



# Advances in Programming Languages

## Lecture 13: Practical Tools for Java Correctness

Ian Stark

School of Informatics  
The University of Edinburgh

Monday 29 October 2018  
Semester 1 Week 7



THE UNIVERSITY  
of EDINBURGH

<https://wp.inf.ed.ac.uk/apl18>  
<https://course.inf.ed.ac.uk/apl>

# Outline

- 1 Opening
- 2 Assertions in Java
- 3 JML, the Java Modeling Language
- 4 Closing

# Outline

1 Opening

2 Assertions in Java

3 JML, the Java Modeling Language

4 Closing

# Topic: Augmented Languages for Correctness and Certification

The next block of lectures cover some language techniques and tools for improving program correctness:

- Specification and Verification
- Practical Java tools for Correctness
- Separation Logic
- Augmented Programming and Certifying Correctness

The focus here is not necessarily on changing what a program does, or making it do that thing faster, or using less memory, or less power. Instead we want to make sure that what it does is the right thing to do.

This lecture shows some methods used in Java programming based on ideas from Hoare Logic.

## Keep doing this! It's working well

- Being enthusiastic
- Homework and pointers to further information
- Getting students to look properly at each of the assignment options
- Concurrency and polymorphism
- Being interesting
- Printed handouts

## Stop this! I don't find it helpful

- Running over time

Actually, I was the one who put that in.

## Start this! I think it's worth a try

- More coding examples
- Exam preparation
- Tutorials

(Yes, I'll do that, final lectures)

(Maybe in a 20-credit version?)

## About you

What steps can you take to improve your own learning in this course?

- Do those homework exercises
- Don't leave coursework until the last minute
- Refresh my understanding of the lambda-calculus



## About you

Which statement best describes how you feel about the level of challenge in this course?



40%




60%



# Homework from Thursday

The lecture on Monday will look at some tools for checking Java programs, including those that apply Hoare Logic and ideas from Design by Contract™. Before then:

## 1. Read this

 Gary Leavens and Yoonsik Cheon  
Design by Contract with JML  
<http://www.jmlspecs.org/jmldbc.pdf>:

## 2. Do this

- Find some information online about *assertions* in Java — a tutorial, Q+A, a discussion, a blog post, ...
- Send me a link to this by email.

# Homework results: Links about Java assertions

WikiBooks: Java Programming

[https://en.wikibooks.org/wiki/Java\\_Programming/Keywords/assert](https://en.wikibooks.org/wiki/Java_Programming/Keywords/assert)

Oracle: Programming With Assertions

<http://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>

Geeks for Geeks: Assertions in Java

<https://www.geeksforgeeks.org/assertions-in-java/>

StackOverflow: Java `assert` vs. JUnit Assertions?

<https://stackoverflow.com/questions/2966347/assert-vs-junit-assertions/6267182>

Grégory Pakosz: Assertions or Exceptions?

<http://pempek.net/articles/2013/11/16/assertions-or-exceptions/>

# Outline

- 1 Opening
- 2 Assertions in Java**
- 3 JML, the Java Modeling Language
- 4 Closing

- What it is.
- When to use it.
- Characteristics.

# Java **assert**: What It Is

## Simple assertion

```
assert Expression1 ;
```

Here **Expression1** is a boolean expression. When executed, if **Expression1** evaluates to false then the assertion throws an **AssertionError**.

## Assertion with explanation of failure

```
assert Expression1 : Expression2 ;
```

Here:

- **Expression1** is a boolean expression.
- **Expression2** is an expression returning a value.

When executed, if **Expression1** evaluates to false then the assertion throws a **AssertionError(Expression2)**.

## Java assert: When to Use It?

- |  |       |                          |
|--|-------|--------------------------|
| ● To check arguments on the way in to a method?    | No    | (write code to handle)   |
| ● To check results on the way out of a method?     | Yes   |                          |
| ● To check an invariant inside a loop?             | Yes   |                          |
| ● For user input sanitization?                     | No    | (write code to sanitize) |
| ● To check a data structure is in a standard form? | Maybe | (if wholly private)      |
| ● To check file contents are in a standard form?   | No    | (write code to handle)   |

Use exceptions to check for problems that *might* happen; use assertions to check for problems that *mustn't*.

# Java assert: Characteristics

- Expressed in Java.  
(So you can use Java code in the thing to be tested and the error reported)
- Checked at runtime.  
(So effectiveness depends on coverage of tests.)
- Are disabled by default.  
(Why? Then what's the point of them?)
- Must be free of side-effects.  
(Why? What about new objects, or time spent on execution?)
- Can call arbitrary other Java code.  
(Provided it is free of side-effects)

Program testing can be used to show the presence  
of bugs, but never to show their absence!  
*E. W. Dijkstra, 1969*



# Outline

- 1 Opening
- 2 Assertions in Java
- 3 JML, the Java Modeling Language**
- 4 Closing

## Hoare Triple

$$\{P\} C \{Q\}$$

- A Hoare assertion  $\{P\} C \{Q\}$  states that if *precondition*  $P$  holds and code  $C$  runs to completion then *postcondition*  $Q$  will hold afterwards.
- Assertions  $\vdash \{P\} C \{Q\}$  can be derived using Hoare *rules*; they may also be tested against a *semantics*  $\models \{P\} C \{Q\}$ .
- Hoare assertions allow logical reasoning about program behaviour: notably in formal *specification* and *verification*.
- Hoare assertions are widely used in tools and languages for formal methods.
- Assertions may be strengthened to *contracts* for code, placing obligations on both caller and callee.

# Model-Based Specification

*Modeling* is an abstraction method for system design and specification.

A *model* is a representation of the desired system.

A *formal model* is one precisely described in a mathematical language.

A model differs from an implementation in that it might:

- capture only some aspects of the system (e.g., call graph);
- be partial, leaving some areas unspecified;
- not be executable.

Any implementation of the system can be assessed against the model. Sometimes the model is iteratively refined to give an implementation.

Sample applications of modeling in computer software development:

- VDM** the *Vienna Development Method*;
- B** the *B method*, *B language* and *Event B*;
- UML** the *Unified Modeling Language*;
- Z** the *Z notation* for formal specification.

# The Java Modeling Language

The *Java Modeling Language*, JML, combines model-based and contract approaches to specification.

[openjml.org](http://openjml.org) / [jmlspecs.org](http://jmlspecs.org)

Some design features:

The specification lives close to the code

Within the Java source, in *annotation comments* `/*@...@*/`

Uses Java syntax and expressions

Rather than a separate specification language.

Common language for many tools and analysis

Tools add their own extensions, and ignore those of others.

JML tools go well beyond runtime assertion checking: some prove correctness before compilation; some generate unit tests; others identify counterexamples to show possible problems.

## JML by Example: Basics

```
public class Account {  
    public int credit;  
  
    /*@ requires credit > amount && amount > 0;  
       @ ensures credit > 0 && credit == \old(credit) - amount;  
    @*/  
  
    public int withdraw(int amount) {  
        ...  
    }  
}
```

JML conditions combine logical formulae ( $\&\&$ ,  $\text{==}$ ) with Java expressions (`credit`, `amount`). Expressions used in JML must be *pure*: no side-effects.

## JML by Example: Logical Formulas

```
public class IntArray {
    public int[] contents;

    /*@ requires (\forall int i,j;
        @           0<=i && i<j && j<contents.length;
        @           contents[i] <= contents[j]);
        @
        @ ensures (contents[\result] == value) || (\result == -1);
    @*/
    public int search (int value) { ... }
}
```

The `search` routine requires that array `contents` be sorted on entry. This would, for example, be necessary if it used binary chop to locate `value`.

## JML by Example: Class Invariants

```
public class IntArray {
    public int[] contents;
    /*@ invariant (\forall int i,j;
        @           0 <= i && i < j && j < contents.length;
        @           contents[i] <= contents[j]);
    @*/

    /*@ ensures (contents[\result] == value) || (\result == -1);
    @*/
    public int search (int value) { ... }
}
```

Now **contents** must be sorted whenever it is visible to clients of **IntArray**.

## JML by Example: Assumptions and Assertions

```
/*@ assume  $j*j < contents.length$  @*/  
contents[j*j] = j;
```

...

```
a[0] = complexcomputation(a,v);  
/*@ assert ( $\backslash$ forall int  $i$ ;  $0 < i \ \&\& \ i < 10$ ;  $a[0] < a[i]$ ) @*/
```

An *assumption* may help a static analysis tool.

An *assertion* must always be satisfied — similar to Java's runtime **assert**.

Unlike Java **assert**, JML can use arbitrary logical formulas and need not be directly executable.



## JML by Example: Model Fields

```
public class IntArray {  
    public int[] contents;  
  
    /*@ model int total;  
       @ represents total = arraySum(contents)  
       @*/  
    ...  
}
```

A *model* field represents some property of the model that does not appear explicitly in the implementation.

Once defined, other JML expressions can refer to **total** just like other fields.

## JML by Example: Model Methods and Classes

```
/*@ ensures \result = (\sum int i; 0<i && i<a.length; a[i])  
  @  
  @ public model int arraySum(int[] a);  
  @*/  
  
/*@ public model class JMLSet { ... } @*/
```

Specifications may refer to *model methods* and even entire *model classes* to represent and manipulate desired system properties.

JML provides specifications for the standard Java classes, as well as a library of model classes for mathematical constructions like sets, bags, integers and reals.

Java has implementations with similar names, but these model classes are for the mathematical concepts, with arbitrary size and precision.

# JML: Behavioural Subtyping

Recall from earlier: *behavioural subtyping* is that if  $S$  is a subtype of  $T$  then any  $S$  can be substituted in place of a  $T$ .

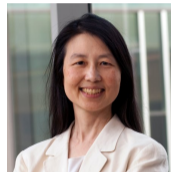
Liskov's principle of substitutivity:

... *properties that can be proved using the specification of an object's presumed type should hold even though the object is actually a subtype of that type.*

[Liskov and Wing, 1994]



Barbara Liskov  
2008 Turing Award



Jeannette Wing  
Columbia/CMU/MSR

This can be captured by requiring that when  $S$  **extends**  $T$ :

- Each precondition for  $S.m$  is implied by the preconditions of  $T.m$ ;
- The postconditions for  $S.m$  imply each postcondition for  $T.m$ ;
- The invariants of class  $S$  extend the invariants of class  $T$ .

In JML this is captured by *specification inheritance*, which enforces substitutivity between classes and subclasses.

[Leavens, 2015]

## Static JML Tools: Checking and Verification

JML annotations can be used to check code before it is executed, or even within an IDE before it is compiled.

**ESC/Java 2** carries out a range of static checks on Java programs. These include formal verification of JML annotations using a fully-automated theorem prover.

Controversially, the checker is neither sound nor complete: it warns about many potential bugs, but not all actual bugs.

This is by design: the aim is to find many possible bugs, quickly.

**KeY** is a formal software development tool based on a dynamic logic for Java. KeY accepts JML for the specification of Java code.

**OpenJML** includes tools to check JML specifications before compilation using any of the *Z3*, *CVC4* or *Yices* theorem provers (specifically, these are *SMT solvers*).

## Dynamic JML Tools: Running and Testing

JML annotations can drive various checks on running code.

**jml4c** is a compiler which inserts runtime tests for every JML annotation; if an assertion fails, an error message provides dynamic information about the failure.

**JMLUnitNG** creates unit tests based on preconditions, postconditions and invariants. These automatically exercise and test assertions made in the code.

**JMLOK2** generates both tests and runtime *oracles*, which are then run to identify places where code does not meet its specification.

**OpenJML** includes tools to compile JML annotations into runtime checks.

# Does your program do what it is supposed to do?

OpenJML is a program verification tool for Java programs that allows you to check the specifications of programs annotated in the Java Modeling Language.

[Download OpenJML](#)

```
// Can you spot the two errors
// in this program?

public class MaybeAdd {
    //@ requires a > 0;
    //@ requires b > 0;
    //@ ensures \result == a+b;
    public static int add(int a, int b){
        return a-b;
    }

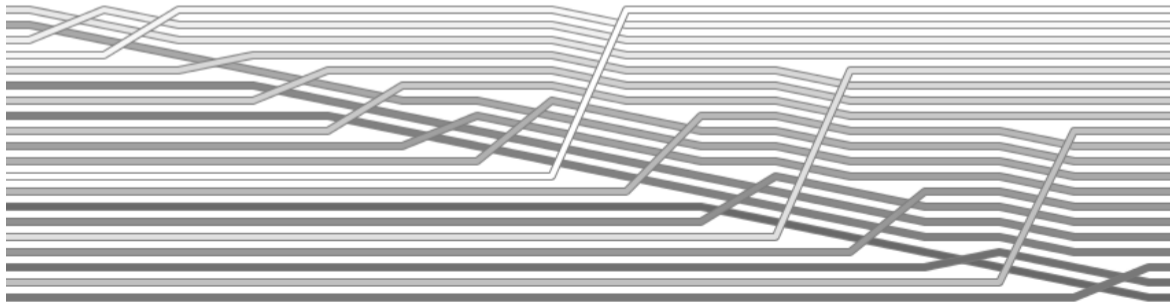
    public static void main(String args[]){
        System.out.println(add(2,3));
    }
}
```



The **TimSort** algorithm performs a stable sort on an array in memory. Based on mergesort, it's highly adaptive where some data is already sorted.

Space and time performance of TimSort is typically excellent, and even in the worst-case remains good.

The original ideas go back to 1993. TimSort was implemented for Python in 2002, soon becoming the default. It's been the default sorting algorithm for Java and Android since 2011.



In 2015 smart folk in the EU *Envisage* project had been using JML in the **KeY** tool to verify some standard sorting algorithms in Java.

They moved on to the more complex TimSort implementation, and were unable to prove an important invariant of the algorithm. This turned out to be because it wasn't true. It wasn't true in the Python implementation either.

The smallest Java counterexample is a 67,108,864-element array. The smallest Python one requires at least  $2^{49}$  elements (0.5 petaelement).

Java TimSort has been adjusted so that the smallest counterexample is now bigger than the largest **int**-indexed array.

Python TimSort has been fixed so that the invariant is true.

<https://is.gd/timfix> <http://www.envisage-project.eu/timsort-specification-and-verification>

<http://www.envisage-project.eu/key-deductive-verification-of-software>



# Outline

- 1 Opening
- 2 Assertions in Java
- 3 JML, the Java Modeling Language
- 4 Closing

## Assertions in Java

- Statements in code of programmer expectations.
- Can be checked at runtime during testing.
- Supplement, but are different from, standard exceptions.

## The Java Modeling Language

- JML combines model-based and contract-style specification
- Annotations within code: **requires**, **ensures**, ...
- Provides model fields, methods and classes.
- Common input language for many tools: runtime checks, static analyses, test generators, invariant guessers, etc.

# Homework

JML is just one of many frameworks for specifying and verifying code in various programming languages.

For example: Java annotations, FindBugs<sup>TM</sup>, QuickCheck, the C specification language ACSL and Frama-C platform, Spec#, ...

Your homework is to use a small part of one such system.

## 1. Read This

Find out about *Type Annotations and Pluggable Type Systems* by reading the Oracle Java Tutorial lesson with that name.

## 2. Do this

Try out the `@NonNull` and `@Nullable` annotations on the *Checker Framework* live demo:

- <http://eisop.uwaterloo.ca/live>

The *Checker Framework* provides twenty different checkers for Java source code through *pluggable type-checking*.

- <http://checkerframework.org/>

Browse the manual there to see some of the example checkers.

Work through one of the tutorials — for these you will need to install the Checker Framework and use it through Eclipse or another Java IDE.

- <https://types.cs.washington.edu/checker-framework/tutorial/>
- <https://github.com/glts/safer-spring-petclinic/wiki>