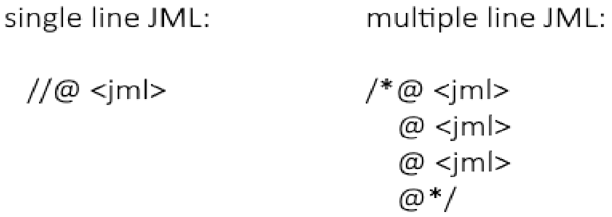
JML specifications imply that all the methods of a particular data type ensure an outcome as long as the conditions are satisfied

Preconditions are specified using the "*requires*" clause and postconditions via the "*ensures*" clause.

Throughout the code if we need to use an invariant, we can do so by using the "*invariant*" keyword.

## Bare-bones structure example

The JML annotations which are contained inside the Java source code, are put inside comments that start with the symbol "@"



single line JML:

```
//@ <jml>
```

multiple line JML:

```
/*@ <jml>
  @ <jml>
  @ <jml>
  @*/
```

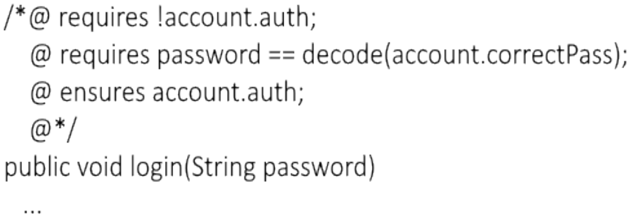
## JML Annotations

Just like Java, JML also have multiple types of annotations, some examples being:

- Modifiers - "**pure**" , "**spec_public**" , "**helper**" and so on.
- Clauses
- Types - "\**real**", "\**bigint**", etc.
- Expression tokens - can be either single words like \**result** or function-like, such as \**old(x)**

## Annotation placement

These annotations are generally located above the method they specify, similar to invariants, which are also usually placed before the fields it mentions. Example below



```
/*@ requires !account.auth;
  @ requires password == decode(account.correctPass);
  @ ensures account.auth;
  @*/
public void login(String password)
  ...
```

## Lightweight vs Heavyweight

In JML, specifications can be either **lightweight** or **heavyweight**. Lightweight specifications are where the user specifies only what they deem as important. For the unspecified attributes, both specifications give default values however the heavyweight expects the user to know the default values as they are not as forgiving.

## Lightweight vs Heavyweight

In order to go from a lightweight specification to a heavyweight one, the user must include this at the beginning of the annotation:

```
[optional privacy modifier] behavior-keyword
```

Where "*behavior-keyword*" should be replaced by behavior, normal_behavior or exceptional_behavior

## Lightweight example

```
public abstract class Light {
protected boolean init_state, end_goal;
protected int helper;
/*@
  @ requires init_state;
  @ assignable helper;
  @ ensures end_goal;
  @*/
protected abstract int some_method();
}
```

## Heavyweight example

```
public abstract class LightAsHeavy {
protected boolean init_state, end_goal;
protected int helper;
/*@ protected behavior
  @ requires init_state;
  @ diverges false;
  @ assignable helper;
  @ working_space \not_specified;
  @ ensures end_goal;
  @ signals_only \nothing;
  @ signals \not_specified;
  @*/
protected abstract int someMethod();
}
```

## Default return values of unspecified clauses

A fairly important detail is how there are different default values for unspecified clauses. These are determined by the type of specification they are a part of. The listing will specify first the lightweight first and heavyweight second.

## Default return values of unspecified clauses

- **requires** & **ensures**. "\*not_specified*" - "*true*"
- **diverges**. "*false*" for both.
- **assignable**. "\*not_specified*" - "\*everything*"
- **accessible**. "\*everything*" for both.
- **callable**. "\*everything*" for both.
- **working_space**. "\*not_specified*" for both.
- **signals_only**. "\*nothing*" for both.
- **signals**. "\*not_specified*" - "*true*"
- **when**. "\*not_specified*" - "*true*"

## How to start implementing JML into your programs

To write JML specifications we need to place them into specially formatted Java comments. This means that a Java compiler will ignore the JML text. JML specifications have to be written in comments that either

1. begin with "//@ "
2. begin with "/*@ " and end with "@*/"

## How to start implementing JML into your programs

It is common practice for lines within such a block comment to have the first non-whitespace characters be a series of @ symbols followed by a space.

```
public class ClassName {
    /*@ requires requirement1;
      @ ensures assurance;
      @*/
    void method() { ... }
}
```

## Implementing JML for BubbleSort

Let us take Bubble Sort as an example method. This method takes one argument, namely an array, which must not be NULL. Therefore, below the declaration of the method and above the first line of code, we have to specify a precondition using the requires keyword.

```
public class BubbleSort {
 //@ requires arr != null;
 public static void sort(int [] arr) {
   for (int i = 0; i < arr.length; i++) {
     for (int j = arr.length-1; j > i; j--) {
       if (arr[j-1] < arr[j]) {
         int tmp = arr[j];
         arr[j] = arr[j-1];
         arr[j-1] = tmp;
         }
       }
     }
   }
 }
```

However, we will not stop here. This method should sort the given array, which means that we will also have to add a postcondition using the ensures keyword and due to that, we must also change the comment from a single line to a multiple line one.

```
public class BubbleSort {
/*@
  @ requires arr != null;
  @ ensures \forall int k; 0 <= k && k < arr.length -1;
  @              arr[k] > arr[k+1];
  @*/
 public static void sort(int [] arr) {
   for (int i = 0; i < arr.length; i++) {
     for (int j = arr.length-1; j > i; j--) {
       if (arr[j-1] < arr[j]) {
          int tmp = arr[j];
          arr[j] = arr[j-1];
          arr[j-1] = tmp;
       }
     }
   }
  }
 }
```

With the method definition done, we shall now move into the insides of the method. However, worth noting is that we use the \forall expression which in the given example will increment k by one, as long as it satisfies the given bounds.

```
public static void sort(int [] arr) {
  /*@ final ghost int n = arr.length;
    @ loop_invariant 0 <= i <= n;
    // elements up-to i are sorted
    @ loop_invariant \forall int k; 0<= k < i;
    @                \forall int l; k < l < n;
    @                 arr[k] >= arr[l];
    @ decreasing n-i;
    @*/
  for (int i = 0; i < arr.length; i++) {
     for (int j = arr.length-1; j > i; j--) {
       if (arr[j-1] < arr[j]) {
         int tmp = arr[j];
         arr[j] = arr[j-1];
         arr[j-1] = tmp;
       }
     }
  }
}
```

We now have arranged the first loop too, therefore we can move onto the second. But, before that, we can see that in the scope above, a ghost variable has been used. A ghost variable is one that can only be accessed inside of the particular specification and it does not necessarily mirror a value of the class, nor of the method. The decreasing keyword signifies that the "$n - i$" calculation will decrease after every iteration and will be different from zero.

```
for (int i = 0; i < arr.length; i++) {
  /*@ loop_invariant i <= j <= n-1;
    // j-th element is always the largest
    @ loop_invariant \forall int k; j <= k < n;
    @                   arr[j] >= arr[k];
    // elements up-to i remain sorted
    @ loop_invariant \forall int k; 0 <= k < i;
    @                  \forall int l; k < l < n;
    @                  arr[k] >= arr[l];
    @ decreasing j;
    @*/
     for (int j = arr.length-1; j > i; j--) {
       if (arr[j-1] < arr[j]) {
         int tmp = arr[j];
         arr[j] = arr[j-1];
         arr[j-1] = tmp;
        }
      }
}
```

And with that, our implementation of JML in the BubbleSort algorithm is now complete and ready to be ran.

## Miscellaneous

The presentation you have been given comes alongside a paper on JML which goes over the same aspects but in more detail. In addition, the full BubbleSort JML code and another example with a more in-depth insight will be found in there together with a well-explained step-by-step tutorial on how to set up JML.

## Conclusion

To conclude, we will be answering a very common question:

- "*Should I use JML?*"

And the answer to that is **Yes!**

Unlike a decade ago, needs for tools to fix mismanagement of and wrongly implemented code have arisen and in furtherance of future large scale projects, of which code has to be proven correct before execution or launch, it is recommended to consider specification languages such as JML.

# Thank you for your attention.