A JML Tutorial

Modular Specification and Verification of Functional Behavior for Java

Gary T. Leavens[1] Joseph R. Kiniry[2] Erik Poll[3]

[1]School of Electrical Engineering and Computer Science
University of Central Florida

[2]School of Computer Science and Informatics
University College Dublin

[3]Computing Science Department
Radboud University Nijmegen

March 27, 2008 / JML Tutorial / jmlspecs.org

Objectives

You'll be able to:

- Explain JML's goals.
- Read and write JML specifications.
- Use JML tools.
- Explain basic JML semantics.
- Know where to go for help.

Tutorial Outline

1. JML Overview
2. Reading and Writing JML Specifications
3. Abstraction in Specification
4. Subtyping and Specification Inheritance
5. ESC/Java2
6. Conclusions

Introduce Yourself, Please

Question

Who you are?

Question

How much do you already know about JML?

Question

What do you want to learn about JML?

Outline

Java Modeling Language

Currently:

- Formal.
- Sequential Java.
- Functional behavior of APIs.
- Java 1.4.

Working on:

- Detailed Semantics.
- Multithreading.
- Temporal Logic.
- Java 1.5 (generics).

Java Modeling Language

Currently:

- Formal.
- Sequential Java.
- Functional behavior of APIs.
- Java 1.4.

Working on:

- Detailed Semantics.
- Multithreading.
- Temporal Logic.
- Java 1.5 (generics).

JML's Goals

- Practical, effective for detailed designs.
- Existing code.
- Wide range of tools.

Detailed Design Specification

Handles:

- Inter-module interfaces.
- Classes and interfaces.
- Data (fields)
- Methods.

Doesn't handle:

- User interface.
- Architecture, packages.
- Dataflow.
- Design patterns.

Detailed Design Specification

Handles:

- Inter-module interfaces.
- Classes and interfaces.
- Data (fields)
- Methods.

Doesn't handle:

- User interface.
- Architecture, packages.
- Dataflow.
- Design patterns.

Basic Approach

"Eiffel + Larch for Java"

- Hoare-style (Contracts).
- Method pre- and postconditions.
- Invariants.

A First JML Specification

```
public class ArrayOps {

  private /*@ spec_public @*/ Object[] a;

  //@ public invariant 0 < a.length;

  /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
  public void init(Object[] arr) {
    this.a = arr;
  }
```

Field Specification with spec_public

```
public class ArrayOps {

  private /*@ spec_public @*/ Object[] a;

  //@ public invariant 0 < a.length;

  /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
  public void init(Object[] arr) {
    this.a = arr;
  }
```

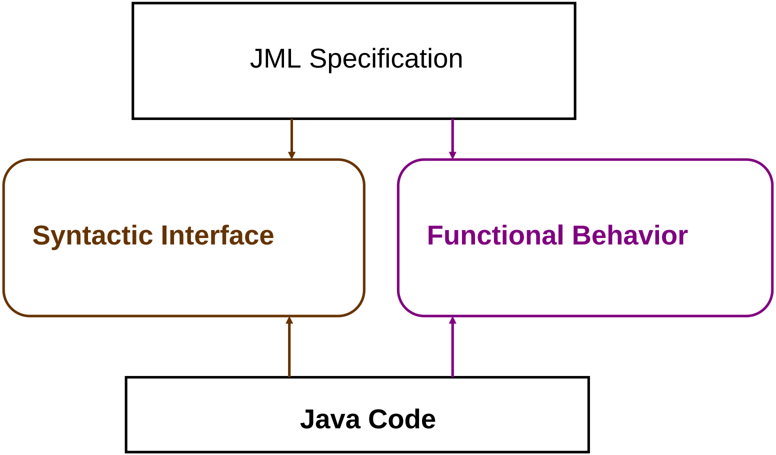
Object Invariant

```
public class ArrayOps {

  private /*@ spec_public @*/ Object[] a;

  //@ public invariant 0 < a.length;

  /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
  public void init(Object[] arr) {
    this.a = arr;
  }
```

Method Specification with requires, ensures

```
public class ArrayOps {

  private /*@ spec_public @*/ Object[] a;

  //@ public invariant 0 < a.length;

  /*@ requires 0 < arr.length;
    @ ensures this.a == arr;
    @*/
  public void init(Object[] arr) {

    this.a = arr;
  }
```

Interface Specification



Interface Specification

```
/*@  requires 0 < arr.length;
  @   ensures this.a == arr;   @*/
public void init(Object[] arr);
```

```
public void init(Object[] arr);
```

```
requires 0 < arr.length;
ensures this.a == arr;
```

```
public void init(Object[] arr)
{ this.a = arr; }
```

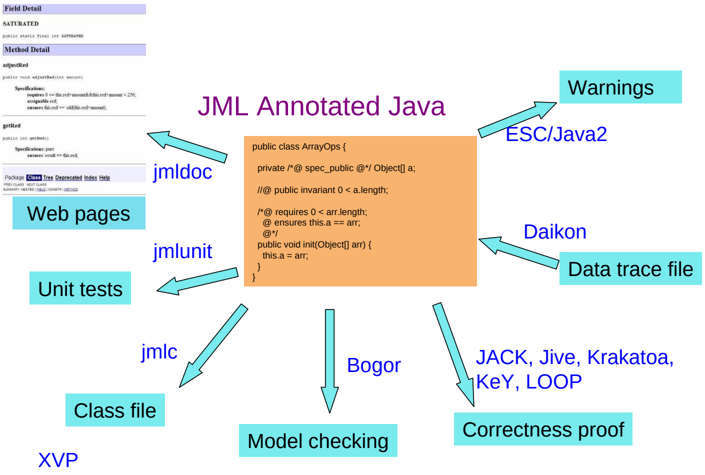
Like . . . But for Java and . . .

- VDM, but
  - OO features
- Eiffel, but
  - Features for formal verification
- Spec#, but
  - Different invariant methodology
  - More features for formal verification

Unlike OCL and Z

- More Java-like syntax.
- Tailored to Java semantics.

Many Tools, One Language



How Tools Complement Each Other

- Different strengths:
  - Runtime checking — real errors.
  - Static checking — better coverage.
  - Verification — guarantees.
- Usual ordering:
  1. Runtime checker (jmlc and jmlunit).
  2. Extended Static Checking (ESC/Java2).
  3. Verification tool (e.g., KeY, JACK, Jive).

Interest in JML

- Many tools.
- State of the art language.
- Large and open research community:
  - 23 groups, worldwide.
  - Over 135 papers.

See jmlspecs.org

Advantages of Working with JML

- Reuse language design.
- Ease communication with researchers.
- Share customers.

Join us!

Opportunities in Working with JML

Or: What Needs Work

- Tool development, maintenance.
- Extensible tool architecture.
- Unification of tools.

Where to Find More: jmlspecs.org

Documents:

- "Design by Contract with JML"
- "An overview of JML tools and applications"
- "Preliminary Design of JML"
- "JML's Rich, Inherited Specifications for Behavioral Subtypes"
- "JML Reference Manual"

Also:

- Examples, teaching material.
- Downloads, sourceforge project.
- Links to papers, etc.

Outline

JML Annotation Comments $\neq$ Java Annotations

JML annotation comments:

- Line starting with `//@`
- Between `/*@` and `@*/`, ignoring `@`'s starting lines.

First character must be `@`

JML Annotations Comments $\neq$ Java Annotations

Question

What's wrong with the following?

```
// @requires 0 < arr.length;
// @ensures this.a == arr;
public void init(Object[] arr)
```

Most Important JML Keywords

Top-level in classes and interfaces:

- **`invariant`**
- **`spec_public`**
- **`nullable`**

For methods and constructors:

- **`requires`**
- **`ensures`**
- **`assignable`**
- **`pure`**

Example: BoundedStack

Example

Specify bounded stacks of objects.

BoundedStack's Data and Invariant

```
public class BoundedStack {

  private /*@ spec_public nullable @*/
     Object[] elems;
  private /*@ spec_public @*/ int size = 0;

  //@ public invariant 0 <= size;
  /*@ public invariant elems != null
    @   && (\forall int i;
    @              size <= i && i < elems.length;
    @              elems[i] == null);
    @*/
```

BoundedStack's Constructor

```
/*@ requires 0 < n;
  @ assignable elems;
  @ ensures elems.length == n;
  @*/
public BoundedStack(int n) {
  elems = new Object[n];
}
```

BoundedStack's push Method

```
/*@ requires size < elems.length-1;
  @ assignable elems[size], size;
  @ ensures size == \old(size+1);
  @ ensures elems[size-1] == x;
  @ ensures_redundantly
  @      (\forall int i; 0 <= i && i < size-1;
  @                  elems[i] == \old(elems[i]));
  @*/
public void push(Object x) {
  elems[size] = x;
  size++;
}
```

BoundedStack's pop Method

```
/*@ requires 0 < size;
  @ assignable size, elems[size-1];
  @ ensures size == \old(size-1);
  @ ensures_redundantly
  @      elems[size] == null
  @   && (\forall int i; 0 <= i && i < size-1;
  @                elems[i] == \old(elems[i]));
  @*/
public void pop() {
  size--;
  elems[size] = null;
}
```

BoundedStack's top Method

```
  /*@ requires 0 < size;
    @ assignable \nothing;
    @ ensures \result == elems[size-1];
    @*/
  public /*@ pure @*/ Object top() {
    return elems[size-1];
  }
}
```

spec_public, nullable, and invariant

spec_public

- Public visibility.
- Only public for specification purposes.

nullable

- field (and array elements) may be null.
- Default is **non_null**.

invariant must be:

- True at end of constructor.
- Preserved by each method.

requires and ensures

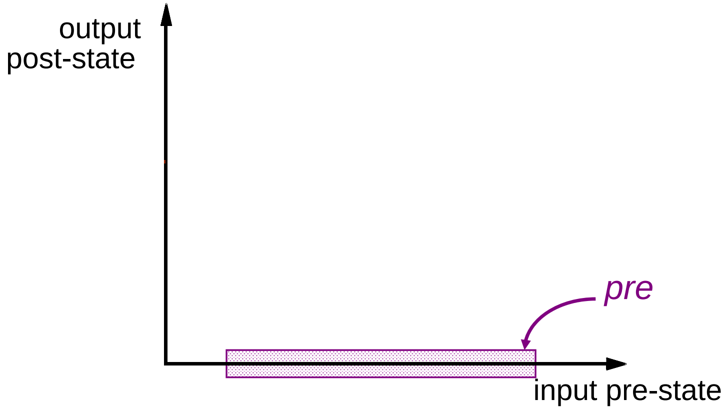
`requires` clause:

- Precondition.
- Obligation on callers, after parameter passing.
- Assumed by implementor.

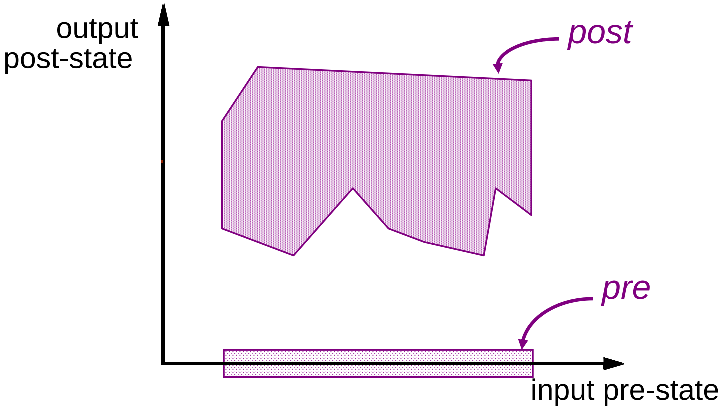
`ensures` clause:

- Postcondition.
- Obligation on implementor, at return.
- Assumed by caller.

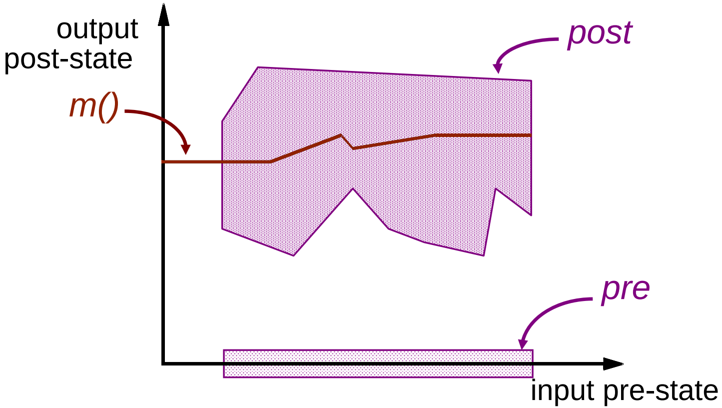
Semantics of Requires and Ensures



Semantics of Requires and Ensures



Semantics of Requires and Ensures



assignable and pure

`assignable`

- Frame axiom.
- Locations (fields) in pre-state.
- New object fields not covered.
- Mostly checked statically.
- Synonyms: **`modifies`**, **`modifiable`**

`pure`

- No side effects.
- Implies **`assignable \nothing`**
- Allows method's use in specifications.

Assignable is a Shorthand

```
assignable gender;
ensures gender.equals(g);
```

means

```
ensures \only_assigned(gender)
     && gender.equals(g);
```

Redundant Clauses

E.g., **ensures_redundantly**

- Alerts reader.
- States something to prove.
- Must be implied by:
  - **ensures** clauses,
  - **assignable** clause,
  - **invariant**, and
  - JML semantics.

Also **requires_redundantly**, etc.

Multiple Clauses

Semantics:

```
requires P;
requires Q;
```

is equivalent to:

```
requires P && Q;
```

Similarly for **ensures**, **invariant**.

Note: runtime checker gives better errors with multiple clauses.

Defaults for Omitted Clauses

- `invariant true;`
- `requires true;`
- `assignable \everything;`
- `ensures true;`

Expression Keywords

- `\result` = method's return value.
- `\old(`E`)` = pre-state value of E.
- `(\forall T x;` P; Q`)` = $\bigwedge\{Q \mid x \in T \wedge P\}$
- `(\exists T x;` P; Q`)` = $\bigvee\{Q \mid x \in T \wedge P\}$
- `(\min T x;` P; E`)` = $\min\{E \mid x \in T \wedge P\}$
- `(\sum T x;` P; E`)` = $\sum\{E \mid x \in T \wedge P\}$
- `(\num_of T x;` P; Q`)` = $\sum\{1 \mid x \in T \wedge P \wedge Q\}$
- ...

Steps for Specifying a Type for Public Clients

1. Specify data (`spec_public` fields).
2. Specify a `public invariant`.
3. Specify each public method using:
   1. `requires`.
   2. `assignable` (or `pure`).
   3. `ensures`.

Exercise: Specify BagOfInt (7 minutes)

Exercise

Specify the following:

```
public class BagOfInt {
  private                       int[] a;
  private                       int n;

  /** Initialize to contain input's elements. */
  public BagOfInt(int[] input);
  /** Return the multiplicity of i. */
  public int occurrences(int i);
  /** Return and delete the minimum element. */
  public int extractMin();
}
```

My Solution: BagOfInt's Data

```
public class BagOfInt {
  /** Elements. */
  private /*@ spec_public non_null @*/ int[] a;
  /** Number of active elements in a. */
  private /*@ spec_public @*/ int n;

  //@ public invariant 0 <= n && n <= a.length;
```

My Solution: BagOfInt's Constructor

```
/** Initialize to contain input's elements. */
/*@ assignable a, n;
  @ ensures n == input.length;
  @ ensures (\forall int i; 0 <= i && i < n;
  @                   a[i] == input[i]);   @*/
public BagOfInt(/*@ non_null @*/ int[] input);
```

My Solution: Method occurrences

```
/** Return the multiplicity of i. */
/*@ ensures \result
  @      == (\num_of int j; 0 <= j && j < n;
  @                         a[j] == i);        @*/
public /*@ pure @*/ int occurrences(int i);
```

My Solution: Method extractMin

```
/** Return and delete the minimum element. */
/*@ requires 0 < n;
  @ assignable n, a, a[*];
  @ ensures n == \old(n-1);
  @ ensures \result ==
  @   \old((\min int j; 0 <= j && j < n; a[j]));
  @ ensures (\forall int j; 0 <= j && j < \old(n);
  @      (\old(a[j]) != \result
  @        && occurrences(\old(a[j]))
  @            == \old(occurrences(a[j])))
  @  || (\old(a[j]) == \result
  @        && occurrences(\old(a[j]))
  @            == \old(occurrences(a[j])-1))); @*/
public int extractMin();
```

Goals of the Tools

jmlc: Find violations at runtime.

jmlunit: Aid/automate unit testing.

ESC/Java2: Warn about likely runtime exceptions and violations.

Getting the Tools

Links to all tools:

- jmlspecs.org's download page.

Individual tools:

- Common JML tools
  sourceforge.net/projects/jmlspecs/
- ESC/Java2
  http://kind.ucd.ie/products/opensource/ESCJava2/
- The Mobius Program Verification Environment (PVE)
  http://kind.ucd.ie/products/opensource/Mobius/

Using jmlc, the Runtime Checker

Example

```
$ jmlc -Q -e BagOfInt.java BagOfIntMain.java
$ jmlrac BagOfIntMain
```

Writing Tests Using Assert

```
int[] mine
   = new int[] {0, 10, 20, 30, 40, 10};
BagOfInt b = new BagOfInt(mine);
System.out.println(
   "b.occurrences(10) == "
    + b.occurrences(10));
//@ assert b.occurrences(10) == 2;
//@ assert b.occurrences(5) == 0;
int em1 = b.extractMin();
//@ assert em1 == 0;
int em2 = b.extractMin();
//@ assert em2 == 10;
int em3 = b.extractMin();
//@ assert em2 == 10;
```

Using jmlc, the Runtime Checker

```
org...JMLInternalExceptionalPostconditionError:
 by method BagOfInt.occurrences regarding spec...s at
   File "BagOfInt.jml", line 21, character 14, when
    'jml$e' is ...ArrayIndexOutOfBoundsException: 6
   at BagOfInt.main(BagOfInt.java:2120)
Exception in thread "main"
```

```
  /*@ ensures \result
    @      == (\num_of int j; 0 <= j && j < n;
    @                         a[j] == i);        @*/
  public /*@ pure @*/ int occurrences(int i);
```

Using jmlc with jmlunit

Example

CLASSPATH includes:

- `.`
- `junit.jar` (version 3.8.1)
- `JML/bin/jml-release.jar`

```
$ jmlunit -i BagOfInt.java
```

Edit `BagOfInt_JML_TestData.java`

```
$ javac BagOfInt_JML_Test*.java
$ jmlc -Q -e BagOfInt.java
$ jmlrac BagOfInt_JML_Test
```

Using jmlc with jmlunit

```
.....F.F.F.F.F.F.F.F.F.F.F.F.F.F.......F.F.....
Time: 0.01
There were 16 failures:
1) occurrences:0(BagOfInt_JML_Test$TestOccurrences)
  junit.framework.AssertionFailedError:
    Method 'occurrences' applied to
    Receiver: {3, 4, 2, 3, 3}
    Argument i: 0
Caused by: ...JMLExitExceptionalPostconditionError:
by: method BagOfInt.occurrences regarding spec...s at
   File "BagOfInt.jml", line 21, character 14, when
    'jml$e' is ...ArrayIndexOutOfBoundsException: 5
```

Using ESC/Java2

Example

```
$ CLASSPATH=.
$ export CLASSPATH
$ escjava2 -nonNullByDefault BagOfInt.java
```

Using ESC/Java2

```
BagOfInt ...
  Prover started:0.03 s 15673776 bytes
    [2.013 s 15188656 bytes]

BagOfInt: BagOfInt(int[]) ...
---------------------------------------------------
BagOfInt.java:11: Warning:
      Postcondition possibly not established (Post)
  }
  ^
Associated declaration is
".\BagOfInt.jml", line 14, col 6:
    @ ensures (\forall int i; 0 <= i && i < n;
      ^
```

Tip: Use JML Assert Statements

JML assert statements

- All JML features.
- No side effects.

Java assert statements

- Only Java expressions.
- Can have side effects.

Tip: Use JML Assume Statements

`assume` *P*;

- Claims *P* is true.
- Checked by the RAC like **`assert`** *P*;
- Blame other party if false.
- Assumed by ESC/Java and static tools.

Assume Statements and Verification

```
//@ requires P;
//@ ensures Q;
public void m() {
  S
}
```

generates:

```
public void m() {
  //@ assume P;
  S
  //@ assert Q;
}
```

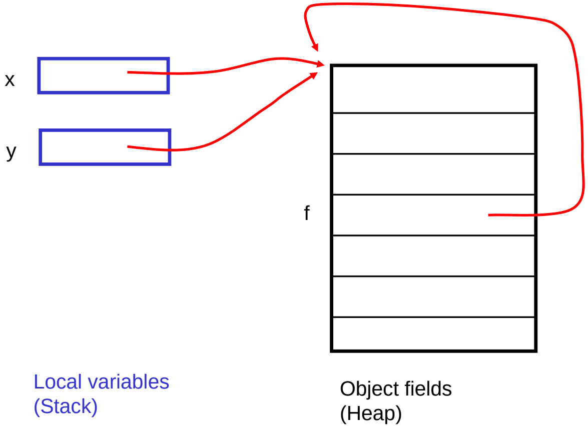
Assume Statements and Verification

```
//@ requires P;
//@ ensures Q;
public void m() {
  S
}
```

generates:

```
//@ assert P;
o.m();
//@ assume Q;
```

Pitfall: Aliasing in Java



Aliasing and Object Identity

JML Uses Java's Indirect Model for Objects

For objects x and y, $x == y$ means:

- x and y have same address.
- x and y are aliased.
- Changing of $x.f$ also changes $y.f$.

Aliasing caused by:

- Assignment ($x = y$).
- Method calls
  - Passing field $o.y$ to formal x.
  - Passing both x and y to different formals.
  - Etc.

Pitfall: Aliasing

Question

What's wrong with this? How to fix it?

```
public class Counter {

  private /*@ spec_public @*/ int val;

  //@ assignable val;
  //@ ensures val == \old(val + y.val);
  //@ ensures y.val == \old(y.val);
  public void addInto(Counter y)
  { val += y.val; }
}
```

Pitfall: Aliasing

Question

What's wrong with this? How to fix it?

```
public class Counter {

  private /*@ spec_public @*/ int val;

  //@ assignable val;
  //@ ensures val == \old(val + y.val);
  //@ ensures y.val == \old(y.val);
  public void addInto(Counter y)
  { val += y.val; }
}
```

Revised Counter to Fix the Problem

```
public class Counter2 {

  private /*@ spec_public @*/ int val;

  //@ requires this != y;
  //@ assignable val;
  //@ ensures val == \old(val + y.val);
  //@ ensures y.val == \old(y.val);
  public void addInto(Counter2 y)
  { val += y.val; }
}
```

Pitfall: Representation Exposure

```
class SortedInts {
  private /*@ spec_public @*/ int[] a;

  /*@ public invariant (\forall int i, j;
    @        0 <= i && i < j && j < a.length;
    @        a[i] <= a[j]);       @*/

  /*@ requires 0 < a.length;
    @ ensures \result == a[0];
    @ ensures (\forall int i, j;
    @        0 <= i && i < a.length;
    @        \result <= a[i]);      @*/
  public /*@ pure @*/ int first()
  { return a[0]; }
```

Pitfall: Representation Exposure

Question

What's wrong with this? How to fix it?

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

/*@ requires (\forall int i, j;
  @        0 <= i && i < j && j < inp.length;
  @        inp[i] <= inp[j]);
  @ assignable a;
  @ ensures a == inp;           @*/
public SortedInts(int[] inp)
{ a = inp; }
```

Pitfall: Representation Exposure

Question

What's wrong with this? How to fix it?

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

/*@ requires (\forall int i, j;
  @        0 <= i && i < j && j < inp.length;
  @        inp[i] <= inp[j]);
  @ assignable a;
  @ ensures a == inp;          @*/
public SortedInts(int[] inp)
{ a = inp; }
```

Revised SortedInts Using Universes (jmlc)

```
class SortedInts2 {
  private /*@ spec_public rep @*/ int[] a;
```

Revised Using Universes (jmlc)

```
/*@ requires (\forall int i, j;
  @     0 <= i && i < j && j < inp.length;
  @     inp[i] <= inp[j]);
  @ assignable a;
  @ ensures \fresh(a);
  @ ensures a.length == inp.length;
  @ ensures (\forall int i;
  @       0 <= i && i < inp.length;
  @       a[i] == inp[i]);          @*/
public SortedInts2(int[] inp) {
  a = new /*@ rep @*/ int[inp.length];
  for (int i = 0; i < a.length; i++) {
      a[i] = inp[i];
} }
```

Revised Using Owner (ESC/Java2)

```
class SortedInts3 {
  private /*@ spec_public @*/ int[] a;
  //@ public invariant a.owner == this;
```

Revised Using Owner (ESC/Java2)

```
/*@ requires inp.owner != this;
  @ requires (\forall int i, j;
  @     0 <= i && i < j && j < inp.length;
  @     inp[i] <= inp[j]);
  @ assignable a;
  @ ensures \fresh(a);
  @ ensures a.length == inp.length;
  @ ensures (\forall int i;
  @       0 <= i && i < inp.length;
  @       a[i] == inp[i]);          @*/
public SortedInts3(int[] inp) {
```

Revised Using Owner (ESC/Java2)

```
public SortedInts3(int[] inp) {
  a = new int[inp.length];
  //@ set a.owner = this;
  for (int i = 0; i < a.length; i++) {
      a[i] = inp[i];
  }
}
```

Pitfall: Undefined Expressions

Question

What's wrong with this? How to fix it?

```
public class ScreenPoint {

  private /*@ spec_public @*/ int x, y;
  //@ public invariant 0 <= x && 0 <= y;

  //@ requires 0 <= cs[0] && 0 <= cs[1];
  //@ assignable x, y;
  //@ ensures x == cs[0] && y == cs[1];
  public ScreenPoint(int[] cs)
  { x = cs[0]; y = cs[1]; }
```

Protective Version of ScreenPoint

```
public class ScreenPoint2 {

  private /*@ spec_public @*/ int x, y;
  //@ public invariant 0 <= x && 0 <= y;

  //@ requires 2 <= cs.length;
  //@ requires 0 <= cs[0] && 0 <= cs[1];
  //@ assignable x, y;
  //@ ensures x == cs[0] && y == cs[1];
  public ScreenPoint2(int[] cs)
  { x = cs[0]; y = cs[1]; }
```

Writing Protective Specifications

- Clauses evaluated left to right.
- Short-circuit operators can prevent evaluation.
  - *G* `&&` *P*, *G* `||` *P*
  - *G* **==>** *P*, *G* **<==** *P*
- Use multiple clauses (equivalent to `&&`).

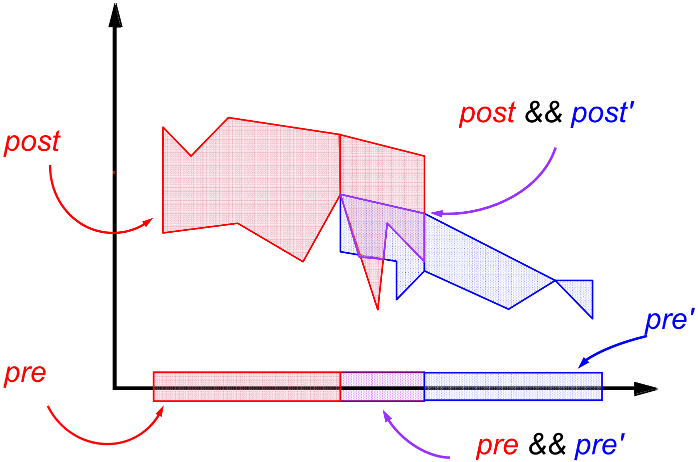
Multiple Specification Cases

- For different preconditions.
- May overlap.
- Used to specify exceptions.
- Used with specification inheritance.

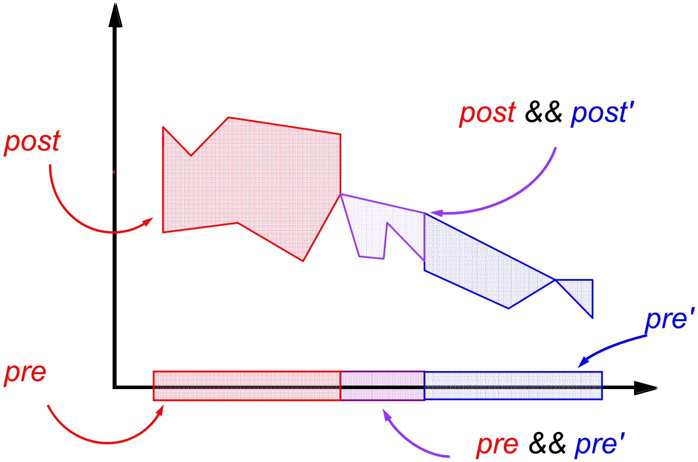
Multiple Specification Cases

```
private /*@ spec_public @*/ int age;

/*@   requires 0 <= a && a <= 150;
  @   assignable age;
  @   ensures age == a;
  @ also
  @   requires a < 0;
  @   assignable \nothing;
  @   ensures age == \old(age);
  @*/
public void setAge(int a)
{ if (0 <= a && a <= 150) { age = a; } }
```

Semantics of Multiple Cases



Semantics of Multiple Cases



Meaning of 'also'

```
  requires 0 <= a && a <= 150;
  assignable age;
  ensures age == a;
also
  requires a < 0;
  assignable \nothing
  ensures age == \old(age);
```

Meaning of 'also'

```
  requires 0 <= a && a <= 150;
  assignable age;
  ensures age == a;
also
  requires a < 0;
  assignable age;
  ensures age == \old(age)
      && \only_assigned(\nothing);
```

Meaning of 'also'

```
requires (0 <= a && a <= 150) || a < 0;
assignable age;
ensures \old(0 <= a && a <= 150)
     ==> (age == a);
ensures \old(a < 0)
     ==> (age == \old(age)
          && \only_assigned(\nothing));
```

Notation for Method Specification in T

```
public interface T {
  //@ requires pre;
  //@ ensures post;
  void m();
}
```

$T \triangleright (pre, post)$

Join of Specification Cases, $\sqcup^S$

Definition

If $T' \triangleright (pre', post')$, $T \triangleright (pre, post)$, $S \leq T'$, $S \leq T$, then

$$(pre', post') \sqcup^S (pre, post) = (p, q)$$

where $p = pre' \;\texttt{||}\; pre$
and $q = (\texttt{\textbackslash old}(pre') \;\texttt{==>}\; post') \;\texttt{\&\&}\; (\texttt{\textbackslash old}(pre) \;\texttt{==>}\; post)$
and $S \triangleright (p, q)$.

Client's View of Multiple Cases

Client can verify by:

- Picking one spec case.
  - Assert precondition.
  - Assume frame and postcondition.
- Picking several cases.
  - Compute their join.
  - Assert joined precondition.
  - Assume frame and joined postcondition.

Implementor's View of Multiple Cases

- Verify each case, or
- Verify their join.

Background for Specifying Exceptions

Java Exceptions:

- Unchecked (RuntimeException):
  - Client avoidable (use preconditions).
  - Implementation faults (fix them).
- Checked:
  - Clients can't avoid (efficiently).
  - Condition simultaneous with use (permissions).
  - Alternative returns (not found, EOF, ...).

When to Specify Exceptions

Unchecked exceptions:

- Don't specify them.
- Just specify the normal cases.

Checked exceptions

- Specify them.

JML Features for Exception Specification

- `exceptional_behavior` spec cases.
- `signals_only` clause.
- `signals` clause.

Exceptional Specification Example

```
public class Actor {

  private /*@ spec_public @*/ int age;
  private /*@ spec_public @*/ int fate;

  //@ public invariant 0 <= age && age <= fate;
```

Exceptional Specification Example

```
/*@   public normal_behavior
  @     requires age < fate - 1;
  @     assignable age;
  @     ensures age == \old(age+1);
  @ also
  @   public exceptional_behavior
  @     requires age == fate - 1;
  @     assignable age;
  @     signals_only DeathException;
  @     signals (DeathException e)
  @              age == fate;
  @*/
public void older()
  throws DeathException
```

Underspecification of Exceptions

Question

How would you specify this,
ignoring the exceptional behavior?

Underspecification of Exceptions

```
/*@ public normal_behavior
  @    requires age < fate - 1;
  @    assignable age;
  @    ensures age == \old(age+1);
  @*/
public void older()
  throws DeathException
```

Heavyweight Behavior Spec Cases

Presumed Complete

`normal_behavior`, `exceptional_behavior`

- Say how method can terminate.
- Maximally permissive/useless defaults.

`behavior`

- Doesn't specify normal/exceptional.
- Can use to underspecify normal/exceptional.

Lightweight Specification Cases

Presumed Incomplete

- Don't use a behavior keyword.
- Most defaults technically `\not_specified`.

Semantics of signals_only

- **`signals_only`** $T_1, \ldots, T_n$`;`
  - Exception thrown to caller must subtype one $T_1, \ldots, T_n$.
- Can't use in **`normal_behavior`**
- At most one **`signals_only`** clause per spec case.
- Default for omitted clause
  - if method declares **`throws`** $T_1, \ldots, T_n$,
    then **`signals_only`** $T_1, \ldots, T_n$`;`.
  - else **`signals_only`** `\nothing;`.

Signals Clause

- Specifies, when exception thrown,
    - State of exception object.
    - Other state.
- Not very useful.
- Tip: normally omit.

Pitfalls in Exceptional Specification

- Can’t return normally *and* throw exception.
- So preconditions shouldn’t overlap.

Question

What happens if they overlap?

Exercise Using Multiple Cases

Exercise

Specify the $3x + 1$ *or "hailstone" function, h, such that:*

$$h(n) = \begin{cases} (3 \times n + 1)/2, & \textit{if } n > 0 \textit{ is odd} \\ n/2, & \textit{if } n > 0 \textit{ is even} \end{cases}$$

and h is undefined on negative numbers.

My Answer

```
/*@    requires 0 < n;
  @    requires n % 2 != 0;
  @    ensures \result == (3*n+1)/2;
  @ also
  @    requires 0 < n;
  @    requires n % 2 == 0;
  @    ensures \result == n/2;
  @*/
public static /*@ pure @*/ int h(int n)
```

My Answer, Using Nesting

```
/*@ requires 0 < n;
  @ {|
  @     requires n % 2 != 0;
  @     ensures \result == (3*n+1)/2;
  @ also
  @     requires n % 2 == 0;
  @     ensures \result == n/2;
  @ |}   @*/
public static /*@ pure @*/ int h(int n)
```

Outline

Abstraction in Specification

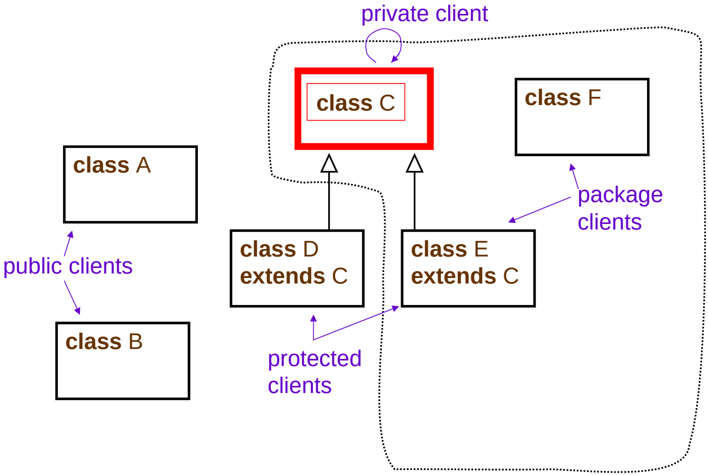
Why use abstraction?

- Ease maintenance by information hiding.
- Readability:
  - Avoid quantifiers.
  - Repeated expressions.
- Specify when no fields available
  Java `interface`s.

Features Supporting Abstraction

- `model` fields and `represents` clauses.
- `pure model` methods.
- `pure` methods.
- `protected` invariants, spec cases, etc.
- `private` invariants, spec cases, etc.

Kinds of Clients



Views of Specifications

Modifier	Declarations in *C* visible to code in:
Private	*C*
(None = package)	*C*'s package
Protected	*C*'s subclasses, *C*'s package
Public	all

Privacy and Modular Soundness

Specifications visible to module *M*:

- Can only mention members visible to *M*.
  - For maintenance.
  - For understandability.
- Must contain all of *M*'s obligations.
  - For sound modular verification.

Privacy and Modular Soundness

Question

Can private fields be mentioned in public specifications?

Question

Can non-trivial preconditions be hidden from clients?

Question

What should a client assume is the precondition of a method with no visible specification cases?

Question

If invariant inv depends on field f, can inv be less visible than f?

Privacy and Modular Soundness

Question

Can private fields be mentioned in public specifications?

Question

Can non-trivial preconditions be hidden from clients?

Question

What should a client assume is the precondition of a method with no visible specification cases?

Question

If invariant inv depends on field f, can inv be less visible than f?

Privacy and Modular Soundness

Question

Can private fields be mentioned in public specifications?

Question

Can non-trivial preconditions be hidden from clients?

Question

What should a client assume is the precondition of a method with no visible specification cases?

Question

If invariant inv depends on field f, can inv be less visible than f?

Privacy and Modular Soundness

Question

Can private fields be mentioned in public specifications?

Question

Can non-trivial preconditions be hidden from clients?

Question

What should a client assume is the precondition of a method with no visible specification cases?

Question

If invariant inv depends on field f, can inv be less visible than f?

Model Fields for Data Abstraction

Model fields:

- Just for specification.
- Abstraction of Java fields.
- Value from `represents`.

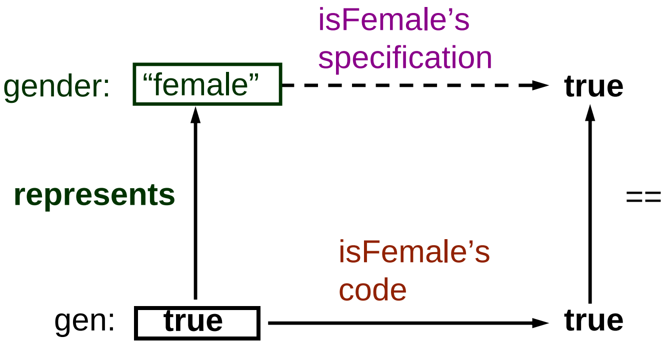
Model Field in an Interface

```
public interface Gendered {
  //@ public model instance String gender;

  //@ ensures \result == gender.equals("female");
  public /*@ pure @*/ boolean isFemale();
}
```

Represents Clauses

```
public class Animal implements Gendered {
  protected boolean gen; //@ in gender;
  /*@ protected represents
    @      gender <- (gen ? "female" : "male");
    @*/
  public /*@ pure @*/ boolean isFemale() {
    return gen;
  }
```

Correctness with Model Fields



Example of Using Model Fields

Question

Is Animal's constructor (below) correct?

```
protected boolean gen; //@ in gender;
/*@ protected represents
  @     gender <- (gen ? "female" : "male");
  @*/

/*@ requires g.equals("female")
  @        || g.equals("male");
  @ assignable gender;
  @ ensures gender.equals(g);  @*/
public Animal(final String g)
{ gen = g.equals("female"); }
```

Example of Using Model Fields

Yes!

```
protected boolean gen; //@ in gender;
/*@ protected represents
  @     gender <- (gen ? "female" : "male");
  @*/

/*@ requires g.equals("female")
  @         || g.equals("male");
  @ assignable gender;
  @ ensures gender.equals(g);  @*/
public Animal(final String g)
{ gen = g.equals("female"); }
```

Semantics of spec_public

```
protected /*@ spec_public @*/ int age = 0;
```

shorthand for:

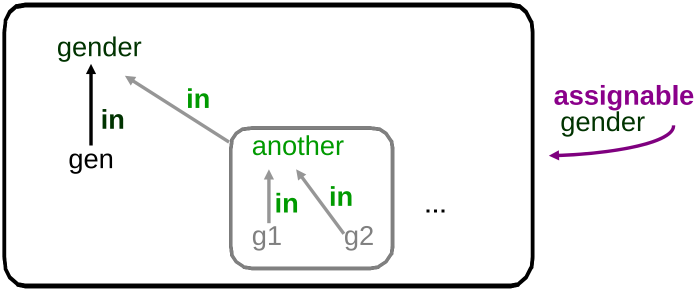
```
//@ public model int age;
//@ protected int _age = 0; //@ in age;
//@ protected represents age <- _age;
```

and rewriting Java code to use `_age`.

Data Groups for Assignable Clauses

- Each field is a data group.
- Membership by `in` clauses.
- Model field's group contains fields used in its `represents`.

Data Groups and Assignable Picture



The Semantics of Assignable

assignable x, y;

means:

method only assigns to (concrete) members of $DG(x) \cup DG(y)$.

Question

What does **assignable** `gender;` *mean?*

In Clauses for Declarations

```
private T x; //@ in g;
```

- Immediately follows declaration
- Same visibility as declaration.

JML ensures that:

- If $f \in DG(g)$, then g visible where f is.
- If f and g visible, can tell if $f \in DG(g)$.

Data Group Visibility and Reasoning

Question

Can assigning to `age` *change* `gender`*?*

Type-Level Specification Features

- fields, `in`, `represents`
- `invariant`
- `initially`
- `constraint`

Initially Clauses

- Hold in constructor post-states.
- Basis for datatype induction.

```
import java.util.*;
public class Patient extends Person {
  //@ public invariant 0 <= age && age <= 150;

  protected /*@ spec_public rep @*/ List log;
  //@ public initially log.size() == 0;
```

History Constraints

- Relate pre-states and post-states.
- Justifies inductive step in datatype induction.

History Constraints

```
import java.util.*;
public class Patient extends Person {

  protected /*@ spec_public rep @*/ List log;

  /*@ public constraint
    @       \old(log.size()) <= log.size();
    @ public constraint (\forall int i;
    @       0 <= i && i < \old(log.size());
    @       log.get(i).equals(\old(log.get(i))));
    @*/
```

Helper Methods and Constructors

A **`helper`** method or constructor is:

- **`private`**
- Exempt from invariants and history constraints.
  - Cannot assume them.
  - Need not establish them.

Ghost fields and Local Variables

- Specification-only data.
- No `represents` clause.
- Value from initialization and `set` statements.
- Locals useful for loop invariants, termination, etc.

Owner is a Ghost Field

Declaration:

```
public class Object {
  //@ public ghost Object owner = null;
  /* ... */
}
```

Assignment:

```
  //@ set a.owner = this;
```

Outline

Problems

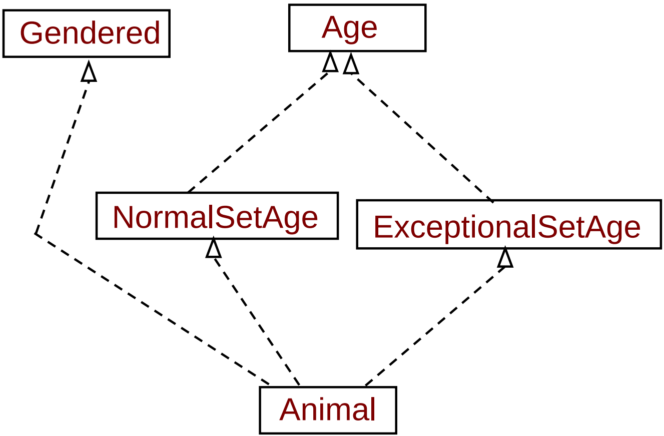
- Duplication of specifications in subtypes.
- Modular verification when use:
  - Subtyping, and
  - Dynamic dispatch.

Specification Inheritance Approach

Inherit:

- Instance fields.
- Type specifications.
- Instance methods.
- Method specification cases.

Multiple Inheritance Example



Age and NormalSetAge

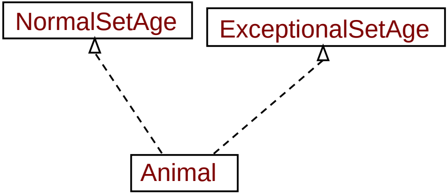
```
public interface Age {
  //@ model instance int age;
}

public interface NormalSetAge
           implements Age {
  /*@ requires 0 <= a && a <= 150;
    @ assignable age;
    @ ensures age == a;   @*/
  public void setAge(int a);
}
```

ExceptionalSetAge

```
public interface ExceptionalSetAge
             implements Age {
  /*@ requires a < 0;
    @ assignable \nothing;
    @ ensures age == \old(age);    @*/
  public void setAge(int a);
}
```

What About Animal's setAge method?

- It's both.
- Should obey both specifications.



Single Inheritance also

Question

What is the specification of Animal's `isFemale` method?

```
public interface Gendered {
  //@ ensures \result == gender.equals("female");
  public /*@ pure @*/ boolean isFemale();
}

public class Animal implements Gendered {

  public /*@ pure @*/ boolean isFemale() {
    return gen;
  }
}
```

Adding to Specification in Subtype

Use of ‘also’ Mandatory

```
import java.util.*;
public class Patient extends Person {
  protected /*@ spec_public @*/
     boolean ageDiscount = false;  //@ in age;

  /*@ also
    @   requires (0 <= a && a <= 150) || a < 0;
    @   ensures 65 <= age ==> ageDiscount;  @*/
  public void setAge(final int a) {
    super.setAge(a);
    if (65 <= age) { ageDiscount = true; }
  }
```

Method Specification Inheritance

Question

What is the extended specification of `Patient`*'s* `setAge` *method?*

Extended Specification of SetAge

```
/*@    requires 0 <= a && a <= 150;
  @    assignable age;
  @    ensures age == a;
  @ also
  @    requires a < 0;
  @    assignable age;
  @    ensures age == \old(age);    @*/

/*@ also
  @    requires (0 <= a && a <= 150) || a < 0;
  @    ensures 65 <= age ==> ageDiscount;  @*/
```

Avoiding Duplication of Preconditions

```
/*@    requires 0 <= a && a <= 150;
  @    assignable age;
  @    ensures age == a;
  @ also
  @    requires a < 0;
  @    assignable age;
  @    ensures age == \old(age);    @*/

/*@ also
  @    requires \same;
  @    ensures 65 <= age ==> ageDiscount;   @*/
```

Method Specification Inheritance

Question

In JML, can you override a method
and make its precondition more restrictive?

No, You Can't Strengthen Preconditions

Can Point Out Special Cases

```
public class Person extends Animal {

  /*@ also
    @   requires 65 <= age;
    @   assignable age, ageDiscount;
    @   ensures ageDiscount;   @*/
  public void setAge(final int a);
```

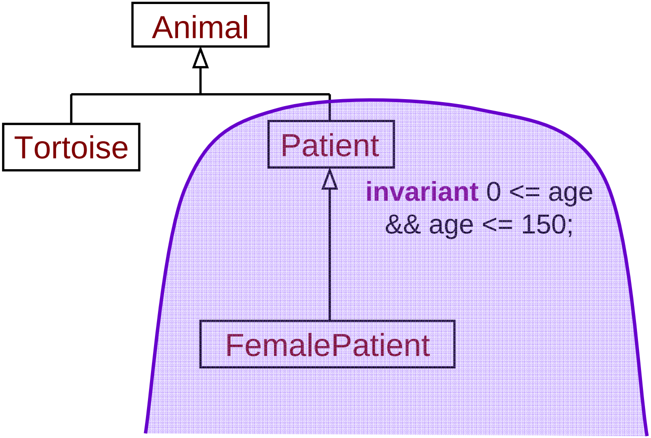
Inheritance of Type Specifications

Obeyed by all subtypes:

- Invariants.
- Initially Clauses.
- History Constraints.

Invariants Obeyed by Subtypes

Not a Syntactic Sugar



Notation for Describing Inheritance

T's Added Specification

Declared in T (without inheritance):

$added\_inv^T$	invariant
$added\_hc^T$	history constraint
$added\_init^T$	initially predicate
$added\_spec^T_m$	m's specification

Other Notations:

$$supers(T) = \{U \mid T \leq U\}$$

$$methods(\mathcal{T}) = \{m \mid m \text{ declared in } T \in \mathcal{T}\}$$

Specification Inheritance's Meaning

Extended Specification of T

Methods: for all $m \in methods(supers(T))$

$$ext\_spec_m^T = \sqcup^T \{added\_spec_m^U \mid U \in supers(T)\}$$

Invariant: $ext\_inv^T = \bigwedge\{added\_inv^U \mid U \in supers(T)\}$

Constraint: $ext\_hc^T = \bigwedge\{added\_hc^U \mid U \in supers(T)\}$

Initially: $ext\_init^T = \bigwedge\{added\_init^U \mid U \in supers(T)\}$

Invariant Inheritance

```
public class FemalePatient extends Patient {
  //@ public invariant gender.equals("female");
```

Extended Invariant:

$$\mathit{added\_inv}^{\texttt{Gendered}} \;\&\&\; \mathit{added\_inv}^{\texttt{Animal}}$$
$$\&\&\; \mathit{added\_inv}^{\texttt{Patient}}$$
$$\&\&\; \mathit{added\_inv}^{\texttt{FemalePatient}}$$

Invariant Inheritance

```
public class FemalePatient extends Patient {
  //@ public invariant gender.equals("female");
```

Extended Invariant:

```
  true && true
  && 0 <= age && age <= 150
      && (\forall int i;
            0 <= i && i < log.size();
            log.get(i) instanceof rep String)
  && gender.equals("female")
```

Modular Verification Problem

Reasoning about dynamic dispatch:

```
Gendered e = (Gendered)elems.next();
if (e.isFemale()) {
  //@ assert e.gender.equals("female");
  r.add(e);
}
```

How to verify?

- Avoiding case analysis for all subtypes.
- Reverification when add new subtypes.

Supertype Abstraction

Use static type's specification.
Example:

```
Gendered e = (Gendered)elems.next();
if (e.isFemale()) {
  //@ assert e.gender.equals("female");
  r.add(e);
}
```

- Static type of `e` is `Gendered`.
- Use specification from `Gendered`.

Static Type's Specification

```
public interface Gendered {
  //@ public model instance String gender;

  //@ ensures \result == gender.equals("female");
  public /*@ pure @*/ boolean isFemale();
}
```

Supertype Abstraction in General

Use static type's specifications to reason about:

- Method calls.
- Invariants.
- History constraints.
- Initially predicates.

Supertype Abstraction Summary

T o = createNewObject();
//@ **assume** $o.ext\_init^T$ && $o.ext\_inv^T$;

/* ... */

//@ **assert** $o.ext\_pre_m^T$;
$o.m()$;
//@ **assume** $o.ext\_post_m^T$;
//@ **assume** $o.ext\_inv_m^T$ && $o.ext\_hc^T$;

Reasoning Without Supertype Abstraction

Case analysis:

- Case for each potential dynamic type.
- Can exploit dynamic type's specifications.

Case Analysis + Supertype Abstraction

- Use `instanceof` for case analysis.
- Downcast, use supertype abstraction.

Case Analysis + Supertype Abstraction

```
/*@ requires p instanceof Doctor
  @        || p instanceof Nurse; @*/
public boolean isHead(final Staff p) {
  if (p instanceof Doctor) {
    Doctor doc = (Doctor) p;
    return doc.getTitle().startsWith("Head");
  } else {
    Nurse nrs = (Nurse) p;
    return nrs.isChief();
  }
}
```

Supertype Abstraction's Soundness

Valid if:

- Invariants etc. hold as needed (in pre-states), and
- Each subtype is a behavioral subtype.

Assumption about Invariants

```
assert Pre;
```

Assumption about Invariants

```
assert Pre;
                assume Pre && Inv;
                assert Post && Inv;
assume Post;
```

Invariant Methodology

Potential Problems:

- Representation exposure
- Reentrance

Relevant invariant semantics:

- Ownership type system
- Re-establish invariant when call

Guarantees:

- Invariant holds at start of method

Open Problems

- Blending with similar Spec# methodology.
- Extension to History Constraints and Initially Predicates.

Validity of Supertype Abstraction

Client's View

T o = createNewObject();
//@ **assume** $o.ext\_init^T$ && $o.ext\_inv^T$;

/* ... */

//@ **assert** $o.ext\_pre^T_m$;
$o.m()$;
//@ **assume** $o.ext\_post^T_m$;
//@ **assume** $o.ext\_inv^T_m$ && $o.ext\_hc^T$;

What Happens at Runtime

Suppose we have

```
public T createNewObject() {
    return new T'();
}
```

Validity of Supertype Abstraction

Client's View

```
T o = createNewObject();
//@ assume o.ext_init^T && o.ext_inv^T;

/* ... */

//@ assert o.ext_pre^T_m;
o.m();
//@ assume o.ext_post^T_m;
//@ assume o.ext_inv^T_m && o.ext_hc^T;
```

Validity of Supertype Abstraction

Implementation (Subtype) View

```
T o = createNewObject(); // new T'()
//@ assert o.ext_init^T' && o.ext_inv^T';

/* ... */

//@ assume o.ext_pre_m^T';
o.m();
//@ assert o.ext_post_m^T';
//@ assert o.ext_inv_m^T' && o.ext_hc^T';
```

Behavioral Subtyping

Definition

Suppose $T' \leq T$. Then
T' *is a strong behavioral subtype of* T if and only if:

- for all instance methods m in T,

$$ext\_spec_m^{T'} \sqsupseteq^{T'} ext\_spec_m^{T}$$

- and whenever **`this`** has type T':

$$ext\_inv^{T'} \Rightarrow ext\_inv^{T},$$
$$ext\_hc^{T'} \Rightarrow ext\_hc^{T}, \text{ and}$$
$$ext\_init^{T'} \Rightarrow ext\_init^{T}.$$

Method Specification Refinement

With respect to T'

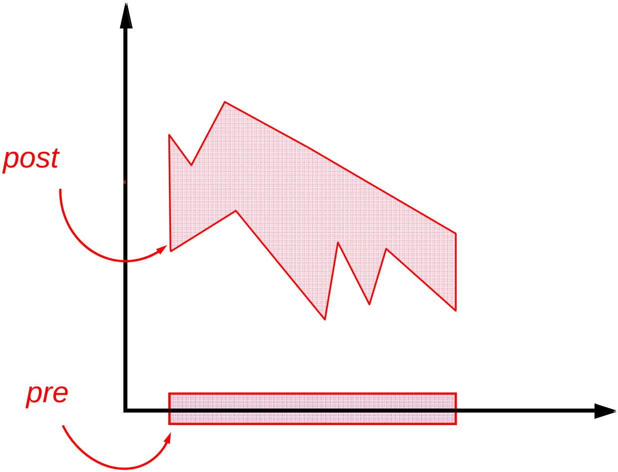
Notation:

$$(pre', post') \sqsupseteq^{T'} (pre, post)$$

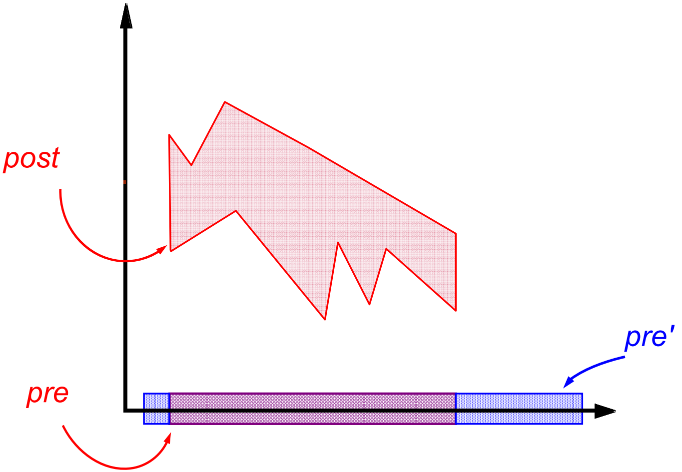
Means:

- Every correct implementation of $(pre', post')$ satisfies $(pre, post)$.

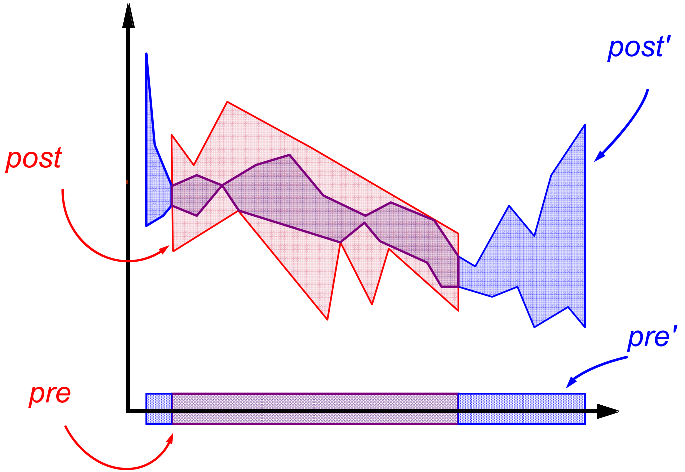
Method Specification Refinement



Method Specification Refinement



Method Specification Refinement



Proving Method Refinements

Theorem

Suppose $T' \triangleright (pre', post')$ *and* $T \triangleright (pre, post)$ *specify m. Then*

$$(pre', post') \sqsupseteq^{T'} (pre, post)$$

if and only if:

$$Spec(T') \vdash pre \texttt{ \&\& (this instanceof } T'\texttt{)} \Rightarrow pre'$$

and

$$Spec(T') \vdash \texttt{\textbackslash old(}pre \texttt{ \&\& (this instanceof } T'\texttt{))} \Rightarrow (post' \Rightarrow post).$$

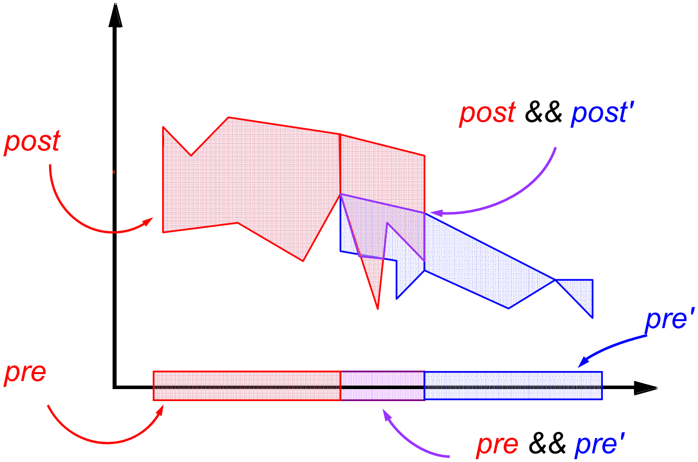
`also` Makes Refinements

Theorem

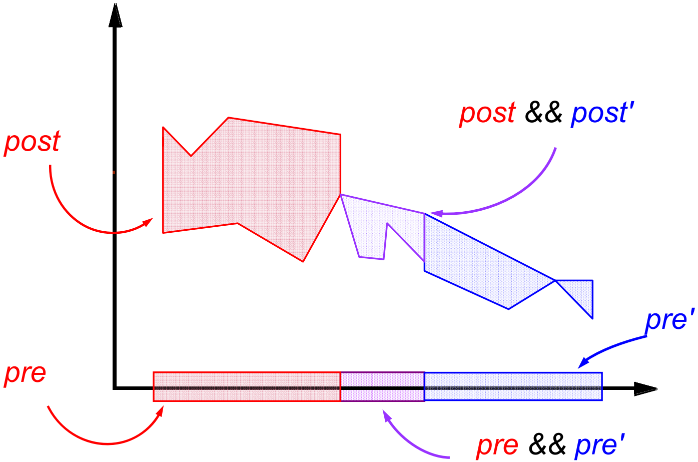
Suppose `\old` *is monotonic. Suppose* $T' \leq T$, *and* $T' \triangleright (pre', post')$ *and* $T \triangleright (pre, post)$ *specify m.*
Then

$$((pre', post') \sqcup^{T'} (pre, post)) \sqsupseteq^{T'} (pre, post).$$

Semantics of Multiple Cases



Semantics of Multiple Cases



Spec. Inheritance Forces Behavioral Subtyping

Theorem

Suppose $T' \leq T$. Then the extended specification of T' is a strong behavioral subtype of the extended specification of T.

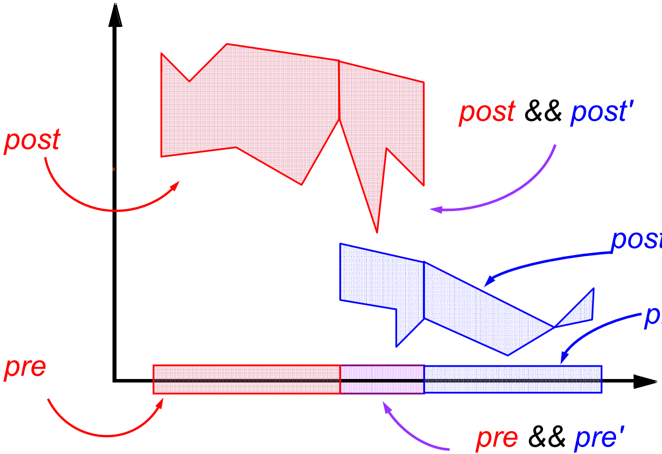
Discussion

Behavioral Subtyping and Spec. Inheritance

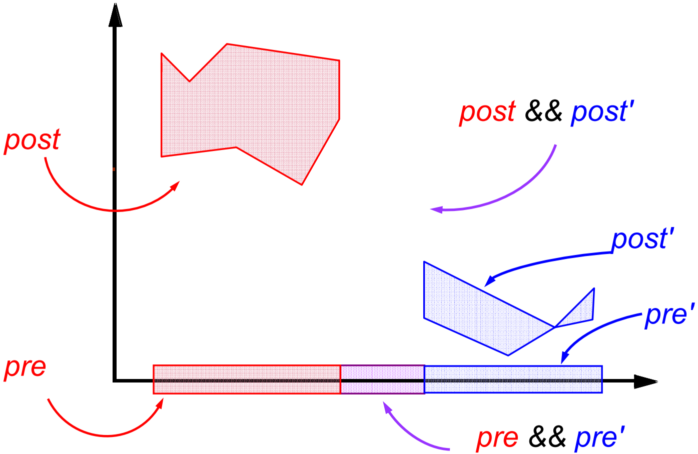
In JML:

- Every subtype inherits.
- Every subtype is a behavioral subtype.
  - Not all satisfiable.
  - Supertype must allow refinement

Unsatisfiable Refinements



Unsatisfiable Refinements



Binary Method Specification

Question

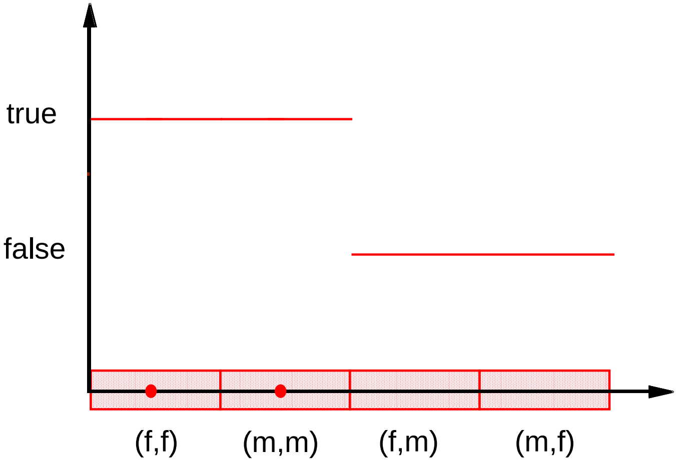
What is wrong specifying `Gender`'s `equals` method as follows?

```
/*@ also
  @    ensures obj instanceof Gendered
  @         ==> (\result
  @                == gender.equals(
  @                     ((Gendered)obj).gender));
  @*/
public /*@ pure @*/
boolean equals(/*@ nullable @*/ Object obj);
```

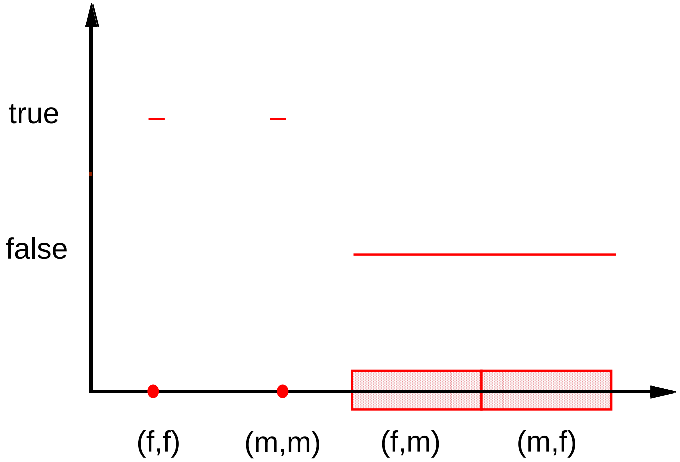
What's Wrong With It?

- Says that only gender matters.
- Refinements can't use other attributes.

Bad Equals Specification



Bad Equals Specification



Binary Method Specification

Question

How to fix it?

```
/*@ also
  @    ensures obj instanceof Gendered
  @        ==> (\result
  @              == gender.equals(
  @                   ((Gendered)obj).gender));
  @*/
public /*@ pure @*/
boolean equals(/*@ nullable @*/ Object obj);
```

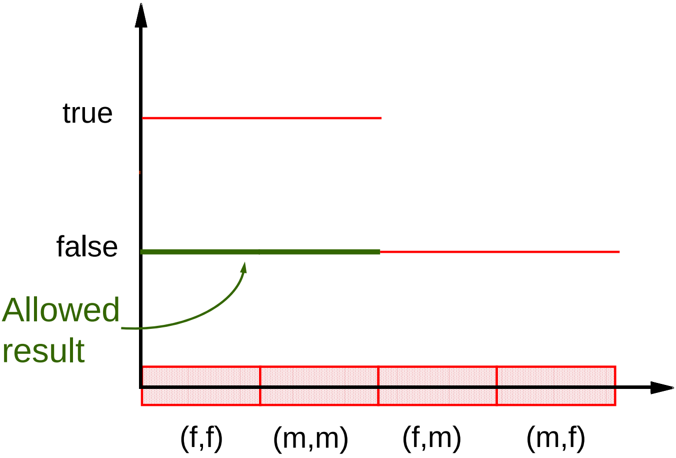
Better, Refinable Specification

Using Underspecification

```
/*@ also
  @   ensures obj instanceof Gendered
  @         ==> (\result
  @                ==> gender.equals(
  @                     ((Gendered)obj).gender));
  @*/
public /*@ pure @*/
boolean equals(/*@ nullable @*/ Object obj);
```

Better, Refinable Specification

Using Underspecification



Conclusions About Subtyping

- Supertype abstraction allows modular reasoning.
- Supertype abstraction is valid if:
  - methodology enforced, and
  - subtypes are behavioral subtypes.
- JML's `also` makes refinements.
- Specification inheritance in JML forces behavioral subtyping.
- Supertype abstraction automatically valid in JML.
- Supertype specifications must be permissive.

Outline

What Makes ESC/Java Unique?

- Encapsulates automatic theorem prover (Simplify).
- Aims to help programmers.
  - Not sound.
  - Not complete.
- Rigorously modular.

What Makes ESC/Java2 Different?

- Nearly full JML syntax parsed.
- Most JML semantics checked.
- Integrates many more static checkers.
- Multiple logics and provers.
- Eclipse integration.

Stengths of Extended Static Checking

- Push-button automation.
- Tool robustness.
- User feedback with no user specifications.
- Integration with popular IDE (Eclipse).
- Popularity in FM community.

ESC/Java's Main Weaknesses

- False positives and false negatives.
- Tool and documentation problems.
- Need for fairly complete specifications.
- Feedback hard for naive users.

Kinds of Messages Produced by ESC/Java2

Cautions or errors, from:

- Parsing.
- Type checking.

Warnings, from:

- Static checking, with Simplify (or others).

Where to Put Specifications

Put specifications in:

- A `.java` file, or
- A specification file.
    - Suffix `.refines-java`, `.refines-spec`, or `.refines-jml`.
    - No method bodies.
    - No field initializers.
    - `Foo.refines-java` starts with:

    ```
    //@ refine "Foo.java";
    ```

    - In the CLASSPATH.

ESC/Java Checks Modularly

Example

```
public abstract class ModularityDemo {

  protected byte[] b;

  public ModularityDemo()
  { b = new byte[20]; }

  public void m()
  { b[0] = 2; }
}
```

Modularity Summary

Properties you want to assume about

Fields: use a modifier (`non_null`), `invariant`, or `constraint`.

Method arguments: use a modifier (`non_null`), or `requires`.

Method results: use a modifier (`pure`, `non_null`), `assignable`, or `ensures`.

When to use `assume`

Assumptions say "fix me"

- Not sure if field or method property.
- You don't want to specify more about:
  - Domain knowledge.
  - Other libraries.
- The prover isn't smart enough.

Best to avoid `assume`.

Need for Assignable Clauses

```
public void move(int i, int j) {
  moveRight(i);
  //@ assert x == \old(x+i);
  moveUp(j);
  //@ assert y == \old(y+j);
  //@ assert x == \old(x+i); // ??
}
```

Assignable Clauses Localize Reasoning

```
//@ requires 0 <= j;
//@ requires y+j < Integer.MAX_VALUE;
//@ assignable y;
//@ ensures y == \old(y+j);
public void moveUp(int j)
```

Kinds of Warnings

Exceptions:

Runtime: Cast, Null, NegSize, IndexTooBig, IndexNegative, ZeroDiv, ArrayStore.

Undeclared: Exception.

Specification violations:

Method: Precondition, Postcondition, Modifies.

Non-null: NonNull, NonNullInit

Loop: LoopInv, DecreasesBound.

Flow: Assert, Reachable.

Class: Invariant, Constraint, Initially.

Exception Warning Example

Example

```
public class Ex {
  public void m(Object o) {
    if (!(o instanceof String)) {
      throw new ClassCastException();
    }
  }
}
```

Exception Warning Example

Output:

```
Ex: m(java.lang.Object) ...
-----------------------------------
Ex.java:6: Warning:
 Possible unexpected exception (Exception)
   }
   ^
Execution trace information:
  Executed then branch in ..., line 3, col 32.
  Executed throw in "Ex.java", line 4, col 6.
```

Turning Off Warnings

Preferred:

- Declare (e.g., runtime exceptions).
- Specify (e.g, **`requires`**).

Alternatively:

- Use **`nowarn`**.

```
//@ nowarn Exception;
```

- Use command line options (`-nowarn Exception`).

Other Kinds of Warnings

Not Covered Here

- Multithreading.
- Ownership.

Counterexample Information

- Violations can give counterexample context.
- Explain how warning could happen.
- State what prover "thinks" could be true.
- Can be hard to read.
- More details with `-counterexample` option.

Example for Reading Counterexamples

Example

```
public class Alias {
  private /*@ spec_public non_null */ int[] a
      = new int[10];
  private /*@ spec_public @*/ boolean noneg
      = true;

  /*@ public invariant noneg ==>
    @     (\forall int i;
    @         0<=i && i < a.length;
    @         a[i] >= 0);
    @*/
```

Example for Reading Counterexamples

Example

```
//@ requires 0<=i && i < a.length;
public void insert(int i, int v) {
  a[i] = v;
  if (v < 0) { noneg = false; }
}
```

Reading ESC/Java2's Feedback

```
Alias.java:17: Warning:
Possible violation of invariant (Invariant)

Associated declaration is ..., line 7, col 13:
  /*@ public invariant noneg ==>  ...
             ^
Possibly relevant .. from counterexample context:
  (vAllocTime(brokenObj) < alloc) ...

Execution trace information:
    Executed then branch in ..., line 16, col 15.

Counterexample context:
    (intFirst <= v:14.32) ...
```

Reading Relevant Items

Item	Meaning
`brokenObj`	object violating invariant
`typeof(brokenObj)`	its type
`brokenObj.(noneg:4.38)`	its `nonneg` field
`brokenObj.(a@pre:2.44)`	its `a` field
`tmp0!a:15.4`	another object

State Described By Relevant Items

Question

What does this mean?

```
typeof(brokenObj) <: T_Alias
brokenObj.(noneg:4.38) == @true
brokenObj.(noneg:4.38<1>) == @true
brokenObj.(a@pre:2.44) == tmp0!a:15.4
brokenObj != this
```

Reading Counterexample Context

Look at:

- **`this`**
- `brokenObj`

```
brokenObj.(noneg:4.38<1>) == @true
this.(noneg:4.38<1>) == bool$false
brokenObj.(a@pre:2.44) == tmp0!a:15.4
this.(a@pre:2.44) == tmp0!a:15.4
...
this != null
brokenObj != this
brokenObj != null
```

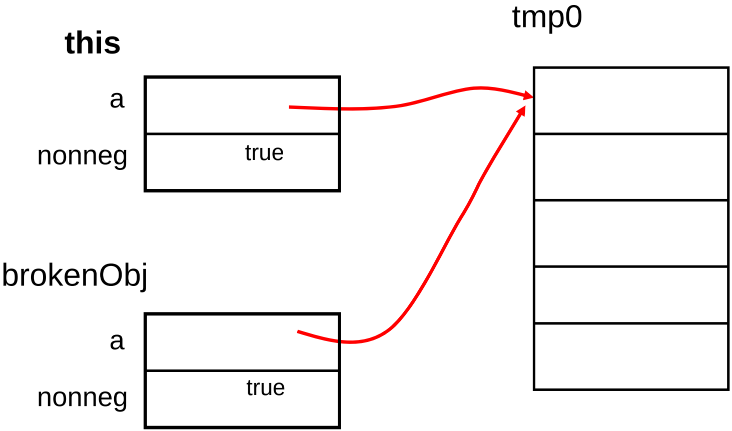
Reading Counterexample Context

Question

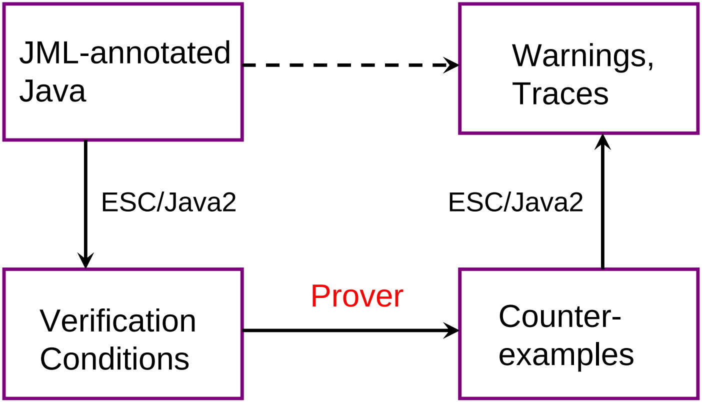
What does the context tell you?

```
brokenObj.(noneg:4.38<1>) == @true
this.(noneg:4.38<1>) == bool$false
brokenObj.(a@pre:2.44) == tmp0!a:15.4
this.(a@pre:2.44) == tmp0!a:15.4
...
this != null
brokenObj != this
brokenObj != null
```

It Tells About Aliasing



ESC/Java as a VC Generator



ESC/Java2 and Provers

Current release supports:

- Fx7 prover.
- Coq.

VC formats:

- Simplify.
- SMT-LIB.

Other Efforts

- Specification-aware dead code detector.
- Race Condition Checker.
- Houdini (creates specifications).

Outline

Advantages of Working with JML

- Reuse language design.
- Ease communication with researchers.
- Share customers.

Join us!

Opportunities in Working with JML

Or: What Needs Work

- Tool development, maintenance.
- Extensible tool architecture.
- Unification of tools.

Current Research on JML

Semantics and Design Work:

- Ownership and invariants (Peter Müller, Spec# folks)
- Multithreading (KSU group, INRIA).
- Frameworks, callbacks (Steve Shaner, David Naumann, me)

Tool Work

- Mobius effort (Joe Kiniry and others)
- Annotation Support (Jass group, Kristina Boysen)
- Testing (Mark Utting, Yoonsik Cheon, . . .).

Future Work on JML

- Tools.
- Java 1.5 support.
- Eclipse support.
- Documentation.
- Concurrency support.
- Semantic details.
- Theorem proving tie-ins, Static analysis tie-ins.
- Inference of specifications.
- Tools that give more benefits.

What Are You Interested In?

Question

What kinds of research or collaborations interest you?

Acknowledgments

Thanks to Joseph Kiniry, Erik Poll, David Cok, David Naumann, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, Patrice Chalin, Peter Müller, Werner Dietl, Arnd Poetzsch-Heffter, Rustan Leino, Al Baker, Don Pigozzi, and the rest of the JML community.

Join us at...

jmlspecs.org

Modular Reasoning

- Prove code using specifications of other modules.
- Sound, if each module satisfies specification.

Scales better than whole-program reasoning.

Supertype Abstraction for Initially

Given:

```
public class Patient extends Person {
  protected /*@ spec_public rep @*/ List log;
  //@ public initially log.size() == 0;
```

Verify:

```
  Patient p;
  if (b) { p = new Patient("male"); }
  else { p = new FemalePatient(); }
  //@ assert p.log.size() == 0;
```