Contents

18.1 Axiomatic Semantics
18.2 Formal Methods Tools: JML
18.2.1 JML Exception Handling
18.3 Correctness of Object-Oriented Programs
18.3.1 Design by Contract
18.3.2 The Class Invariant
18.3.3 Correctness of a Stack Application
18.3.4 Final Observations
18.4 Correctness of Functional Programs
Background

OO systems focuses on classes and objects

*Methods and messages are subordinate*

The state of a system is the set of *all active objects and their values* at any moment of run time.

Formal specifications $P$ and $Q$ are therefore *logical expressions about an object’s state.*

Tools for formal specifications:

- Specifications: *Java Modeling Language (JML)*
- Design: *Unified Modeling Language (UML) and JML*
- Coding: *Java and JML*
- Verification: *Java and JML*
Specifications in OO Programs

Where?

- Method level: *pre- and post-conditions, loop invariants*
- Class level: *class invariant (class state)*
- System level: *intra-class invariants (system state)*

When (in the OO design process)?

- Specification and design phases:
  Write specifications for all classes and methods (UML/JML)
- Coding phase:
  Develop code from the specifications (UML/JML/Java)
- Verification phase:
  Prove that specifications and code are *equivalent* (JML/Java)
What is JML? (www.jmlspecs.org)

History
  – Emerged in early 2000s out of ESC/Java2

Goals
  — Infuse formal methods into the software process
  — Make formal specification accessible to programmers
  — Provide direct support for “design by contract” methodology
  — Integrate with a real language (Java)

JML is a language for writing specifications in Java
  — Preconditions
  — Postconditions
  — Loop invariants
  — Class invariants
18.2 Formal Methods Tools: JML

JML specifications are *special comments* in a Java program:

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

for one-liners

for multiple-liners

The Hoare triple

\[
\{P\} \ s_1; \ s_2; \ ...; \ s_n \ \{Q\}
\]

is written in JML/Java as

```
/*@ requires \(P\); 
ensures \(Q\); 
*/

type method (parameters) {
  local variables
  \(s_1; \ s_2; \ ...; \ s_n\)
}
```

\(P\) and \(Q\) are extensions of Java *boolean* expressions; they may use parameters, locals, and class variables as arguments.
## JML Language Summary

<table>
<thead>
<tr>
<th>JML Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>requires p;</code></td>
<td>( p ) is a precondition for the call</td>
</tr>
<tr>
<td><code>ensures p;</code></td>
<td>( p ) is a postcondition for the call</td>
</tr>
<tr>
<td><code>signals (E e) p;</code></td>
<td>when exception ( e ) is raised by the call, ( p ) is a postcondition</td>
</tr>
<tr>
<td><code>loop_invariant p;</code></td>
<td>( p ) is a loop invariant</td>
</tr>
<tr>
<td><code>invariant p;</code></td>
<td>( p ) is a class invariant</td>
</tr>
<tr>
<td><code>\result == e;</code></td>
<td>( e ) is the result returned by the call</td>
</tr>
<tr>
<td><code>\old v</code></td>
<td>the value of ( v ) at entry to the call</td>
</tr>
<tr>
<td><code>\product int x ; p(x); e(x))</code></td>
<td>the product of ( e(x) ) for all ( x ) that satisfy ( p(x) )</td>
</tr>
<tr>
<td><code>\sum int x ; p(x); e(x))</code></td>
<td>the sum of ( e(x) ) for all ( x ) that satisfy ( p(x) )</td>
</tr>
<tr>
<td><code>\forall int x ; p(x))</code></td>
<td>states that ( p(x) ) holds for all values of ( x )</td>
</tr>
<tr>
<td><code>p ==&gt; q</code></td>
<td>( p \supset q )</td>
</tr>
</tbody>
</table>

*Note*: \( p \) is a good old fashioned Java boolean expression.
E.g., consider the Hoare triple:

```c
{n ≥ 1} int Factorial (int n) {
    int f = 1;
    int i = 1;
    {1 ≤ i ∧ i ≤ n ∧ f = i!} while (i < n) {
        i = i + 1;
        f = f * i;
    } return f;
} {f = n!}
```

- **Precondition** $P$: \{n ≥ 1\}
- **_loop invariant** $R$: \{1 ≤ i ∧ i ≤ n ∧ f = i!\}
- **Postcondition** $Q$: \{f = n!\}
Here is the JML encoding

```java
/*@ requires 1 <= n ; 
   ensures \result == (\product int i; 1<=i && i<=n; i) ; @*/
int Factorial (int n) {
    int f = 1;
    int i = 1;
    /*@ loop_invariant i<=n && f==(\product int j; 1<=j && j<=i; j); @*/
    while (i < n) {
        i = i + 1;
        f = f * i;
    }
    return f;
}
```
JML software tools

1. Compiling (use jmlc instead of javac)
   - Does syntactic and type checking, and byte code generation for all JML assertions and Java code

2. Static checking (ESC/Java2)

3. Runtime assertion checking (use jmlrac instead of java)
   - Checks precondition $P$ at entry to every call
   - Checks postcondition $Q$ at exit from every call
   - Checks loop invariant $R$ before every iteration
   - Issues a Java Exception when any of these is not true

4. Proof assistance tools (Daikon, LOOP)
JML Eclipse Environment

1. Compiling

3. Runtime assertion checking
Let’s try some tests with our example:

```java
public class myFactorial {
   /*@ requires 1 <= n;
    ensures \result == (\product int i; 1<=i && i<=n; i);
    @*/
    static int Factorial (int n) {
        ...
    }
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        System.out.println("Factorial of " + n + " = " + Factorial(n));
    }
}
```
Here’s a compile and two runs:

% jmlc -Q myFactorial.java
% jmlrac myFactorial 3
Factorial of 3 = 6
% jmlrac myFactorial -5
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError:
by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 3, character 15 when
'\n' is -5
at myFactorial.checkPre$Factorial$myFactorial(myFactorial.java:240)
at myFactorial.Factorial(myFactorial.java:382)
at myFactorial.main(myFactorial.java:24)
# JML Exceptions

<table>
<thead>
<tr>
<th>JML Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMLEntryPreconditionError</td>
<td>A method call’s arguments do not satisfy the method’s <code>requires</code> clause.</td>
</tr>
<tr>
<td>JMLEntryPostconditionError</td>
<td>A method call exits normally, but its result does not satisfy its <code>ensures</code> clause.</td>
</tr>
<tr>
<td>JMLExceptionPostconditionError</td>
<td>A method call exits abnormally, raising an exception defined by its <code>signals</code> clause.</td>
</tr>
<tr>
<td>JMLLoopInvariantError</td>
<td>Some execution of a loop’s body does not satisfy its <code>loop_invariant</code> clause.</td>
</tr>
<tr>
<td>JMLInvariantError</td>
<td>A method call does not leave the object in a state that satisfies its <code>invariant</code> clause.</td>
</tr>
</tbody>
</table>
JML helps identify errors

Example 1: Suppose we change the while loop from

```java
while (i < n)
```

to

```java
while (i <= n)
```

so that \( n! \) will be computed incorrectly.

```shell
% jmlrac myFactorial 3
Exception in thread "main"
  org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError:
  by method myFactorial.Factorial regarding specifications at
    File "myFactorial.java", line 9, character 24 when
    'n' is 3
  at myFactorial.internal$Factorial(myFactorial.java:102)
  at myFactorial.Factorial(myFactorial.java:575)
  at myFactorial.main(myFactorial.java:211)
```

invariant not satisfied
JML Example 2

Suppose we change the while loop from

```java
while (i < n) to
while (i <= n)
```

and also remove the JML loop invariant. Now we get:

```plaintext
% jmlrac myFactorial 3
Exception in thread "main"
    org.jmlspecs.jmlrac.runtime.JMLNormalPostconditionError:
    by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 4, character 23 when 'n' is 3 '
result' is 24
at myFactorial.checkPost$Factorial$myFactorial(myFactorial.java:321)
at myFactorial.Factorial(myFactorial.java:392)
at myFactorial.main(myFactorial.java:24)
```

postcondition not satisfied
JML Example 3

Disagreement between a JML specification and a program may signal an error in the specification. E.g., if the loop invariant had been \( j \leq i \) rather than \( j < i \) the following result would occur:

% jmlrac myFactorial 3
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLLoopInvariantError:
by method myFactorial.Factorial regarding specifications at
File "myFactorial.java", line 9, character 24 when 'n' is 3
at myFactorial.internal$Factorial(myFactorial.java:101)
at myFactorial.Factorial(myFactorial.java:573)
at myFactorial.main(myFactorial.java:209)
But beware... JML is no silver bullet

jmlrac doesn’t trap all errors... here are two “normal” runs:

% jmlrac myFactorial 21
Factorial of 21 = -1195114496
% jmlrac myFactorial 32
Factorial of 32 = -2147483648

Wrong answers, but no JML messages!

Recall: (1) Java has no ArithmeticOverflow exception, but
(2) Factorial(n) for n > 12 should give a result > \(2^{31}-1\)

Note: jmlrac computes the same wrong result (when it checks the postcondition)
as the Factorial method computes, so this error goes undetected.

Conclusion: the program and its specifications are both wrong.
## Design by Contract

<table>
<thead>
<tr>
<th></th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>Arguments for each method/constructor call must satisfy its <code>requires</code> clause</td>
<td>The call delivers correct result, and the object keeps its integrity</td>
</tr>
<tr>
<td>(caller)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>Result for each call must satisfy both its <code>ensures</code> clause and <code>INV</code></td>
<td>Called method/constructor doesn’t need extra code to check argument validity</td>
</tr>
<tr>
<td>(object)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note*: Blame can be assigned if obligations aren’t met!
The contract for a Stack class

<table>
<thead>
<tr>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client (caller)</strong></td>
<td><strong>Benefits</strong></td>
</tr>
<tr>
<td>Arguments for every call to Stack(), push, pop, top, isEmpty, and size</td>
<td>Every call to Stack(), push, pop, top, isEmpty, and size</td>
</tr>
<tr>
<td>must satisfy its requires clause</td>
<td>delivers a correct result, and the object keeps its integrity</td>
</tr>
<tr>
<td><strong>Class (object)</strong></td>
<td></td>
</tr>
<tr>
<td>Result from each call must satisfy both its ensures clause and the class</td>
<td>No method or constructor needs extra code to check argument validity</td>
</tr>
<tr>
<td>invariant ( n = \text{size}() )</td>
<td></td>
</tr>
</tbody>
</table>
18.3.2 The Class Invariant

A class $C$ is formally specified if:

1. Every constructor and public method $M$ in the class $C$ has preconditions and postconditions, and
2. $C$ has a special predicate called its class invariant $INV$ which, for every object $o$ in $C$, argument $x$ and call $o.M(x)$, must be true both before and after the call.

Note: During a call, $INV$ may temporarily become false.

Why are we doing this?

i) Formal specifications provide a foundation for rigorous OO system design (e.g., “design by contract”).

ii) They enable static and dynamic assertion checking of an entire OO system.

iii) They enable formal correctness proof of an OO system.
18.3.3 Correctness of a Stack Application

```java
public class Stack {
    private class Node {
    }
    private Node theStack = null;
    private int n = 0;
    public void push(int v) {
    }
    public int pop() {
    }
    public int top() {
    }
    public boolean isEmpty() {
    }
    public int size() {
    }
}
```

- **public constructor** $C = \text{Stack}()$
- "helper" class
- state variables help define $INV$
- public methods $M$
Adding Class-Level Specifications

/*@ public model Node S;
   private represents S <- theStack;
   public invariant S == null || n == this.size();
@*/

private /*@ spec_public @*/ Node theStack = null;
private /*@ spec_public @*/ int n = 0;

Notes: 1) JML model variables allow a specification to
distance itself from the class’s implementation details.
2) spec_public allows JML specifications to treat a Java
variable as public without forcing the code to do the same.
Adding Method Specifications

/*@ requires n > 0;
   ensures result==\old(S).val &&
   S==\old(S).next && n==\old(n)-1;
@*/

public /*@ pure @*/ int pop( ) {
  int result = theStack.val;
  theStack = theStack.next;
  n = n-1;
  return result;
}

Notes: 1) \old denotes the value of S at entry to pop.
2) The ensures clause specifies that pop removes
   the top element and returns it.
Similar specifications for push

/*@ ensures S.next==old(S) && S.val==v; @*/

public /*@ pure @*/ void push(int v) {
    theStack = new Node(v, theStack);
    n = n+1;
}
“Pure” methods

/*@ requires n > 0;
   ensures \result==S.val && S == \old(S);
@*/
public /*@ pure @*/ int top() {
    return theStack.val;
}

*Note*: A method is *pure* if:
1) it has no non-local side effects, and
2) it is provably non-looping.
public class myStackTest {
    public static void main(String[] args) {
        MyStack s = new MyStack();
        int val;
        for (int i=0; i<args.length; i++)
            s.push(Integer.parseInt(args[i]));
        System.out.println("Stack size = " + s.size());
        System.out.print("Stack contents = ");
        for (int i=1; i<=n; i++) {
            System.out.print(" " + s.top());
            s.pop();
        }
        System.out.println();
        System.out.println("Is Stack empty? " + s.isEmpty());
    }
}
Class and System Correctness

1. A class $C$ is (formally) correct if:
   a. It is formally specified, and
   b. For every object $o$ and every constructor and public method $M$ in the class with precondition $P$ and postcondition $Q$, the Hoare triple
      \[
      \{ P \land \text{INV} \} \ o.M(x) \ {Q \land \text{INV}}
      \]
      is valid for every argument $x$.

2. A system is correct if all its classes are correct.
Correctness of pop() 

/*@ requires n > 0; 
ensures \result==\old(S).val &&
S==\old(S).next && n==\old(n)-1; */

public /*@ pure @*/ int pop( ) {
  int result = theStack.val;
  theStack = theStack.next;
  n = n-1;
  return result;
}

INV: n = size()
P \wedge INV: n > 0 \wedge n = size()
Q \wedge INV: result = old(S).val \wedge
  S = old(S).next \wedge
  n = size()
A “loose” correctness proof for pop()

1. “Loose” because
   a. We assume the correctness of size(), and
   b. We omit some details.

2. The assignments in pop(), together with $P \land INV$, ensure the validity of $Q \land INV$:
   a. Steps 1 and 4 establish $\text{result} = \text{old}(S).\text{val}$
   b. Step 2 establishes $S = \text{old}(S).\text{next}$
   c. Step 3 and our assumption establish $n = \text{size}()$:
      I.e., $n = \text{old}(n) - 1$
      and $\text{size}() = \text{old}(\text{size}()) - 1$
      So, $n = \text{size}()$, since $\text{old}(n) = \text{old}(\text{size}())$
Final Observations

1. Formal verification:
   a. is an enormous task for large programs.
   b. only proves that specifications and code agree.
   c. only proves partial correctness (assumes termination).

2. Tools exist for:
   a. Statically checking certain run-time properties of Java programs (ESC/Java2)
   b. formally verifying Ada programs (Spark)

3. Tools are being developed to help with formal verification of Java programs (Diacron, LOOP)

4. What is the cost/benefit of formal methods?